
# WCAE 2003

*Proceedings of the*

# Workshop on Computer Architecture Education

*in conjunction with*

**The 30th International Symposium on Computer Architecture**

*and*

2003 Federated Computing Research Conference

Town and Country Resort and Convention Center b

San Diego, California

June 8, 2003

# Workshop on Computer Architecture Education

## Sunday, June 8, 2003

**Session 1.** Welcome and Keynote 8:45–10:00


8:45 Welcome Edward F. Gehringer, workshop organizer
8:50 Keynote address, "Teaching and teaching computer Architecture: Two very different topics (Some opinions about each)," Yale Patt, teacher, University of Texas at Austin 1


**Break** 10:00–10:30

**Session 2.** Teaching with New Architectures 10:30–11:20


10:30 "Intel Itanium floating-point architecture," Marius Cornea, John Harrison, and Ping Tak Peter Tang, Intel Corp. ..... 5
10:50 "DOP — A CPU core for teaching basics of computer architecture," Miloš Bečvář, Alois Pluháček and Jiří Daněček, Czech Technical University in Prague ..... 14
11:05 Discussion


**Break** 11:20–11:30

**Session 3.** Class Projects 11:30–12:30


11:30 "Superscalar out-of-order demystified in four instructions," James C. Hoe, Carnegie Mellon University ..... 22
11:50 "Bridging the gap between undergraduate and graduate experience in computer systems studies," Lori Carter and Scott Rae, Point Loma Nazarene University ..... 28
12:05 "Integration of computer security laboratories into computer architecture courses to enhance undergraduate curriculum," Jayantha Herath and Ajantha Herath, St. Cloud State U. and U. of Dubuque ..... 36
12:20 Discussion


**Lunch** 12:30–2:00

**Session 4.** Teaching Techniques 2:00–3:30


2:00 "Combining learning strategies in a first course in computer architecture," Patricia Teller, Manuel Nieto, and Steve Roach, U. of Texas at El Paso ..... 41
2:25 "Building resources for teaching computer architecture through electronic peer review," Edward F. Gehringer, North Carolina State University ..... 49
2:40 "Laboratory options for the computer science major," Christopher Vickery and Tamara Blain, Queens College of CUNY ..... 57
2:55 "Activating computer architecture with Classroom Presenter," Beth Simon, U. of San Diego; and Richard Anderson and Steven Wolfman, U. of Washington ..... 64
3:15 Discussion


**Break** 3:30–4:00

**Session 5.** Simulation Environments 4:00–5:40


4:00 "The Liberty simulation environment as a pedagogical tool," Jason Blome, Manish Vachhajarani, Neil Vachhajarani, and David I. August, Princeton University ..... 72
4:25 "Multimedia components for the visualization of dynamic behavior in computer architectures," Peter Marwedel and Brigit Sirocic, U. of Dortmund ..... 79
4:40 "Didactic architectures and simulator for network processor learning," Henrique Cota de Freitas and Carlos Augusto P. S. Martins, Pontifical Catholic U. of Minas Gerais ..... 86
5:00 "On the introduction of reconfigurable hardware into computer architecture education," Ross Brennan and Michael Manzke, Trinity College, Dublin ..... 96
5:15 "Use of HDLs in teaching of computer hardware courses," Zvonko Vranesic and Stephen Brown, U. of Toronto ..... 103
5:30 Discussion

# Teaching and Teaching Computer Architecture: Two Very different topics (Some Opinions about each)

Yale N. Patt
teacher, The University of Texas at Austin

## Abstract

*This year's Computer Architecture Education workshop is remarkable in its recognition that to teach computer architecture well, one has to pay attention to two things (a) teaching and (b) computer architecture. Having been doing both for a good number of years, I harbor a fair number of opinions on what one should do and what one should not do with respect to each. This talk will get into some of those opinions. With respect to teaching, I will discuss some of my Ten Commandments of Good Teaching, what I think of distance learning, political correctness, emphasis on memorization, the inability of American students to write English, the value of having students study in groups, and what I feel is often the sad misuse of technology. Most importantly, I will discuss my motivated bottom up approach to learning. With respect to teaching computer architecture, I believe the single most important point to get across is that computer architecture, if it is a science at all, is a science of tradeoffs. The student is best served if he/she thoroughly understands the fundamental principles so as to be able to make the appropriate tradeoffs in reaching a particular design objective. I also plan to discuss the use (and too often, misuse) of measurements, simulation, and real ISAs as opposed to concocted ISAs.*

## 1 Introduction

I have been asked to provide copies of the transparencies I will use in my Keynote Address. I have added some annotated text to hopefully provide some context.

Knowing from past experience that the dynamic schedule of my talks usually bears only casual resemblance to the static schedule that I prepared ahead of time, I provide these copies somewhat timidly.

Figure 1 is from a talk I gave last fall at the annual Visions Lecture of the Computer Sciences Department at The University of Texas at Austin. The subject was Education. I was asked to give my dream for an ideal future with respect to education.

- Distance Learning produces better education, not cheaper education
- We pay teachers enough that those who would opt for this career don't opt for medical school instead
- We teach high school English teachers enough English that students at the University can write two consecutive coherent sentences
- We get past this insane preoccupation with political correctness, so we can get on with the business of teaching and learning
- We stop canonizing the use of high tech education. Bad pedagogy is NOT good pedagogy if draped in technology
- We stop rewarding memorization ability, so maybe students will learn to think,...and perhaps understand

**Figure 1. Visions (Re: Education)**

The remaining figures deal with the two parts of my talk, a focus on teaching, and a focus on teaching computer architecture.

## 2 Focus on Teaching

Teaching involves at least three things: how to teach (Section 2.1), what to teach (Section 2.2), and what aids to use in the process (Section 2.3).

### 2.1 My Ten Commandments of Good Teaching

Someone suggested I come up with a set of commandments for good teaching. For historical reasons, they thought ten would be a good number. So, I set out to do it, and came up with nine. Ergo, note the tenth one. On sober reflection, that one in itself is a tenth command-

ment. So, the list on Figure 2 really does contain ten, not nine as some have commented.

- Know the material
- Want to teach
- Genuinely respect your students and show it
- Set the bar high; students will measure up
- Emphasize understanding; de-emphasize memorization
- Take responsibility for what is covered
- Don't even try to cover the material
- Encourage interruptions; don't be afraid to digress
- Don't forget those three little words
- Reserved for future use

**Figure 2. My Ten Commandments of Good Teaching**

## 2.2 Emphasis on the Fundamentals

My emphasis on teaching the fundamentals has been part of me forever. No doubt my PhD students get tired of hearing about it. For example, I believe that an appropriate PhD qualifier is not a written test on some advanced course in the graduate curriculum, but rather an oral exam on the fundamental concepts found in relevant senior level undergraduate courses.

My view is simple: research, development, problem solving are all about breaking problems down into small pieces and working with the small pieces until the

- Top-down design, Bottom-up learning for understanding
- Abstraction is vital, but...
- Not bottom-up, but "motivated" bottom-up
- Engineering is about DESIGN, first understand the components
- From Concrete to Abstract (Dijkstra notwithstanding)
- Cut through protective layers
- Memorizing is not understanding
- Students do better working in groups

**Figure 3. Some thoughts on what is important**

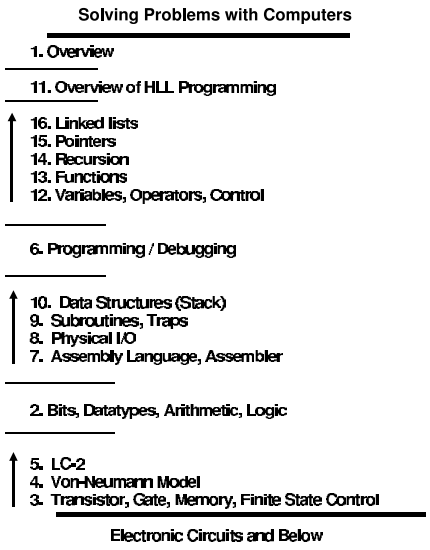


**Figure 4. My motivated bottom-up approach**

"Aha!" happens. How well someone can do that depends on how well that person has mastered the fundamentals.

My views crystallized particularly strongly with the development of our "new" introduction to computing, which we first developed at Michigan in the mid-90s[1] and later turned into a textbook, published by McGraw-Hill[2]. My view of what is important is expressed in Figure 3. More specific detail of the motivated bottom-up approach is shown in Figure 4.

## 2.3 High Tech in the Classroom

There seems to be a preoccupation with using technology in the classroom. Certainly, there is much that can be done with technology to improve learning. I am concerned that in our leap to technologize everything, we are developing some very bad pedagogy under the umbrella of "using technology." Figure 5 shows some common uses. Figure 6 lists some serious concerns.

# 3 Focus on Teaching Computer Architecture

We are lucky. We get to teach computer architecture. Some will tell you that computer architecture is dead, that the microprocessor is to computing like a brick is to buildings. Wrong. Computer architecture is the interface between what technology can provide and what the marketplace demands. Technology continues to provide more and more. We are told that within a very few

- Email
- Web site
- Power Point
- Document Reader
- Animations
- Plato, vintage 2003
- Clever attendance mechanism
- Other bookkeeping
- Text+Voice (WOW Factor, see Shriver's CDROM)[3]

**Figure 5. Some uses of high tech**

- Baseline Power Point
- Cost
- Extemporaneous Effect
- Visual/voice disconnect
- Attendance vs. Participation

**Figure 6. Some caveats associated with using high tech**

years, each chip will contain more than a billion transistors. And the marketplace continues to demand more. In fact, the higher and high performance chip becomes an important enabler. As we develop more, the marketplace dreams of more things it needs.

Contrary to being dead, computer architecture is in a constant state of high volatility. The state-of-the-art examples we studied yesterday are boring today.

Computer architecture will always be alive and healthy as long as people continue to dream up new needs for our future products. The design points may change. Not just higher performance, but higher reliability, availability, cheaper cost, and more power-sensitive designs, for example.

Within that framework, what do we teach. In my view, we focus on three things: the fundamental principles (which do not change, or change very, very slowly), the tradeoffs that always result, and the concrete implementation of those principles.

Figure 7 identifies a number of the fundamentals, Figure 8 (levels of transformation) and Figure 9 (three parts of a microarchitecture) elaborate on two of them. Figure 10 lists some concerns.

- The transformation hierarchy
- Three parts of a Microarchitecture
- The DSI
- IPC vs. cycle time
- Partitioning

**Figure 7. Some fundamentals of computer architecture**

- Problems
- Algorithms
- Programs
- ISA
- Microarchitecture
- Circuits
- Devices

**Figure 8. Levels of Transformation**

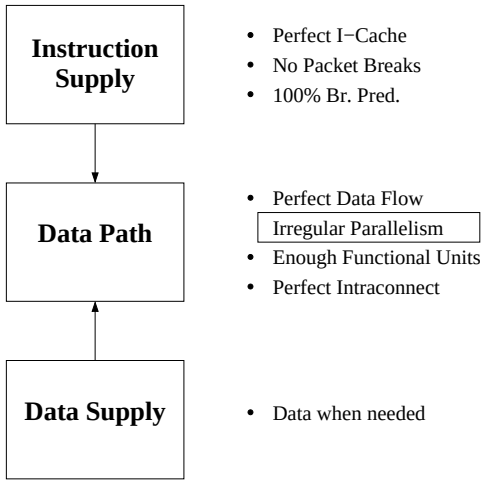


**Figure 9. The Microarchitecture**

- Focus on Measurements
- Use of Simulation
- Real ISA vs. Concocted ISA

**Figure 10. Some concerns**

- the new data path
- internal fault tolerance
- asynch and synch co-existing
- different cycle times for different functions
- SSMT (aka helper threads)
- Block-structured ISA
- uarch support for CAD
- greater use of microcode
- greater impact of the compiler
- compiler/uarch communication

**Figure 11. The Microprocessor ten years from now (perhaps)**

Finally, because a course in Computer Architecture is not only about what is, but also about preparing the student for what will be, it should also give our best current guess into the future (see Figure 11).

## References


[1] Y. N. Patt. The First Computing Course for CS, CE, and EE Majors at Michigan. In *The Interface*, pages 1–3, November 1998.

[2] Y. N. Patt and S. J. Patel. *Introduction to Computing Systems: From Bits and Gates to C and Beyond*. McGraw-Hill, 2001.

[3] B. D. Shriver and B. Smith. *The Anatomy of a High Performance Microprocessor: A Systems Perspective*. IEEE Computer Society Press, 1998.

# Intel® Itanium® Floating-Point Architecture

Marius Cornea, John Harrison, and Ping Tak Peter Tang

Intel Corporation

## Abstract

The Intel® Itanium® architecture is increasingly becoming one of the major processor architectures present in the market today. Launched in 2001, the Intel Itanium processor was followed in 2002 by the Itanium 2 processor, with increased integer and floating-point performance. Measured by the SPEC CINT2000 benchmarks, the Itanium 2 processor still trails by about 25% the Intel P4 processor in integer performance, albeit P4 runs at more than three times Itanium's clock frequency. However, its floating-point performance clearly leads in the SPEC CFP2000 charts, and its rating is about 25% higher than that of the P4 processor. While the general features of the Itanium architecture such as large register sets, predication, speculation, and support for explicit parallelism [1] have been presented in several papers, books, and mainstream college textbooks [2], its floating-point architecture has been less publicized. Two books, [3] and [4], cover well this area. The present paper focuses on the floating-point architecture of the Itanium processor family, and points out a few remarkable features suitable to be the focus of a lecture, lab session, or project in a computer architecture class.

## Introduction

The performance of today's processors continues to increase. But the physical limits for the manufacturing technology will eventually be reached, rendering Moore's Law inapplicable. Substantial further advances can be attained only by allowing a processor to operate on more bits at a time, and to execute more instructions in parallel. This was the motivation that led to the design of the Itanium processor family. Based on the EPIC (Explicitly Parallel Instruction Computing) design philosophy [5], the Itanium architecture was co-developed by Intel Corporation and Hewlett-Packard Company, combining the best in the RISC and VLIW architectures, while also adding several features originating from recent research studies in processor architecture. The result is a processor architecture that can handle a large amount of work based on its ability to feed instructions quickly to several execution units.

To date, two implementations of the Itanium architecture have been introduced by Intel Corporation. The Itanium processor provided hardware manufacturers and software writers with a first development vehicle. The second implementation, represented by the Itanium 2 processor, increased the performance level of the Itanium processor by a factor of 1.5 to 2 in several cases.

Itanium processors target the most demanding enterprise and high-performance computing applications, addressing the growing needs for data communications, storage, analysis and security, while also providing performance, scalability and reliability advantages at significantly lower costs than before.

Common desktop applications have no immediate need for the computing power or addressing capabilities of a 64-bit processor, but an increasing number of mid-range and high-end applications already do, or will soon, require such capabilities. These are mainly programs that demand a lot of memory space and/or perform a large amount of computation. Examples include applications accessing large databases or delivering Internet content, programs that use 64-bit long integers, and data-intensive applications solving scientific and engineering problems. Itanium processor features that benefit the latter category will be the focus of the present paper.

Scientific and engineering applications that can take advantage of the increased floating-point performance of Itanium processors include among others quantum chromodynamics (QCD), quantum mechanics, molecular simulation, cell research, or new drug discovery applications, computer-aided design tools, and solvers for large equation systems used in a variety of scientific and technical problems. Digital content creation applications that require high bandwidth, large memory, and powerful floating-point performance are also going to benefit from running on Itanium processors. Such applications can run very slowly on workstations based on 32-bit processors because of the smaller data item size, and also because of the continuous data traffic between storage disks and the memory system. Reduced swapping between memory and disk on Itanium-based systems are likely to increase performance of some applications by up to two orders of magnitude.

## Itanium Floating-Point Architecture

The Itanium floating-point architecture has been designed to combine high performance and good accuracy. A large floating-point register set of 128 registers is provided, and almost all operations can read their arguments from, and write their results to,

arbitrary registers. Together with register rotation for software-pipelined loops, this large number of registers allows the encoding of common algorithms without running short of registers or needing to move data between them in elaborate ways. Registers can store floating-point numbers in a variety of formats, and the rounding of results is determined by a flexible combination of several selectable defaults and additional instruction completers.

The basic arithmetic operation, the *floating-point multiply-add* (*fused multiply-add*), allows higher accuracy and performance in many common algorithms. Several additional features are also present to support common programming idioms. The fused multiply-add operation combines two basic floating-point operations in one, with only one rounding error. Besides the increased accuracy, this can effectively double the execution rate of certain floating-point calculations, as the fused multiply-add operation forms an efficient computation core that maps perfectly to several common algorithms used for technical and scientific purposes. The fused multiply-add operation creates the possibility of implementing new algorithms, such as software-based division and square root operations. As execution units are pipelined, a division or square root operation does not block the floating-point unit for the entire duration of the computation, and several other operations can be initiated or carried out in parallel.

The large number of floating-point registers available, of which some are static and some are rotating, allows for efficient implementation of complicated floating-point calculations. An illustration of software and hardware interaction in the Itanium architecture, this is achieved on one side by avoiding frequent accesses to memory, and on the other through software pipelining of loops containing floating-point computations. For example, the throughput for division operations can be as high as one result for every 3.5 clock cycles on the Itanium and Itanium 2 processors.

## Floating-Point Formats

The IEEE Standard 754-1985 for Binary Floating-Point Arithmetic [6] mandates precisely defined floating-point formats referred to as single precision and double precision. As well as these IEEE-mandated formats, Intel architectures have traditionally supported a double-extended precision type, with 64 bits of precision and a 15-bit exponent field. In current IA-32 implementations, results computed in the floating-point register stack may be rounded to 24, 53 or 64 bits of precision. Although the first two precisions coincide with the IEEE single and double precision, the precision control setting in IA-32 processors does not affect the exponent range, as the exponent uses a 15-bit field until the number is actually written back to memory. Although the greater exponent range is normally advantageous, it can lead to subtle variations in underflow and overflow behavior depending on exactly when a result is written to memory (which may be compiler-dependent and hard to predict).

In order to maintain the useful extra exponent range but allow the user complete control over rounding, the Itanium architecture allows for *both* conventional single and double precision formats *and* formats with the same precision but a 15-bit exponent field. In addition, a still wider exponent field of 17 bits is provided in each case, a very useful feature for intermediate calculations with double-extended precision numbers. This means that there are actually eight floating-point formats directly supported by the Itanium architecture, shown in Table 3-1.

Table 3.1. Floating-Point Formats Available in the Itanium Architecture

| Format | Precision | Exponent Bits | Exponent Range |
|---|---|---|---|
| Single | 24 | 8 | –126 to 127 |
| Double | 53 | 11 | –1022 to 1023 |
| Double extended | 64 | 15 | –16382 to 16383 |
| IA-32 stack single | 24 | 15 | –16382 to 16383 |
| IA-32 stack double | 53 | 15 | –16382 to 16383 |
| Register single | 24 | 17 | –65534 to 65535 |
| Register double | 53 | 17 | –65534 to 65535 |
| Register | 64 | 17 | –65534 to 65535 |

## Register and Memory Encodings

The Itanium architecture specifies 128 floating-point registers f0, f1, ..., f127. Register f0 is hardwired to +0.0 and f1 to +1.0, and both are read-only, but all other registers are available for reading and writing. Each register is 82 bits long, with a 64-bit significand (using an explicit integer bit), a 17-bit exponent field and a 1-bit sign. The exponent bias has the value 65535, or 0xFFFF (hexadecimal).

Certain values, such as NaNs, are neither negative nor positive. Special encodings, such as zeros, infinities, pseudo-zeros, pseudo-denormals, NaNs, pseudo-NaNs, pseudo-infinities, or NaTVal are all possible. Some of these special categories are explained below.

The minimum (biased) exponent value of 0 is reserved for double-extended real denormalized numbers (*denormals*), and for *pseudo-denormals*. The maximum (biased) exponent value of 131071, or 0x1FFFF, is reserved for special numbers such as infinities and NaNs.

Other exponent values, between 0 and 0x1FFFE in biased form, are used for finite numbers. The value in a floating-point register with sign *s*, biased exponent *e* and significand $m_0m_1m_{2...}m_{63}$ is determined by the following formula for biased 17-bit exponents between 1 and 0x1FFFE:

$$(-1)^s \cdot 2^{e-65535} \cdot m_0.m_1m_{2...}m_{63}$$

and the following for biased exponents that are zero:

$$(-1)^s \cdot 2^{-16382} \cdot m_0.m_1m_{2...}m_{63}$$

The register encoding is redundant: the same real value can sometimes be represented in several different ways. This is a consequence of the presence of an explicit integer bit, and is true of all floating-point formats that support it. For example, one can have positive *pseudo-zeros* with significand equal to zero but exponent from 0x000001 to 0x1FFFD rather than zero. Most of these alternative representations of the same number are equally acceptable as inputs to floating-point operations, the only exceptions being the *unsupported* numbers with exponent 0x1FFFF and integer bit 0 (pseudo-infinities and pseudo-NaNs). In particular, the user can freely operate on arguments of mixed format without any time-consuming format conversions. This is often useful, especially when:

- Using double-extended intermediate precision calculations to compute a double precision function. The double precision input argument can be freely combined with double-extended intermediate results.

- Computing functions involving constants with few significant digits. Whatever the precision of the computation, the short constants can be stored in single precision.

However, *results* of floating-point operations, and floating-point values loaded from memory, are always mapped to fixed canonical representatives in the register.

Note that the subsets of positive and negative register format numbers are almost symmetrical, with only two exceptions. First, NaTVal, the special *Not a Thing Value* quantity used to track floating-point computations that encounter failed speculative loads, has an encoding as an otherwise unused positive floating-point number: positive sign, biased exponent of 0x1FFFE and significand of 0 (a pseudo-zero). Second, encodings with a positive sign and a biased exponent of 0x1003E (corresponding to the unbiased decimal value of 64) are used also for *canonical integers*, and for *SIMD[1] floating-point numbers* (pairs of 32-bit single precision numbers). These are stored in the significand portion of a floating-point register.

The register encoding used differs from the encoding used when floating-point values are stored in memory. Single precision and double precision floating-point numbers are stored in the memory format specified by the IEEE Standard, with exponent biases of 127 (0x7F) and 1023 (0x3FF) respectively, and no explicit integer bit. Double-extended and register format numbers are stored in a more direct mapping of the register contents (the exponent bias for double-extended values is 0x3FFF).

For example, the value of a single precision floating-point number with sign *s*, biased exponent *e* and significand $m_0m_1m_{2...}m_{23}$ stored in memory is determined by the following formula for biased 8-bit exponents between 0x1 and 0xFE:

$$(-1)^s \cdot 2^{e-127} \cdot m_0.m_1m_{2...}m_{23}$$

and the following for biased exponents that are zero:

$$(-1)^s \cdot 2^{-126} \cdot 0.m_1m_{2...}m_{23}$$

For double precision values, the exponent bias to subtract from the exponent *e* is 1023, and denormals have an exponent of –1022. For double-extended precision values, the exponent bias is 16383, and denormals have an exponent of –16382.

## Status Fields and Exceptions

Given the number of floating-point formats available in the Itanium architecture, it is important to have a flexible means of specifying the desired floating-point format for a particular result to be rounded into, as well as the direction of rounding (e.g. rounding to nearest or truncation). Moreover, in accordance with the IEEE Standard, floating-point operations on the Itanium architecture not only produce results, but may optionally trigger exceptions or record exceptional conditions by setting sticky status flags. It would be impractical to encode all this information into the

[1]SIMD is an acronym for Single Instruction and Multiple Data, a form of parallel computing in which one operation is performed in parallel on multiple sets of operands.

format of each instruction, so some global status and control word is necessary for specifying options as well as recording exception flags. On the other hand, having only a single record would be inconvenient where there are several parallel threads of control, or where exceptions in some intermediate instructions need to be ignored. Therefore, the Itanium architecture features four different s*tatus fields* which can be specified by completers in most floating-point instructions. An instruction with a given status field completer is then controlled by, and records certain information in, that status field.

A 64-bit *Floating-Point Status Register* (FPSR) controls floating-point operations and records exceptions that occur. The FPSR contains 6 *trap disable* bits that control which floating-point exception conditions actually result in a trapped exception (where control passes to the OS and possibly to a user handler), and which are merely recorded in sticky status flags. These bits control the five IEEE Standard exceptions: invalid operation (vd), division by zero (zd), overflow (od), underflow (ud) and inexact result (id), as well as the additional *denormal/unnormal operand* exception (dd), which occurs if an input to a floating-point instruction is an unnormalized number. In addition to this field, the FPSR contains four 13-bit status fields, denoted in the assembly language syntax by s0, s1, s2 and s3.

Each status field can be divided into two parts: *flags* and *controls*. The six flags are bits that record the occurrence of each of the 6 exceptions mentioned above, when exceptions are masked, or, for the overflow, underflow or inexact result exceptions, also when they are *enabled* (*unmasked*). These flags are *sticky*, meaning that later operations that do not cause exceptions will not set flags back to 0, so the occurrence of an exception anywhere in a computation sequence will be apparent at the end of that sequence. Of the control part, one bit (td) allows all exceptions to be disabled irrespective of the individual trap disable bits from the FPSR (often useful in intermediate calculations). The remaining 6 bits control the rounding mode, precision and exponent width, and the flushing to zero of tiny[2] results.

[2]The IEEE Standard allows for two methods of determining whether a result is tiny. Intel architecture processors choose to define a result as being tiny if the exact value rounded to the destination precision *while assuming an unbounded exponent* is less than the smallest normal value that can be represented in the given floating-point format.

The pc and wre fields together determine the floating-point format into which the result will normally be rounded. The rounding control rc determines the IEEE rounding mode.

Although the status fields determine the default rounding behavior of operations, it is often possible to override them by explicit completers. This applies, for example, to many of the instructions to be discussed below. If an instruction has an explicit .s or .d completer, then the destination format is single or double precision respectively, except if the wre flag is set, in which case register single or register double is used.

Software conventions for the FPSR determine many of the appropriate applications for particular status fields. Typically, s0 is the main user status field used for most floating-point calculations. Status field s1, with wre enabled and all exceptions disabled, is used for intermediate calculations in many standard numerical software kernels such as those for division, square root, and transcendental functions. Status fields s2 and s3 are also commonly used for speculation. The default setting of the FPSR is such that all status fields use the 64-bit precision, the round-to-nearest rounding mode, and have floating-point exceptions and the flush-to-zero mode disabled. Only status field s1 uses the widest-range exponent.

## The Floating-Point Multiply-Add

In most existing computer architectures, there are separate instructions for floating-point multiplication and floating-point addition. In the Itanium architecture, these are subsumed by a more general instruction, the *floating-point multiply-add* or *fused multiply-accumulate*, which takes three arguments, multiplies two of them and adds in the third. The basic assembly syntax is:

(*qp*) fma.*pc*.*sf* $f_1 = f_3, f_4, f_2$

which sets $f_1 = f_3 \cdot f_4 + f_2$. Note that no intermediate rounding is performed on the result of the multiplication, and the result is written to $f_1$ as if it were first computed exactly and then rounded, in a natural extension of the way conventional arithmetic operations are specified to behave in the IEEE Standard. The rounding of the result and the triggering of exceptions is controlled by the status field specified by the *sf* completer and possibly by the FPSR trap disable bits, except that the rounding precision from *sf* may be overridden by an optional precision control completer *pc*.

Since the floating-point registers f0 and f1 are hardwired to the values +0.0 and +1.0 respectively, addition and multiplication can easily be implemented

as special cases of the fma: $x + y = x \cdot 1 + y$ and $x \cdot y = x \cdot y + 0$. In fact, the floating-point addition and multiplication assembly instructions

(*qp*) fadd.*pc*.*sf* $f_1 = f_3, f_2$
(*qp*) fmpy.*pc*.*sf* $f_1 = f_3, f_4$

are simply pseudo-operations that expand into

(*qp*) fma.*pc*.*sf* $f_1 = f_3$, f1, $f_2$
(*qp*) fma.*pc*.*sf* $f_1 = f_3, f_4$, f0

respectively. In order to change signs, there are two variants of the fma: the fms (floating-point multiply-subtract) and fnma (floating-point negative multiply-add). The instructions

(*qp*) fms.*pc*.*sf* $f_1 = f_3, f_4, f_2$
(*qp*) fnma.*pc*.*sf* $f_1 = f_3, f_4, f_2$

compute $f_1 = f_3 \cdot f_4 - f_2$ and $f_1 = -f_3 \cdot f_4 + f_2$ respectively. Floating-point subtraction

(*qp*) fsub.*pc*.*sf* $f_1 = f_3, f_2$

is likewise a pseudo-operation for

(*qp*) fms.*pc*.*sf* $f_1 = f_3$, f1, $f_2$

An even more degenerate instance of fma, called fnorm (floating-point normalize) can be used to round a result into a given floating-point format. This can be used as a 'lowering' operation to convert a value to a smaller floating-point format, but the most common use is just to ensure that the number is normalized. (This is often useful, because processing unnormalized values is slower in most cases than performing an fnorm followed by the intended operation.) This rounding to a canonical value is accomplished by the standard fma behavior, and so fnorm.*pc*.*sf* $f_1 = f_3$ is simply a pseudo-operation for fma.*pc*.*sf* $f_1 = f_3$,f1,f0.

It was stated above that the fma behaves in accordance with the IEEE Standard. Strictly speaking, that standard does not cover the fma instruction, but all the stipulations are extended to it in a natural way. However, there is some subtlety over the signs of zero results.

If the result of an fma without the final rounding would be nonzero, then should it round to zero, the sign of the final zero will reflect the sign of the exact result. This of course is the 'correct' decision, but is a non-trivial extrapolation of the IEEE Standard. Here, the sign rules for multiplications and divisions are obvious (the exclusive or of the input signs). And for addition and subtraction, when the rounded result is nonzero, the exact result must be too (in a fixed floating-point format), so only the special case of exactly zero results needs to be dealt with.

Now consider the case when the result of an fma instruction without rounding is exactly zero. Normally, the sign of $x \cdot y + z$ is determined by multiplying the signs of $x$ and $y$ to give a sign for the intermediate result, then using the rules of the IEEE Standard, treating $w + z$ as if it were an ordinary sum. However, this is not appropriate for considering the ordinary product a special case of the fma. For example, $(+1.0) \cdot (-0.0) + (+0.0)$ would give +0.0, whereas the IEEE-specified product is (–0.0). This difficulty is circumvented as follows: if the third argument to the fma is actually register zero (f0), then the sign of zero is determined by the IEEE rules for products. Otherwise, the sign of zero results is decided as specified above for fma, even if the third argument to fma is not the special register zero f0 but nevertheless contains the value zero. This applies equally to the variants fms and fnma.

A floating-point multiply-add is a very valuable architectural feature, for reasons of both speed and accuracy. In typical implementations, the final addition can be combined into the floating-point multiplication operation without significantly increasing its latency. Thus, a single fma is faster than a multiplication and an addition executed successively. Since additions and multiplications are heavily interleaved in many important floating-point kernels (the evaluation of inner, or dot, products of vectors or the evaluation of polynomials for example), the use of an fma can lead to significant performance improvements. For example the vector dot product $x \cdot y$:

$$p = \sum_{i=0}^{N-1} x_i \cdot y_i$$

can be evaluated by a succession of fma operations of the form

$$p = p + x_i \cdot y_i$$

requiring only $n$ floating-point operations, whereas with a separate multiplication and addition it would require $2n$ operations, with a longer overall latency.

Apart from its speed advantage, the fact that no intermediate rounding is performed on the product also tends to reduce overall rounding errors. In common cases this difference may be relatively unimportant, but in special situations, the lack of an intermediate rounding makes possible a number of techniques that are difficult or costly on a traditional architecture. The floating-point division and square root implementations provide ample illustration of this fact, but here are three other characteristic examples.

## Exact Arithmetic

In certain applications it is important to perform arithmetic to very high precisions, perhaps hundreds of bits. A natural way of manipulating very precise

numbers is as *floating-point expansions*; that is, sums of standard floating-point numbers of decreasing magnitude. In order to perform efficient computations on such expansions, the basic building blocks are operations that compute exact arithmetic operations on individual pairs of floating-point numbers. For example, it is known (Moller [7], and Dekker [8]) that if $|x| \geq |y|$ the exact sum $x + y$ can be obtained as a 2-piece expansion Hi + Lo by the following sequence of floating-point adds:

$$Hi = x + y$$
$$tmp = x - Hi$$
$$Lo = tmp + y$$

This is straightforward to implement on traditional architectures, though features of the Itanium architecture make it significantly more efficient. However, on traditional architectures there is no similarly easy way of obtaining the exact product of floating-point numbers as an expansion; this requires fairly complicated and inefficient methods based on splitting the numbers into high and low parts by masking and performing numerous sub-computations. However, with the fms instruction, this computation is simple and efficient:

$$Hi = x \cdot y$$
$$Lo = x \cdot y - Hi$$

This sequence always results in $Hi + Lo = x \cdot y$ exactly with $Lo$ a rounding error in $Hi \approx x \cdot y$.

## Accurate Remainders

It is often the case that given a floating-point number $q$ approximately equal to the quotient $a / b$ of two floating-point numbers, one wants to know the remainder $r = a - b \cdot q$. This arises whenever evaluation of a quotient to higher precision is needed, for example, in floating-point expansions. Provided the approximation $q$ is good enough, it can be shown that $r$ is always representable exactly as a floating-point number. However, that does not mean it is always straightforward to obtain it on traditional architectures. In fact, if $a - b \cdot q$ is computed by a multiplication and a subsequent subtraction, the rounding error in the multiplication may be comparable in size to $r$ itself, rendering the result essentially meaningless. Thus, complicated masking and multiple computations are necessary. But in the Itanium architecture, evaluating $a - b \cdot q$ by an fnma instruction will give an exact answer provided $q$ is accurate enough.[3]

## Accurate Range Reduction

A similar situation arises when one has an integer approximation to the exact quotient. Many algorithms for mathematical functions, in particular trigonometric functions such as *sin*, begin with an initial range reduction phase, subtracting an integer multiple of a constant such as $\pi / 2$. With the fma this can be done in a single instruction $x - N \cdot P$ yielding an accurate result. Without the fma however, the rounding error in the multiplication could severely distort the result, so it might be necessary to represent $P$ as the sum of two numbers with fewer significant bits. (Each of these numbers can be multiplied by $N$ without error, and after several operations the main result can be obtained.) The fma is also useful for obtaining the appropriate $N$ rapidly in the first place. Typically, one wants to perform some operation such as

$$y = Q \cdot x$$
$$N = rint\ (y)$$
$$r = x - N \cdot P$$

where *rint* $(y)$ denotes the rounding of $y$ to an integer, and $Q \approx 1 / P$. Rather than using the special fcvt instructions to convert $y$ to an integer, the integer conversion can be performed by adding and subtracting a large constant like $S = 2^{p-1} + 2^{p-2}$ where $p$ is the floating-point precision, for example $p = 53$ for double precision. (Adding such a constant fixes the most significant bit of the sum and hence performs integer rounding of $y$, provided $|y| \leq 2^{p-2}$; the use of $2^{p-2}$ makes the technique work for both positive and negative $y$.) Using the fma the multiplication by $Q$ and the addition of $S$ can be combined, and hence the reduced argument can be obtained by just three fma operations:

$$y = S + Q \cdot x$$
$$N = y - S$$
$$r = x - N \cdot P$$

This approach has the additional advantage of avoiding some rare problems with the intermediate rounding of the product $Q \cdot x$.

## Comparison and Classification

Floating-point comparisons are similar to the integer comparisons. The basic instruction is

[3] It suffices for $q$ to be accurate to one *unit in the last place* (ulp).

(*qp*) fcmp.*fcrel.fctype* $p_1, p_2 = f_2, f_3$

Here the *fcrel* completer, which is compulsory, determines the relation that is tested for. The mnemonics differ slightly from those used in the integer comparison: eq for $f_2 = f_3$, lt for $f_2 < f_3$, le for $f_2 \le f_3$, gt for $f_2 > f_3$, ge for $f_2 \ge f_3$,and unord for $f_2 ? f_3$. There is no signed/unsigned distinction but there is a new possibility, shown in the last case ($f_2 ? f_3$): two values may be *unordered*, since a NaN (Not a Number) compares false with any floating-point value, even with itself. Mnemonics are also provided for the complements of all these conditions, although in the actual instruction encoding these simply swap the predicate registers and/or the input floating-point registers.

The *fctype* field has two possible values, *none* (i.e. the field is omitted in the assembly syntax), and unc. If omitted, the result of the comparison and its complement are written to the designated predicate registers in the usual way. If the completer unc is used, however, then the behavior is the same if the qualifying predicate *qp* of the instruction is true, but *both* the predicate registers $p_1$ and $p_2$ are cleared if *qp* is false.

It is often desirable to classify a floating-point number, for example to abort a calculation if an input is infinite or NaN. A comprehensive instruction for classifying the floating-point value in a register is fclass:

(*qp*) fclass.*fcrel.fctype* $p_1, p_2 = f_2,$ *fclass*

The result of classifying the contents of $f_2$ is written to the predicate registers $p_1$ and $p_2$, controlled by the optional *fctype* in the same way as for comparisons (i.e. its values can be *none* or unc). The *fcrel* field may be m ($f_2$ is a member of the class specified by *fclass*) or nm ($f_2$ is not a member of the class specified by *fclass*). The actual classification is encoded as a 9-bit field whose bits are interpreted to determine whether the floating-point value is: positive or negative; zero, unnormalized, normalized or infinity; NaN or NaTVal.

## Division and Square Root

There are no instructions specified in the Itanium architecture (except in IA-32 compatibility mode) for performing floating-point division or square root operations. Instead, the only instruction specifically intended to support division is the *floating-point reciprocal approximation* instruction, frcpa, which given floating-point numbers *a* and *b*, normally returns an approximation to 1 / *b* good to about 8 bits. The syntax of this instruction is as follows:

(*qp*) frcpa.*sf* $f_1, p_2 = f_2, f_3$

Similarly, the only instruction to support square root is the *floating-point reciprocal square root approximation* instruction frsqrta, which given a floating-point number *a*, normally returns an approximation to 1 / √a good to about 8 bits.

(*qp*) frsqrta.*sf* $f_1, p_2 = f_3$

In special cases such as $b = 0$ for frcpa or $a = 0$ for frsqrta, these instructions actually return the full IEEE-correct result for the relevant operation (the full quotient in the case of frcpa), and indicate this by clearing the output predicate register $p_2$. Usually, however, the initial approximations need to be refined to perfectly rounded quotients or square roots by software, and this is indicated by setting the predicate register $p_2$. Consequently, one can simply predicate the software responsible for refining the initial approximation by this predicate register. Thanks to the presence of the fma instruction, quite short straight-line sequences of code suffice to do this correction. There are several reasons for relegating division and square root to software.

- By implementing division and square root in software, they immediately inherit the high degree of pipelining in the basic fma operations. Even though these operations take several clock cycles, new ones can be started while others are in progress. Hence, many division or square root operations can proceed in parallel, leading to much higher throughput than is the case with typical hardware implementations.

- Greater flexibility is afforded because alternative algorithms can be substituted where it is advantageous. It is often the case that in a particular context a faster algorithm suffices, for example because the ambient IEEE rounding mode is known at compile time, or even because only a moderately accurate result is required (this might arise in some graphics applications).

- In typical applications, division is not an extremely frequent operation, and so it may be that the die area on the chip would be better devoted to something else.

Intel Corporation distributes a number of recommended algorithms for various precisions and performance constraints, so the user will not ordinarily have to be concerned with the details of how to implement these operations. As an example, consider the single precision division algorithm, optimized for throughput (it has the smallest possible number of floating-point instructions, resulting in the minimum latency per result in software-pipelined loops): The algorithm calculates $q = a/b$ in single precision, where *a* and *b* are single precision numbers, *rn* is the IEEE round to nearest mode, and *rnd* is any IEEE rounding mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

(1) $y_0 = 1 / b \cdot (1+\varepsilon_0)$, $|\varepsilon_0| < 2^{-8.886}$ table lookup
(2) $d = (1 - b \cdot y_0)_{rn}$ 82-bit register format precision
(3) $e = (d + d \cdot d)_{rn}$ 82-bit register format precision
(4) $y_1 = (y_0 + e \cdot y_0)_{rn}$ 82-bit register format precision
(5) $q_1 = (a \cdot y_1)_{rn}$ 17-bit exponent, 24-bit mantissa
(6) $r = (a - b \cdot q_1)_{rn}$ 82-bit register format precision
(7) $q = (q_1 + r \cdot y_1)_{rnd}$ single precision (IEEE)

The assembly language implementation follows [9], assuming the input values are in floating-point registers f6 and f7, and the result in f8:

```
      frcpa.s0 f8,p6=f6,f7;; // Step (1)  y0=1/b in f8
(p6) fnma.s1 f9=f7,f8,f1;;  // Step (2) d = 1-b*y0 in f9
(p6) fma.s1 f9=f9,f9,f9;;   // Step (3) e = d+d*d in f9
(p6) fma.s1 f8=f9,f8,f8;;  // Step (4) y1 = y0+e*y0 in f8
(p6) fma.s.s1 f9=f6,f8,f0;;  // Step (5) q1 = a*y1 in f9
(p6) fnma.s1 f6=f7,f9,f6;;  // Step (6) r = a-b*q1 in f6
(p6) fma.s.s0 f8=f6,f8,f9;; // Step (7) q = q1+r*y1 in f8
```

Support for software pipelining on Itanium processors allows for this algorithm to be scheduled without any additional code, so that one result is generated every 3.5 clock cycles (since there are 7 floating-point instructions to schedule on 2 floating-point units on Itanium and Itanium 2 processors). This is a lot more efficient than on most present-day processor architectures.

Table 3.2 shows the Itanium 2 processor cycle times for the division root algorithms of various precisions (a similar table is available for square root [9]). For algorithms optimized for latency, the operation latency is given, in number of clock cycles. For operations optimized for throughput, the number of clock cycles required to generate one result is given.

Table 3.2. Latency and Throughput for Floating-Point Division on the Itanium 2 Processor

| Division | Single Precision | Double Precision | Double-Extended Precision |
|---|---|---|---|
| Optimized Latency | 24 | 28 | 32 |
| Optimized Throughput | 3.5 | 5 | 7 |

The square root algorithms rely on loading constants, and the time taken to load these constants is not included in the overall latencies. If the function is inlined by an optimizing compiler, these loads should be issued early as part of normal operation reordering. For comparison, note that on the Itanium 2 processor, a floating-point add/subtract, multiply, or fused multiply-add operation has a latency of 4 clock cycles, and a throughput of 0.5 clock cycles (meaning that two results can be generated every clock cycle, for example in a software-pipelined loop).

### Additional Features

The Itanium architecture includes a number of other useful floating-point instructions that have not been mentioned, which are covered in detail in [4]. They include:

- transferring values between floating-point and integer registers by means of the getf and setf instructions
- floating-point merging, useful in order to combine fields of multiple floating-point numbers to give a new number using the fmerge instruction
- floating-point to integer and integer to floating-point conversion using the fcvt instructions
- integer multiplication and division - the Itanium architecture does not specify a full-length integer multiplication or division instruction; instead, such operations are intended to be implemented using the floating-point unit, by first transferring the arguments to floating-point registers, performing the multiplication or division there, and transferring the result back
- floating-point maximum and minimum, using the fmax and fmin instructions

## Conclusion

The Itanium floating-point architecture was designed so that its high performance, accuracy, and flexibility characteristics make it ideal for technical computing. Floating-point enhancements include a high precision and wide range basic floating-point data type, the fused floating-point multiply-add operation, software division and square root operations, and a large number of floating-point registers. Floating-point code can also draw on other generic Itanium architecture features such as predication, register rotation, high memory bandwidth, and speculation.

All floating-point data types are mapped internally to an 82-bit format, with 64 bits of accuracy and a 17-bit exponent. This affords calculations that are more accurate, and do not underflow or overflow as often as on other processors. The great flexibility in using and combining various floating-point formats and computation models makes it easy to implement complex numerical algorithms more efficiently than before.

The fused multiply-add operation combines two basic floating-point operations in one, with only one rounding error. Besides the increased accuracy, this can effectively double the execution rate of certain floating-point calculations, as the fused multiply-add operation forms an efficient computation core that maps perfectly to several common algorithms used for technical and scientific purposes.

The large number of floating-point registers available, of which some are static and some are rotating, allows for efficient implementation of complicated floating-point calculations. An illustration of software and hardware interaction in the Itanium architecture, this is achieved on one side by avoiding frequent accesses to memory, and on the other through software pipelining of loops containing floating-point computations.

The highest SPEC CFP2000 score for a single processor system, of 1431, belongs currently to an Itanium 2 system running at 1GHz - the Hewlett-Packard HP Server RX2600. The best performing P4 system, running at 3.06 GHz, has a score of 1092. The SPEC CINT2000 scores are in reverse order though – 810 and 1099 respectively. This gap will likely decrease and the advantage is expected to be on the Itanium processor family side as its core frequencies will get higher - today's Itanium processors run at relatively low frequencies, and as the compiler technology on which Itanium processors depend so much continues to evolve.

## References


[1] Intel(R) Itanium(TM) Architecture Software Developer's Manual, Revision 2.0, Vol 1-4, Intel Corporation, December 2001

[2] John Hennessy, David Patterson, "Computer Architecture - A Quantitative Approach", Morgan Kauffman Publishers, Inc., third edition, 2002

[3] Peter Markstein, ''IA-64 and Elementary Functions: Speed and Precision", Hewlett-Packard/Prentice-Hall 2000

[4] Marius Cornea, John Harrison, Ping Tak Peter Tang, "Scientific Computing on Itanium-based Systems", Intel Press 2002

[5] John Crawford, Jerry Huck, "Motivations and Design Approach for the IA-64 64-Bit Instruction Set Architecture", Oct. 1997, San Jose, http://www.intel.com/pressroom/archive/speeches/mpf1097c.htm

[6] ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, IEEE, New York, 1985

[7] O. Moller, "Quasi double-precision in floating-point addition", BIT journal, Vol. 5, 1965, pages 37-50

[8] T. J. Dekker, "A Floating-Point Technique for Extending the Available Precision", Numerical Mathematics journal, Vol. 18, 1971, pages 224-242

[9] "Divide, Square Root, and Remainder Algorithms for the Itanium Architecture", Intel Corporation, Nov. 2000, http://www.intel.com/software/products/opensource/libraries/numnote2.htm, http://developer.intel.com/software/products/opensource/libraries/numdown2.htm

# DOP – A CPU CORE FOR TEACHING BASICS OF COMPUTER ARCHITECTURE

**Milos Becvar, Alois Pluhacek and Jiri Danecek**

*Department of Computer Science and Engineering*
*Faculty of Electrical Engineering*
*Czech Technical University in Prague,*

Abstract: A simple 16-bit processor core called DOP and its teaching environment is presented. The DOP processor illustrates the basic principles of computer organization and is therefore used in the introductory hardware course. Its major features are simplicity, availability of an FPGA implementation and a C compiler. This paper presents the description of the core, HW and SW tools and teaching methodology.

## 1. INTRODUCTION

An introductory computer hardware course should teach students to the fundamental principles of computer internal functionality. Students, who are familiar with programming in high-level languages, are required to understand the interaction between a processor, a memory and I/O devices, an internal organization of processor, computer arithmetic and basics of digital design. Our experience has shown that it is not an easy task for most of them. The functionality of the processor executing instructions seems to be something almost mythical and totally unrelated to intuitive execution of a program in high-level language.

It is obvious that we have to provide some practical experience helping to understand these abstract principles. One common way is to let students to create a program in high-level language, which simulates a simple processor. However, this approach is not good for illustrating the relation between high-level languages, compilers, program in assembly language and actual "binary" program executed by processor.
Another approach uses visualization simulators and HW emulators (Bruschi, 1999; Yurcik *et al*, 2001; Brorson, 2002; Ellard *et. al,* 2002).

We present a simple 16-bit processor core called DOP that is currently used in our introductory computer organization course. The processor is simple, yet fully operational, and could be used in the embedded applications, which do not require an excessive computing performance. The DOP processor core was developed at our department together with various SW and HW visualization tools (Danecek *et al.,* 1994a).

The goal of this paper is to describe this processor core and its learning environment for teaching basics of computer organization. The paper is organized as follows - section 2 outlines the introductory course and characterizes the students, section 3 describes the DOP processor core, section 4 describes the SW and HW tools supporting this processor and finally section 5 outlines the use of the DOP in our introductory course.

## 2. COURSE REQUIREMENTS

The introductory computer organization is a one-semester course, which is mandatory for all undergraduate students of 3rd year of computer science and engineering. Almost 200 students take this course every year. The course covers the basic principles of computer functionality, the data representation, computer arithmetic and the controller design. Furthermore it introduces basics of practical digital design.

Students have relatively strong background in high-level languages and assembly language of x86 and are mostly SW-oriented. The majority of students thoughts that the only computer is an Intel x86 PC. The minority of students has experience in implementing simple digital circuits and wants to choose HW specialization in graduate study.

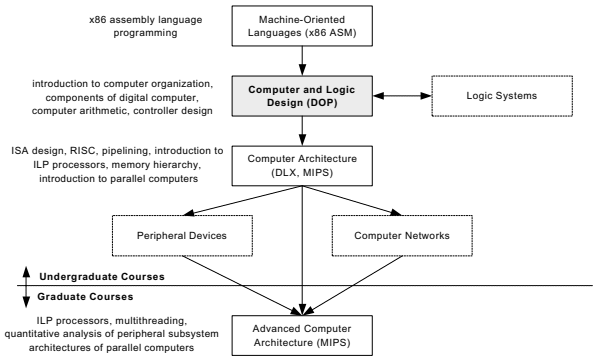


Fig.1. Computer architecture course flow at CTU

Figure 1 shows the place of "Computer and Logic Design" course in the overall Computer Architecture flow at Czech Technical University. It also outlines the main topics covered at each level and a reference platform.

The main goal of "Computer and Logic Design" course is to explain the components of digital computer and functionality of sequential processor built from a simple datapath and controller. The reference processor should illustrate these principles. The processor internal organization should be reasonably simple to be understood without deep knowledge in the digital design (note that "Logic Systems" course is unfortunately scheduled in parallel).

The next undergraduate course traditionally called "Computer Architecture" revisits the processor architecture and introduces the concept of pipelining and basics of techniques used in modern ILP processors as well as other concepts found in modern computers. Some topics are covered in separated courses "Peripheral Devices" and "Computer Networks". Undergraduate courses together cover all topics in the popular reference book "Computer Organization and Design – HW/SW Interface" (Patterson and Hennessy, 1998). Some topics from the area of peripheral subsystem and computer networks are covered in more detail than in this book. The main graduate course "Advanced Computer Architectures" is obligatory only for students of HW specialization. This course is based on the book (Patterson and Hennessy, 2002).

With this course flow in mind, we can describe the organization of the DOP, which is used to illustrate the basic principles of computer organization and design. The relation between the DOP and more advanced courses is discussed in the section 5.4.

## 3. DOP ARCHITECTURE OVERVIEW

The abbreviation "DOP" means "Danecek's Original Processor" according to one of its proponents. The DOP processor has not been primarily designed to be an educational platform. Its ISA was designed as a result of experience with writing HLL compilers for 8-bit and 16-bit microcontrollers like 8051, 68HC11, PIC16C5x, SAB80C166. These simple processors were designed for assembly language programming and writing efficient compilers for them is difficult and sometimes even impossible (Danecek *et al*., 1994b).
The DOP was intended to be a simple 16-bit processor core suitable for embedded systems and implementation in FPGA (Danecek *et al,* 1994a). The main feature of the processor is the simple compilation of high-level languages. From the nature of applications comes the requirement to optimize the program size over the speed of execution.

It is not surprising that the result of this development is an accumulator-oriented processor with variable length instruction encoding. Its main characteristics are outlined in Table 1. The processor contains only few programmer-visible registers. Local variables, temporaries and parameters are allocated on the stack in the main memory. This arrangement is valuable for illustrating relation between HLL programs and actual "binary" program executed by the processor.

The DOP processor is connected to the byte-organized main memory and peripheral subsystem by 16-bit Address Bus and 8-bit Data Bus. (Multibyte data are stored using Little Endian format) The main memory is common for the data and instructions.

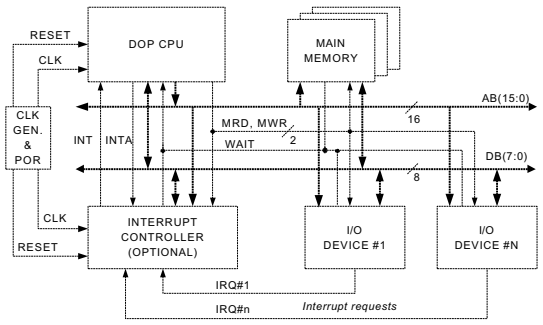


Fig.2. DOP system level overview

Peripheral devices for DOP could be memory mapped and processor supports single external maskable interrupt signal. The interrupt subsystem could be further expanded by an external interrupt controller. Interrupt subsystem was added to illustrate the interrupt service cycle at the HW level. Figure 2 shows the

interconnection between DOP processor, memory and peripherals. Some parts of this system exists as an FPGA implementation, others are only modeled in SW or exist only in the specification (peripheral devices). For further discussion of HW and SW tools please refer to section 4.

Table 1 DOP Characteristics

| | |
|---|---|
| DOP ALU width | 16 bit |
| Internal bus width | 16 bit |
| Address bus width | 16 bit |
| Data bus width | 8 bit |
| Encoding of signed numbers | 2's complement |
| Data types supported by ISA | Word (16 bit), unsigned byte (8 bit), signed short (8 bit) |
| Multibyte data storage format | Little Endian |
| I/O subsystem | Memory Mapped |
| Programmer visible 16-bit registers | PC, SP, W, S, D |
| Programmer visible 8-bit registers | L (loop counter), F (Flags) |
| External interrupts | 1 (maskable),<br>16 interrupts with external interrupt controller |

### *3.1 DOP Instruction Set Architecture*

The DOP ISA is an example of an accumulator-oriented instruction set with several enhancements.
First operand of ALU instruction is always an accumulator – register W. Second operand can be register or memory location (typically on the stack where local variables and temporaries are located).
The instruction set also includes a dedicated instruction LLA, which computes the address of local variable on the stack. Figure 4 shows the example of computation with local variables.

DOP instruction set supports three data types – 16-bit word, 8-bit unsigned byte and 8-bit signed short integer. Bytes and short integers are internally extended to word length and all operations are performed with these 16-bit operands. This is another solution than in x86 ISA, which provides separated instructions for 8-bit operands. Moreover, all data manipulation instructions support all three data types. This regularity simplifies the task of code generation and is also a good educational example.

DOP ISA also includes several provisions for efficient support of operands longer than 16-bit word and addition and subtraction of operands of different sizes (see Fig.5). Firstly, lower words of the two operands are added or subtracted (if the shorter operand is an 8-bit short integer, it is sign-extended). In the same time, the sign of the second ALU operand is stored in the special Auxiliary Flag (AF) (see also AUXF signal on the fig. 6). The addition or subtraction is finished by applying instruction AAF to the remaining words of the longer operand (instruction adds extended sign XAF of shorter operand stored in the AF and carry flag CF from the previous operation.)

There are also two special instruction prefixes modifying the behavior of the following ALU instruction. First prefix is an UCF (use carry flag) this prefix enforces the use of the CF in the following ALU instruction. For example the prefix UCF followed by an ADD instruction is equivalent to the ADDC instruction (add with carry) whereas the UCF followed by the SUB behaves like the SUBB (subtract with borrow). Second prefix is the SWW (suppress write to W) that allows synthesizing the comparison and test instruction. The SWW followed by the SUB behaves like the CMP (only flags are set, W is not modified) and the SWW followed by an AND is similar to the TEST instruction.

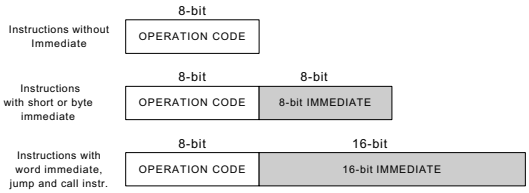


Fig.3. DOP instruction formats

The DOP is oriented to the high instruction encoding density and uses three formats of instruction. Most of instructions occupy only a single byte (1st format); other formats are used for instructions with 8-bit or 16-bit immediate. The DOP encoding density is superior over comparable processors. It has been reported that programs compiled for DOP occupy less than 60 % of memory space than for 8051. This feature can be very valuable for embedded systems (Danecek *et. al*, 1994c)

```
LLA S, 0x04     ;  S <= SP – 0x04 (address of a)
LLA W, 0x07     ;  W<=SP – 0x07 (address of b)
LLA D, 0x03     ;  W<=SP – 0x03 (address of c)
LD W, [W]       ;  W<=Mem[W]
ADD [S+]        ; W<= W + Mem[S], S++
ST [D]          ; Mem[D]<=W
```

Fig. 4 Example of c:=a+b in DOP assembly language (*a,b and c* are local variables allocated on stack)

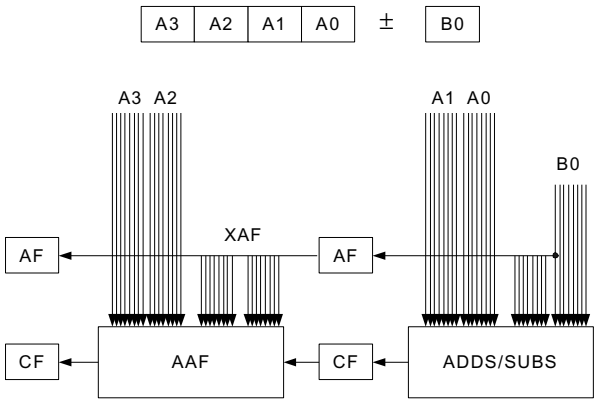


Fig.5 Example of addition/subtraction of short integer (8-bit signed) to doubleword (32-bit signed)

*3.2 DOP Arithmetic and Logic Unit*

The DOP Arithmetic and Logic Unit is based on the 16-bit W register which serves as an accumulator. This organization is very simple yet efficient for this class of processor. The second operand for ALU could be an internal register or an operand read from memory (to temporary register). The second operand is connected by the internal 16-bit bus to the second ALU input.

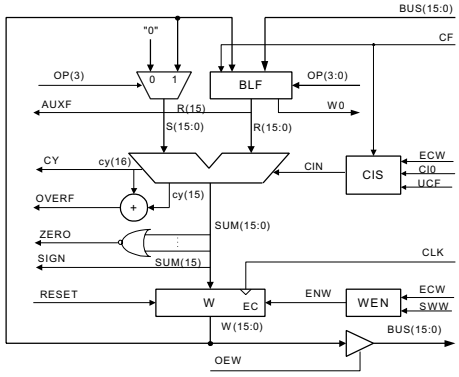


Fig.6. DOP ALU organization

The ALU internally contains the Block of Logic Functions (BLF) and 16-bit binary adder. This organization implements all basic binary arithmetic and logical operations (addition, subtraction, logical and, or, xor, negation and shifts). The logical operations are implemented in BLF, while the second input of adder is connected to zero. More complex operations such as multiplication or division could be synthesized by SW routine (library of these routines is available).
The blocks labeled CIS (carry in selection) and WEN (write enable) implement the prefixing instructions UCF and SWW.

The ALU also contains Flags – Carry Flag, 2's complement Overflow Flag, Zero Flag, Sign Flag and Auxiliary Flag used for addition or subtraction of operands longer than 16-bit word. All Flags are set after each ALU operation.

*3.3 DOP Registers*

Beside mandatory 16-bit PC (program counter) register the DOP contains only a few programmer-visible registers - SP (stack pointer), W (accumulator), which is a part of the ALU, S (source register) and D (destination register). Both S and D register could be used as general-purpose data storage during expression evaluation or address computation. All registers could be connected via 16-bit internal bus to ALU or to 16-bit external address bus and serve as an address for the main memory. The SP, S and D register support autoincrement and autodecrement addressing modes for accessing arrays and longer operands. Therefore these registers are implemented as bi-directional counters.

The DOP also includes the 8-bit L register, which is used as loop counter for an easy compilation of short *for* loops. The L register could be accessed separately or together with the FLAGS register as a 16-bit PSW (program status word).

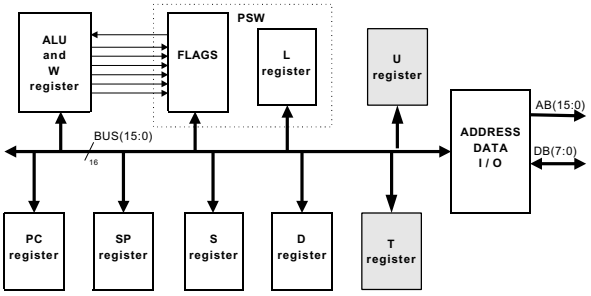


Fig. 7. DOP datapath

The whole datapath is outlined in figure 7. Note that U and T registers are not visible to the programmer and could be used as a temporary storage for the complex instructions. These registers were added only for educational purposes (original DOP ISA could be fully implemented without these registers).

*3.4 DOP Controller*

The DOP processor was originally implemented with optimized hardwired controller. However, this controller is not very suitable for practical experiments of our students. Consequently, a very simple horizontal microprogrammed controller was designed. The DOP controller currently needs the 58-bit wide microinstruction for all control signals (actual width is 64-bit including several spare control signals). Each

microinstruction corresponds to a single clock cycle. An Address of the next microinstruction is specified as a field in the current microinstruction. Three the lowest significant bits of this address could be modified by the condition multiplexer. This organization allows branching to the 8 destination addresses in a single microinstruction depending on the status of various internal and external signals. The DOP controller uses the condition multiplexer for gradual instruction decoding (decoding takes 2-3 clock cycles per instruction). The condition multiplexer is also employed for the testing of flags and external signal WAIT (from main memory or I/O) and INT (from interrupt controller).

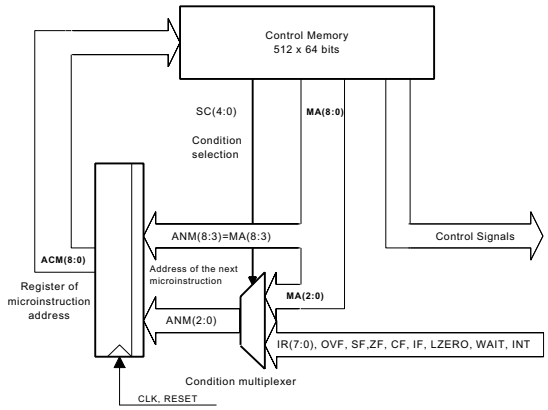


Fig.8. DOP microprogrammed controller

The complete implementation of the DOP instruction set needs less than 270 microinstructions in the control memory. This controller is definitely not the fastest possible microprogrammed one for this processor. However, its major advantage is the simplicity and regularity.

## 4. DOP HW AND SW TOOLS

### *4.1 DOP simulators and models*

Four SW models of the DOP currently available are characterized in Table 2. Each of them allows the analyzing of the processor behavior on the different level of abstraction for various purposes. The first two simulators are executable on almost any personal computer, whereas the VHDL models require complex and relatively expensive VHDL simulator. Therefore only a limited number of students could use the VHDL models simultaneously. Recently, the major FPGA vendors offers relatively cheap versions of VHDL simulators but it is still unlikely that every student can buy one and run it at home on his computer.

Table 2 DOP SW simulators and models

| Type of simulator | Modeling Language | Purpose |
|---|---|---|
| Instruction Cycle Accurate | C++ | Assembly program debugging, compiler debugging |
| Functional Clock Cycle Accurate | Pascal | Microprogram development, observation of the int. function of the processor |
| VHDL RTL model | VHDL | Design verification, detailed view of signal flow |
| VHDL post P&R model | VHDL + SDF | Timing verification, detailed view of real delays on FPGA |

The instruction-cycle accurate simulator can be used for debugging of DOP assembly program and the compiler development. The functional clock-cycle accurate simulator is used for development of DOP microprogram (firmware) for each instruction. The third type of model is synthesizable RTL VHDL model including memory and an interrupt controller. This model could be used to confirm results of functional clock-cycle accurate simulation. It allows more detailed view of the CPU behavior in time (namely signal sequences). The most accurate model is the post-place and route generated VHDL Vital model with SDF file. It shows the real delays of signals on FPGA.

### *4.2 HW emulator board*

A FPGA based emulator board was designed for the DOP processor. The board contains the control memory 8 k x 64 bits made from 8 SRAM chips. The size of control memory is significantly higher than necessary for the DOP. It allows reusing the emulator board for different processors. The rest of the DOP is implemented in the Main FPGA (XC4013E-PQ160). The board also includes the additional 8k x 8bit SRAM circuit as the main memory for the program and data and some interface circuitry with the host system (implemented in separate Interface and Control FPGA). SW on the host computer controls all functionality of the emulator board.

Having configured the Main FPGA, the control memory could be downloaded with the microprogram. The SW on the host computer generates clock signal for CPU. The status of each register could be read out from the DOP processor after each clock cycle. Besides this debugging mode, the DOP processor is able to execute the sequence of instructions independently and later generates an interrupt to the host system.

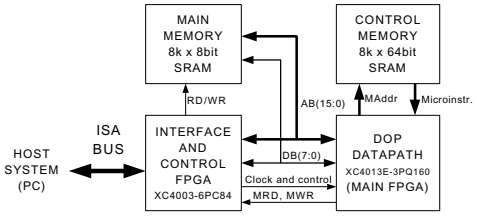


Fig. 9. DOP HW emulator board block diagram

Moreover, the HW emulator board offers the possibility of experiments that are not possible with the SW simulators. For example, it is possible to extend the basic DOP processor by additional functional units such as multiplier or divider and control them by currently unused signals in the control memory.

## 5. USE OF DOP IN COMPUTER AND LOGIC DESIGN COURSE

### *5.1 Overview of seminars*

The "Computer and Logic Design" course is in the typical format of CTU. It takes one semester (14 weeks). Every week is a single 90-minutes lecture and 90-minutes seminar. Seminars are held in classrooms or in laboratories. The table 4 describes the current schedule of these course seminars. Most of the seminars are held in classroom and there are only two laboratory seminars. This is not optimal, but it is the result of limited availability of laboratories, which are used by parallel Logic Design course.
It can be also seen that DOP currently occupies approximately half of the semester.

Table 4 Example of Computer and Logic Design seminar schedule

| Seminar | Scope |
|---|---|
| 1 | Introduction to DOP processor, Instruction Set Architecture and Data Types |
| 2 | Principles of synchronous design, Datapath of DOP - design of registers, ALU and interface circuitry, arithmetic |
| 3 | DOP controller implementation, principles of horizontal microprogram., discussion of possible enhancements |
| 4 | DOP basic cycle, DOP firmware, homework assignment |
| 5 (laboratory) | Exercise with DOP cycle accurate simulator |
| 6 (optional) | Evaluation of homework on VHDL simulator and HW emulator |

### *5.2 DOP classroom seminars*

During the first four seminars the internal organization of the DOP processor is explained to students. This is done with aim to maximally involve students in explaining the DOP schematic diagrams. These seminars have a strong link to lectures and goal is to illustrate the topics of lectures on practical examples.
For example: during the DOP ALU description, arithmetic is exercised and other possible organizations are discussed. Similar approach is used for explaining the DOP controller and basic cycle.

### *5.3 DOP laboratory seminar*

After explaining the DOP schematics, the SIMDOP – functional clock cycle accurate simulator is introduced during laboratory seminar. Students are let to write a short program in DOP assembly language, translate instructions into hexadecimal form and execute them step-by-step on the simulator. Students are also shown that the same program is executed in the VHDL simulator and most importantly on the HW emulator board. The majority of students does not understand the VHDL simulator and the HW emulator board but they appreciate that the processor "really exists".

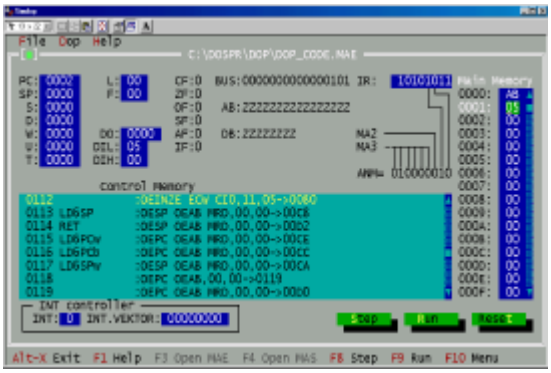


Fig. 10. SIMDOP – functional cycle accurate simulator

### *5.4 DOP homework assignment*

The main educational method based on the DOP is the homework, which is solved by every student independently. Each student has to write a microprogram implementing some complex instruction, which can possibly extend the DOP instruction set. Currently, we have collection of around 50 complex instructions usable as homework. Most instructions are extending DOP arithmetic capabilities (e.g. multiplication and division, operations on long operands). Typical complexity of homework is between 10 and 20 microinstructions. One of the requirements is

preserving functionality of the original instructions. Two registers U and T were added to the DOP for making the implementation of these complex instructions easier.

The microprograms are developed using functional cycle accurate simulator (SIMDOP), which is available to all students (see figure 10). For easier writing of horizontal microinstructions symbolic language - *microassembler* and its compiler were developed. Microassebler represents each microinstruction as a set of active signals and uses labels to represent addresses in Control Memory (see Control Memory section on figure 10). Compiler is responsible for allocation of microinstruction in the Control Memory and translates the microprogram to the binary format. It also does some correctness checks on the microprogram – for example it checks the control of tri-state buffers to avoid the most frequent mistakes coming from the collision on the internal bus.

This compiler simplifies the student task but can also lead to misunderstanding of the real content of Control Memory (binary format of microinstructions is created as a side effect of compilation, this format can be used with VHDL models).

Besides the implementation of the instruction, student has to write a report, which can become a part of DOP's documentation for a programmer. Students have to describe the number of clock cycles and typical use of the added instruction.
Homework are reviewed at the end of semester by the teaching assistants and become part of students evaluation.

*5.4 Optional seminar*

At the end of semester one optional laboratory seminar is offered. There is usually no time to present this for every student. Volunteers are typically students more interested in HW. During this seminar, homework are evaluated on the VHDL models and downloaded to the HW emulator.

5.5 *Experience with the DOP*

Before the DOP processor was chosen a simple CPU based on AMD 2900 CPU slices had been used for the same purpose. It means that this approach of teaching this course has a relatively long tradition.
Student reports shows that t he main pedagogical tool is the homework. It has to be stressed that nowadays the main goal of the homework is not to teach microprogramming but make the students understand how the processor works. Microprogrammed controller offers the possibility to easy modify the processor functionality, which is advantage over the hardwired controller.

Student reports shows very frequently that they were afraid from the complexity of the homework but finally found it simple and interesting. They claim that homework helps them to finally understand how the processor works. It also seems that classroom seminars are not very efficient way to explain the DOP and students forgot most of it before they start to solve the homework assignment.

It is also interesting to note the most frequent mistakes students make in homework. Typically they use some dedicated register as PC or SP for intermediate storage in instruction and do not understand that it is not possible. Second most frequent mistake is made in reports where students are not able to write a reasonable description of instruction for a programmer. Description usually contains a lot of implementation details but important requirements for the programmer are omitted (state of input registers, modified registers, and flags).

*5.6 Relation of the DOP to more advanced courses*

It has been shown that the DOP reasonably describes the basic principles of computer organization, which are explained in the "Computer and Logic Design" course. It introduces some old concepts (accumulator-oriented ISA, microprogramming), which are not currently used in the mainstream general-purpose computers On the other hand similar processors are still used in the area of embedded systems. It seems more appropriate to show these principles on this type of architecture than building some artificial combination of RISC ISA and non-pipelined datapath with microprogrammed controller.

After introducing the quantitative approach and ILP in the following "Computer Architecture" course, students can see that the main idea behind the DOP (orientation to simple HLL compilation and high instruction encoding density) has lead to simple processor but with poor performance. More advanced concepts such as pipelining and superscalar execution are explained on the RISC processors. This is perhaps not a direct way to the contemporary computer architecture but it explains the evolution of processor architectures and ISA and relation between them.

## 6. CONCLUSIONS AND FUTURE WORK

The DOP core presents an example of a simple processor that could be used to illustrate the basics of computer organization and digital design. In comparison with commercial products, it has simple and orthogonal architecture. The DOP is currently used in introductory computer organization course seminars. Students participate on the design of the DOP datapath and controller and finally write a microprogram for DOP, implementing some complex instruction. Students use the cycle accurate simulator for homework assignment and could later evaluate the results on more accurate VHDL model or HW emulator board. The feedback from our students is mostly positive. They claim that homework assignment finally helped them to understand the processor functionality. More experienced students appreciated introduction to FPGA and VHDL style of circuit description.

In the future we want to increase the number of laboratory seminars which are more efficient than explaining the DOP processor in the classroom. We prepare new laboratory seminar for experimenting with C compiler and DOP instruction-cycle accurate simulator. The HW emulator board can be also used more efficiently. New experiments with this emulator would allow extending the DOP by additional units such as multiplier and controlling it by spare bits in the microinstruction. However, this requires significant rearrangement of the seminars and other courses.

At the same time the DOP is used also in more advanced digital design courses for experiments with FPGAs and as a simple target for developing compiler from subset of C in the introductory to compilers course. Currently, we plan to make the DOP documentation and tools available via Internet and JAVA version of SIMDOP is prepared.

## REFERENCES


Brorson, M. (2002). MipsIT – a Simulator and Development Environment using Animation for Computer Architecture Education, In: *Proc. of Workshop of Computer Architecture Education*, Anchorage, USA

Bruschi, S. M. (1999). Simulation as a Tool for Computer Architecture Teaching, In: *Proc. of SCS Summer Simulation Conferenc*e, pp. 81-86., Orlando, USA

Danecek, J., Drápal, F., Pluhácek, A., Salcic, Z., Servít, M. (1994a): DOP – A Simple Processor for Custom Computing Machines. In: *Journal of Microcomputer Applications ,* vol. 17*,* pp. 239-253, Academic Press Limited

Danecek, J., Drápal, F., Pluhácek, A., Servít, M. (1994b) The Architecture of General-Purpose Processor Cell. In: *Proc. of 4th International Workshop on Field-Programmable Logic and Applications, FPL94*, pp. 321-325, Prague

Danecek, J., Drápal, F., Pluhácek, A., Salcic Z.,Servít, M. (1994c) Methodologies for Computer Aided Hardware/Software Co-Design Using Field Programmable Gate Arrays. In: *Research Report.* Department of Computers, CTU Prague

Drápal, F., Danecek, J., Pluhácek, A., Servít, M. (1995), Implementation of a General-Purpose Processor Macro, In: *Proc. Design Methodologies for Microelectronics,* pp. 89 –97, Smolenice, Slovakia

Ellard, D., Holland, D., Murphy, N., Seltzer M. (2002) On the Design of a New CPU Architecture for Pedagogical Purposes In: *Proc. of the Workshop of Computer Architecture Education*, Anchorage, USA

Patterson, D., Hennessy, J. (1998), Computer Organization and Design: The Hardware/Software Interface, 2nd edition, Morgan Kaufmann Publishers , San Francisco, USA

Patterson, D., Hennessy, J. (2002), Computer Architecture A Quantitative Approach, , 3rd edition, Morgan Kaufmann Publishers, San Francisco, USA

Yurcik, W., Wolffe, G., Holliday, M. (2001). A Survey of Simulators Used in Computer Organization/Architecture Courses, In: *Proc. of the 2001 Summer Computer Simulation Conference (SCS 2001)*, Orlando, USA

# Superscalar Out-of-Order Demystified in Four Instructions

James C. Hoe
Computer Architecture Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213
jhoe@ece.cmu.edu

*Abstract—This paper describes a processor design project intended to illustrate the detail inner workings of modern superscalar out-of-order processors. In the project, the students implement a cycle-accurate RTL-level model of an out-of-order core—including rename, issue, execute, completion and retirement stages—based on the MIPS R10000. The processor core only supports four instruction types. First, the basic integer subtract instruction is included to exercise the mechanisms related to register-renamed out-of-order execution. Second, two types of branch instructions, resolving correctly and incorrectly respectively, exercise speculative execution and branch rewind capabilities. Lastly, an exception-triggering instruction tests the support for precise exceptions. The project is designed to be completed in six weeks by a team of two to three students with solid background and strong interest in computer architecture and digital design. This project has been used twice in an advanced graduate computer architecture course* (CMU 18-744 Hardware Systems Engineering) *and has received favorable feedback from students and industry recruiters. The project handout and required Verilog source files can be downloaded from* http://www.ece.cmu.edu/~jhoe/superscalar.

## I. Introduction

Implementing an n-stage in-order pipelined processor is the staple design project in undergraduate introductory computer architecture courses. Such a hands-on project is very instructive in that the students walk away with an in-depth understanding of not just the abstract principle of pipelining but also the exact mechanisms that make a real instruction pipeline work well (e.g., stalling, squashing and forwarding). Modern superscalar out-of-order microarchitecture, on the other hand, is a central topic in most graduate-level computer architecture courses. Unfortunately, due to the complexity of the subject, the material is often presented at a fairly high level; rarely are students required to work out a coherent, complete datapath in a hands-on fashion. It is our contention that most students are only able to walk away with a "warm-and-fuzzy" understanding of how an out-of-order core really operates. In other words, many students may comprehend the basic principle of microdataflow instruction scheduling, but very few students would be able to accurately describe the intricacies in the instruction issue and data forwarding logic that permit two instructions with read-after-write dependency to execute in back-to-back cycles.[1]

In this paper, I described a project designed to impart students with *precise* and *accurate* understanding of modern superscalar out-of-order processor design. In this project, the students implement a cycle-accurate RTL-level model of an out-of-order core, i.e., rename, issue, execute, completion and retirement stages. To stay within a reasonable workload, students only need to support the execution of four simple instruction types. First, the integer subtract instruction is sufficient to exercise all of the mechanisms involved in register-renamed out-of-order execution. Second, two types of branch instructions, resolving correctly and incorrectly respectively, exercise speculative execution and branch rewind capabilities. Lastly, an exception-triggering instruction tests the support for precise exceptions. The RTL model produced by the student must be both simulatable and synthesizable. The students' final RTL implementations are evaluated both in terms of IPC performance and hardware cost.

[1] Data dependence between a pair of dependent single-cycle ALU instructions is resolved in the same cycle when the producer instruction is selected for issue. This way, the dependent instruction itself is eligible for scheduling in the next cycle while the producer instruction is still being executed.

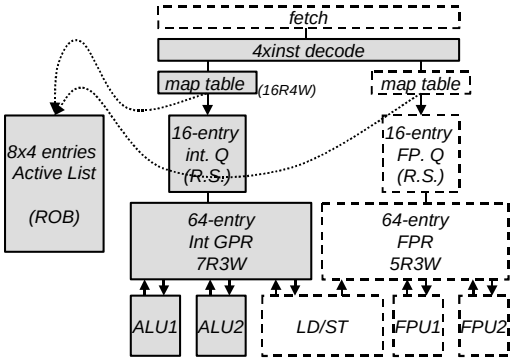


Fig. 1. MIPS R10000 Block Diagram

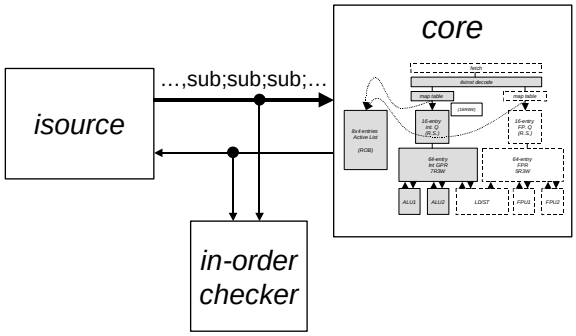


Fig. 2. The Simulation Environment

The project can be completed in six weeks by a team of two to three graduate (or advanced undergraduate) students who have solid background and strong interest in computer architecture and digital design. This project has been used twice in an advanced graduate computer architecture course (*CMU 18-744 Hardware Systems Engineering*, Spring 2002 and Spring 2003). CMU 18-744 is a depth course in our ECE department's graduate computer architecture curriculum. This course has as prerequisite our first-year graduate computer architecture course (*CMU 18-741 Advanced Computer Architecture*). The project is liked by the students who have taken the class and has gotten positive comments from industry recruiters who talked to the students.

**Paper Outline:** Following this introduction, the remainder of the paper is organized as follows. Section II gives an overview of the project specification. Section III and IV provide details in the project setup and execution, respectively. Section V explains the project's stated objectives and acceptance criteria. Section VI suggests ways to extend or expand the project in the future. Section VII concludes with a few remarks regarding our experience in running this project.

## II. Project Overview

The project calls for the Verilog RTL implementation of a superscalar speculative out-of-order core based on the MIPS R10000 microarchitecture. Figure 1 gives a high-level sketch of MIPS R10000's out-of-order core. The details of the microarchitecture are described extensively in [5]. The MIPS R10000 microarchitecture is selected for the project because similar microarchitectural arrangements—most notably the use of a common physical register pool to serve as both rename registers and architectural registers—are employed by most of the recent superscalar out-of-order processors.

The scope of the project only covers the mechanisms for renaming, microdataflow scheduling, data forwarding, branch rewind and exception recovery. Datapath elements relevant to this project are highlighted in gray in Figure 1. Memory and floating-point portions of the datapath are left out in the current version of the project. For the purpose of testing and simulation, a synthetic randomized instruction source that mimics the instruction fetch stage provides test stimulus to the out-of-order core.

The synthetic instruction source emits a randomized instruction stream composed of 4 instruction types in the MIPS ISA [3]. The out-of-order core only needs to support the execution of the integer subtract instruction (SUB rd, rs, rt). A sequence of SUB instructions with randomized source and destination registers is sufficient to exercise the hardware related to register renaming, microdataflow scheduling, and data forwarding.

The synthetic instruction source also emits ADD, BNE and BEQ instructions. The semantics of these instructions, however, have been *redefined* for the purpose of exercising the mechanisms for precise exception and branch rewind.

- The execution of an ADD should always lead to an exception. The ADD instruction itself cannot be finished. The state of the out-of-order core should rewind to just before the ADD instruction prior to continuing with the corrected instruction fetch stream.
- The execution of a BEQ should always confirm its corresponding branch prediction, requiring no change to the incoming instruction stream.
- The execution of a BNE should always reverse its corresponding branch prediction. The state of the out-of-order core should rewind to just after the BNE prior to continuing with the corrected instruction fetch stream.

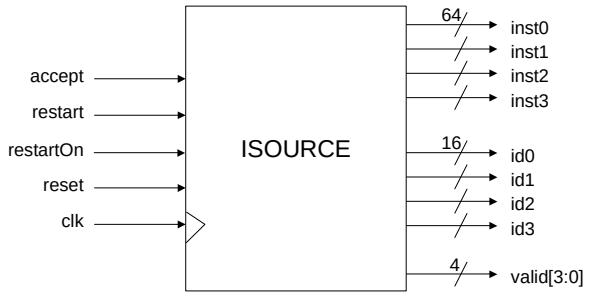


Fig. 3. The isource Module Interface

The frequency of these special instructions can be adjusted as necessary. The out-of-order core implementation is not allowed to take advantage of the special semantics of the ADD, BEQ and BNE instructions except when the instructions are being executed. In other words, until ADD, BEQ and BNE reach the execution unit, they must be treated normally as if they were expected to complete; similarly, the speculatively fetched wrong-path instructions following an ADD or BNE must also be treated normally (although they must be later discarded) until the exceptional condition is determined in the execution unit.

## III. PROJECT SETUP

The students are provided with two behavior-level Verilog modules that constitute the testbench environment for developing their core (Figure 2). The first is a synthetic instruction source, and the second is a checker module.

### *A. Instruction Source Module*

The isource module mimics a 4-wide instruction fetch buffer. The interface to the isource module is depicted in Figure 3. On each cycle, the isource module presents a sequence of 0 to 4 randomly generated instruction words on its four inst ports. The corresponding bits in the 4-bit valid mask indicate the validity of individual instruction words. If less than four instructions are valid, the valid instructions are always clustered together toward inst0. The output of the inst and valid ports do not change until the accept input port is asserted on a clock edge. (See example waveform in Figure 4). In other words, the instruction fetch stream can be stalled by deasserting accept. The instructions that follow a BNE instruction are, by our redefinition, wrong-path instructions and hence must be discarded after the BNE instruction is later executed. After branch

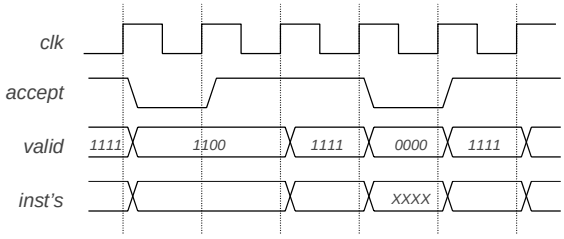


Fig. 4. Stalling Fetch by Deasserting accept

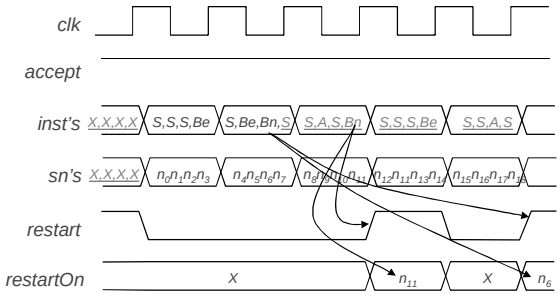


Fig. 5. Restarting Fetch by Asserting restart

rewind, the correct instruction stream is resumed by asserting the restart input port for 1 clock edge. (See example waveform in Figure 5.) The new instruction stream begins immediately on the following cycle. Instruction fetch is restarted in the same way following a precise exception caused by an ADD instruction. The isource module generates a sequencing ID for each instruction in the stream. The sequence ID of the exceptional instruction must accompany the assertion of the restart signal to properly resolve the situation when nested branch mispredictions are resolved out of program order.

### *B. Checker Module*

The checker module maintains a shadow copy of the architecture register file. The checker module passively monitors the activities on all input and output ports of the isource module. The checker module computes the correct in-order-state of the register file according to the observed instruction stream. The checker module executes the instructions in order. Instruction processing is skipped following an ADD or BNE instruction until the restart signal is asserted for the correct exceptional instruction. The checker module provides a reference state to verify the execution of the out-of-order core. The checker module also collects and displays basic performance and instruction stream statistics (e.g., IPC, instruction mix, and the number of exceptions) during

simulation.

## IV. Project Execution

Four major milestones demarcate the different phases of the project. These milestones both help pace the students' effort and also steer the students' attention.

- **Phase 1:** Develop a one-instruction-wide out-of-order core for just the SUB instruction. The core only needs to handle one instruction per cycle in each of the decode, dispatch, execute and writeback stages. The emphasis in this step is to develop the register renaming and dataflow algorithms. This step is allotted 2.5 weeks, which includes allowance for getting up to speed on the MIPS R10000 microarchitecture.
- **Phase 2:** Extend the one-instruction-wide core to support branch instructions (BNE and BEQ) and branch rewinds.
- **Phase 3:** Extend the one-instruction-wide core to also support precise exceptions. Step 2 and 3 are together allotted 2 weeks.
- **Phase 4:** Extend the fully-capable core from one-wide to superscalar operations in all stages. This step is allotted 1.5 weeks.

An appropriately restricted isource module is provided for in each phase to facilitate testing of the restricted core in the first three phases.

The project can run alongside of a normal lecture sequence. However, a part of each lecture should be reserved to discuss and clarify project related issues. As necessary, a number of the lectures can also be devoted to covering the more subtle details of the MIPS R10000 design. Another option is to have the students take turn presenting different aspects of the MIPS R10000 core, as described in [5].

The RTL models produced for the project must not only simulate correctly but also be synthesizable. Synthesizabilty is a project acceptance criterion to ensure the students do not include unrealistic hardware structures in their processor models. The students can only implement the core using the synthesizable subset of Verilog Hardware Description Language [2]. In this regard, students with RTL design experience have a significant advantage. Therefore, it is important each team includes at least one member who is familiar with the RTL design flow.

Students are encouraged to follow a top-down design flow where they begin with a very high-level, possibly behavioral, model for the major datapath structures. Next, they can refine the datapath structures piecewise from behavioral Verilog down to synthesizable RTL code. After each refinement step, the students can immediately simulate against the testbench environment to ensure the correctness of the most recent changes.

Although the core must be implemented using the synthesizable subset of Verilog, students are encouraged to incorporate behavioral-level Verilog code for monitoring and debugging purposes. These out-of-band behavioral debugging code can dynamically examine the processor state cycle-by-cycle and compute elaborate runtime invariant conditions. In our experience, carefully designed runtime invariants are very powerful debugging aids. These invariants help flag an erroneous operation in a timely manner such that they allow much better localization of the origin of that error.

## V. Project Objectives

### A. General Implementation Guidelines

It is suggested that the students base their core on the MIPS R10000. As a general guideline, their RTL models should capture all details explicitly mentioned in [5]. Nevertheless, the students also need to improvise in several places where design decisions are not spelled out explicitly. In addition to the general guideline above, the following set of specific criteria must be met by the students' RTL models.

1) The execution of the out-of-order core must correspond to the reference behavior of the checker.
2) In the out-of-order core, pending instructions must issue as soon and as fast as possible after (true) data dependence and structural hazards have been cleared.
3) Back-to-back dependent instructions must be capable of issuing on consecutive cycles (in the absence of structural hazards).
4) Branch rewind must be fast, i.e., taking O(1) time, independent of the number of instructions to rewind.[2] A branch rewind must start and complete as soon as possible, without waiting for older instructions to retire.
5) Exception rewind can be slow, i.e., taking O($n$) time where $n$ is the number of instructions to rewind.
6) The core cannot make use of the special semantics of ADD and BNE instructions until they are in the functional unit.

[2] This criterion forces the student to implement a "branch rewind stack".

Other aspects of this design project are essentially left to the students' whim.

### B. Evaluations

The quality of the resulting RTL model is judged on correctness, IPC performance and hardware cost. For the early milestones, the acceptance criterion is simply for the out-of-order core to simulate correctly against the test environment (isource and checker) for an agreed upon number of instructions. During simulation, the checker keeps count and reports the progress. After a sufficient number of instructions has elapsed, instruction fetch from isource is stopped, and the out-of-order core is switched into the "drain" mode. The students can next verify that the out-of-order core's committed register file state agrees with the reference register file state in the checker.

In Phase 4, the out-of-order core must also achieve a minimum IPC performance. The IPC lower bound helps diagnose performance bugs. For example, we had seen a case where one team did not handle back-to-back execution of dependent instruction. The consequence of this oversight is obvious in their abysmal IPC relative to the other teams. During Phase 4, the different teams in the class are routinely informed of each other's latest IPC achievements. This effectively turned the last phase of the project into a competition. The students were self-driven to fine-tune things like the issue priority logic and the branch rewind process to stay ahead of the rest of the class.

Although the RTL models produced by the students are synthesizable, we do not use the synthesis outcome to evaluate hardware cost. Synthesized storage structures (RAM, CAM, and FIFO) are much less efficient that the custom blocks instantiated in real processor implementations. As a compromise, we compute an estimate manually using empirical values. We assume each bit of storage (a bit cell) costs 1 unit and the integer ALU costs 50,000 units. To compute the final cost of a storage structure, the raw bit-cell cost must be further multiplied by the number of normal read or write ports and by two times the number of associative lookup ports. (For logical structures that require different types of references in its different columns, the students would have to break the logical structure into its physical components to get an accurate estimate.) This very coarse grain model does not account for random logic, registers or routing congestions. Nevertheless, it does force the students to be aware of the tremendous cost and tradeoffs of adding additional ports and associative lookup.

## VI. Project Extensions

The project as described is designed to be completed by groups of two to three students in a half semester (~six weeks). The duration and scope of the project can be extended by including other aspects of modern high-performance processors, such as branch prediction, aggressive load/store ordering and cache hierarchy. Here, we briefly suggest some specific ideas.

1. Instead of MIPS R10000, one could also retarget the project to be based on Alpha 21264 with clustered datapath. This microarchitecture is as described in detail in [3].
2. One could augment a baseline implementation of MIPS R10000 with support for Simultaneous Multithreading (SMT) [1]. The project would serve to clarify the implementation consequences of SMT support.
3. The project currently does not handle loads and stores. The memory subsystem in modern superscalar processors can easily be made into a similar but stand-alone project.
4. Similarly the project can also be extended to examine instruction fetch and prediction issues in modern superscalar processors (e.g. wide fetch using a trace cache).

The key is to carefully confine the scope of the project so the students are exposed to all of the important details but without unnecessary tedium.

## VII. Conclusion

The intent of this project is for students to gain an in-depth understanding of modern superscalar microarchitecture through hands-on practice. In addition, the project also gives the students a chance to experience issues in project teaming and participate in a full engineering cycle of specification, implementation, validation and analysis. We believe this course is invaluable training for students who are headed for either industry or graduate research.

This project is challenging and time-consuming. Although the students often gripe about its load and difficulty, in our experience, the students really did enjoy spending the time to work out the details, especially for the moments when a fuzzy concept in their head suddenly becomes crystal clear in their implementation.

## REFERENCES


[1] S. J. Eggers, J. S. Emer, H. M. Leby, J. L. Lo, R. L. Stamm and D. M. Tullsen, "Simultaneous multithreading: a platform for next-generation processors," *IEEE Micro*, vol 17 no 5, pp 12-19, Sep/Oct 1997.

[2] *HDL Compiler for Verilog Reference Manual*, Synopsys, Inc., 2000.

[3] G. Kane and J. Heinrich, *MIPS RISC Architecture*, New Jersy:Prentice Hall, 1991.

[4] R. E. Kessler, "The Alpha 21264 microprocessor," IEEE Micro, vol 16, no. 2, pp 24-36, Mar/Apr 1999.

[5] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Mic*ro, vol. 16, pp. 28-41, Apr 1996.

# Bridging the Gap between the Undergraduate and Graduate Experience in Computer Systems Studies

Lori Carter and Scott Rae
Department of Math and Computer Science
Point Loma Nazarene University
San Diego, California 92106
{lcarter, srae}@ptloma.edu

## Abstract

This paper presents the contents of a Special Topics Class used to introduce undergraduate students to the different approach to learning and thinking found in the graduate level Computer Systems environment. During the first half of the semester, the students received instruction in reading and writing technical documents, and had experience with professor-guided self-study. During the second half of the semester, the students presented an idea for a simulator, built it, and used it to conduct simple experiments. After completing the experiments, they were encouraged to brainstorm variations to the algorithms or architectures they simulated, which would improve performance. Student evaluations revealed that for 50% of the students, their interest in graduate school increased as a result of the class.

## 1 Introduction

The Computer Science undergraduate experience tends to be one of passive learning, where the professors are the producers and the students the consumers. Although in early classes this approach is necessary, and in later classes it is certainly changing [11] [8] [9] there is still a chasm between the undergraduate style of learning, and the graduate school experience, where the student becomes the producer. The disparity between the two levels of education affects both the number of students considering graduate school and the quality of the students entering graduate programs. For some students, obtaining a Ph.D. is not even a consideration. With a bachelors degree in hand, they are competent programmers and tired of sitting in classes. Completely unaware of the challenges and opportunities that await, the graduate school scenario they view as "more of the same" is unappealing. Conversely, if the student has had some exposure to the opportunities for fascinating research, creativity and autonomy found at the Ph.D. level, the tendency is to wonder if their undergraduate education has been adequate to make them a viable candidate for such an undertaking. If these students do pursue graduate education, they are, in fact, often ill-equipped to succeed.

This paper describes a class that was designed to bridge the gap between the undergraduate approach to Computer Systems studies, and the graduate level approach. Where the undergraduate curriculum can be quite broad and structured, the graduate counterpart tends to be more correctly characterized as narrow, directed self-study. Successful graduate students must be able to find the necessary information and produce the desired results with a modicum of guidance from an instructor or advisor. They should have the ability to think "outside of the book", dreaming of new approaches rather than simply embracing what has gone before. In addition, graduate students are expected to be able to read, write, and present technical papers; using very different skills than those taught in English Composition and Speech 101.

In an effort to introduce students to the graduate level approach to learning and thinking, and to improve their confidence and competence with skills required at that level, a Special Topics course was used to introduce Juniors and Seniors to Computer Systems research. During the course of the semester, the students were required to learn Java (their first language was C++), using the directed self-study method. The end product of this class for each of the students was a Java Applet that simulated some component of Computer Architecture or Operating Systems, and which was amenable to simple experimentation. In preparation for building the simulator, each student presented a powerpoint presentation on the proposed project, receiving suggestions on design and presentation skills from their peers. They were introduced to technical papers written on

their topics, and received instruction on, and practice in, reading these papers. The students wrote a proposal and final paper on the chosen simulation using the technical style exemplified by the papers they read. Student evaluations revealed that for most of the students, their interest in graduate school had increased as a result of their experiences. The vast majority of students indicated that confidence in their ability to find the solution independently had dramatically increased. The remainder of the paper discusses the various components of the class, and the responses of the students involved.

## 2 Syllabus

The semester long Special Topics course was divided into 2 distinct halves, with grade weights assigned as follows:

**Phase 1 (individual):**

| | |
|---|---|
| Java, DOS labs | 30% |
| Technical Paper Summaries | 15% |
| Paper/Java/DOS Quizzes | 15% |
| Project Proposal | 40% |

**Phase 2 (group if desired):**

| | |
|---|---|
| Powerpoint presentation | 20% |
| Progress Meetings | 30% |
| Final project (paper, demo) | 50% |

The content of the first half of the semester was to be completed on an individual basis, while the second half of the semester could be completed in groups of 2 if desired. It is interesting to note that of the 14 students involved in the class, only 3 groups were formed. Most of those that chose to do the work as individuals indicated that the motivation for this choice was to reduce their dependence on others, and to increase their individual learning.

During the first 8 weeks, the students split their classroom time between laboratories and lectures. The lecture time was used for the introduction of Java topics and learning technical communication skills, while the labs focused mainly on self-learning of the Java language. Many of the Java Labs were based on the Sun Java Tutorial [17]. The schedule for lectures and labs is shown in Figure 1. Weeks 9-16 were spent building the simulator, beginning with a powerpoint presentation of their proposed design. Students met with the professor weekly to check progress and discuss changes. At the end of the semester, each student produced a simulation demonstration and final paper detailing their simulator design, experimental results, and future work.

| Week | Lecture | Lab |
|---|---|---|
| 1 | Intro to DOS, Java | Intro to DOS, "A Cup of Java" (Sun Tutorial) |
| 2 | Intro to Technical Papers | Group Analysis of organization, content, language of 3 systems papers |
| 3 | Basis Java Constructs, Discuss paper 1 | Sun Tutorial using "Click Me" applet |
| 4 | Intro to Java Interfaces, Discuss paper 2 | Modifying applets to implement Listeners |
| 5 | Java Threads, Layouts Discuss paper 3 | Animation Lab |
| 6 | Writing Proposal, Finding background papers | Finish Animation Lab Locate background papers |
| 7 | Writing a Simulator | Analysis of Turing Machine applet simulation |
| 8 | Technical Presentations | proposal draft meetings |

Figure 1: Lecture and Lab schedule for the first 8 weeks.

## 3 Directed Self-Study of the Java Environment

When this class was originally envisioned, an extensive search for a suitable Java textbook was conducted. Although many fine books exist, the conclusion was reached that part of the goal of this class was to encourage life-long self-learning. Consequently, the class was taught without any textbook, using internet-available resources and encouraging self-discovery rather than "spoon-feeding". Initially this was very uncomfortable for the students. However, as they began to work though the on-line tutorials, following links to various "rabbit trails" on related topics, most began to realize that there was a huge amount of information available on the internet, and that it was more current than what is available in a textbook. In addition, they became adept at looking for and finding examples of Java code that performed tasks similar to what they were desiring to do, modifying it to fit their needs. Relying on a textbook would have limited the examples that many students would have considered.

As can be seen from the schedule in Figure 1, Java was introduced to the students through lectures, tutorials, modification and analysis of existing Java Applets and through the experience of original applet creation. The goal of the lectures was to present basic differences between C++ and Java, and to provide introduction to Java Applet requirements, components and capabilities. The use of the Sun Java tutorials guided the student through learning a specific topic, while also providing links to be followed as the student's interest was piqued with regard to

a particular subject. As a part of other labs, students were directed to sources on the internet that exemplified specific features of Java (interfaces, layouts, animation) [2] [12]. Requiring the students to answer questions about, and modify, existing applets, forced them to understand how the features worked, and provided a model for original implementations. Written analysis of an existing simulation [14] provided a framework for putting together a larger project.

# 4 Reading and Writing Technical Papers

In a course focused on introducing students to the post-graduate style of learning, students must be taught to digest and comprehend technical papers, the primary source of information for most recent technical developments. To develop those skills, students first reviewed three sample papers: "A Comparison of Software Code Reordering and Victim Buffers[1]," "WSClock – A Simple and Effective Algorithm for Virtual Memory Management[6]," and "Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation[4]." The papers extended Computer Architecture and Operating Systems concepts that the students had encountered in prerequisite or corequisite courses.

To introduce the reading process, a brochure obtained from the internet [7] outlined basic steps to comprehension, such as underlining, outlining, note-taking, and defining unfamiliar words. A worksheet provided by the professor helped focus attention on five basic sections contained in some form in most papers: the abstract, the introduction, background information, methodology, and results. Students were asked to compare the structures of the three papers to see how different authors might represent the same basic information. Other questions, such as, "*How are the results presented? Text? Tables? Graphs?*" and, "*What is the purpose of the figures in each paper?*" led the students to evaluate both the contents and the presentation of the papers.

For the second phase of this component of the class, students summarized the papers' contents in a one-page write-up, based on the major points they had identified. In the summary, they were asked to begin to mimic the technically-oriented style modelled in the papers they had read. The act of condensing the information to a readable summary helped in data retention. In addition, to encourage good note-taking practices, quizzes were administered that tested the students' ability to extract details and data they were unlikely to simply retain. Students were allowed access to both their summaries and the papers, but a time limit ensured that only information that had been highlighted, noted, or outlined previously, would be helpful to the student in answering the questions.

During the second half of the semester, students had the opportunity to apply what they had learned. In preparation for designing their applet projects, the students were required to research their chosen simulation topic by locating and reading at least two existing papers closely related to the topic. At this point, the students moved from reading technical papers to writing them. After carefully researching the subject, class members wrote draft proposals in research format, detailing both the algorithmic and design requirements for the projects. The selected background papers were referenced to support the projects' relevance. Subsequent to simulation completion, final papers were written, modelled after those the students had studied, complete with figures showing actual simulation interfaces, and graphs detailing the results of their experiments.

# 5 Presentation

A second reason for not having a course text was economic. The variety of topics presented would have necessitated the purchase of multiple textbooks. Again, the internet was the source of expert information on good presentations. A very readable, practical essay on technical presentations was located [2]. Students were required to read the essay, and make a check-list of features that should exist in presentation. The following class period, the professor gave a powerpoint presentation on what was to be included in the student's presentation, and simulation project. The professor's presentation was critiqued by the class, based on their check-lists.

Finally, each student gave a Powerpoint presentation on their proposed simulations. This was used not only as an opportunity to practice presentation skills, but as an opportunity for peer feedback on their proposed simulator interface designs. The presenter as well as the audience received a grade for the exchange.

# 6 Simulation and Experiments

As was previously mentioned, the final project for this course was a Java Applet that simulated some aspect of Computer Architecture or Operating Sys-

tems. This project consumed most of the second half of the semester. During this time period, few formal class sessions were held. Instead, class time was filled with individual meetings between the student and professor, much the way an advisor-advisee meeting would occur at the graduate level. Requirements for the simulators were as follows:

- Must be completely original work, completed as a Java Applet (or Japplet)
- Must be amenable to experimentation, allowing user interaction
- Must be on a pre-approved topic
- Must be animated, showing a progression through an algorithm etc.
- Must display some kind of results

## 6.1 Why Design Simulators?

The final project for this course was a simulation for a variety of reasons. First, much of the systems research is completed using simulators [15]. Clearly, the simulator instructs and informs the user, particularly when the process simulated is depicted visually. A less obvious benefit is available to the constructor of the simulator. The building of the simulator improves the understanding of the topic simulated, and the process of re-creating a technique often has the effect of causing the programmer to consider other, better ways of accomplishing the simulated task.

Second, the simulator provides a target for research. Prior to designing the simulator, the students were required to find (and read) several technical papers on their subject, to improve their expertise. After construction, the simulators became a platform for conducting simple experiments. Although traditional Computer Architecture research is conducted on comprehensive simulators such as Simple Scalar [5], there are certainly arguments supporting the benefits of simpler component simulators [3]. Recent textbooks include access rights to web pages with links to applets supporting the concepts taught in their books[10].

The third reason for this project this is that it was meaningful for the students. The simulations will be used as learning tools for upcoming students in Computer Architecture and Operating Systems courses.

## 6.2 Simulator Implementations and Experiments

The interfaces for 4 of the simulators created are shown in Figures 2, 3, 4, and 5. Figure 2 shows the interface of a simulator designed for instruction on and experimentation with cache associativity. The user of the simulator inputs a string of memory references and the 2 levels of associativity to be compared. The user can request that the simulation run to completion and display the results (cache misses and total access time), or to step through each cache access. If the latter is chosen, the simulator displays which memory addresses fill the cache lines in each associativity version for each address of the string of references. The user learns how associativity works, which addresses are brought into the cache with a single address request, and can experiment with how different levels of associativity perform with different patterns of memory requests (random, sequential, clustered etc.).

Figure 3 displays an interface for an applet that simulates different CPU Scheduling algorithms. The user inputs tuples of information about several processes entering the ready queue for CPU time. Each tuple includes the process ID, the time of arrival in the system, and the duration of the current CPU burst for the process [13]. The output of the simulation is an animated Gantt chart displaying the CPU use of the processes. In addition, there is a text area reporting the ultimate turnaround time and response time for each process, as well as the throughput for the entire set of processes. The user learns how each of the algorithms work, and can experiment with the effectiveness of each algorithm for improving turnaround time, response time and throughput on different patterns of process requests.

Figure 4 depicts a simulator that can be used to explore different paging algorithms for the virtual memory system. The user can choose between the *FIFO* algorithm for page replacement, and the *Clock* (also known as *Second Chance*) algorithm [13] with 1 or 2 reference bits, and choose the number of frames assigned to the process . In addition, the user can enter a string of pages in the order in which they are required by the process. The output of the simulation is a visual display of each page entering its assigned frame, and the number of page faults resulting from the use of each algorithm. The project allows the user to experiment with different patterns of page accesses for several page replacement algorithms: *FIFO*, and several different implementations of *Clock* (1 bit, and 2 implementations with 2 reference bits).

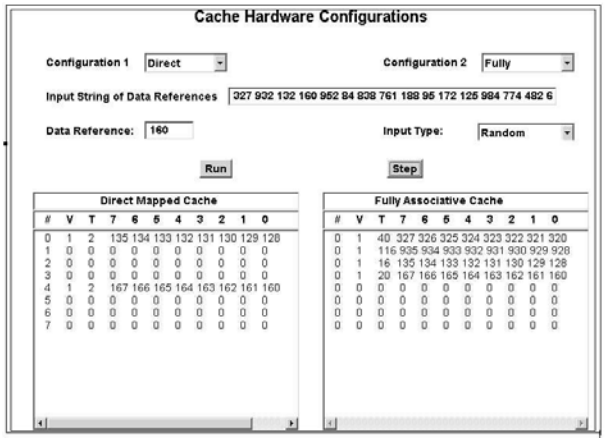


Figure 2: Simulation compares behavior of caches with different levels of associativity

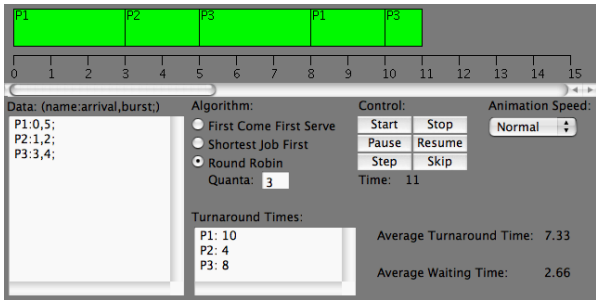


Figure 3: Interface for simulation of CPU scheduling

Although most of the students designed simulations based on ideas with which they were already familiar, some students became intrigued with new ideas based on more advanced material. Figure 5 shows an interface for a simulator demonstrating power dissipation and fan out. The user can choose a circuit and set the value for each of the circuit inputs. The simulation shows the propagation of the values through the gates, coloring outputs with red for 1 and blue for 0. Experiments using this simulator can help provide insight as to which combination of gates provides the optimal circuit in regard to power usage. Another student designed a simulator dealing with victim caches, a topic she became interested in after reading one of the introductory papers early in the semester [1]. Her experiments centered around the size or even existence of the victim cache.

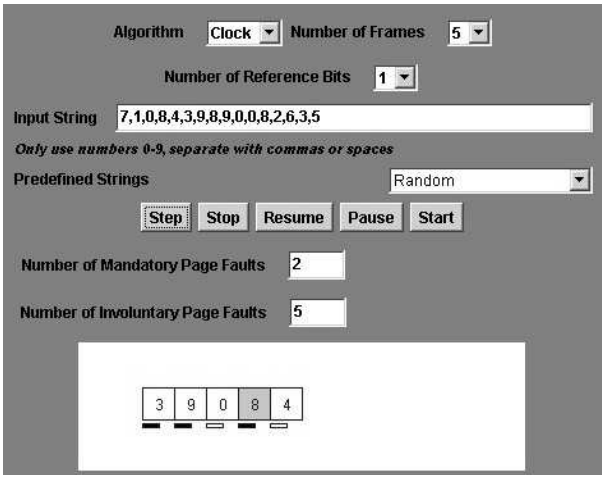


Figure 4: Simulation depicting page replacement

### 6.3 Progress Meetings

During the weekly progress meetings students were expected to present to the professor evidence of their progress, showing that they had completed the tasks discussed at the last meeting. In addition, they could receive individual help from the professor, and discuss possible implementation changes. Notice that this also served to assure the integrity and originality of the students' code. They were required to implement the suggestions that had been discussed the prior week. Thirty percent of the grade for the second half of the semester was based on their performance at these meetings.

## 7 Student Response

Overall, student response to the course was extremely positive. All agreed that it was very different from what they had previously experienced. A common thread that ran through student responses to most aspects of the class was that it was harder, but ultimately more enjoyable and satisfying than other Computer Science courses. The students learned a lot about their learning styles, and their ability to work independently. The students were asked to evaluate the learning experience provided by each component of the class on a scale of 1 to 5, where 1 is described as "fairly worthless" and 5 being "an excellent learning experience". Figure 6 shows the average ranking provided by the students for each component.

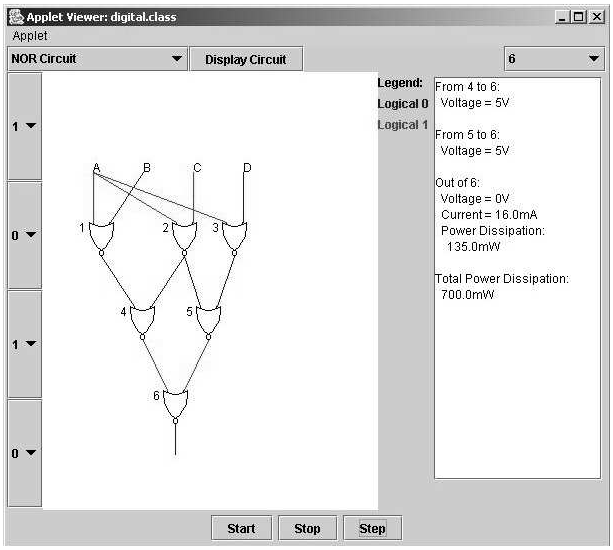


Figure 5: Interface of simulation demonstrating power dissipation

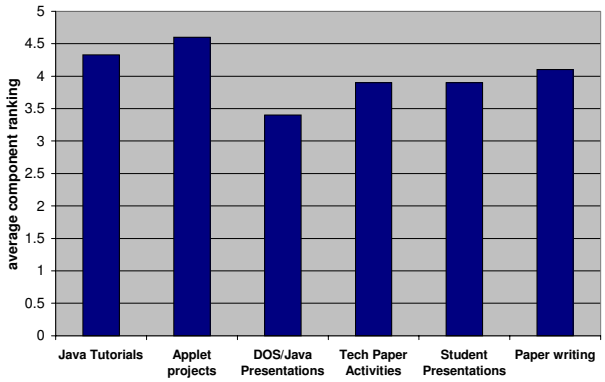


Figure 6: Student rankings of component learning experiences.

It is interesting to note that the DOS and Java presentations, the major lecture component of the class were ranked the lowest by the students, while the active, independent learning experiences (tutorials and projects) were ranked the highest. One aspect of the class, student presentations, involved the students sharing with each other things they had learned in the process of completing applet project assignments. This did not get as high a rating as was expected. When questioned about their responses, the students indicated that, although the material was interesting, it was hard to appreciate and apply when they weren't working on the same exact project.

Another questions that was asked on the evaluation was stated as follows:

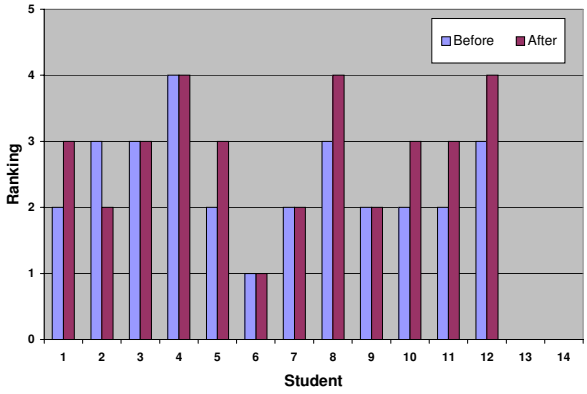


Figure 7: Student rankings on graduate school consideration before and after class

> *This class has introduced a number of things usually found only at the graduate level (reading, analyzing and writing technical papers, directed self-study, thinking more deeply about a subject than what is introduced in a textbook.) We're interested in knowing if this class has in any way influenced your desire to seek further computer science education at the graduate level. Please indicate your interest in graduate school prior to this class, and at this point in the semester on a scale of 1-5, where 1=grad school? No way, never!; 2=well, it isn't completely distasteful; 3=sounds kinds of interesting; 4=I would seriously consider it; 5=count me in!.*

Figure 7 shows the student responses for both before and after. There were 14 students involved in the class, and 2 did not take the survey. The remainder of this section describes in more depth the responses of the students to some of the class components.

## 7.1 Response to Self-directed Study Leading to Simulation Project

As is clear from the results shown in Figure 6, learning Java through tutorials and applet projects was generally well received. The internet-based assignments (tutorials and code modification) presented initial insights to the students as to where they

could locate the information required to complete the larger simulator project. It was still a giant leap to the implementation of this larger project. Many students were initially frustrated. They were not used to persevering when the information they wanted was not immediately available. Students approached this dilemma in 3 different ways:

1. They found examples of code that did tasks similar to what they wanted to do, copied it in to their project, and acted mystified when it didn't work.

2. They found simple examples of components of the tasks they wanted to accomplish, and tried to understand them and then put them all together, with varying degrees of success.

3. They studied complex examples of similar tasks until they really understood what was happening, and then made an overall plan for the completion of their project using similar techniques.

As can be expected, the first approach led to the most frustration. The more initial understanding that was gained, the greater the success, and ultimately, the less time was spent on completing the project. The students taking approach number 1 also gave up easily on the applet modification projects earlier in the semester. They were far too conditioned to being given a formula, and just plugging in different numbers without taking the time to understand the concept.

Fortunately, the majority of students did not take approach number one, and ultimately all were extremely proud of their successful completion of the assignment. Student comments included "it was like self-learning with a safety net - you could come to the professor if you were really stuck"; "I learned so much more this way. I had something to apply my discoveries to, and I found the answers myself, so the information will stick". Even the students who initially rebelled at the idea of self-learning gratefully received and read (on their own) texts on Java and Java applets and caught up with the rest of the class. Once they realized they weren't going to be spoon-fed, they learned how to feed themselves.

### 7.2 Response to Technical Paper Reading/Writing

The class members were required to read 3 technical papers over the course of 3 weeks. At first, they were resistant and frustrated. By the last paper, they had gotten quite good at extracting the important information. Several students indicated that they were intrigued with the information found in the papers, information not found in the textbook realm, and wanted to pursue the topics further. Most were delighted that they were able to make sense of the material by the third paper.

The students' writing improved dramatically over the course of the semester, beginning with the summaries and ending with the final paper. Some of the students who weren't terribly gifted in Composition 100 realized they had a talent for clear, concise technical writing. One of the students was coauthor on this paper.

### 7.3 Response to Simulator Building and Experimentation

Most students found the simulator building to be a very satisfying experience. Each student had the experience of becoming an expert in a small area of Computer Systems. They admitted to recognizing that their initial knowledge of their topic was imprecise. The processes of building the simulators required that they hone their knowledge of the algorithm or architecture. They seemed to enjoy the process of composing hypotheses and determining their validity.

## 8 Conclusions and Future Work

The Special Topics Class was designed to equip undergraduate students to be self-learners, creative thinkers and competent technical readers, writers, and presenters. It has appeared to be, for the most part, a very successful experiment. Fifty percent of the students responding to the evaluation noted an increased interest in graduate school. More informally, almost all of the students indicated experiencing a heightened level of confidence in their abilities to learn and perform.

The next time this class is taught, the number of lectures on DOS and Java would probably be decreased, freeing up time to do more in the form of tutorials, applet modification and code analysis. More attention would be paid to making the transition from simple applets to an extensive project less abrupt. In addition, the student presentations on "what I learned" would be reserved for the second half of the semester. During this time period, as the simulators were being constructed, students were much more eager to exchange information.

The internet is already an excellent source for current material in teaching computer systems [16]. We plan, in the near future, to add our contribution in the form of a web page with links to all of the student simulation projects for the purpose of teaching simple computer systems concepts.

Although this class involved only 14 students, it could easily scale to a larger class size with the help of graduate assistants. This would actually provide invaluable experience to Ph.D. students who, themselves, would like to pursue academia as a career.

## 9 Acknowledgements

Paul Kelly, Jeremy Bradney, Richard Trager, Steven Potter and Jenni Sapp contributed interface designs to this paper.

## References

[1] I. Bahar, B. Calder, D. Grunwald, "A comparison of software code reordering and victim buffers," *Third Workshop on Interaction between Compilers and Computer Architectures*, October 1998.

[2] K. Boone, "The k-zone," http://www.kevinboone.com

[3] P. Bose, "Simulation in the small: the case for simpler models and testcases in computer architecture education and research", *Proceedings WCAE 2002, Workshop on Computer Architecture Education,* Vancouver, BC, June 10, 2000

[4] R. P. Brent, "Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation," *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 3, July 1989.

[5] D. Burger, T.M. Austin and S. Bennett, "Evaluating future microprocessors: the SimpleScalar tool set," Tech. Rept. CS-TR-96-1308. Univ. of Wisconsin-Madison, July 1996.

[6] R. Carr and J. Hennessy, "WSClock - a simple and effective algorithm for virtual memory management," *Proceedings of the Eighth ACM Symposium on Operating System Principles,* December 1981.

[7] M. J. Hanson, D. McNamee, "Efficient reading of papers in science and technology," http://www.cse.ogi.edu/~dylan/efficientReading.html

[8] J. Herath, S. Ramnath, A. Herath, S. Herath, "An active learning environment for intermediate computer architecture courses," *Proceedings WCAE 2002, Workshop on Computer Architecture Education,* Anchorage, AK, May 26, 2002

[9] W. T. Hsu, "Experiences integrating research tools and projects into computer architecture courses," *Proceedings WCAE 2000, Workshop on Computer Architecture Education,* Vancouver, BC, June 10, 2000

[10] J. Kurose, K. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*, Addison-Wesley, 2003

[11] I. Papaefstathiou, C. P. Sotiriou, "Read, use, simulate, experiment and build: an integrated approach for teaching computer architecture," *Proceedings WCAE 2002, Workshop on Computer Architecture Education,* Anchorage, AK, May 26, 2002

[12] R. Sebesta, *Programming the World Wide Web,* Addison-Wesley, 2002

[13] A. Silberschatz, P. Galvin, G. Gagne, *Applied Operating Systems Concepts,* John Wiley and Sons, 2000.

[14] S. Skinner, "Turing machine simulator applet," http://www.wap03.informatik.fh-wiesbaden.de/weber1/turing/tm.html

[15] C. Weaver, E. Larson, T. Austin, "Effective support of simulation in computer architecture instruction," *Proceedings WCAE 2002, Workshop on Computer Architecture Education,* Anchorage, AK, May 26, 2002

[16] W. Yurcik, E. Gehringer, "A survey of web resources for teaching computer architecture," *Proceedings WCAE 2002, Workshop on Computer Architecture Education,* Anchorage, AK, May 26, 2002

[17] "The java tutorial," http://www.java.sun.com/docs/books/tutorial

# Integration of Computer Security Laboratories into Computer Architecture Courses to Enhance Undergraduate Education

Jayantha Herath, Susantha Herath, Ajantha Herath*
St. Cloud State University, St. Cloud, MN 56301
*University of Dubuque, Dubuque, IA 52807
jherath@stcloudstate.edu

**Abstract**

**Most computer science and engineering programs have two or more required computer architecture courses but lack suitable interfacing laboratory experience for other upper-level classes. Information assurance and network security tracks have been developed over the recent years without providing necessary and sufficient background knowledge in logic, storages and processor architecture. Integration of real-world applications is always a better approach to not only to excite the passive student body but also to explore the computer architecture subject area. At the intermediate level, architecture knowledge can be extended to provide information and network security experiences to students. Such extensions to the course will provide proper interfacing to networking, operating systems, databases and other senior level security related courses. This paper describes possible integration of security and privacy concepts into computer architecture course sequence with hands-on classroom activities, laboratories and web-based assignments.**

## 1. Introduction

One day a tenured Professor of Medicine received an e-mail from a close friend via Yahoo e-mail, with an attachment file. He opened this e-mail with other messages, and continued to work as usual. The attachment contained a worm that causes automatic transmission of more e-mail messages. A few days later his computer was confiscated, his supervisors accused him of creating and transmitting a virus from his computer, and the FBI was called in. His entire address list had received e-mails with a worm, automatically. One and half years later he found himself indicted by a grand jury for violating Federal law 18 USC 1030 a 5 (a). The grand Jury charged that this Professor "knowingly caused the transmission of a program, information, code or command, and as a result of such conduct, did intentionally cause damage without authorization to a protected computer, which is used in interstate and foreign commerce and communication, and, by such conduct, caused loss to one or more persons during a one-year period aggregating at least $5000.00 in value". The Professor had to find an attorney to represent himself. Many attorneys asked him to pay $150,000 and another asked for $60,000 upfront and $450 per hour to represent him in this case.

The case above illustrates the complexity of conflicting technical and legal issues. Interestingly, most lawyers and judges do not understand the technologies involved and need help from technical expert witnesses to find out what happened in the computer using the forensic evidence available in the storage. And often the technical experts do not know much about the legal issues involved. Finding the hidden evidence in storage systems and presenting it in an acceptable form to the court is a hard problem for a computer architect to solve. For this reason, it is important to provide basic computer forensic techniques in a computer architecture setting, to provide a way to recover information in storage systems that ensures its integrity.

In general, knowledge gained in computer architecture courses serves as the gateway for upper level undergraduate computer science courses. The main objective of this study is to develop computer architecture course modules with computer security applications for undergraduate students. One of the challenges facing us in the classroom is finding experiments to engage the interests of students while improving the quality of computer architecture courses [1]. The goal has been and continues to be helping them become good computer scientists in a relatively short period of time with a solid grounding in both theoretical understanding and practical skills so that they can enter the profession and make valuable contributions to the society. The proposed active learning modules aim to provide students with an exciting learning environment and the necessary tools and training to become proficient in the computer architecture subject matter with applications in security and privacy. The following sections outline the details of course plan, examples, assessment plan, future work and summary.

## 2. Detailed Course Plan

To help master computer architectures, our curriculum provides three semester courses. The first course in this sequence covers the fundamentals of digital logic circuit design [2]. This foundation course helps the students develop component integration skills from gate-level to register-transfer-level, when designing a circuit to perform a particular task. The laboratories for this course consist of hardware and VHDL software simulations of combinational and sequential digital logic circuits. The intermediate level course introduces both complex instruction set and reduced instruction set processor architectures. The laboratories for this course consist of hardware and software simulations of basic programming constructs in CISC and RISC architectures. The third course focuses on advanced concepts in special purpose architectures to provide both depth and breadth to the subject matter. One could, conceivably, introduce security protocols for storage and system-on-a-chip related laboratories at this level.

***Integrating Computer Architecture with Security***

The focus of the intermediate-level course so far has been on implementing techniques of basic programming constructs such as I/O, arithmetic expressions, memory operations, register operations, if-else and switch conditional operations, for-while iterative computation controls, simple functions and recursive functions in several different instruction set processor architectures. This approach provides interfacing for cs-1 and cs-2 courses taken in the previous semesters. Increasing the performance of the processor by reducing program execution time is considered at the gate, register and functional levels of the processor design [3] [4]. At the end of the semester, students in this course design a pipeline processor using VHDL as their final project.

It is observed that learning processor architecture alone is insufficient at this level. Students should start to understand the importance of storage systems and applying classroom knowledge to solve real-life problems. A course sequence with extensions to security applications would help students develop such skills using several different architectures before their graduation. The following examples constitute a way to integrate real-life problem solving to this level of students. Example 1 illustrates a closed laboratory designed to help students understand the changes in the content of the memory. The open laboratory followed by this lab, a packet-sniffing example, would provide necessary interfacing with a network security course module. Example 3 shows an application of the knowledge acquired in logic-design class to provide interfacing with database security course module. It describes a simple logic extension used to break into database systems. Example 4 describes the application of the clock values in a legal issue. This could provide interface to secure operating system course module.

```
TUTOR 1.32> MS 2000 'ABCDEFGHIJKLMNOPWRSTUVWXYZ'

TUTOR  1.32> MS 2020 'abcdefghijklmnopwrstuvwxyz'

TUTOR  1.32> MS 2040 '0123456789'

MEMORY DISPLAY

TUTOR  1.32> MD 2000 256
002000    41 42 43 44 45 46 47 48  49 4A 4B 4C 4D 4E 4F 50  ABCDEFGHIJKLMNOP
002010    57 52 53 54 55 56 57 58  59 5A FF FF FF FF FF FF  WRSTUVWXYZ......
002020    61 62 63 64 65 66 67 68  69 6A 6B 6C 6D 6E 6F 70  abcdefghijklmnop
002030    77 72 73 74 75 76 77 78  79 7A FF FF FF FF FF FF  wrstuvwxyz......
002040    30 31 32 33 34 35 36 37  38 39 FF FF FF FF FF FF  0123456789......
....
002090    12 EB 00 13 12 EB 00 0E  12 FC 00 20 12 EA 00 02  .k...k...|. .j..
0020A0    12 EA 00 12 12 EA 00 03  12 EA 00 02 12 FC 00 21  .j...j...j...|.!
....
0020F0    FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
```

**Figure 1. Setting and Displaying the Content of Storage**

***Example-1***

To understand how the processor operates, students need to recognize the contents of registers and memory, learn the limitations of instruction sets and how the basic programming constructs are implemented in processor architecture. Figure 1 depicts the first classroom activity used to help students understand the addresses and content of the memory. In this example, memory is set by using MS and the content of the memory is displayed using MD. In addition, students also perform translations of arithmetic expressions, data transfers, if-else-for-while control and recursive functions as in-class activity. They perform traces of registers and memory as an in-class activity using Motorola 68000 instruction set architecture.

***Example -2***

The objective of this experiment is to apply the knowledge acquired in interpreting content of memory displays in the previous experiment into network security applications. Network traffic is easy to capture and analyze using the tools available in the web. Network protocol analyzers, such as Ethereal Packet Sniffer, can be used to accumulate both incoming and outgoing network data [11] [12]. Most packet analyzers assemble all the packets in a TCP conversation and represent the data using tcpdump format. Students were asked to capture the user-id and password of their own email accounts using a packet analyzer. Also, they were instructed about the legal issues involved in packet capturing. After doing this experiment, one student observed that rediff login is not secure. However, he noted that hotmail.com is not only using secure login, but also encrypts the traffic. Hence, students could not identify their user-Id, password or message in that communication. Moreover, they found that encryption will significantly increase network traffic by observing the amount of data captured. This experiment would be a good interfacing for a Network-security class. Figure 2 shows the memory display of the captured data.

```
00000000  50 4f 53 54 20 2f 63 67  69 2d 62 69 6e 2f 6c 6f POST /cg i-bin/lo
00000010  67 69 6e 2e 63 67 69 20  48 54 54 50 2f 31 2e 31 gin.cgi HTTP/1.1
00000020  0d 0a 41 63 63 65 70 74  3a 20 61 70 70 6c 69 63 ..Accept : applic
00000030  61 74 69 6f 6e 2f 76 6e  64 2e 6d 73 2d 65 78 63 ation/vn d.ms-exc
00000040  65 6c 2c 20 61 70 70 6c  69 63 61 74 69 6f 6e 2f el, appl ication/
00000050  6d 73 77 6f 72 64 2c 20  61 70 70 6c 69 63 61 74 msword,  applicat
```

**Figure 2(a) Packet Sniffer Output - www.rediff.com**

```
00000000  16 03 00 04 79 02 00 00  46 03 00 2f ed 29 44 2a ....y... F../.)D*
00000020  7a b2 b5 95 40 08 c3 74  ae 70 98 20 49 08 00 00 z...@..t .p. I...
00000030  82 32 61 be ad eb b1 27  ee 5e 93 e6 b3 1e ac 79 .2a....' .^.....y
00000040  7e 80 31 0b d2 2e b9 70  3b e5 55 b3 00 03 00 0b ~.1....p ;.U.....
00000050  00 03 5a 00 03 57 00 03  54 30 82 03 50 30 82 02 ..Z..W.. T0..P0..
00000060  bd a0 03 02 01 02 02 10  3c f4 4e cc 7b c3 e6 34 ........ <.N.{..4
00000070  b0 3f 2d 8e b8 78 41 27  30 0d 06 09 2a 86 48 86 .?-..xA' 0...*.H.
00000080  f7 0d 01 01 05 05 00 30  5f 31 0b 30 09 06 03 55 .......0 _1.0...U
```

**Figure 2 (b) Packet Sniffer Output - www.hotmail.com**

***Example -3***

In general, database systems track their own users before allowing the access. Access to database systems is controlled using a user-id and a password for legitimate users. However, some databases can be attacked without a valid user-id and password. Such attacks can be performed by applying the knowledge of Boolean logic expressions learned in an introductory digital-logic design class. In one such successful database intrusions, the attacker entered the system by converting user-id and password expressions combined with one AND operation into two arbitrary expressions combined with an AND operation followed by an OR operation with an expression that always evaluated to a True value [8][9].

***Example -4***

Often legal experts have difficulties interpreting the forensic data available in the memories. Utah vs. Payne presents an interesting application of clock values. In this case, the prosecution presented data to the courts without analyzing its content. The defense team accurately analyzed the data column that represented clock values to illustrate the time line of events. The defendant was found not guilty [10].

***Other Architecture-Security Applications***

In addition to the network-database-security experiments described above, it is possible to search, develop and integrate introductory experiments for intrusion detection, forensic analysis of storages, sanitizing storages, web security and network security. Providing hardware and software support, and specifying protocols to make the processor and storage secure can also be considered at any stage of the course sequence. The most time-consuming task in solving security issues, similar to the one described in the introductory part of this paper, will be the postmortem analysis. There is a need to perform forensic analysis of the data in the storage to determine the source of e-mail transmissions and decode other communications. A computer architecture course with security and privacy

related applications, such as the ones described above, and protocols to provide security and privacy to storage will enhance students' higher level skills: teamwork, analysis, synthesis and active participation in the classroom. These course modules will help students learn architectural concepts in an active learning environment, thus providing students an opportunity to function well in an increasingly competitive society in which security is highest priority.

***Difficulties***

Incorporating security and privacy related issues in many subject areas into the architecture course may overload the students and faculty. Selecting a series of projects that increase enthusiasm among a diverse body of students will not be an easy task. To overcome this difficulty, we plan to work with information security faculty to develop suitable laboratory experiments.

***Course Assessment***

Once developed, the course material can be evaluated by soliciting criticism from faculty and students. Student learning can be evaluated in many different ways. Background knowledge can be performed in the form of a simple questionnaire/worksheet that the students fill out prior to completing the lab assignments. Students will be asked to explain the concepts they learned. Recording experiences from laboratory assignments is an essential part of student work. Group-work evaluations will also be used to assess the course. The faculty and teaching assistants regularly observe the team work. There are opportunities to test course materials within a large university system that could possibly extend use to other faculty and students.

***Information Security Symposium***

Two computer security symposiums were organized at the end of the Spring 2003 semester to stimulate our students, computer science, information systems and engineering faculty in five neighboring states as well as businesses and industry. Invited speakers from West Point Military Academy, Carnegie Melon University, University of Idaho, the University of Minnesota, the University of Iowa, the University of Wisconsin and the University of North Dakota delivered lectures based on their work. The symposium was well attended by students, faculty and industry representatives. These symposiums helped our efforts to develop a curriculum emphasizing secure storages, forensics, network-database security that presents an integrated view of hardware, software and security issues to the undergraduate students [6][7].

## 3. Summary and Future Work

Traditionally, computer architecture courses are presented to a less-than-enthusiastic student body and often delivered without indicating real world applications in a relatively passive classroom environment. One of the reasons for diminished student interest of learning the subject is poor interfacing with other courses in the curriculum. In general, learning takes place if the student can both integrate what he is learning in the classroom into real-life applications and understand how the subject pertains to learning other subjects in the degree program. To promote this in the classroom and to overcome the above-mentioned deficiencies, an intermediate computer architecture course can be developed with hands-on classroom activities and laboratories involving architecture and security. Computer security issues are important for businesses, industries and government. There is a need for increased computer architecture education with security and privacy emphasis through undergraduate level computer science curricula. This need can be met through several stages. Computer architecture laboratories should have application interfaces to other courses in the curriculum such as operating systems, network, databse security. And further training of students can be accomplished by developing mathematics, operating systems, networking and database courses with an emphasis on security. Regular regional symposiums are very helpful to share the laboratory and curriculum development efforts with other colleges and universities. The availability of properly designed and developed course materials, with a series of hands-on laboratories as well as classroom activities, will reduce both instructors' preparation time and multiple copying stages, and increased students' ability to absorb the subject matter. Developing a series of system-on-chip experiments and/or security protocols for storages would be useful for the laboratories in the third course of the sequence. These objectives can be accomplished through the optimal use of available resources. Through such platforms, students will learn to appreciate instruction set architecture.

**Acknowledgments**

Our computer architecture project has been supported by the MnSCU Center for Teaching and Learning through the Bush/MnSCU Learning by Doing Program. Many students helped to make our extensions to the courses successful. We would like to specially thank to Professor Larry Grover, Dr. Splittgerber, Erik Cramer, Mark Ebersole, WeiShin, Mohammad Khan, Amit Parnerkar for helping us throughout the years to accumulate experimental data.

## 4. References

1. Gehringer, E.F. A Web-Based Computer Architecture Course Database,

http://www.csc.ncsu.edu/eos/users/e/efg/archdb/FIE/2000CACDPaper.pdf
2. Computer Architecture I http://web.stcloudstate.edu/jherath/CompArch-1
3. Hennessy, J. L., Patterson, D. A. Computer Organization and Design: Hardware/Software Interface, Second Edition, 1997 http://www.mkp.com
4. Computer Architecture II http://web.stcloudstate.edu/jherath/CompArch-2
5. Hardware/Software Interfacing for High Performance Symposium -02 http://web.stcloudstate.edu/jherath/Conference.htm
6. Symposium on Information Assurance and Security -2003 http://web.stcloudstate.edu/sherath/SIAS2003
7. Information and Network Security Workshop-2003 http://web.stcloudstate.edu/sherath/INSW2003
8. Pfleeger, Security in Computing, http://www.prenhall.com
9. http://www.cs.uwec.edu/~wagnerpj/security/
10. http://www.simson.net/2002-11-Forensics.ppt
11. http://winpcap.mirror.ethereal.com/install/default.htm
12. http://www.ethereal.com/distribution/win32/

# Combining Learning Strategies and Tools in a First Course in Computer Architecture

**Patricia J. Teller, Manuel Nieto, and Steve Roach**
The University of Texas at El Paso
Department of Computer Science
{pteller, manueln, sroach@cs.utep.edu}

**Abstract** When instructors learn about participatory learning strategies such as cooperative, active, problem-based, or team-based learning, often the reaction is something along the lines of "sounds great but it would not work in my class—with such an approach I could not cover the required volume of material." We are pleased to report, via this paper, that this is not necessarily the case. With some retraining on the part of the instructor, as well as the students, a significant initial investment on the part of the instructor, the incorporation of tools that support the learning process, and a CQI (continuous quality improvement) process in place, it does work.

## 1.0 Introduction

Undergraduate computer architecture courses have a large volume of material that students must learn. In addition, we want our students to develop written and oral communications skills, hone deductive reasoning and critical analysis skills, and be prepared to work in teams. To achieve these goals, it is necessary to employ a variety of learning strategies and tools, such as cooperative, active, problem-based, and team-based learning [2, 4, 1, 5, 9]. Although these strategies provide rich opportunities for a diverse student body to master the required course material and skills, they can be classroom-time intensive. This leads to challenges for both the instructor and the students. In general, the students must take on a more active role inside and outside the classroom and the instructor must carefully design the course, keeping in mind that the learning process is student-centric rather than instructor-centric.

Four significant challenges face an instructor when delivering a course using student-centric, team-based approaches.

1. When participatory learning strategies are employed, the quantity of material that can be covered during a class session is limited. Certainly, it is less than that which can be covered by a traditional lecture.
2. It is difficult to accurately assess individual mastery when assessing work products that are team generated.
3. Each student is different. It is necessary to adjust the course dynamically to meet the needs of the students in a particular class.
4. Some students need to be encouraged to utilize and develop critical thinking and analysis skills required in a student-centric course.

This paper focuses on ways to meet these challenges. After briefly outlining how cooperative, active, and problem-based, and team-based learning are used in a first course in computer architecture, it concentrates on two specific approaches that we found to be particularly effective, Readiness Assessment Tests (RATs) [5, 9] and problem-based discovery learning. The remainder of the paper is organized as follows. Section 2 briefly describes the course in terms of content and our general philosophy towards delivery. Section 3, after briefly describing the structure of the course, focuses on RATs, used to motivate students to prepare for course activities, and problem-based discovery learning techniques and the software and hardware tools used in the labs. Section 4 reports on the results of both informal and formal assessments of the course and the learning strategies and tools employed. And, finally, in Section 5, we conclude with our view of our accomplishments and our plans for the future.

## 2.0 Course Philosophy and Content

The subject course, which is taken after a course in digital systems, provides students with an introduction to computer architecture and prepares them for a second, more advanced course in computer architecture, which uses the Patterson and Hennessy textbook entitled "Computer Organization and Design: The Hardware/Software Interface" [10]. Given the two preceding courses, the intent is that the first three chapters of the Patterson and Hennessy textbook will be covered fairly quickly and mostly out of class.

The mantra of the instructors, in this course as well as in their other courses, is attributed to Albert Einstein: "Make things as simple as possible, but not simpler." During this course, students peel away the layers of abstraction starting with high-level concepts and ultimately visualizing the execution of a program and controlling a robot via an assembler language program. Along the way, the instructor guides the students to discovery, feeding student curiosity as students deduce answers to questions.

Students enrolled in the subject course learn the general method by which a computer executes a program. They gain an understanding of basic computer architecture and work with assembler language and machine code. Related concepts covered in the course include the mapping of high-level programming constructs to low-level constructs, assembly and disassembly of instructions (instruction formats, addressing modes, effective address calculation), interpretation of Boolean algebra descriptions of instruction functionality, stored program concept (including, introduction to linking and loading), subroutine linkage, introduction to text, data, and stack segments, fetch/decode/execute process, machine state, evaluation of execution time (in cycles and time), and identification of exceptional behavior, e.g., overflow. Additionally, students are introduced to I/O interfacing, interrupt handling, and robot control. And, they are reintroduced to basic concepts of software engineering (program design, implementation (modularization), testing, debugging, and documentation). The 68HC11 is used as a model processor architecture. Supporting materials include a textbook [6], a Motorola 68HC11 processor manual [7], and the software and hardware described in Sections 3.3.1 and 3.3.2.

The 68HC11 provides access to a 64KB address space (RAM) to store text, data, and stack, and four 16-bit and three 8-bit general-purpose registers. Memory-mapped I/O is used to access I/O ports, e.g., the analog-to-digital and serial communication ports. The ISA provides immediate, extended, direct, indexed, inherent, and relative addressing modes.

**3.0 Course Structure**

The course is scheduled for three hours of lecture and three hours of laboratory time per week. In-class sessions utilize a combination of learning techniques including cooperative, active, problem-based, and team-based learning [2, 4, 1, 5, 9]. After an introductory lecture (no longer than 10-12 minutes), students (formed into base groups, or sometimes groups formed in one of a variety of ways) are given a related task to accomplish, cooperatively, in a specified amount of time. While accomplishing the task, each group member adopts a different role, e.g., timekeeper, recorder, facilitator (ensuring participation by all group members), and questioner (ensuring that each group member understands and agrees with the group outcome). When the time expires, the instructor randomly selects a person from one group to describe her/his group's outcome. To add variety to class meetings, active learning [4], used on a one-on-one basis, also is employed.

Some of the virtues of cooperative learning (which is used in conjunction with problem-based and team-based learning) include honing of technical and communication skills via discussion and teaching, enhancement of critical analysis skills via discussion/debate, development of team skills, and growth of confidence (in particular, for students who feel more comfortable articulating the group's answer, rather than their own answer). Despite its benefits, when using cooperative learning, a facilitator must guide students' behavior (e.g., point out inappropriate behavior such as domination), and incorporate individual accountability so that all group members are held responsible for their own progress in the course.

In general, each major unit of the course is organized as a sequence similar to the following:

1. reading assignment (outside class)
2. RAT (in class, described next in Section 3.2)
3. lecture (in class, maximum 10-12 minutes)
4. in-class active/cooperative//problem-based/team-based learning using simple problems (described briefly above)
5. out-of-class problem solving using simple problems (cooperative and individual)
6. in-class active/cooperative//problem-based/team-based learning using more complex problems (described briefly above)
7. out-of-class problem solving using more complex problems (cooperative and individual)
8. simulator or robot lab (in lab, described in Section 3.3)
9. lecture (in class, maximum 10-12 minutes)
10. in-class active/cooperative/problem-based/team-based learning
11. assessment (for example, a quiz, which builds in individual accountability)

Depending on the targeted course material, steps 3 (optional), 4, and 5 and steps 6 and 7 may need to be repeated several times. Also, a lecture and/or problem-solving sessions may need to be added to address hurdles encountered during RATs, problem-solving sessions, labs, quizzes, and exams; similarly steps 9 and 10 are used for this purpose. Individual accountability is built in to the course via weekly quizzes and three examinations, which are taken individually, and a final project, which is described in Section 3.3. Out-of-class problems can be worked on in groups or individually; one strategy is to alternate between the two.

The basis for a final grade depends on the instructor. An example of one used in the course is RATs 15%, quizzes 10%, exams 40% (a minimum average of 65 is required to pass the course), labs 10% (a minimum average of 60 is required to pass the course), homework 0% (optional; answer sheets are provided but homework is not graded), programming assignments 15%, and final project 10%.

### 3.1 Student Responsibility and Discovery Learning

Two major hurdles to increased student involvement in class are the difficulty in covering the required material in the limited time available and the need for students to utilize and develop critical thinking and analysis skills. To address the first issue, students must come to class prepared. In a traditional lecture setting, students achieve in-depth understanding of the subject matter outside of the classroom. Lectures introduce topics, and homework reinforces them—whether applying the concepts or preparing for a quiz or exam. Using the approach described in this paper, we attempt to reverse the in-class and out-of-class roles: the students introduce themselves to the topics by reading and working simple problems outside of class; in-depth understanding comes from applying their knowledge to more complex problems in and outside the classroom and in the lab, thus, utilizing and developing critical thinking and analysis skills. In class, the instructor observes students and becomes aware of difficulties that they encounter—this allows for intervention when necessary. The 12-minute lecture motivates the subsequent problem solving that will take place subsequently, explains a concept at a deeper level than would not be possible without student preparation, and/or addresses issues that have proven (by observation in class or lab, or by performance on out-of-class problem solving, quizzes, or exams) to be difficult. Our techniques for motivating students to prepare for class activities are described in Section 3.2, while our strategies for encouraging critical thinking are exemplified in Section 3.3.

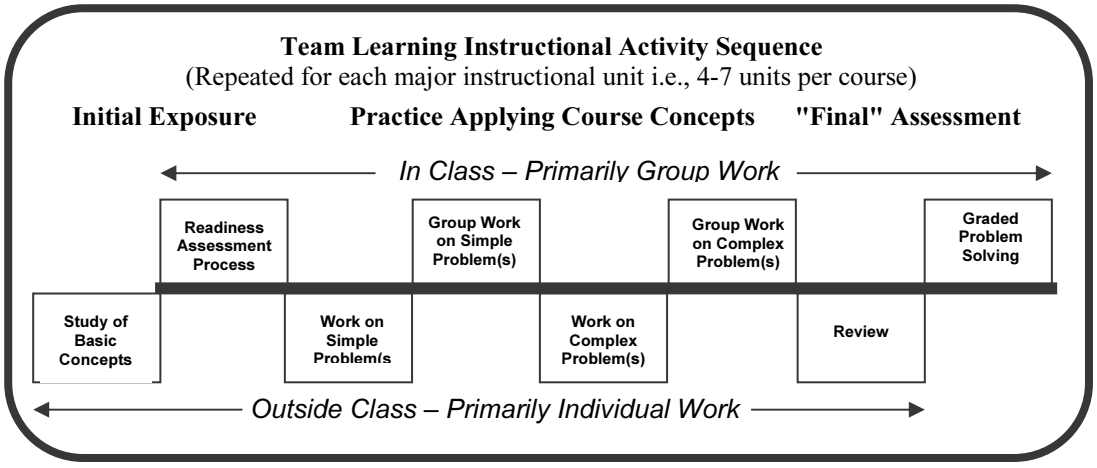


**Figure 1. Team learning instructional activity sequence.**

### 3.2 Motivating Outside Preparation

Given the best effort of an instructor with respect to designing an effective, participatory class session, it is doomed to failure without adequate student preparation. A technique that addresses this problem is RATs, Readiness Assessment Tests [5, 9]. Using team-based learning [5, 9], a major instructional unit is addressed using the activity sequence depicted in Figure 1, which includes RATs as part of the readiness assessment process and is very similar to the sequence adopted in the subject course. A RAT is a quiz, taken immediately after a reading assignment; it tests whether or not a student did (with a reasonable amount of depth) the assigned reading. The assigned reading, in turn, prepares students for a class session that is meant to enhance the learning accomplished via the reading. The questions are true/false and/or multiple-choice. A sample question might be "Overflow cannot occur when the numbers are not of the same sign. True or False?" An inappropriate question is one that requires the understanding of a key concept, for example, a multiple-choice question regarding how overflow is detected. Save these types of questions for quizzes!

First, a RAT is taken individually; students record their answers prior to handing in the RAT. Immediately thereafter, the same RAT is taken as a team (all RATs are taken by the same "base" team). (Here the term "team" is used, rather than "group", because the intent is that the conditions created during the life of the course will turn the base "group" into a "team".) After the first RAT, the student body comes to consensus on how much the individual and team RATs are to be weighed w.r.t. each student's grade. For example, the individual RAT score might be worth 40% of a student's score, while the team RAT is worth 60%. Of course, the instructor can set a threshold. Scantron tests make this whole process easier.

If a team does not agree with the grading of a question, they are permitted to submit a written appeal [5, 9], which the instructor evaluates. If the instructor agrees with the appeal, then only that team receives an adjusted score—this rewards critical analysis and effective communication. Of course, if the instructor's answer is incorrect, all teams' RATs are regarded.

According to our observations, RATs are effective in several ways. RAT scores were generally high; for example, in spring 2002 they were in the 80th percentile for more than 70% of the students. Discussions during team RATs, which are animated and passionate, indicate that the vast majority of students do, indeed, read the assigned material. Students seem to cover a sufficient amount of material on their own and are better prepared for collaborative, active, problem-based, and team-based learning. The taking of a RAT as a team incorporates many of the virtues of collaborative learning.

### 3.3 Supporting Discovery Learning

The laboratory sessions, which are based on those described in [11, 12], focus on problem-based discovery learning. This approach supports the learning process in several ways. It is used to introduce, explore, and strengthen understanding of new concepts. Simulator- and robot-based laboratory assignments (labs) guide learning via different levels of abstraction. The well-defined, proctored, cooperative learning labs meet twice a week. They are experimental labs, similar to labs in physics or chemistry, that are designed to lead students down a path of self-discovery, affording them opportunities to teach themselves and others via the scientific method. Students work in pairs using, for example, brain storming to decide on an approach to use to discover an answer or to understand the results of an experiment.

The goal of each lab is stated explicitly, and the lab is written in such a way that students are required to experiment, critically analyze experimental results, and use deductive reasoning to arrive at correct answers. Working in pairs, and sometimes in larger groups, reinforces their confidence in their understanding of jointly constructed hypotheses, experimental designs, and predictions, their analysis of experimental results, and their understanding of the concepts that underlie the lab. Of course, this also helps improve their communication and team skills as well.

This course uses two types of tools in labs: a simulator and robots. Simulator labs focus on basic machine organization and assembler language using the Visual 6811 simulator [8]. Robot labs focus on memory-mapped I/O, serial communication, interrupt handling, and I/O interfacing. A lab sheet, which describes the goals of the lab and what is expected of the students, is presented to students prior to a lab; the deliverable of each lab is either a completed lab sheet (that includes answers to questions, as described below) or a demo. Students unable to complete the lab during the allotted time, must finish it on their own time; this is a typical situation for robot labs, which anticipate complex problem solving being done outside of the labs. Simulator and robot labs, and the tools they employ, are described in the following sections.

#### 3.3.1 Simulator Labs and Visual 6811

Simulator labs are carried out in pairs by writing and executing assembler language programs, examining assembler listings, symbol tables, and cross-reference tables, and simulating the execution of an assembler language program using the Visual 6811 simulator, a simulator for the Motorola HC6811 microprocessor. Each simulator lab brings together different pairs of students. This permits students to get to know other students in the class and provides challenges with respect to communication and team skills.

A simulation permits the examination of register and memory contents, including the text, data, and stack segments, and execution time in cycles. Via simulator labs, students can analyze, among other things:

- representation of instructions (in machine code),
- initialized and uninitialized data,
- instruction length and instruction format differences attributable to addressing modes,
- effects of instruction execution,
- effective address formation,
- program control flow,
- behavior of subroutine calls and return instructions, and their effect on the runtime stack,
- composition of activation records,
- differences between call-by-value and call-by-reference semantics,
- contents of interrupts vectors, and
- algorithmic complexity in terms of execution time.

Visual 6811 is composed of two independent subsystems that work together to provide the functionality to debug a program and to simulate the execution environment of a program written in the 68HC11 assembler language. The two subsystems are a simulation engine, which is composed of a library of functions, and a GUI, which controls the simulation of the program.

The simulation engine can simulate the complete 68HC11 ISA. (Note, however, that some underlying subsystems such as the asynchronous serial communication interface and interrupts are not yet implemented.) Simulation is at the instruction level, as opposed to the cycle level. The library provides access via the GUI to fetch and modify simulated components such as registers, memory, elapsed cycles, etc.

The GUI controls the execution of a simulation, displays changes to the execution environment of a program as it executes, and provides debugging functionality. The GUI subsystem is platform independent; Visual 6811 can run on any system

running the Windows, Unix, or Linux operating systems and that supports the Java Runtime Environment (JRE). Figure 2 shows a screen shot of Visual 6811 while simulating a program. The screen shot shows a breakpoint that is set (instruction highlighted in red) and the next instruction to be simulated (instruction highlighted in cyan). After a program is assembled, the GUI displays a program listing (see the area of Figure 2 denoted by the red circle marked "1", i.e., section 1 of Figure 2).

The assembler listing is composed of the program's assembler instructions and the corresponding machine code.

third indicate, in hexadecimal format and ASCII representation, respectively, the contents of the next 16 bytes of memory starting at that block's memory address. Section 5 is included as an aid to see how the top of the stack changes as a program executes and so that both the text and data segments can be viewed at the same time. Instead of displaying the stack in its usual abstract notation (a stack of values with the most recent value stored at the top and the oldest value stored at the bottom), the stack is displayed in the same format as memory to solidify the notion that the stack is just another area of memory.

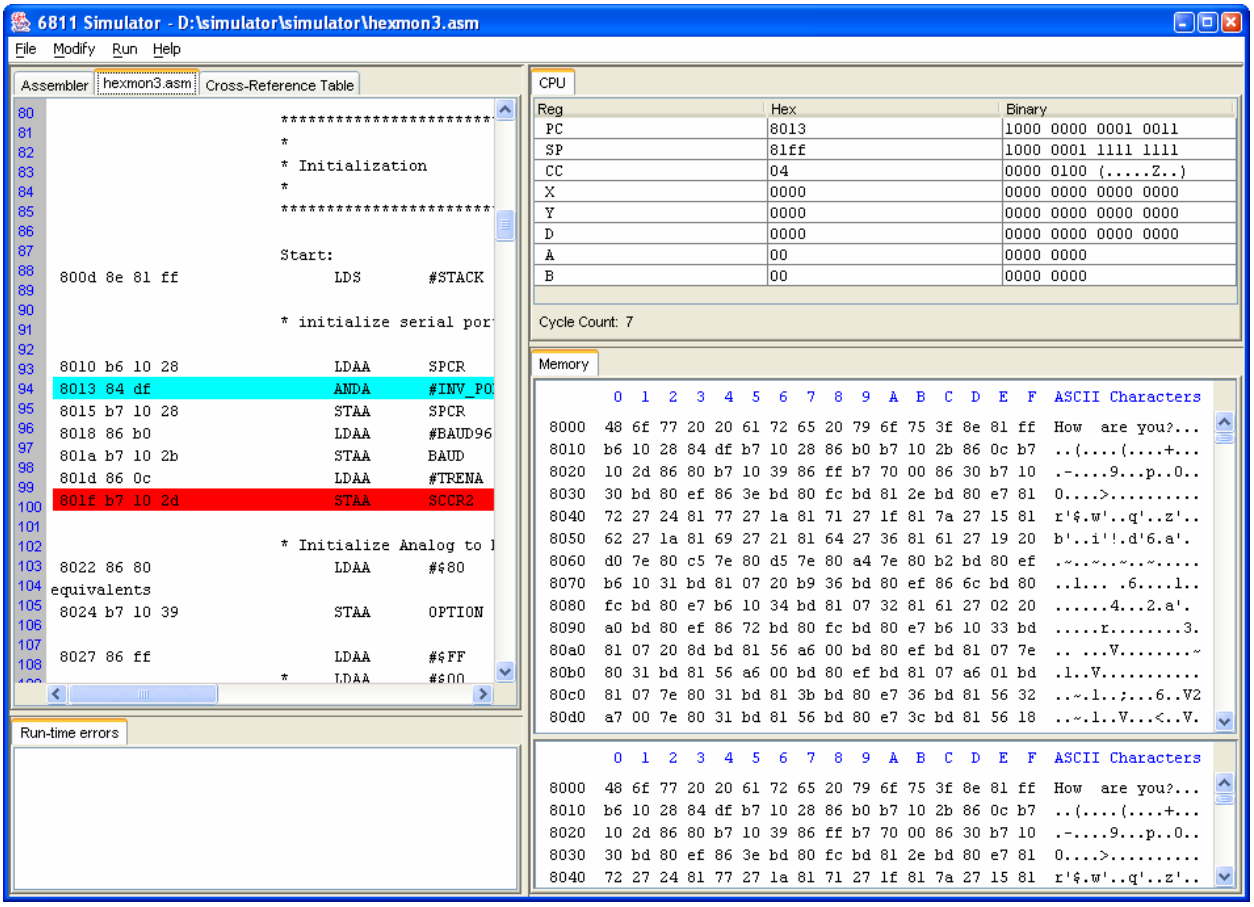


**Figure 2. Visual 6811 while simulating a program.**

Section 2 of Figure 2 shows a tab to activate the cross-reference table. Section 3 displays run-time errors that occur while a program is simulated. An error is generated, for instance, if a bad opcode is fetched. Section 4 displays the names of registers and their contents (in both hexadecimal and binary).

As a program is simulated, register contents are updated in response to the (simulated) execution of instructions. Sections 5 and 6 display the contents of memory. The memory is laid out to show 16-byte memory blocks. Each block is divided into three parts: the first identifies the block's starting memory address and the second and

Visual 6811 provides the following features to debug and control the simulation of a program:

- *assemble a program*: Assemble the selected program and display the assembler listing. After this step, the program can be simulated.
- *reassemble last assembled program*: Reassemble the last file that was assembled (successfully or not).
- *modify memory contents*: Modify the contents of memory by entering a new value to store at a specified memory address.

- *modify register contents*: Modify the contents of any of the simulated general-purpose registers. This and the previous option make it easy to modify the execution environment of a program while it is simulated and quickly see the execution behavior that results from these changes.
- *step through the execution of a program*: Simulate a program one instruction at a time, permitting the observation of changes in the execution environment as a result of instruction execution.
- *run a program to completion*: Execute a program to completion, as opposed to one instruction at a time. This is most useful when breakpoints are set.
- *toggle breakpoints*: Set or clear breakpoints in a program. This is useful when debugging large programs and when analyzing specific code sections.
- *stop a simulation*: Interrupt a simulation that is running to completion. This is useful, for instance, if a program has entered in an infinite loop, so the only way to get out of the loop is by stopping the simulation.
- *reset execution of a program*. Restart a simulation by loading into memory the machine code of the last program assembled.

### 3.3.2 Robot Labs

After acquiring some competence at programming the 68HC11, students are challenged to control small robots that are controlled by 68HC11s. The TJPro robots [3], depicted in Figure 3, have infrared (IR) emitters and detectors, and bumper sensors, expanding the possibilities for programming the different interfaces, such as the analog to digital interfaces. Robot labs require students to download programs, via the serial communication interface, to a 68HC11 microprocessor. These programs interface with the 68HC11's I/O ports and permit students to analyze the behavior of, among other things:

- memory-mapped I/O,
- the serial communication interface,
- motors connected to the I/O ports,
- digital and analog sensors,
- programmable timers and counters, and
- interrupts.

The robot labs allow students, paired into fixed teams of two (which are formed based on students' input re: their top six choices), to leap from abstract concepts and simulations to a more hands-on experience. In addition, they permit students to become involved in fun and challenging projects. For instance, the first robot lab has students use the serial communication interface to display the contents of the robot's memory (the 68HC11) on the monitor of the host PC. Pressing a key on the host's keyboard, which instructs the 68HC11 to read from or write to a memory byte or word, drives the related program.

Final projects have been very successful in challenging students to incorporate in a program all the knowledge they have acquired throughout the semester. The first final project had students program robots to navigate a maze. To make this more challenging, not only did the robots have to find their way out of the maze, they had to remember the path that they followed so that when the robots were next placed at the beginning of the maze, they navigated the maze without bumping into walls. Another challenging final project was to program two robots equipped with IR detectors: a *wimp* and a *follower*. The *follower*'s goal was to detect the *wimp* and tap it. The *wimp*'s goal was to avoid being detected by the *follower* and exit the arena before getting tapped by the *follower*.

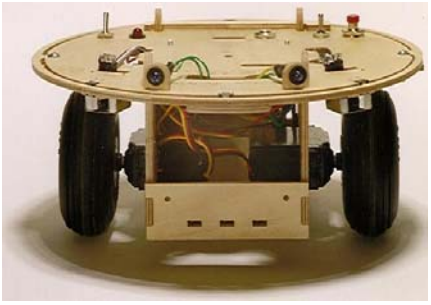

**Figure 3. TJPro robot.**

Individual accountability is achieved by defining projects that (1) are comprised of multiple interfacing functional components, which when complete meet the project specifications, and (2) have multiple subgoals that can be achieved by a subset of complete functional components. In this way, if one team member does not do her/his part, the other team member does not get penalized in so far as the project grade is concerned.

Besides all the benefits attributable to collaborative, problem-based discovery learning, the robot labs require students to operate at a higher level, especially with respect to debugging. Considering debugging techniques as a continuum ranging from random (uneducated, uninformed), to brute force (uneducated, informed), to guessing (educated, uniformed), and finally to experimental (educated, informed), the complexity and difficulty of robot-based labs make the first three forms of debugging ineffective. This leaves the students with only one viable alternative: an experimental, analytical approach. As a result, the student must

- be clever about creating a hypothesis,
- be resourceful in testing hypotheses,
- discuss hypotheses in terms of observed behavior, not the underlying computer architecture, and

- discuss solutions in terms of the underlying computer architecture.

Just what we want!

**4.0 Assessment**

The strategies and tools used in this course have been assessed both informally and formally. The informal assessment concentrates more on the learning strategies employed in the course, while the formal assessment focuses on Visual 6811, the simulator used in the course. In addition, student evaluations from fall 2001 and spring 2002 indicate that the course was well received.

A 14-question informal assessment tool was used the second time the course was taught in this way (in spring 2002) to assess the effectiveness of the learning strategies and the simulator and robot labs employed in the course; the course has been taught this way twice a year since fall 2001. In general, the students (27 in number) were most positive about discovery learning via the simulator and robots. Opinions were divided with respect to working in groups during class and labs. On the positive side was the opportunity to share ideas, opinions, and reach some conclusions; on the negative side was the frustration experienced as a result of unprepared group members, who might affect their grades. Weekly quizzes were strongly favored—students stated that quizzes help them study for quizzes and exams (especially when they include questions that focus on the same concepts), and keep concepts they learn fresh in their minds. On the other hand, some students thought quizzes increased stress. The majority of students thought it was a good idea to use the robots in the course. Programming the robots helped to reinforce the concepts they learned and to see (physically) the power of the 68HC11 architecture in action. Some interesting student comments follow.

With respect to the simulator:

- "[The simulator] helped me achieve knowledge about the material covered in the class."
- "[It] helped me to understand what was happening."
- "We could test and see what happened for ourselves."
- "[It] helped me learn how to assemble and disassemble."
- "[It] helped me with programming."

With respect to the robots:

- [The introduction of the robots] was a good idea."
- "[They] enforced the concepts learned in class."
- "[They] gave me confidence that I can do something that seems impossible at first."
- "[They] were fun!"
- "[They] made it possible to see the power of the 68HC11 in action."

Recently a formal assessment, which focuses on Visual 6811, was conducted. The assessment is based on the response of 25 students, who either just took the course this semester (spring 2003) or who took it in the past two years. The assessment indicates the following:

- The simulator was straightforward to use and had a very low learning curve (23 students, i.e., 23/25).
- Being able to view the assembler source and the virtual representation of memory was helpful in understanding how memory changes as a program is executed (19/25).
- The simulator was particularly helpful for understanding stack manipulations and the differences among the various addressing modes (22/25).
- The simulator facilitated debugging by permitting the user to step through program execution and, while doing so, observe how memory and registers changed (18/25).
- Given the choice of using the simulator or not, students said they would have chosen to use the simulator since it helped them to debug and understand the execution of a program (21/25).

In fall 2001/spring 2002, of 24/19 responses to a student evaluation, out of a 1-5 rating, where 5 is excellent:

- The varied use of questions, discussions, lectures, and/or group work in the class was rated 4.8/4.9 (average rating).
- The relevance of course materials to state course objectives was 4.4/4.8.
- The relevance of class assignments was 4.5/4.7.
- The estimation of how much learned in the course was 4.4/4.6.
- The effectiveness of the course in challenging the student intellectually was 4.7/4.7.
- The overall rating of the course was 4.5/ 4.7.

**5.0 Conclusions and Future Work**

More powerful assessment is needed in order to quantify the success of the learning strategies and tools used in the course. But if student evaluations, student comments, and instructor observations are considered, the course indeed has been effective. In our opinion, the most powerful of the learning strategies employed in the course are the RATs and the cooperative, problem-based discovery labs (both simulator and robot). In addition, it cannot be denied that collaborative, active, problem-based, and team-based learning, i.e., learning that actively involves students, is far superior to lecture.

The next step in the development of the course is to extend the capabilities of Visual 6811 to simulate both the functionality of the serial communications interface and interrupt handling. In the very long run and with sufficient interest, a textbook that incorporates both the technical information conveyed via this course and the strategies and tools used as the vehicles of conveyance is envisioned.

**6.0 References**


[1] Davis, B. *Tools for Teaching*, San Francisco: Jossey-Bass, 1993.

[2] Johnson, D. W., R. T. Johnson, and E. J. Holubec, *Cooperation in the Classroom*, Edina, MN: Interaction Book Company, 1992.

[3] http://www.mekatronix.com/

[4] Meyers, C., and T. B. Jones, *Promoting Active Learning*, San Francisco, Jossey-Bass Publishers, 1993.

[5] Michaelsen, L. K., and R. H. Black, "Building Learning Teams: The Key to Harnessing the Power of Small Groups in Higher Education," in *Collaborative Learning: A Sourcebook for Higher Education*, Vol. 2, S. Kadel and J. Keehner (Eds.), State College PA: National Center for Teaching, Learning, and Assessment, pp. 65-81, 1994.

[6] Miller, G. H., *Microcomputer Engineering*, Upper Saddle River, NJ: Prentice Hall, 1999.

[7] Motorola M68HC11 Reference Manual, 2001.

[8] Nieto, Manuel, *Visual 6811: A GUI for Simulation of the Motorola 68HC11 Microprocessor Architecture*, Master's thesis, University of Texas at El Paso, Department of Computer Science, 2003.

[9] www.ou.edu/idp/teamlearning/

[10] Patterson. D. A., and J. L. Hennessy, *Computer Organization and Design: The Hardware /Software Interface*, San Francisco, CA: Morgan Kaufman Publishers, 1997.

[11] Teller, P., "Experimental, Cooperative Labs in a First Course in Computer Architecture," *Proceedings of the 1997 Frontiers in Education Conference (FIE '97)*, Pittsburgh, PA, CD-ROM, November 1997.

[12] Teller, P., and T. Dunning, "Mobil Robots Teach Machine Programming and Organization," *Proceedings of Supercomputing `95*, San Diego, CA, CD-ROM, December 1995.

# Building Resources for Teaching Computer Architecture Through Electronic Peer Review

Edward F. Gehringer
Depts. of ECE and Computer Science
North Carolina State University
efg@ncsu.edu

## 1. Abstract

Electronic peer review is a concept that allows students to get much more feedback on their work than they normally do in a classroom setting. Students submit assignments to the system, which presents them to other students for review. Reviewer and author then communicate over a shared Web page, and the author has a chance to submit revised versions in response to reviewer comments. At the end of the period, the reviewer gives the author a grade. Each author gets reviews from several reviewers, whose grades are averaged. At the end of the review period, there is a final round when students grade each other's *reviews*. Their grade is determined by the quality of both their submitted work and their reviewing.

This paper reports on our use of peer review in two computer architecture courses, a microarchitecture course and a parallel-architecture course. Students in these courses engaged in a variety of peer-reviewed tasks: Writing survey papers on an aspect of computer architecture, making up homework problems over the material covered in class, creating machine-scorable questions on topics covered during the semester, animating and improving graphics in the lecture presentations, and annotating the lecture notes by inserting hyperlinks to other Web documents. Students generally found these exercises beneficial to their learning experience, and they have provided resources that can be used to improve the course. In fact, with such a system, large classes are actually a blessing, since they produce better and more copious educational materials to be used in subsequent semesters.

## 2. Peer Review in the Classroom

Peer review is a concept that has served the academic community well for several generations. Thus, it is not surprising that it has found its way into the classroom. Dozens of studies report on different aspects of peer review, peer assessment, and peer grading in an academic setting. A comprehensive survey can be found in Topp 98. Experiments with peer assessment of writing go back more than 25 years [4]. Peer review has been used in a wide variety of disciplines, among them accounting [8], engineering [7, 10], mathematics [3], and mathematics education [6].

However, *electronic* peer review experiments have been much rarer. Although the Daedalus Integrated Writing Environment [1] is widely used for peer assessment of student writing, only a few computer-mediated peer-review experiments have taken place in other fields. An early project in computer-science and nursing education was MUCH (Many Using and Creating Hypermedia) [9, 11]. The earliest reported software program to support peer evaluation was evidently created at the University of Portsmouth [12]. The software provided organizational and record-keeping functions, randomly allocating students to peer assessors, allowing peer assessors and instructors to enter grades, integrating peer- and staff-assessed grades, and generating feedback for students. One of the early Web-based peer-review experiments was described by Downing and Brown [2]. Their psychology students collaborated to create hypertexts which were published in draft on the World Wide Web and peer reviewed via e-mail. Our project was one of the first to use the Web for both submission and review of student work.

## 3. Peer Review on the Web

There is much to recommend a Web-based approach to peer review. Unlike software that is written for a specific academic field (e.g., English composition), a Web-based application can accept submissions in practically any format, including diagrams, still pictures, interactive demonstrations, music, or video clips. Of course, the student has to understand how to produce such a submission, but for each field, that expertise tends to "come with the territory."

Secondly, the Web is a familiar interface. Most students use the Web in their day-to-day studies, so they can pick up a Web-based application for peer review with minimal effort. In addition, many if not most students are already familiar with tools for producing Web pages; for example, almost all wordprocessors can save files in HTML format.

Thirdly, Web creation skills are of increasing importance in business as well as academia. In producing work for Web-based peer review, students not only learn about the subject of their submission, but also gain valuable experience with software they will use in their later studies and on the job.

Fourthly, a Web interface enables the peer-review program to be used in distance education, which is an important and rapidly growing segment of the education market. On-campus students can review distance-education students, and vice versa, bringing the two groups closer together in their educational experience. With Web-based submission, there is no extra overhead for the instructor or TAs in handling distance-education students.

Finally, Web-based peer review facilitates the production of Web-based resources. The best peer-reviewed work can be turned into materials to help future classes learn. For example, students can create machine-scorable questions for each lecture, with different sets of students choosing different lectures. The best questions on each lecture can be incorporated into daily quizzes delivered via a Web-based testing system such as LON-CAPA [15], Mallard [16], or WebAssign [17].

Or, students can write research papers on various topics assigned by the instructor (e.g., the branch-prediction strategy of a particular processor architecture). The best paper on each topic can then be presented to the next semester's students as background reading on that topic. The writers can be asked to include liberal doses of hyperlinks in their papers, so that later students can read not only their work, but also the analyses of experts.

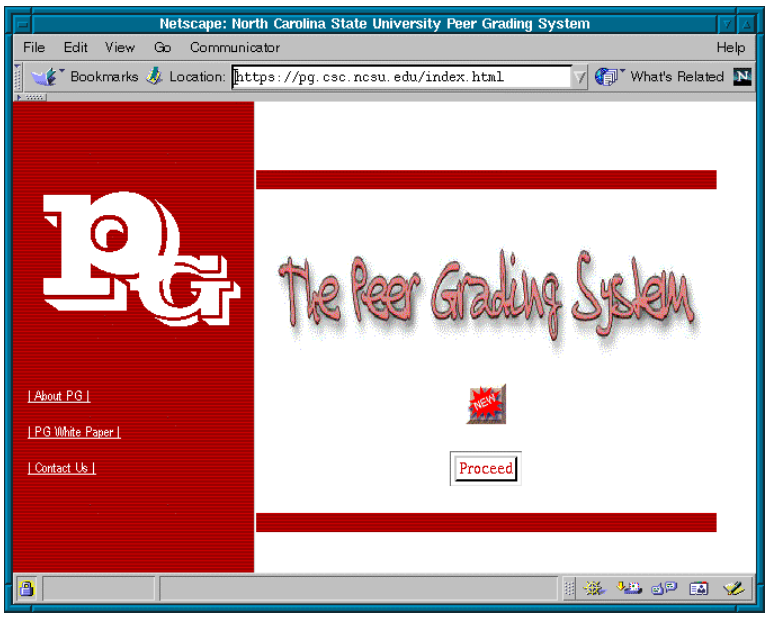


**Figure 1.** PG's welcome page

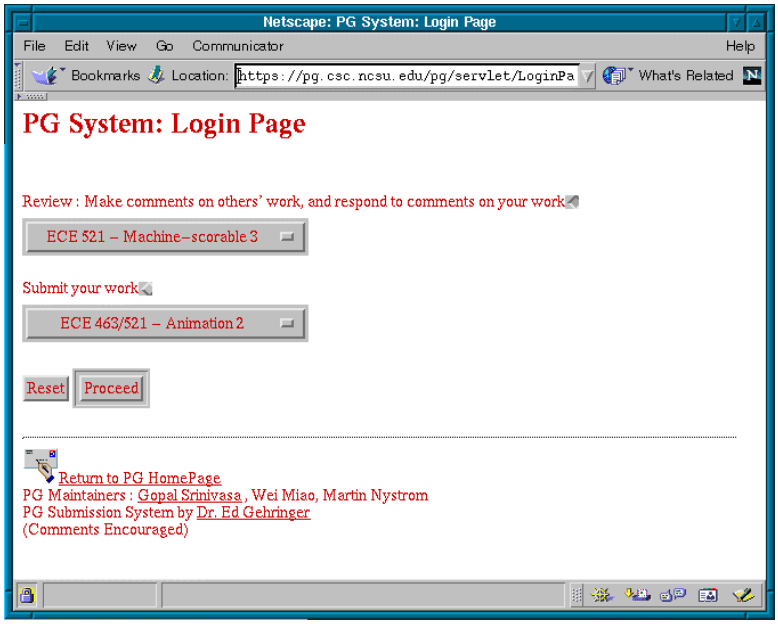


**Figure 2.** PG's login page

## 4. The PG System

PG (Figure 1) is a Web-based application for peer review and grading. It is written in Java and is servlet based. Students submit their work over the Web.

Reviewers can be assigned pseudo-randomly by PG, or by the instructor, using a spreadsheet. The number of reviewers is arbitrary, but usually three or four students are assigned to review each submission. Reviewers and authors communicate double-blindly via a shared Web page. At the end of the review process, the reviewer assigns a grade to each author whose work (s)he has reviewed. A student's grade is the average of the grades given by the reviewers, plus an incentive

described below to encourage careful reviews.

A student entering the PG system (Figure 2) has a choice of whether to submit a new page or review pages submitted by others. If more than one Web page is to be submitted, they may be submitted sequentially, each with a different filename, or submitted in a single Zip or tar file, which PG will unpack into its components. Entire directory hierarchies may be submitted in this manner. Since the files themselves are copied, all work to be reviewed will have a URL beginning with the pathname of the PG system, not the submitter. This ensures that the reviewers will not be able to guess their authors' identities by dissecting the URL. The ability to submit directory hierarchies allows large projects to be submitted as easily as small ones.

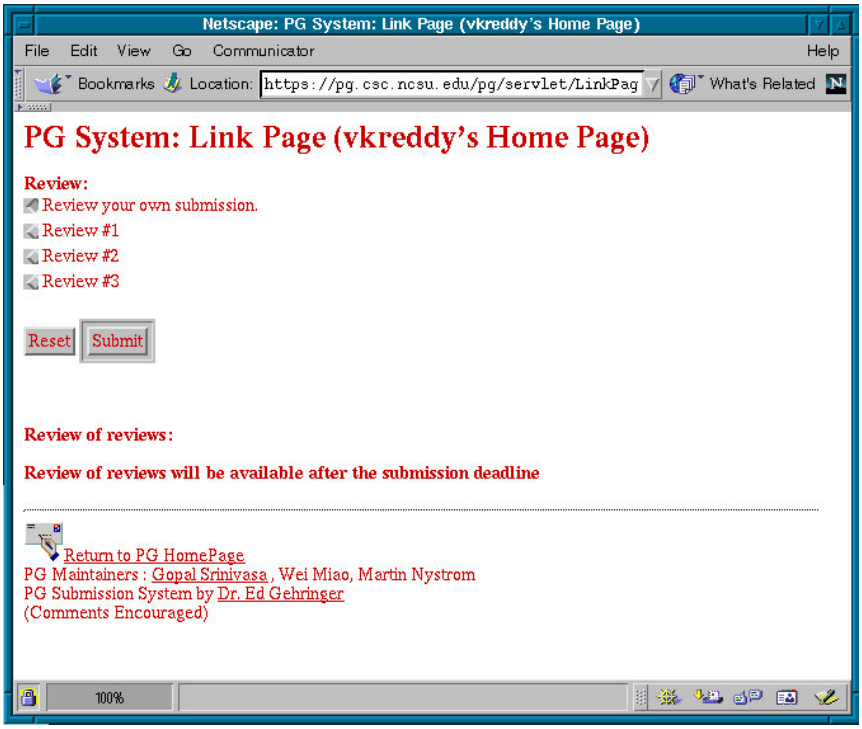


**Figure 3.** Page with links to submissions to be reviewed

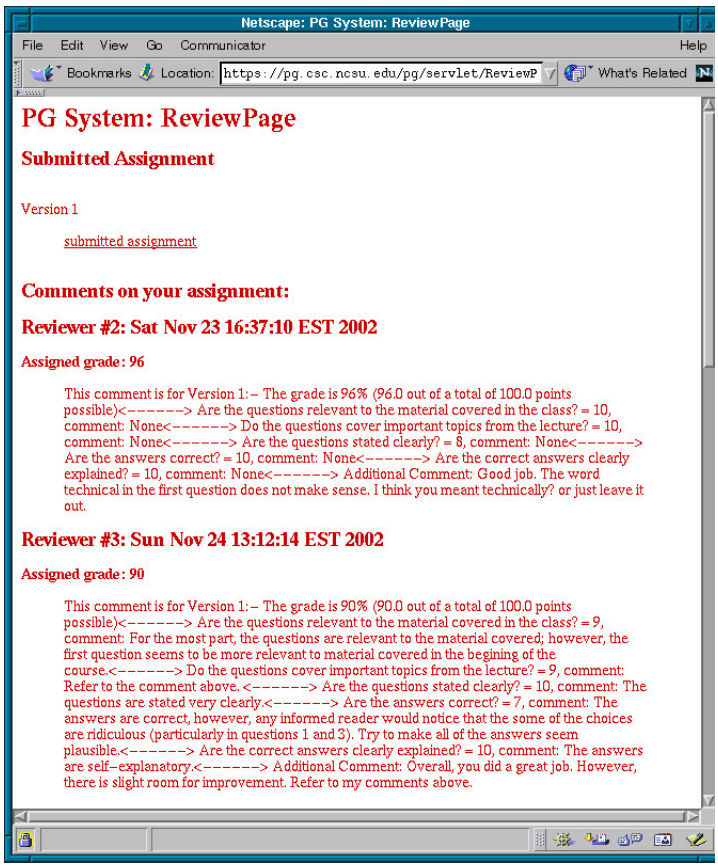


**Figure 4.** Review page

Reviewers communicate with their authors via a shared Web page. There is one such page for each author (Figure 3); the author can view the reviewers' comments and vice versa. The instructor can configure the system either to allow (Figure 4) or not to allow reviewers to see the other reviewers' comments and assigned grades. There are reasons in support of both strategies. Allowing reviewers to see each other's feedback provokes better dialogue over the quality of a submission, but the first reviewer's comments may unfairly influence subsequent reviewers' assessments.

Grading is based on a *rubric* consisting of several questions that the reviewer must answer with a numeric score. The questions may be assigned different weights, if desired. The grade that a particular reviewer gives a student is calculated by summing the product of each score with the corresponding question weight. A rubric-oriented approach is used to insure that all students are graded on the same criteria, and to reduce the chance that a reviewer will give an unrealistically high

grade due to ignoring some of the criteria that the submission is supposed to meet. In addition to giving numeric scores, the reviewer has ample opportunity to give feedback to the student on how to improve. This can be seen in Figure 5.

## 5. The Submit-Review-Publish Cycle

Our experience with PG has led us to a four- to six-phase cycle, capable of producing high-quality peer-reviewed work suitable for Web publication.

1. The *signup* phase (optional): If not all students are to do the same assignment, the students are given a list of potential topics (relating to research, or to a particular lecture, etc.) and sign up for one of them. To assure that all topics are chosen, only a limited number of students is allowed to sign up for any particular topic.
2. The *submit* phase. Students prepare their work and submit it to PG.
3. The *initial feedback* phase. Students are given a certain period of time—usually 3 to 7 days—to make initial comments on all the work This phase was instituted after students complained that their reviewers often did not comment on their work until it was too late to revise it. Reviewers may assign a grade during this period, but they are not required to do so.
4. The *grading* phase. During the next period—again usually 3 to 7 days—students can revise their work in response to reviewers' comments, and reviewers can comment on the revisions. At the end of this give-and-take, reviewers are required to assign a grade. This grade is one component of the author's final grade for the assignment.
5. The *review of review* phase. After the review period is over, each student is presented with a set of reviews to assess. The students grade each *review* based on whether it was a careful and helpful review of the submission. The grades the students receive on their *reviewing* is then factored into their grade for the assignment (usually 25% of their grade is based on their reviewing). This phase was instituted

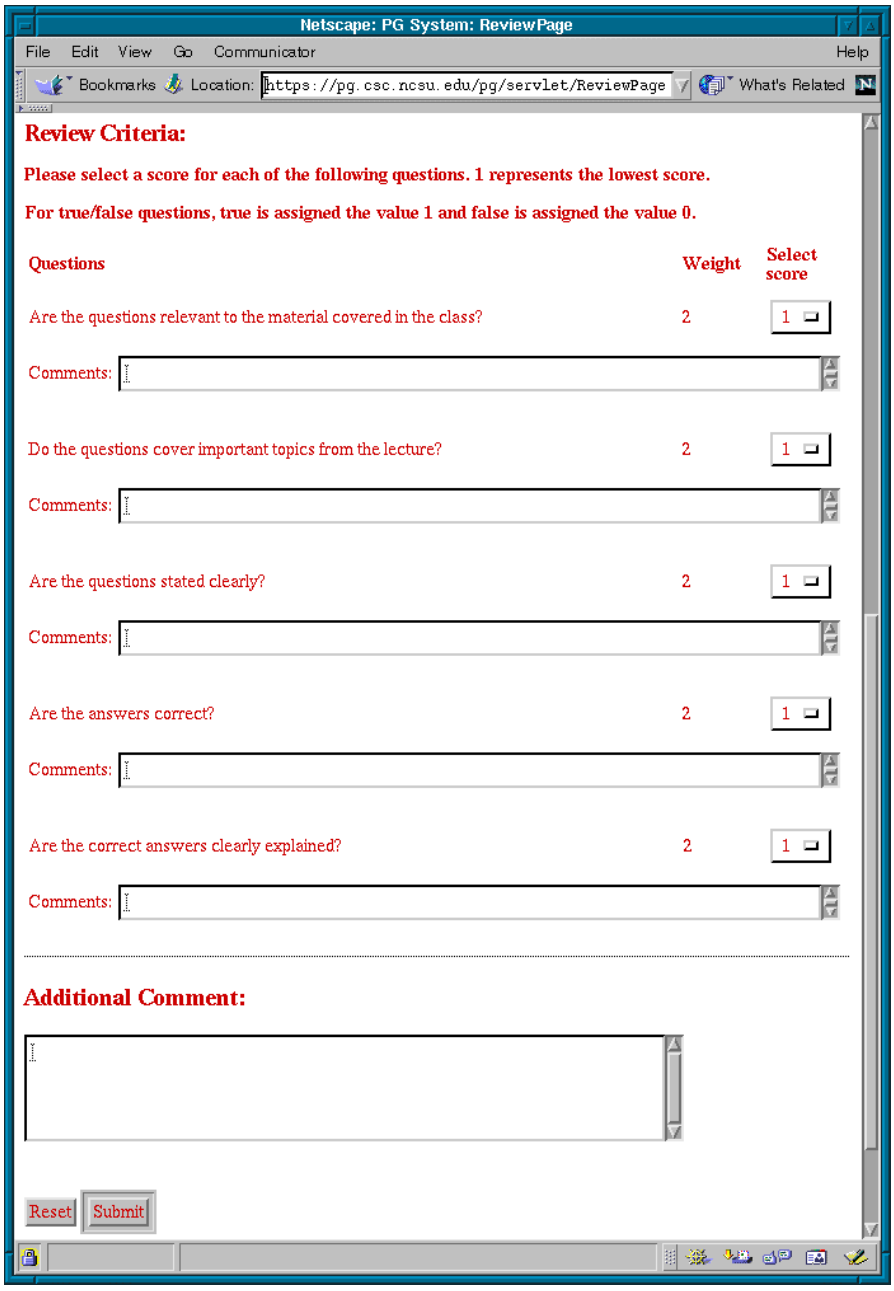


**Figure 5.** Grading rubric

after it was discovered that many students were doing cursory reviews. As will be seen in Section 7, this is a sufficient incentive to be careful in reviewing.

6. The *Web publishing* phase (optional). PG creates a Web page with links to the best student assignment in each category. As described below, this can serve as a useful study tool for future generations of students.

## 6. How Peer Review Has Been Used in Computer-Architecture Classes

There are opportunities to use peer review in almost any course. One of the best opportunities is in evaluating student writing. Prospective employers and thesis advisors widely believe that technical students need frequent opportunities to hone their writing skills. But students need ample feedback in order to improve. Peer review can give more copious feedback than instructor or teaching-assistant review, for the simple reason that each student has only a few submissions to review, rather than several dozen. Moreover, students will be writing for an audience of their peers later in their careers, so it is important for them to learn how to do this.

In computer-architecture courses, I have assigned students to write **reviews of papers** from the technical literature. I always assign two or three related papers so that the students cannot simply summarize a paper, but must instead integrate material learned from different sources. For example, in my microarchitecture course (using the Hennessy-Patterson text *Computer Architecture: A Quantitative Approach*), I assigned these papers on power-aware architectures:

> "Energy-effective issue logic," Daniele Folegnani and Antonio Gonzalez, *28th International Symposium on Computer Architecture*, July 2001, pp. 230-239.
>
> "Drowsy caches: simple techniques for reducing leakage power," Krisztian Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge *29th International Symposium on Computer Architecture*, May 2002, pp. 148-157.

I also have students do **annotations** of my lecture notes, which are on line as PowerPoint or Word files [13]. Each student signs up to annotate one of the lectures during the semester. Depending on how many students there are in the course, two to four annotations of each lecture are produced. This consists of inserting hyperlinks to other Web pages that define the term or describe the topic I am covering. Typically the students insert several dozen hyperlinks in each 75-minute lecture. The best annotation of each lecture (the one with the highest grade) is then made available to students in the next semester. In this way, students in one semester produce a resource that helps students in subsequent semesters to fill in gaps in their understanding of the material.

An excellent way to improve students' understanding of the material is to have them make up questions over what they have studied. I have assigned two different kinds of peer-reviewed questions. The first is **madeup homework problems**. Students are asked to make up a problem similar to those on the problem sets I assign for homework (typically these are problems from the textbook or similar problems). The students then peer-review each other's problems. Students learn by checking each other's work, and the problems they make up are often good enough to be used for subsequent homeworks and exams in the course. For example, in the last three times I've taught my parallel architecture course, I've used 27 problems that were made up by students in previous semesters. Given the fact that most instructors say [14] it is either important or very important to increase their supply of questions beyond what they now have, the usefulness of this approach cannot be denied.

Students' comprehension of lectures can be improved if they are asked a set of questions about the lecture after viewing it. In recent years it has become possible to pose questions and score student answers via a Web assessment and testing system like LON-CAPA [15], Mallard [16], or WebAssign [17]. It would be a major time commitment for the instructor personally to write a set of questions on each lecture, but peer review makes it possible for the students to write the questions themselves. Moreover, these questions come already "pretested" by a small set of students—the peer evaluators. Beginning in Fall 2002, I had students write a set of **machine-scorable questions** over the material in a specific lecture. This produced a "bank" of questions that can be used to create daily quizzes for students in later semesters. Ultimately, they could become a resource for a Web-enhanced version of the textbook we are using.

Computer architecture is a rather visual subject—one's comprehension is often improved by seeing a picture, or a graphical simulation, of a topic or an algorithm. Cache coherence and instruction-level parallelism are examples of such topics. Since some students are gifted in visual arts, I have allowed students to choose an **animation** as one of their peer-reviewed assignments. The best of their animations can then be incorporated into future lectures.

Peer review can be used for **research papers**. Though I have not yet assigned this in a computer-architecture course, in my operating-systems course, I had each student select a research topic from a set that included topics like "Scheduling in Windows NT," "Deadlock handling in Unix or a particular flavor of Unix," and "Virtual memory in Linux." Similar topics in architecture would be the cache-coherence algorithm, branch predictor, or instruction-retirement approach used by a particular architecture.

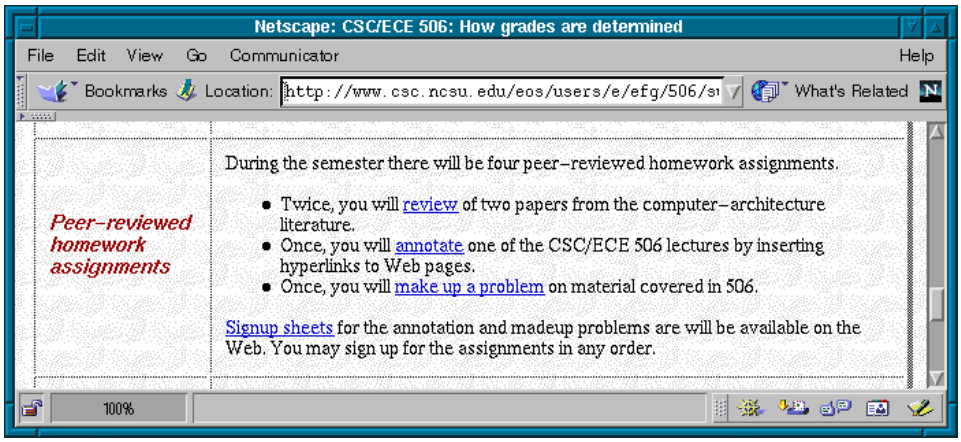


**Figure 6.** Peer-reviewed assignments in parallel-architecture class

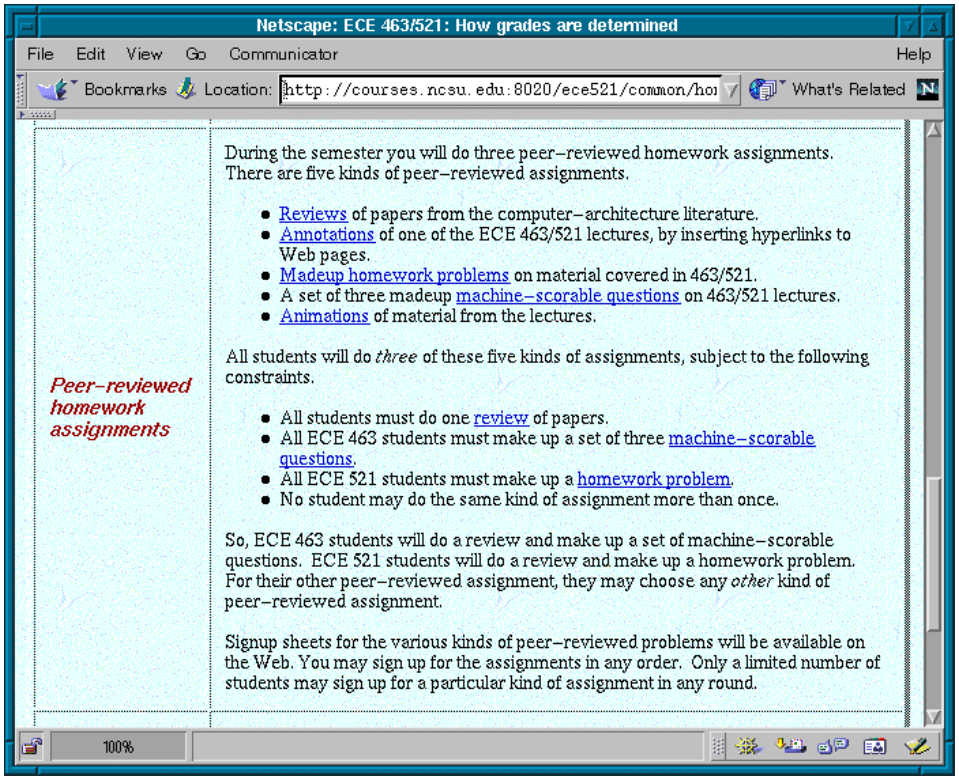


**Figure 7.** Peer-reviewed assignments in microarchitecture class

Assuming that students have the requisite computer skills, electronic peer review is as widely applicable as peer review in general. The author has previously reported on its use in computer science [18] and ethics in computing [19] courses.

Through peer review, each class can stand on the shoulders of previous classes, learning the material with better resources, and producing ever-better tools to teach future classes. In some cases, instead of seeing large classes as a burden, an instructor may come to prefer them because they can create more formidable Web-based resources, and do so without burdening the instructor and with additional grading responsibility. This is an example of "education engineering" [20]—developing methodologies and tools to create educational materials more quickly and in greater volume, and disseminate them without loss of quality to the increasing numbers of students seeking a technologically up-to-date education.

## 7. Choosing Assignment Types

During the semester, I assign several peer-reviewed assignments, and several types of peer-reviewed work. I give the students a choice of which order to do the assignments, subject to the constraint that there is a limit to the number of students doing each type of assignment for each deadline. This strategy is

motivated by a desire to students doing all kinds of assignment soon after each lecture, so that, e.g., while Lecture 10 is fresh in their minds, some students will be making up problems, some will be annotating, and some will be creating animations. This insures that I get problems, animations, etc. over a wide range of lectures, rather than having all submissions concentrated on the lectures that were covered near the time an assignment was announced.

Figures 6 and 7 show the assignments I recently gave in my masters-level parallel-architecture class (CSC/ECE 506) and my combined senior/masters-level microarchitecture class (ECE 463/521).

## 8. Student reaction

Students in both architecture classes were surveyed at the beginning of January 2003. In CSC/ECE 506, 16 of 36 students responded, a rate of 44.4%. In ECE 463/521, 71 of 96 students responded, a rate of 74.0%. The classes did not vary much in their reaction.

The comments provided by students indicate fairly strong support for the concept of peer review, but they take issue with three aspects of the way it was implemented for these two courses.

- They thought they were hurt by the fact thbat a few students did not do their reviews. In fact, the version of PG used at the time did not deduct points for students who failed to do their reviews. During Fall 2002, PG was modified to do this checking, and it will be in the system in coming semesters. This should lead to more reliable reviewing and therefore address this criticism.

- While generally supporting the idea of multiple review deadlines, they sometimes submitted an update that was never re-reviewed by their reviewers. Currently, there is no guarantee that a reviewer won't complete reviewing in Round 2 before an author resubmits. The scheme will be changed in Spring 2003 to have extra deadline, so that there is a review period followed by a resubmission period, followed by a second review period. This should take care of the problem.

- A number of students objected to reviewers who gave low grades but few if any suggestions on how to improve. Now students are told, during the review-of-review period, to downgrade reviewers who deduct points without explaining why.

Note that, in general, students thought that reviews of reviews *were* effective (3.9 on a scale of 5) in motivating careful reviews. This suggests that giving students some guidance in how to evaluate reviews can motivate students to review according to guidelines that they are given.

## 9. Conclusion

Electronic peer review has proved to be an effective technique for teaching computer architecture. It allows the students to get experience writing for their peers, and it facilitates the production of educational resources that can be used by future classes, such as annotated lecture notes, homework and test questions, and daily machine-scorable quizzes. However, effective implementation of peer review is tricky. Reviewers must be given good guidance in how to review and sufficient motivation to do a good job. Authors must be given enough time to revise their work pursuant to reviews, and reviewers must be given enough time to complete their final pass. Our experience with PG has given us many ideas on how to improve the process and outcomes of peer review.

**Table 1.** Student Evaluation of PG

| | | CSC/ECE 506 | ECE 463/521 |
|---|---|---|---|
| 1 | Peer review is helpful to the learning process. | 3.63 | 3.41 |
| 2 | I was satisfied with the reviews of my work. | 3.53 | 3.47 |
| 3 | The feedback I obtained from the reviews helped me to improve my work. | 3.60 | 3.49 |
| 4 | Two review deadlines were imposed, one for the first review and another for the final grade. Did this provide an adequate opportunity for you as an author to respond to the comments of your reviewers? | 3.60 | 3.83 |
| 5 | The knowledge that my reviews would be reviewed motivated me to do a careful job of reviewing. | 3.93 | 3.92 |

## 10. References


[1] The Deadalus Group, Daedalus Integrated Writing Environment, http://www.daedalus.com/info/overtext.html

[2] Downing, T. and Brown, I., "Learning by cooperative publishing on the World-Wide Web," Active Learning 7, 1997, pp. 14-16.

[3] Earl, S. E., "Staff and peer assessment: Measuring an individual's contribution to group performance," *Assessment and Evaluation in Higher Education* 11, 1986, pp. 60-69.

[4] Ford, B. W., *The effects of peer editing/grading on the grammar-usage and theme-composition ability of college freshmen*. Dissertation Abstracts International, 33, 6687.

[5] Gehringer, Edward F., "Strategies and mechanisms for electronic peer review," *Proc. Frontiers in Education 2000*, Kansas City, October 18–21, 2000 (to appear).

[6] Lopez-Real, Francis and Chan, Yin-Ping Rita, "Peer assessment of a group project in a primary mathematics education course," *Assessment and Evaluation in Higher Education* 24:1, March 1999, pp. 67-79.

[7] MacAlpine, J. M. K., "Improving and encouraging peer assessment of student presentations, *Assessment and Evaluation in Higher Education* 24:1, March 1999, pp. 15-25.

[8] Persons, Obeua S., "Factors influencing students' peer evaluations in cooperative learning," *Journal of Business for Education*, Mar.–Apr. 1998.

[9] Rada, R., Acquah, S., Baker, B., and Ramsey, P., "Collaborative learning and the MUCH System," *Computers and Education* 20, 1993, pp. 225-233.

[10] Rafiq, Y., & Fullerton, H., "Peer assessment of group projects in civil engineering," *Assessment and Evaluation in Higher Education* 21, 1996, pp. 69-81.

[11] Rushton, C., Ramsey, P., and Rada, R., "Peer assessment in a collaborative hypermedia environment: A case-study," *Journal of Computer-Based Instruction* 20, 1993, pp. 75-80.

[12] University of Portsmouth, "Transferable peer assessment," in National Council for Educational Technology [ed.], Using information technology for assessment, recording and reporting: Case study

[13] http://courses.ncsu.edu/ece521/common/lectures/notes.html

[14] Gehringer, E., "Reuse of homework and test questions: When, why, and how to maintain security?" submitted to ASEE 2003 Annual Conference, Educ. Res. & Methods Division.

[15] The Learning Online Network with CAPA, http://www.lon-capa.org.

[16] Mallard: Asynchronous Learning on the Web, http://www.ews.uiuc.edu/Mallard/.

[17] WebAssign, http://webassign.net.

[18] Gehringer, E., "Peer review and peer grading in computer-science courses," SIGCSE 2001: Thirty-Second Technical Symposium on Computer Science Education," Charlotte, Feb. 21–25, 2001, pp. 139–143.

[19] Gehringer, E., "Building an Ethics in Computing Website using peer review," 2001 ASEE Annual Conference and Exposition, Session 1461.

[20] Gehringer, E., "A Web-Based Computer Architecture Course Database," 1999 ASEE Annual Conference and Exposition, Session 3232.

# Laboratory Options for the Computer Science Major

Christopher Vickery
Tamara Blain
Queens College of CUNY

*Computer Science and Computer Engineering programs typically converge on the Dynamic-Static Interface (DSI) from opposite directions. Computer Science (CS) introduces students to system architecture and organization so they can have a better appreciation for the mechanisms that make their software work, whereas Computer Engineering (CE) introduces students to software design so they can have a better appreciation for the software that will be using the hardware systems they design. Mindful of this distinction between CS and CE, we chronicle the efforts of our CS department to capitalize on current trends in the design and implementation of digital systems to extend our students' expertise in this area. We summarize the current curriculum in our department, present a survey of the language options we have explored for evolving our curriculum, and conclude with a brief description of the laboratory environment we have adopted, which is centered on the Handel-C hardware implementation language.*

## 1 Introduction

Computer Science and Computer Engineering curricula have traditionally brought significantly different perspectives to bear on what to cover and how to teach computer architecture, the point where the two disciplines meet. Broadly speaking, the CS students bring good software skills to their architecture courses, whereas the CE students bring stronger circuit design skills to theirs. The distinction carries over to the design of digital systems in industry, where a software team and a separate engineering team typically work in parallel during the development of a new product (codesign), with software/hardware integration occurring late in the development cycle.

But the inexorable advance of circuit complexity has caused the traditional engineer/programmer dichotomy to start to break down. We are not talking here about shifting the dynamic-static interface [23, 25] for a particular system design, nor about the dual roles individuals might play in a design effort. Rather, we are responding to changes in the way digital systems are developed due to changes in the functionality of programmable logic devices (FPGAs in particular) and the software tools used to develop systems using them.

Ours is a Computer Science department in a liberal arts college. There is no engineering department on campus. Although the university encourages cooperation among its member colleges, the fact remains that the closest CE courses available to our students are a 90-minute subway ride away from us. In this context of CS isolated from CE, this paper reports on the options we have considered as we adjust our curriculum to provide our undergraduate students with a better understanding of the principles and practices of implementing digital architectures.

## 2 Context: A CS Department

Our undergraduate curriculum prepares students in the broad areas of *i)*software design and implementation, *ii)* formal methods, *iii)* hardware design, and *iv)* applications, in roughly that order of emphasis. Our offerings in the "hardware design" area include a course in assembly language and basic logic design, and a second course that covers additional logic design and an introduction to computer architecture. We have used a number of textbooks for these courses over the years, never finding ones that both students and faculty found completely suitable. The current text for both courses is by Murdocca and Heuring [22].

Our curriculum also includes a "Hardware Laboratory" course, which has not been offered in recent years. This course was developed in the days of SSI and MSI integrated circuits and dropped by the wayside as simulators have allowed us to develop a similar degree of mastery to the old lab course without requiring the students to spend time in the lab itself. Thus, the closest our students have come to a hardware laboratory experience for the past several years has been through simulation assignments in the two courses mentioned above. The student edition of CircuitMaker [2] has served our purposes in this regard, although the free version must be installed only on the students' personal computers, not in college laboratory facilities.

Four years ago, we received NSF funding [16] to revise our curriculum to use HDLs to give our students a more realistic view of circuit design technologies. Our stated goals were, "to give all of our students some knowledge of the methods that are used in designing modern digital circuits [and] to provide those students who are interested with hands-on experience in designing and using digital logic as a method of teaching them about computer architecture."

At the time we prepared our grant proposal, VHDL seemed to be the natural vehicle for introducing CS students to logic design, leveraging their existing software skills to introduce them to hardware design techniques. There were several textbooks based on VHDL available, it was an IEEE Standard, and seemed generally well suited to our needs. Although Verilog also become an IEEE Standard in 1995, at the time of our grant proposal, VHDL seemed like the most straightforward way to go. Since then, there has been a good deal of foment in the CAD world, propelled by the need for tools to adapt to the ever-increasing complexity of digital devices. What follows is a survey of the evolving software development options we have seen.

## 3 Laboratory Options

A first question CS departments have to answer in planning instruction in digital design is whether to focus on simulation only, or to have the students target actual hardware devices. A second question is whether to use commercial development tools or instructional software. Once those questions are resolved, the issues of software and (if hardware devices are targeted) prototyping platforms need to be addressed.

### *3.1.1 Simulation or Hardware?*

Simulation is a critical step in developing new hardware designs, but in an instructional environment, simulation can arguably be the end step in a student's lab experience.

There are several arguments for using only simulation for introducing CS students to the design of digital systems:

- *Cost*. Except for the computers to run the simulations, a relatively abundant commodity, simulation avoids the overhead and costs of purchasing and maintaining prototyping equipment and instrumentation for a laboratory.
- *Ease of debugging.* In addition to avoiding the issues associated with bad connections and failed components, simulation provides a software view of the system under development which is not only more familiar to CS students, but also more flexible than hardware in terms of allowing students to visualize and locate problems in their designs.
- *Simplicity.* Development tools for hardware implementations need to provide a richer feature set than instructional simulations. The result can be that students need to spend more effort learning to use the tool than studying the simulations.

On the other hand, implementing actual circuits can be much more motivating than just running "yet another program."

We have not had good luck with most of the student-oriented simulators for logic design that are available at low or no cost. Many had problems with reliable schematic entry, and many have had unnecessarily poor user interfaces. The student edition of the CircuitMaker schematic entry and simulation software from Altium cited above has been the best we have found so far [2]. Compared to a textbook-only presentation, it's far superior. But it limits the size of the designs students can implement, cannot be installed in departmental labs for free, and it doesn't provide a tie-in to actual hardware implementation.

Another option for those graduates who become interested in chip design is to send them to commercially available short courses that teach digital IC or systems design, ECAD, or hardware/software codesign. But the danger in these courses is that there is "insufficient time to address any topic in the depth required by students to gain proper insight into the subject area". Thus these courses may not adequately provide them with the skill set necessary for designing and implementing complex distributed embedded systems and Systems on Chip (SOCs) [12]. And, of course, this option begs the question of what to do in the context of a university curriculum.

The gamut of options available for implementing circuits in hardware is extremely broad, ranging from inexpensive breadboards with SSI and MSI ICs connected by jumper wires to industrial-grade prototyping systems used in the development of commercial ASIC designs at costs that are prohibitively high for virtually all instructional purposes. FPGA-based development systems strike a middle ground between these two extremes, and are particularly well suited to instructional laboratories. The boards are self-contained units requiring no assembly on the part of the student, although expansion headers are normally available for customized projects. Student designs are prepared and simulated on PCs, and downloaded to the prototyping board through a serial or parallel cable. The use of reprogrammable FPGAs as the implementation target gives students a development cycle familiar to them familiar from the software development world: edit, compile, debug.

Major FPGA vendors, notably Altera and Xilinx, provide inexpensive or free student versions of their commercial tools for FPGA development suitable for use with a variety of prototyping boards from companies such as Xess and Digilent. An inexpensive package available from Altera's University Program, for example, includes a prototyping board with a 20,000 gate

FPGA, several LEDs, displays, and switches mounted on the board, and I/O connectors for a mouse and VGA display. This kit comes packaged with a good tutorial volume [17] featuring a number of interesting projects students can do. The kit includes a student edition of Altera's MAX+Plus development software, which includes schematic, waveform, and HDL text editors for design entry. It should be noted, however, that this kit uses a relatively small FPGA by today's standards, (not large enough to implement a full CPU) and that the MAX+Plus software does not provide the same functionality as Altera's Quartus toolchain. Systems of this type are more appropriate for introductory logic design laboratories rather than CS Computer Architecture courses, where students need to explore architectural design parameters.

Hardware Description Languages (HDLs), most commonly VHDL and/or Verilog, are the most commonly used means for entering designs for platforms like those discussed so far. But today's FPGA devices can have millions of gates instead of tens of thousands, providing architecture students with hardware targets rich enough to support investigations into topics as advanced as pipelining, cache design, and multiprocessor communication, not just basic logic design. Furthermore HDL programming is evolving to deal with the complexity of these newer devices. We review some of these languages below. An appealing alternative to working with an HDL or one of its derivatives, at least for CS students who are approaching the DSI from the software side, is to use a language based on a traditional High Level Programming Language (HLL). After our survey HDLs and their derivatives, we will turn our attention to Handel-C, a hardware implementation language based on C that we are adopting for use by our CS majors.

## 4 HDLs and Their Derivatives

### *4.1.1 Verilog and VHDL*

Hardware design is dominated by the use of Verilog and VHDL. They are most powerful as gate-level implementation languages [1][3]. VHDL allows a multitude of language or user-defined data types, which may mean confusing conversion functions needed to convert objects from one type to the other. All of the logical operators, NAND, NOR, XOR, etc, are included in VHDL but separate constructs, typically defined using the VITAL language, must be used to define cell primitives of ASIC and FPGA libraries. VHDL offers a great deal of flexibility in terms of its abundance of permissible coding styles. It allows for concurrent synchronization schemes, such as semaphores. VHDL is better suited than Verilog to handle very complex designs. It is relatively weaker in lower level designs but superior in higher level and system level designs, which results in slower simulations. Its wealth of constructs, attributes, and types make VHDL a good language for design and verification [7]. It is strongly typed and there are many ways to model the same circuit, features which make it more robust and powerful than Verilog but also more complex. This complexity means it is more difficult to understand and use.

Verilog has adopted many of VHDL's features, thus Verilog is moving towards increasing complexity as well [7]. Verilog is used for high-speed gate-level and register-level circuit descriptions, fast IC modeling and RTL simulation, easy synthesis, and test applications [9]. Gate simulations in Verilog are 10x to 100x faster than the same simulations in VHDL, which means shorter time to verify designs [8]. Compared to VHDL, Verilog data types are simple, easy to use and geared towards modeling hardware structure as opposed to abstract modeling. Because it is simpler, Verilog is easier to learn. On a Verilog vs. VHDL debate forum, an engineer who knows both languages cites: "If you were just taught Verilog syntax, you're in trouble. If you were taught syntax with guidelines, and warned about legal Verilog constructs that should never be used, you can gain expertise in half the time it takes to become proficient in VHDL [8]." Because of its background as an interpretive language, there are no libraries in Verilog whereas VHDL stores compiled entities, architectures, packages, and configurations. Verilog was originally developed with gate-level modeling in mind, and so has very good constructs for modeling at this level and for modeling cell primitives of ASIC and FPGA libraries. For this reason, students may find Verilog more digestible than VHDL at first. Because it is geared towards lower level modeling, it is faster in simulations and effective synthesis. It lacks, however, constructs needed for system level specifications. Verilog's simple, intuitive and effective way of describing digital circuits for modeling, simulation, and analysis purposes make it very popular in the industry.

### *4.1.2 ESL Design*

There is a movement towards system level modeling, also called electronic system level (ESL) design. This is the design of an electronic product at the conceptual level, including hardware/software codesign; design partitioning, and specification writing [20]. It demands being able to describe requirements and functions independently of implementation, and being able to talk about interfaces and protocols without describing the actual hardware [19]. Verilog is neither object-oriented nor strongly typed, which makes it cumbersome for system level design. Also, the previously attractive flexibility of its syntax can lead to difficult to detect

errors. Neither Verilog nor VHDL provides the syntax or semantics to describe a product at the system level [20]. The trend of RTL engineers moving up in abstraction and systems engineer moving down, as well as the fact that both Verilog and VHDL have shortcomings in the requirements of ESL design, has necessitated the need for either a new language, or the extension of an existing language to bridge the gap between specification and implementation. The new topic of debate is the question of which language is right for ESL design [20].

### *4.1.3 Extended HDLs*

**Superlog** is an extension of Verilog that includes features that allow a more abstract description of an electronic system [20]. While most of the semantic elements added were borrowed from VHDL, it retains most of the features of Verilog, including support for hierarchy, events, timing, concurrency, and multi-valued logic [6]. Superlog's major technical advantages over VHDL are a clean and powerful interface to C that allows hardware/software codesign, and C-based constructs for system design and decomposition [1]. It borrows useful features from C and Java, including support for dynamic processes, recursion, arrays, and pointers. It also includes support for communicating processes with interfaces, protocol definition, state machines, and queuing. It has been estimated that Superlog needs one half to one third the number of lines of code to describe a function as Verilog at the same abstraction level, and Superlog can go much higher in abstraction [6].

**System Verilog.** A radically revised version of Verilog was presented at the 2001 International HDL Conference [15]. These changes represent a move towards an even higher level of abstraction for the language and an extension to its capability to verify large designs. SystemVerilog is a blend of Verilog, C/C++, and Superlog that allows module connections at a high level of abstraction [15]. Verilog currently allows the connection of one module to another only through module ports, which can be tedious. SystemVerilog introduces interfaces which makes it possible to begin a design without first establishing all the module connections. C-language constructs, such as globals, are another addition. In Verilog, only modules and primitive names can be global. SystemVerilog allows global variables and functions. SystemVerilog borrows abstract data types from C, such as 'bit', 'char', 'int', and 'logic', which provide more versatility then the existing 'reg' and 'net' types and allows C/C++ code to be included directly in Verilog models and verification routines. Also included is an assertion construct, similar to VHDL's, intended to do away with proprietary assertion languages. Because there's much in Superlog that is not part of SystemVerilog, Superlog will remain a superset of SystemVerilog. With its new additions, SystemVerilog may remove some of the impetus for C-language design, at least for RTL chip designers. The question of whether or not vendors will create tools to support SystemVerilog remains to be seen. [15]

### *4.1.4 HLL Pros and Cons*

Teaching system level design in a High Level Language (HLL) is well suited to students with limited electronics or CAD backgrounds and are unfamiliar with hardware concepts such as signals, voltages, and details of the clock. By starting with either Handel-C or SystemC, hardware/software codesign becomes more accessible to students whose initial programming experience will most likely be C, C++, or Java rather than assembly language [12]. It exposes the students to concurrency, parallelism, software-to-hardware mapping, pipelining, and computer architectures as well programming principles [11]. In Handel-C, for example, each assignment statement takes one clock cycle and each expression evaluation takes no clock cycles, which makes it easy to reason about the number of clock cycles required to execute the code. This relationship encourages efficient compact code form a hardware perspective [11].

However, there is a risk in HLL-based design for the student who already has a software mindset. Specifying hardware using an HDL is not programming, but rather the building of hardware and arrays of gates. Applying general purpose programming tactics to an HDL too often makes too many gates and highly inefficient chip and logic layouts [21].

There are other shortcomings to the HLL approach. One is that it is hard to integrate outside IP with any hardware designed this way. This is due to the fact that close examination of compiler-generated circuits reveals little of their purpose or about how they were generated. Therefore, the "hooks" into the circuitry are not readily apparent. The obfuscated nature of the compiler generated circuits also makes it nearly impossible to hand optimize any of these circuits. These problems stem from the fact that the original HLL on which these new languages are based either cannot express parallelism, or their concepts of memory, methods, and objects map poorly onto real hardware. Thus the new languages are forced to include tools that include the necessary attributes, but at the expense of generating clean hardware. But as one industry expert points out, "elegance of implementation has never triumphed over timesaving hacks. Mnemonics overtook opcodes, compilers overtook assembly, and HDLs overtook schematics. Each time, the old guard maligned the inefficiency of the automated tools vs. the craftsman-

ship of their methods; but each time automation carried the day [26]."

Many ASIC or FPGA based products include a mixture of algorithmic processing most readily expressed in an HLL and other sets of operations most efficiently implemented directly in gates. FPGAs accommodate these designs by providing CPU cores that can be drawn from a library and implemented in the logic fabric of the FPGA as well as the emergence of devices such as Xilinx' Virtex II Pro which include one or even multiple hard CPU cores embedded directly in the device itself. In systems such as these, use of an HLL based implementation language provides a good fit for implementing the entire job [24].

An increasing amount of system functionality is expressed in embedded software; synthesis and layout are linked into one process, and the typical hardware designer is forced by complexity to work at a high level [14]. S/he would use the ultimate design system, where you wouldn't even care what goes into the hardware or software; you'd write C/C++ code and everything else would just happen under the hood because of an intelligent C/C++ compiler [1]. According to some industry experts, this future may present itself in 5 to 10 years, and those whose career paths extend that far would do well to anticipate it [14].

### *4.1.5 C Based Languages*

**SystemC.** SystemC is an open source language that is more a structured class library than a language. An argument for SystemC is that the C language lacks the object-oriented facilities that some complex system designs require [19]. SystemC was developed to support system level design. Its class libraries add hardware design-specific modeling constructs that increase the power of the language to meet the needs of hardware design [3]. The class libraries provide data types appropriate for fixed-point arithmetic, communication channels, which behave like pieces of wire (signals), and modules to break down a design into smaller parts. In addition, the class library contains a simulation kernel - a piece of code that models the passing of time, and calls functions to calculate their outputs whenever their inputs change [10]. The syntax is simple and close enough to C++ that students should find it easily digestible.

SystemC partially addresses the problem that C language design presents by creating a number of classes that mimic hardware primitives and time-domain events [20]. Although at present it offers only modeling support, SystemC is moving towards broader capabilities in synthesis [5]. Future versions of the class library will be extended to cover modeling of operating systems, to support the development of models of embedded software [10].

The major drawback of SystemC is the need to convert a C/C++ based description to Verilog or VHDL in order to synthesize it [20]. The problem is that there is not yet a working behavioral synthesis tool available for commercial use that can accept C++ as it's input language. The conversion process is currently a manual decomposition of the design until the designer gets to a low enough point of abstraction such that a commercially available translator allows the use of RTL synthesis. This process, even if done automatically, is prone to errors that are difficult to find [19].

## 5 Handel-C

**Handel-C** [4] is both a subset and a superset of conventional C. It does not include functional recursion, floating-point data, or any of the Standard C runtime library functions for I/O or string operations. However, its integer type is augmented with a rich set of operators and declarations for field widths, a *par* construct for expressing parallelism, semaphores and communication channels as primitives, and multiple *main()* functions, each with its own clock [12]. Because it is a variation on C rather than on C++, Handel-C is closer to the hardware than SystemC [18].

Handel-C provides a rich set of code structures including functions, arrays of functions, inline functions, macro procedures, and macro expressions. These facilities allow the student to explore time-space tradeoffs in a design. Handel-C is not tied to any particular family of target devices, although it is clearly aimed at FPGA development in general [13].

The Handel-C development environment supports cycle accurate simulation, allowing students to see multiple statements being executed in parallel using a debugging user interface fully reminiscent of traditional software IDEs. Compiling generates an industry standard (EDIF) netlist, which is then imported into the FPGA vendor's toolkit, where VHDL or Verilog based modules can be integrated and simulated with the Handel-C part of the design if desired. The vendor's tools then perform place and route, and generate a bit stream for downloading to the target device. [26].

Handel-C appears to be an ideal development language for CS students with limited experience in hardware design. But adopting it for laboratory use introduces tradeoffs that need to be considered. In particular, prototyping kits that take full advantage of the language's ability to generate complex systems can add considerably to the cost of laboratory seats. For example, one such kit is the RC200 from Celoxica, which includes a standalone prototyping board with a 1M gate Xilinx

FPGA, audio, video, networking, and memory subsystems and peripherals such as a camera and touchscreen. Fully equipped, this kit costs as much as a complete midrange PC.

## 6 Conclusion

Trying to evaluate software development toolchains and/or target platforms can be as daunting a task as trying to track emerging trends in design languages. With the recent emergence of FPGA devices so powerful and fast they challenge the one-time undisputed supremacy of ASICS for high-end designs, inexpensive FPGA-based systems, such as those available from Xess and Digilent, emerge as appealing vehicles for CS programs interested in offering students exposure to mainstream technologies of the day. And the main FPGA vendors, such as Altera and Xilinx provide student versions of their development platforms on very reasonable terms for universities.

But logic design using VHDL or Verilog is not as attractive an option for CS students as C-based development languages. We found the Handel-C development environment from Celoxica Ltd. particularly appealing. Hardware is specified in Handel-C, and the resulting netlist is then imported into an FPGA project (using Altera or Xilinx tools, depending on the target), where HDL modules, including IP cores, can be integrated if desired. While inexpensive prototyping kits are available that provide support for logic designs of modest complexity, we believe the options possible using Handel-C and more complex prototyping platforms like Celoxica's RC200 are more suitable for a CS laboratory in computer architecture.

We are in the process of setting up a CS laboratory based on Handel-C and RC200 development kits and will be offering the first class using the lab during the Fall 2003 semester. A major challenge for us is to develop a set of laboratory exercises that guide students in the effective use of this complex environment in a meaningful way. We look forward to sharing our experiences.

## 7 References


[1] Aldec, Inc. *Evita: Advanced Verilog Tutorial with Applications*. www.aldec.com/Downloads.

[2] Altium Ltd. *CircuitMaker Student Edition.* www.circuitmaker.com.

[3] Bartlett, Joan. "The case for SystemC". *EEDesign*. 7 March 2003.

[4] Celoxica Ltd. *Handel-C Language Reference Manual.* Document Number RM-1003-3.0. 2002.

[5] Clark, Peter. "IP99: Designers see little need to move away from HDLs". *EE Times*. 4 Nov. 1999.

[6] Clark, Peter. "Startup to field next-generation design language". *EE Times*. 31 May. 1999.

[7] Cohen, Ben. *Verification Guild*. Vol. 1, No. 17. 14 Aug. 2000. janick.bergeron.com/guild.

[8] Cummings, Clifford E. *Verification Guild*. Vol. 1, No. 17. 14 Aug. 2000. janick.bergeron.com/guild.

[9] Davidmann, Simon. "It's time for a rethinking of system-on-a-chip design". *EE Times*. 25 Oct. 1999.

[10] Doulos Ltd. *A Brief introduction to SystemC*. www.doulos.com/knowhow/systemc_guide/tutorial/introduction.

[11] Downtown, A.C., Fleury, M., Self, R. P., Sangwine, S. J., and Noakes, P. D. *Hardware/Software Co-Design: A Short Course for Unbelievers*. www.celoxica.com/technical_library/files/"CELCUPACPGENHardware Software Co-Design - A short Course For Unbelievers-01002.pdf."

[12] Downtown, A.C., Fleury, R. P., and Noakes, P. D. *Future directions in computer architectures curricula: Silicon compilation for hardware/software co-design*. www.essex.ac.uk/ese/research/mma_lab/Handelc/21CComputer.pdf.

[13] Gaffar, A. A. "A Survey on the Handel-C Language" *Surprise Project 1999*. www.iis.ee.ic.ac.uk/~frank/surp99/article1/amag97

[14] Goering, Richard. "Rank and file don't like C". *EE Times*. 15 Nov. 1999.

[15] Goering, Richard. "Standardization nears for next-generation Verilog". *EE Times*. 14 Nov. 2001.

[16] Goodman, S. G. and Vickery, C. *A Laboratory for Computer Organization and Architecture.* NSF DUE-9950364, 1999.

[17] Hamblin, J. O. and Furman, M. D. *Rapid Prototyping of Digital Systems: A Tutorial Approach*. Kluwer, 2001.

[18] Hammes, Jeffrey P. *A High Level, Algorithmic Programming Language and Compiler for Reconfigurable Systems*. www.cs.colostate.edu/cameron/Publications/hammes_enregle.pdf.

[19] Moretti, Gabe. "Get a handle on design languages". *EDN Magazine*. 5 July 2002.

[20] Moretti, Gabe. "System-level design merits a closer look." *EDN Magazine*. 21 Feb. 2002

[21] Motorsabbath. *Hardware design in JHDL*. www.slashdot.org, 16 Jan. 2002.

[22] Murdocca, M. J. and Heuring, V. P. *Principles of Computer Architecture.* Prentice Hall, 2000.

[23] Patt, Y. and Patel, S. *Introduction to Digital Systems.* McGraw-Hill, 2001.

[24] Prophet, Graham. "System-level design languages: to C or not to C?" *EDN Europe*. 14 Oct. 1999.

[25] Shen, J. P. and Lipasti, M. H. *Modern Processor Design*. McGraw-Hill, 2003.

[26] Turley, Jim. "The Death of Hardware Engineering". *Embedded.com*. 28 Feb. 2002.

# Activating Computer Architecture with Classroom Presenter

Beth Simon† Richard Anderson* Steven Wolfman*

†Math & Computer Science, U. of San Diego
San Diego, CA 92110
bsimon@sandiego.edu

*Computer Science & Engineering, U. of Washington
Seattle, WA 98195-2350
{anderson,wolf}@cs.washington.edu

## Abstract

In this paper we discuss our experiences using a Tablet PC-based presentation system in an undergraduate computer architecture class. The system allowed us to integrate PowerPoint slides with high quality pen-based writing and to separate the instructor's view of the materials from the students' view. This allowed a more natural and interactive development of class concepts and content.

The system that we used was Classroom Presenter which was developed at University of Washington and Microsoft Research. The system has received substantial use at University of Washington, being used in approximately 15 large courses since Autumn 2002. The successful deployment at the University of San Diego in a small undergraduate course is interesting since the developers of the system viewed Classroom Presenter as most appropriate for large lectures and for distance courses. The deployment at the University of San Diego explored new ground in usage of the system. In this work we present an overview of the system and discuss particular uses and advantages of the system in an undergraduate architecture class setting.

## 1. Introduction

Presentation technology impacts the structure and delivery of lectures. Different technologies support different instruction styles and provide various mechanisms for engaging an audience. In university classrooms predominant presentation technologies include blackboards, whiteboards, overhead projectors, and computers with data projectors. Each of these technologies has properties that may make them more or less suitable to specific instructors or course material. In particular, delivering a computer-based lecture has both advantages and disadvantages. Advantages include the ability to structure material in advance, prepare high-quality examples and illustrations, and to share and re-use material[3]. But, these advantages come at the expense of flexibility during presentation – especially in an undergraduate architecture class where we'd like students to experience for themselves the tradeoffs inherent in the microarchitecture design process.

Classroom Presenter (hereafter, Presenter) is a system developed and deployed by the University of Washington as part of the Conference XP conferencing experience project. The immediate goal of Presenter is to provide an improvement to the computer-based lecturing environment – offering, at a minimum, the benefits of prepared slides and extemporaneous writing and diagramming. In the long term, Presenter aims to enhance learning and teaching through new technologies and software for the classroom. The key components of the current system are the use of an instructor Tablet PC (with high-quality inking support), wireless network connectivity, and a data projector.

In this paper we focus on the presentation issues found in an undergraduate Patterson and Hennessey-style architecture course. We show how the interactive nature of inking on empty (or intentionally incomplete) slides allows students to participate in the microarchitecture design process, rather than having it presented to them "fait accompli". Specifically, we investigate Presenter's ability to support the in-class development and modification of datapath diagrams and active student participation in problem solving. Additionally, we preview future Presenter support for sharing tablet-based in-class group work.

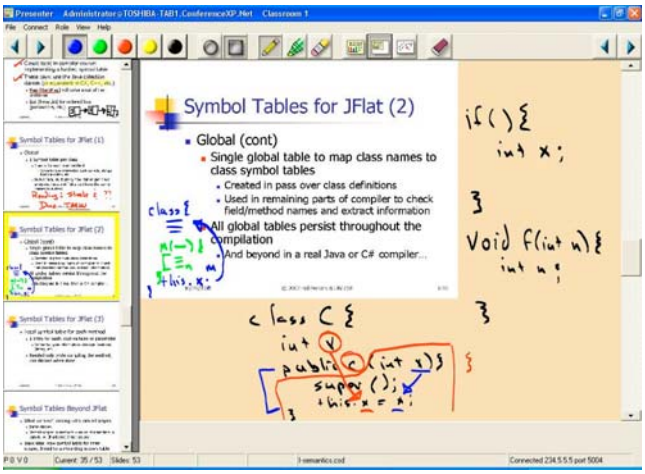


**Figure 1. A screenshot of the instructor's Tablet PC while using Presenter. The slide is minimized to provide extra writing space. The filmstrip is along the left of the figure allowing preview of and navigation among slides. (CSE582, Univ. of Washington, Autumn 2002)**

The rest of the paper is organized as follows. In Section 2 we describe the Presenter system and note three key features that arose in its design process. In Section 3 we describe an initial offering of an undergraduate computer architecture class utilizing Presenter for class lecturing in a small classroom setting (10-15 students). In Section 4 we describe upcoming Presenter features that empower additional classroom interaction and group activities in both the small and large classroom environments. Section 5 describes related work and Section 6 concludes.

## 2. Presenter System

Figure 2 depicts the basic architecture of the Presenter system. The instructor loads a presentation composed of an ordered deck of slides onto a mobile tablet computer. She can write on the slides with the tablet's pen and control the presentation—advancing slides, changing pen color, etc.—using controls displayed on the tablet. The tablet wirelessly transmits control information and writing to a computer driving a data projector, synchronously displaying slides and writing to the entire class. In a distance class, a remote site would have its own computer and projector controlled by the tablet via the Internet.

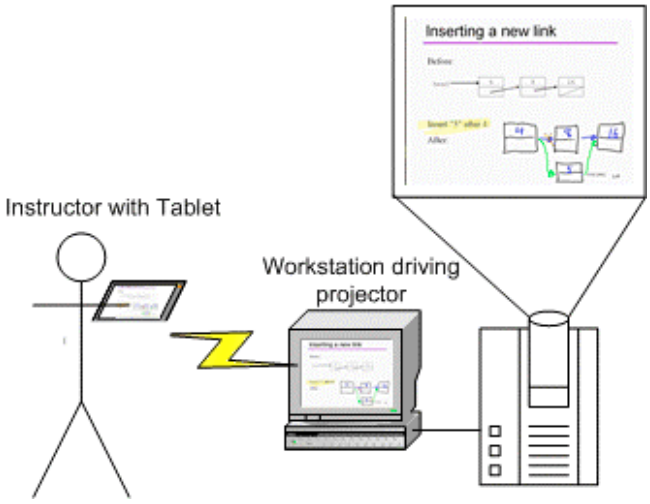


**Figure 2. The architecture of Presenter in the classroom. An instructor controls the presentation from a mobile Tablet PC. The tablet is wirelessly connected to a machine driving the display of the presentation on a data projector.**

Presenter transmits and displays writing and control information in real-time over an 802.11b wireless network. In-class use of Presenter has been tested with most known brands of Tablet computers currently available (including Acer, Toshiba, HP/Compaq, Fujitsu, NEC, and Motion Computing models). The machine controlling the data projector must either have an internet connection or be connected to the same wireless network as the instructor's tablet (no live internet connection required). Assuming a classroom has already been outfitted with a computer and data projector, using Presenter would cost about $2000 for the Tablet PC and $150 for a wireless base station. Both of these could be shared across classes, or the tablet could be used as a personal machine by an instructor.

Three key features of Presenter and, we believe, for lecture presentation systems in general, emerged from our in-class experiences with our system. First, Presenter integrates writing directly on top of slides. The instructor can use the tablet pen to write notes or diagrams directly over the slide, as shown in Figure 1. The instructor can also shrink the visible slide and writing, creating new writing space around the slide. The ability to write in the context of the slides maintains the connection between extemporaneous writing and the prepared content. Sustaining this link enhances the slides' value as a support structure for communication – a "mediating artifact" [16] – in the classroom. Additionally, if the instructor has even more to say, she can jump to a "whiteboard" slide to work an impromptu sample problem or continue discussion beyond the context of the prepared slide.

Second, writing in Presenter is represented with high-quality ink, which renders in real-time and looks and feels natural. This is enabled by the high pen sampling rates and resolution on recent Tablet PCs and Tablet PC software support for smooth curves and pressure-sensitive line thickness. High-quality writing allows full use of the available resolution on the display, increases instructor comfort with writing and eases student comprehension of hand-written text.

Finally, Presenter separates the instructor view of the presentation from the projected display that students see. Actually, Presenter supports three viewing modes: instructor (seen on the "primary" tablet), projector, and student. Instructor and projector modes are described here, while student mode will be further discussed in Section 4.

The instructor uses the instructor view on her tablet while the projector machine ships the display view to the public screen. This separation allows the instructor view to include a wide array of tools—such as pen color and style controls—and information displays—such as the filmstrip pane on the left side of the display, showing miniatures of the slides immediately preceding and following the current slide. Furthermore, with no tether to the data projector, the tablet can go completely wireless, giving the instructor freedom to control her presentation from anywhere in the classroom or even to pass the tablet to a student.

Additionally, Presenter supports "instructor mode objects" – text or drawings visible only on the instructor tablet view and not shown on the projector view. These objects can contain reminders, notes, or hints to the instructor of issues to discuss in relation to the slide or questions to ask the students. These objects can also encapsulate information that the students will be asked to actively *derive* in-class – in contrast to more traditional static "here's the resulting answer" treatments. Pictures, graphs, or diagrams can be annotated with circles, lines, or other drawing objects that the instructor can "draw over" in class to highlight important areas or show modifications.

One unique capability of the Presenter system is that it facilitates the creation of an artifact from a given lecture. Inked notes created in class can be saved in conjunction

with the slides from the class and viewed at a later time. This has implications for allowing instructors to review, in greater detail, the material and discussions covered in a given class period. Students could also be given access to inked notes from class discussions, at the discretion of the instructor.

## 3. Presenter for Undergraduate Architecture

We discuss one semester's experimentation using Presenter in a small-class undergraduate Patterson and Hennessey-style computer architecture class. We give examples of the various usages of Presenter system components in creating a more interactive lecture while still maintaining the organization and re-use features of an electronic presentation. Many of these examples echo recommended practices of modern pedagogy, e.g., active learning [9] and Classroom Assessment Techniques [2]. A survey of the class found strong student approval of the Tablet PC-based system, despite occasional technical issues involved with beta-testing.

### Inking-Over for Emphasis, Notes to Mention

Perhaps the most often used form of interaction enabled by the Presenter system is a simple circling or highlighting of a word or phrase on a slide. This can allow the instructor to visibly drive home an important concept or emphasize a term students should understand. In Figure 3(a) we see a projector view that might result after a discussion of execution time versus throughput. These circles were added at the same time a "verbal clue" was given to the students – showing emphasis or distinguishing from previously discussed concepts. Additionally, instructor-only objects (shown in Figure 3(b) in rounded text boxes) can remind the instructor of additional comments to make or simply encourage the instructor to prompt the class for a verbal response.

Another instructor using Presenter combined inking for emphasis with a simple feature of Presenter to develop a new discussion style. His discussion of a slide would focus on certain features of the material (emphasizing these in ink with highlights, circles, or other marks), next he would erase the ink with the "chalkboard eraser" button in Presenter's top toolbar, and then he would discuss the slide again from a different perspective and with different markings. The rapid erase feature enabled this new style and tempo of discussion.

### Culling Participation from the Class

Next we show an example where the class will be shown two graphs and asked to propose various conclusions that can be drawn. Figure 4(a) shows the instructor view before class discussion, Figure 4(b) shows the instructor view after

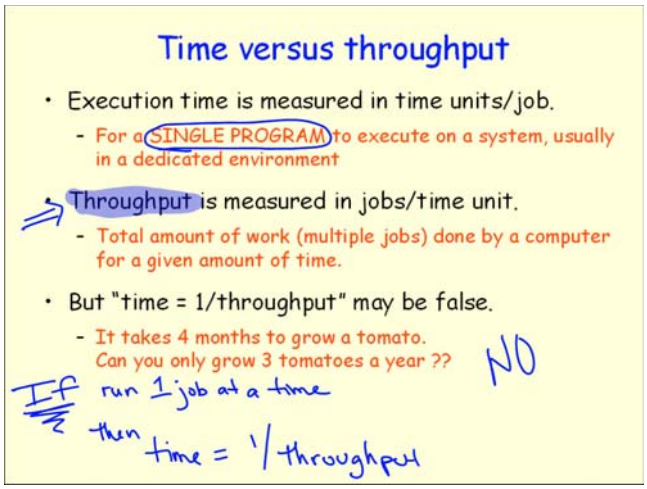


**Figure 3(a). Projector view with key points emphasized via circling and highlighting. Additional notes at bottom of screen emphasize what was, hopefully, made clear in verbal lecture.**

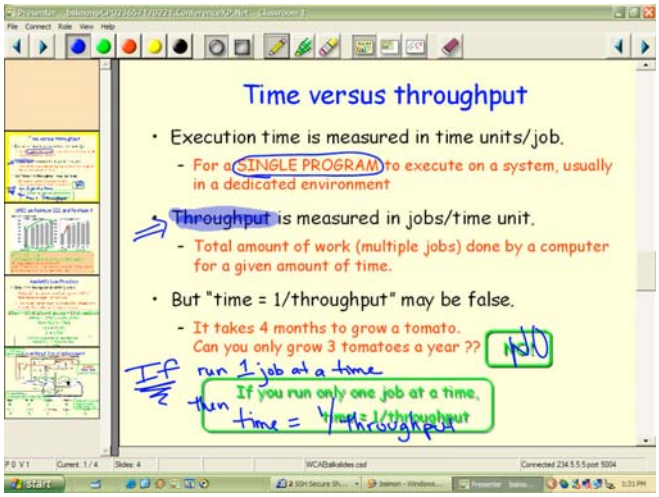


**Figure 3(b). Instructor view after discussion. The rounded text boxes are instructor objects, not seen on the projected display. These objects can hold reminders of points to emphasize in class.**

class discussion, and Figure 4(c) shows the projector view after discussion.

This slide wraps up a discussion of benchmarking as a manner of evaluating performance. Students are encouraged to recall a previous concept then apply it to the given problem. Specifically, students are asked to explain why the doubling of the clock rate doesn't produce a doubling of performance (circles on the left graph remind the instructor where to draw student attention). Instructor notes at the bottom of the slide prompt the instructor to write, one more time, the ET = IC * CPI * CT equation and provide a color-coded reminder of the main topics students should bring up.

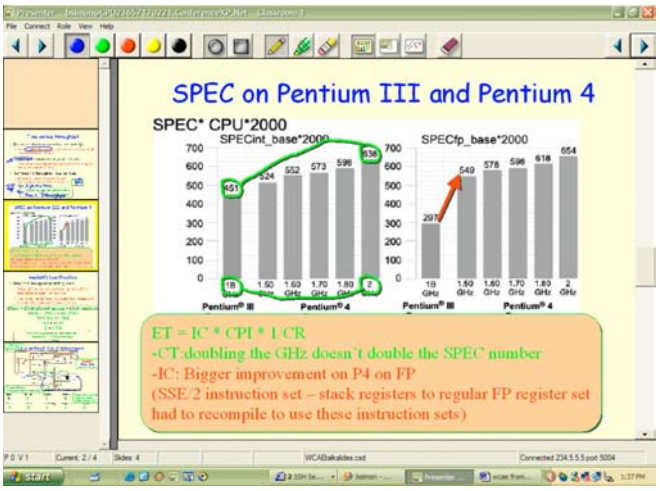


**Figure 4(a). Instructor view before discussion. The "hand drawn" circles and lines, arrow, and text box are instructor objects – not seen on projected slide.**

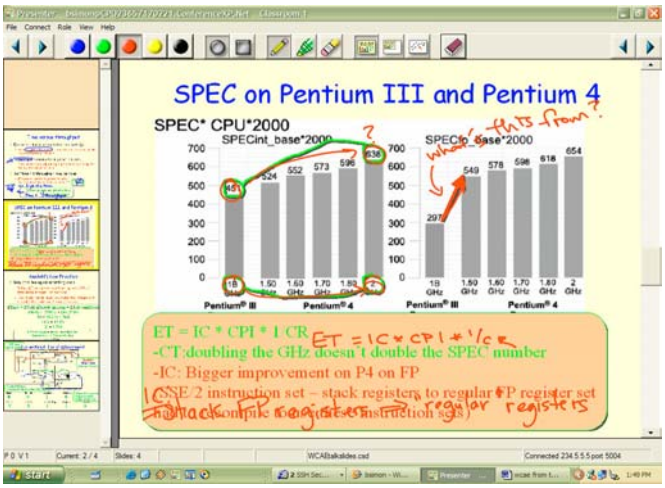


**Figure 4(b) Instructor view after discussion. In-class inking has occurred overtop of "instructor object" inking as issues are raised in class. Some "notes" at bottom have been "copied" for students.**

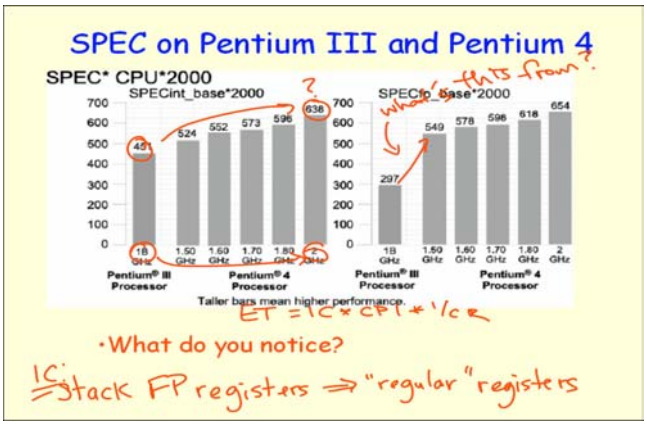


**Figure 4(c). Projector view after discussion. This is what the students see.**

Note that, in class, the instructor can "draw over" the circles and arrow instructor objects – either at the direction of an astute student, or as a hint to the class if no suggestion is forthcoming. If a student brings up some issue other than those "expected" by the instructor, the instructor is free to explore that topic, ignoring his own notes. If, after that discussion concludes, he wants to return to a "clean" version of the slide to discuss the planned topics, he can erase all ink at once using the chalkboard eraser icon on the top toolbar. If he wants to perform a partial erase of certain words, the pencil eraser erases ink one stroke at a time.

### Interactive, But Planned, Problem Solving

Figure 5 shows one example of interactive problem solving where the students can get a first experience with applying Amdahl's Law. In order to be sure to cover all the "basics", there are instructor notes to encourage the instructor to fully set up the equation and to relieve him of the need to concentrate on simple math. A separate instructor note shows an additional calculation that can be discussed or omitted as time allows. A scroll bar on the instructor view allows the instructor to "scroll up" the ink (as on an overhead projector) of the Amdahl's Law solution to provide additional room to solve the speedup equation (alternately, he can shrink the current slide to ¾ size and show additional work around the edges). In Figure 5(b), we see the projector version after the instructor has "scrolled up" some inked notes to solve an additional problem.

### Providing Unexpectedly Needed Review

"You all know how to convert from decimal to binary don't you?" When it becomes clear that one has misjudged the background knowledge of the class, Presenter allows one to easily "break out" of a planned lecture sequence. This can be done either by jumping to a backup slide (perhaps stored at the end of the slide deck and accessed through the filmstrip view) or to a blank "whiteboard" slide to provide a quick review or to recommend a reference for student use.

### Summarizing What We've Learned

Presenter can add new life to the usual "here's what's important from Chapter X" conclusion slides. Simply converting current summary bullets to an instructor object (not seen by students) can force students to take notes as the instructor "overwrites" key topics or allow the class to brainstorm their opinions of the most important material as in the "Empty Outline" Classroom Assessment Technique [2] pp.138-141.

### Adding an Instruction to a Single Cycle Datapath

The freedom of instructor mode objects (rather than just instructor mode text) is especially useful when explaining and modifying charts, graphs, or diagrams. The slide shown in Figure 6 was developed as an in-class review of a homework assignment where several students had produced confused answers. They had been asked to modify the

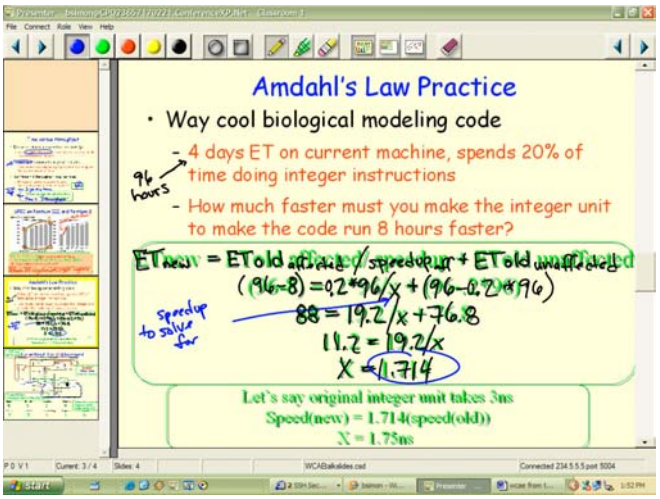


**Figure 5(a). Instructor view where a pre-planned practice problem has been solved in-class. Text in bubbles are instructor objects and do not appear on projected slide. Instructor can "trace over" instructor object equations to avoid skipping through solution too quickly. Notes in bottom instructor object bubble can be discussed or skipped as time permits.**

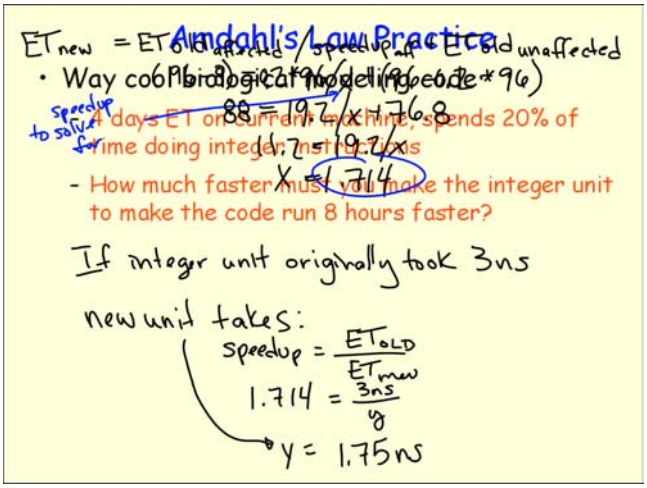


**Figure 5(b). Projector view of 5(a) after "scrolling up" first equation work to make room for new discussion. This new ink addresses the bottom instructor object bubble in 5(a).**

MIPS single cycle datapath (developed in Patterson and Hennessey, chapter 5) to support only lw and sw instructions that had no displacement. In Figure 6(a) we see the instructor view before discussion. The datapath has been "drawn over" with instructor object lines in different colors. Notes are scattered around the slide as reminders of the format of the instruction and as a guide to an ordering for discussing the datapath modifications. In class, we see (Figure 6(b)) that the instructor has inked over the instructor objects as discussion of the problem progresses. Figure 6(c) shows the much less cluttered projected version.

## 4. Upcoming Features and Future Work

### Additional Instructor Control Features

Currently, all instructor notes must fit within the available real estate on the slide, and the built-in notes from PowerPoint are ignored. The on-slide notes offer extra flexibility—shapes, figures, etc. as notes rather than just text; however, off-slide notes would ease space management issues for instructors. In future versions, an optional extra pane will display PowerPoint notes.

Furthermore, instructor notes are currently distinguished from public slide elements by a shadow beneath the note, allowing instructors to quickly determine which slide elements students see and which are invisible to them. However, some instructors prefer other mechanisms such as making instructor object text all one color, or using only the rounded box shape for notes. We plan to support a broader range of mechanisms for distinguishing instructor notes.

### Instructor-Only Inking

Future plans for "instructor-only" inking would allow instructors to make private inked notes during class (visible only on the instructor tablet). These notes could reflect anything that pops into mind in class that the instructor wants to remember afterward. Possibilities include comments on the efficacy of certain slides, points of confusion, and possible homework or test problem ideas.

### Tablets for Student Interaction in the Small Classroom

While in-class group problem solving is often viewed by students as very instructional, instructors often feel the class time spent on these group activities comes at the cost of lecture presentation time. Specifically, if students are to benefit from the evaluation of other students' work, then one must ask different groups to present their results to the class in some form (at the board, etc.). This adds yet more time to the group activity.

Future versions of Presenter will support an additional, "student view" that will be wirelessly transmitted to Tablet PCs scattered throughout the class. While not every student may have a tablet, for the purposes of group work, each group (or a subset of groups) can be given a tablet on which to record their work. The instructor tablet will have a method of previewing the various student tablet screens, and selecting one to be projected by the data projector. In this way, a group can "show their work" instantaneously. The group can be asked to describe their work, possibly using additional ink (circling, etc) to emphasize points in the discussion.

Some of the things students will be able do include working on a "blank screen", solving a problem proposed on a slide, modifying a datapath diagram, or filling in an empty cache.

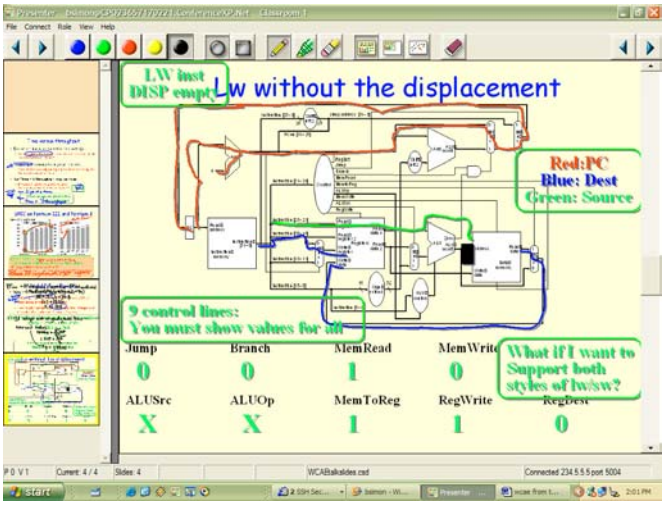


**Figure 6(a). Instructor view before discussion. Many instructor objects are present including text boxes, lines drawn over the datapath, and values for control lines.**

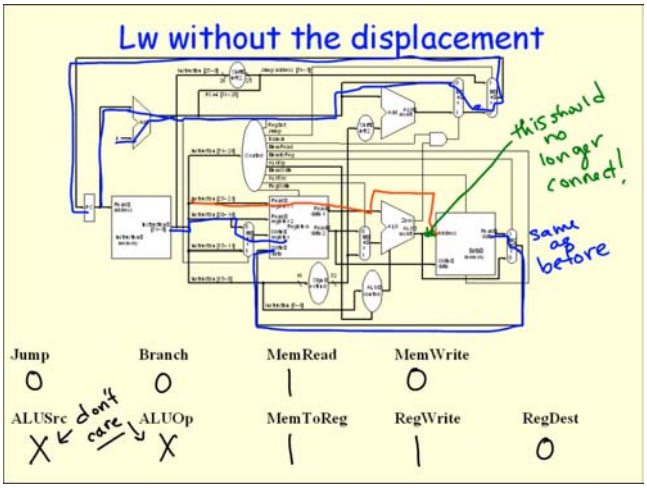


**Figure 6(c). Projector view of modifications to support lw without displacement. Pen color changed to show which parts of the datapath had to be altered to accommodate the new instruction.**

### Wireless Data Projectors

While current implementations require a separate machine to drive the data projector (so as not to tether down the instructor), future versions may discard this requirement through the use of wirelessly enabled data projectors. While still in their infancy, wireless data projectors could set up a connection with the instructor tablet to project only the projector view version of a slide.

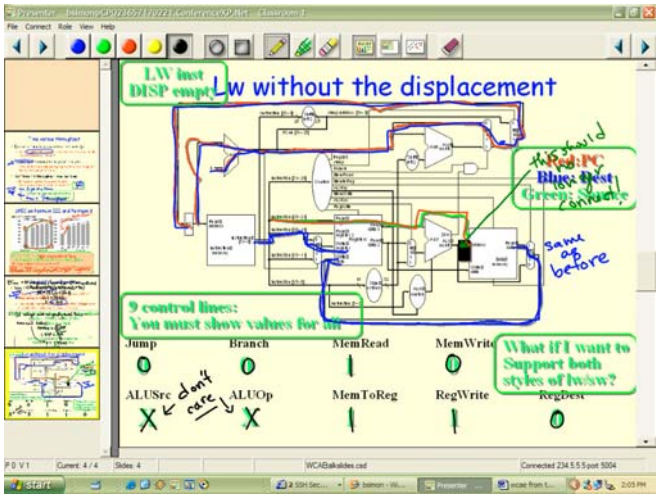


**Figure 6(b). Instructor view after discussion. Datapath lines have been "inked over" in the order recommended in an instructor object text note. The class was asked to supply control line values, which could be "checked" against instructor note values.**

## 5. Related Work

There have been a number of related efforts to deploy technology in the classroom to enhance learning, and to capture the lecture for later playback. eClassroom (formerly Classroom 2000) [1] is a premier project for incorporating technology in the classroom to facilitate note taking, capture, playback, and presentation. While eClassroom includes some effort to improve presentation facilities for the instructor, our work focuses directly on this aspect. Classroom Presenter also differs from eClassroom in that our goal is to deploy in a general data projector-enabled classroom, as opposed to basing our design on a dedicated facility. The Pebbles system [10] was one of the first projects to explore steering a presentation from wireless devices. Our emphasis on writing as part of presentation relates to the broad literature on pen computing and electronic whiteboards [11]. In this stage of our work we are not attempting a semantic interpretation of the ink, as in the Back of the Envelope project [6]. Work on zoomable interfaces has relevance to our work, both in

suggesting alternative ways to view the display surface and the presentation [5].

The importance of actively involving students in classroom activities at regular intervals is supported by studies on student attention spans [2][13]. There has been some educational work attempting to evaluate the use of PowerPoint in university classrooms [7][8][14]. The overall results have been ambiguous with respect to learning outcomes, but the papers have some perceptive comments on the use of PowerPoint and indicate a favorable student response. There has also been an active debate on the lecture style supported by slides with polemics on all sides [15][12][4].

## 6. Conclusions

This work describes the Classroom Presenter lecture presentation system developed at the University of Washington. We highlight some of the specific utilities of the system in the context on an undergraduate computer architecture course. The main benefits of Presenter stem from wireless, high-quality inking over slides during lecture combined with a separation of views between instructor and projector. These features have enabled high levels of spontaneity and interactivity in an undergraduate architecture class. Specifically, the ability to use instructor-only visible objects to annotate diagrams and graphs can encourage the instructor to develop designs jointly with the class rather than presenting them as problems already solved.

## References


[1]Abowd, G. D. Classroom 2000: an experiment with the instrumentation of a living educational environment. *IBM Systems Journal*, 38(4), 1999.

[2]Angelo, Thomas A. and Cross, K. Patricia. *Classroom Assessment Techniques*. Jossey-Bass Publishers, San Francisco, 1993.

[3]Bligh, D. A. *What's the use of lectures?* Jossey-Bass Publishers, San Francisco, 2000.

[4]Creed, Tom. PowerPoint No, Cyberspace Yes. The National Teaching & Learning Forum, 1997, http://www.ntlf.com/html/sf/cyberspace.htm

[5]Good, Lance and Bederson, Benjamin B. CounterPoint: Creating Jazzy Interactive Presentations. HCIL Tech Report #2001-03. University of Maryland, College Park, MD 20742, 2001.

[6]Gross, Mark D., and Do, Ellen Yi-Luen. Drawing on the Back of an Envelope: a framework for interacting with application programs by freehand drawing. Computers & Graphics, 24 pp. 835-849, 2000.

[7]Hozl, J. Twelve tips for effective PowerPoint presentations for the technologically challenged. Medical Teacher, 19, 175-179, 1997.

[8] Lowry, R. B. Electronic presentation of lectures -- effect upon student performance. University Chemistry Education, 3 (1), 18-21. 1999.

[9]McConnell, Jeffrey J. Active Learning and Its Use in Computer Science. SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education (Barcelona, Spain June 2-5, 1996), also published as SIGCSE Bulletin, Vol. 28 Special Issue, 1996, pp. 52-54.

[10]Myers, Brad A. and Stiel, Brad A. and Gargiulo, Robert. Collaboration using multiple PDAs connected to a PC. In *Proceedings of CSCW'98: ACM Conference on Computer-Supported Cooperative Work*, pages 285–294, November 1998.

[11]Mynatt, E., Igarashi, T., Edwards, W. K., and LaMarca A. Flatland: new dimensions in office whiteboards. Proceedings of ACM Human Factors in Computing (CHI 99). New Your: ACM, pp 346-353. 1999.

[12]Rocklin, Tom. PowerPoint is Not Evil! National Teaching and Learning Forum Newsletter, 1997. http://www.ntlf.com/html/sf/notevil.htm

[13]Stuart, John and Rutherford, R. J. Medical Student Concentration During Lectures. The Lancet, September 2, 1978, pp. 514-516.

[14]Szabo, Atilla and Hastings, Nigel, Using IT in the undergraduate classroom: should we replace the blackboard with PowerPoint? Computers & Education, 35 175-187, 2000.

[15]Tufte, E., "The Cognitive Style of PowerPoint", www.edwardtufte.com, 2003.

[16]Vygotsky, L.S. Mind in Society. Cambridge, MA: Harvard University Press, 1978.


## Appendix: Classroom Presenter Feature List

**The Filmstrip.** The Filmstrip is a "preview strip" of slides surrounding the currently displayed slide. The filmstrip facilitates non-linear slide navigation order, which can allow faculty to react to the development of ideas and discussion in class.

**Filmstrip preview.** Another benefit of the filmstrip view is to allow the instructor to make appropriate concluding comments based on the content of the next slide to be displayed. Filmstrip preview makes this more viable by showing an instructor-visible "zoomed in" version of a filmstrip slide when the tablet pen is "waved" over a portion of the filmstrip.

**The Toolbar.** Back and forward buttons are placed at both ends to allow easy movement one slide in either direction of the current slide. Four inking colors are supported as well as two ink "tips": a square and a round tip. The next three buttons select "regular pen" inking, highlighting, or stroke-based erasing. The following three buttons control the view: full slide, ¾ size slide, or blank whiteboard (you can

jump from the whiteboard back to your current slide by clicking on the “full slide” button). The chalkboard eraser erases all ink on the current slide. An upcoming feature will be “undo” which can undo the last selection (particularly important for the full erase).

ACKNOWLEDGMENTS: We thank the University of Washington CS Education and Educational Technology Group for their support and insights.

# The Liberty Simulation Environment as a Pedagogical Tool

Jason Blome Manish Vachharajani Neil Vachharajani David I. August
Departments of Computer Science and Electrical Engineering
Princeton University
{blome, manishv, nvachhar, august}@cs.princeton.edu

## Abstract

*This paper describes how the Liberty Simulation Environment (LSE) and its graphical visualizer can be used in a computer architecture course. LSE allows for the rapid construction of simulators from models that resemble the structure of hardware. By using and modifying LSE models, students can develop a solid understanding of and learn to reason about computer architecture. Since LSE models are also relatively easy to modify, the tool can be used as the basis of meaningful assignments, allowing students to explore a variety of microarchitectural concepts on their own. In lectures where block diagrams are typically displayed, LSE's visualizer can be used instead to not only show block diagrams, but to demonstrate the machine in action. As a result, LSE can ease the burden of conveying complex microarchitectural design concepts, greatly improving the depth of understanding a computer architecture course provides.*

## 1. Introduction

The goal of a computer architecture course is to teach students how microprocessor hardware operates and to give them an opportunity to experiment with microprocessor design. To do this, the students need to learn about existing microarchitecture design techniques ranging from simple concepts such as pipelining to advanced organizations such as out-of-order machines including features such as speculation, branch prediction, and register renaming.

Unfortunately for the student, the expanse of the computer architecture design space is vast and complex. Within a single design, each component plays an important role in facilitating the correct and efficient execution of a program, however, correct execution is only ensured when interaction of all the components is carefully orchestrated. These component interactions are often subtle making it difficult to convey enough information in lecture to foster a deep understanding.

To remedy this, courses are augmented with periodic assignments that encourage the students to discover some of the complexities on their own. Typically, these assignments involve drawing pipeline sketches and evaluating block diagrams. Unfortunately, these assignments do little to foster an understanding of the complex dynamic interactions and instead reinforce the students' understanding of the architecture's high-level structure.

An approach that leads to a much better understanding of a microarchitecture is to have the students design the hardware for a machine and run programs on it in a simulated environment. This way students will discover, on their own, the intricacies of how different parts of the architecture interact to facilitate the correct program behavior. The insight gained from this process of design, test, and debugging is often deeper than any knowledge gained in lecture. Unfortunately, specifying the hardware, even in a synthesizable hardware description language like VHDL or Verilog (as opposed to a gate level description), can take many weeks. As a result, when this method is employed, it is typically limited to a single course-length project. While a project of this type is better than no design experience, it only provides insights about the techniques employed in one particular design, which is often a small subset of the techniques taught in the class.

A promising alternative to specifying the hardware is using, building, and modifying higher-level simulation tools for the microarchitecture. Assignments could consist of asking students to modify a simulator to incorporate new behavior. Unfortunately, current simulation environments are too difficult to modify to make this practical for periodic homework assignments. Furthermore, the most common method of building a simulator (coding it by hand in C or C++) does little to convey the actual hardware structure or the functionality of its components [1]. Consequently the process of reasoning about and building the simulator is very different from the way in which a computer architect would design a microprocessor thus making it unsuitable as a pedagogical tool. Students should think like architects, not like simulator writers.

To be effective for use in assignments, the simulation system should have simulator descriptions that reflect the hardware being modeled. Components used in modeling should correspond to hardware blocks and they should be interconnected via communication channels like hardware blocks. On the other hand the simulation environment should not impose, upon the student, the burdens that hardware description languages like VHDL or Verilog often do. Instead, it should allow rapid construction and modification of models so that working with an executable model can be a part of

regular assignments.

LSE is a simulator construction tool that meets the requirements outlined above. In this paper we give an overview of the Liberty Simulation Environment (LSE) and describe how it can be used in a course to enhance student understanding of computer architecture. In the next section we describe the Liberty Simulation Environment. In Section 3 we describe the LSE visualizer and how it visualizes the LSE descriptions. Then, in Section 4 we give specific examples of how to use LSE in a course. Finally, we conclude in Section 5.

## 2. The Liberty Simulation Environment

The Liberty Simulation Environment (LSE) is an excellent tool for students to learn about and explore microarchitecture. LSE descriptions resemble the hardware they model and are easy to modify. This section describes enough about LSE so that it is possible to understand how LSE can be used for instructional purposes. Details of how LSE enables rapid specification while still resembling the modeled hardware as well as details of all the concepts described in this section can be found elsewhere [1].

As shown in Figure 1, LSE consists of three main parts: the Liberty Structural Specification Language (LSS), a component library, and the Liberty Simulator Constructor. To use the system, the user describes a machine by specifying, in the LSS language, a set of interconnected instances of components. These components, called *modules*, are typically taken from the module library although custom modules can be created if necessary. The user then invokes the simulator constructor, and the constructor reads the specification and the code from the module library and builds an executable simulator for the described machine. This section will discuss in more detail the properties of modules, module communication, and collection of data from a run of the simulator executable.

### 2.1. Modules

Each module is a parameterized template that is instantiated in an LSS machine description to create a *module instance* (or simply instance). Much like components in hardware design, modules can be leaves of a hierarchy, or they can be constructed hierarchically by grouping collections of other interconnected module instances. Like hardware blocks, module instances execute concurrently [2] and communicate with other instances by passing data across communication channels.

However, unlike hardware design, the details of instance behavior (hierarchical or leaf) can be customized via module parameters. When a user instantiates a module in a machine description, the user specifies values for the parameters declared by the module (or accepts the default value specified by the module). These parameters are used to customize the behavior of the module instance for the particular description. Parameters can control simple configuration options (e.g. the cache line size or whether a 2-level branch predictor has a global or per-address predictor table). Further, parameters can also be used to allow control of algorithms allowing users to customize complex behavior. For example, the branch predictor has a parameter that allows users to override the predictor state-machine code to implement a custom predictor if none of the provided predictor options is suitable. Parameters can even control the instantiation of hardware structures in lower levels of the hierarchy.

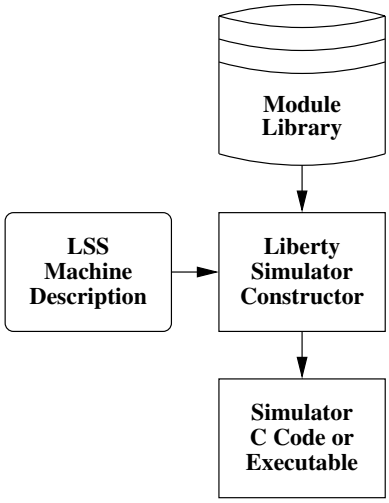


Figure 1: Overview of main LSE components.

### 2.2. Communication Channels

Modules specify a communication interface for module instances by declaring *ports*. Each instance may have one or more *port instances* per port. Each port instance is a communication channel and may have exactly one value sent on it per cycle. For example, the register file module may have an input port on which register read requests are made. Each port instance would accept one register read request per cycle. If two register reads per cycle were needed, there would be 2 port instances of the register read request port, and two instances of the output port on which these read requests were returned. Another example is the `tee` module which is used to fanout a given value to multiple receivers. The `tee` duplicates the value received on its input port instances on multiple output port instances.

Users specify how modules communicate by interconnecting port instances from one module to port instances on the same or other modules. While details regarding ports and the communication system can be found in [1], other work describes LSE's execution and messaging semantics [2].

### 2.3. Data Collection

To allow modularity and flexibility even for data collection, LSE provides a data collection mechanism that avoids

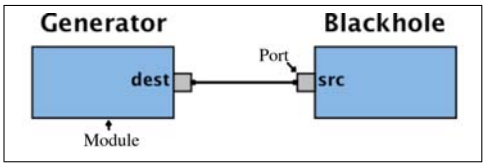


Figure 2: Simple source to sink description.

the pitfalls of embedding instrumentation code directly into the simulator code. Each LSE module may declare that its instances emit certain *events* during the execution of the simulator. Each event includes data related to the event and a dynamic identifier (dynID) that represents the system level object that caused the event to occur. Orthogonal to the declaration of events, users may associate, with any event, a *data collector* which captures the event and records data or computes statistics.

For example, a branch predictor module may emit an event every time it makes a prediction. The event could include information about what prediction was made and what predictor made that prediction. The dynID for the event would identify the dynamic instruction instance that caused the prediction to occur. A user could hook this event with a data collector to count the number of predictions made or to calculate a branch misprediction rate for example.

In addition to recording data or computing statistics, the LSE visualizer (described in Section 3) hooks these events to visualize the flow of data through the machine at runtime. An example of this is described in Section 4.

## 3. Visualization

The LSE Visualizer provides a means to view LSS descriptions as block diagrams. In addition, the LSE Visualizer provides an interface for compiling an LSS design into an executable simulator, and it provides tools for observing, via animation, the execution of the simulator. In this section we will describe the visualizer in enough detail so that it is possible to understand the discussion of how the visualizer can be used in a computer architecture course as described in Section 4.

Figure 2 is an LSE Visualizer screenshot showing the block diagram of a simple system. In this system, a module instance, called Generator, sends data to another module instance, called Blackhole, which discards it. Generator is an instance of the `datasource` module, and Blackhole is an instance of the `sink` module. Both the `datasource` and `sink` modules are provided by the LSE module library. During simulator execution, a unit of data will be transfered from Generator to Blackhole in each and every cycle until the simulation is terminated manually.

In Figure 2 the module instances are represented by the large boxes, while their ports are represented by the small boxes. The single line between the box labeled `dest` and the box labeled `src` represents a connection between the respective port instances on the module instances Generator and Blackhole.

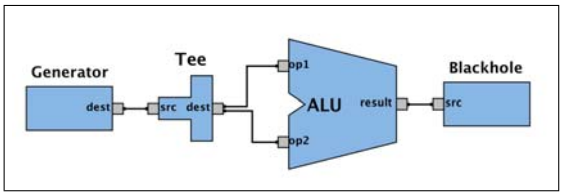


Figure 3: Simple $x + x$ description.

Figure 3 shows a screenshot of a slightly more sophisticated machine configuration. Here, the `datasource` module instantiated as Generator is connected to an instance of the `tee` module named Tee. The Tee in turn fans out the data originating from Generator into port instances of ports `op1` and `op2` of the instance ALU. ALU then computes the sum of the values passed into it on these port instances and sends the result to Blackhole to be thrown out. The function of this machine is simply to compute the value of $x + x$, where $x$ is the value generated by Generator, and then throw away the result.

In this diagram, there are a few interesting features to note. First, notice that Tee's `dest` port is connected twice, meaning that there are two port instances of the `dest` port. Also notice that the Tee and ALU instances have been given custom shapes for their visual representation. In general each module instance can be given a custom shape. This feature allows the visualized modules to be recognizable on sight instead of having each module be a nondescript blue rectangle. We use this feature in the next section to make the machine visualizations resemble diagrams found in computer architecture textbooks.

In addition to the above schematic rendering features, the visualizer can interface the generated simulator executable and display execution information as it occurs. This is useful for following instructions as they flow through a pipeline or observing the status of ports in the system. Screenshots of the visualizer interacting with the generated simulator are shown in the next section.

## 4. Applications

In this section, we will give examples of how LSE can be used in lecture to illustrate computer architecture concepts and how LSE can be used to formulate assignments that allow students to explore the myriad of interactions between architectural techniques. All the examples are centered around a simple Tomasulo-style [3] machine that executes the DLX [4] instruction set.

### 4.1. LSE in the Classroom

Standard presentation tools do a fine job of illustrating the static aspects of a design. However, dynamic interactions are generally only briefly described or illustrated with static pipeline diagrams. The LSE system and its visualizer, however, can demonstrate the *dynamic* behavior of the ma-

chine by displaying, over time, events produced by the executable machine model. In a lecture environment, this can be used to show the flow of data and update of state through the modeled architecture.

To illustrate this we will show a few screenshots of the Visualizer showing the flow of instructions through a simple Tomasulo-style pipeline. A screen shot of the Visualizer is shown in Figure 4. Notice that the structure of the machine is fairly obvious. The block labeled `Register File` is the register file, horizontally stacked tall vertical blocks are the reservation stations, the ALU looks like an ALU, and the shifter, LSU, and branch unit are clearly labeled. The machine does not support precise exceptions or speculation and so there is no need for a reorder buffer.

Figure 5 shows a screen shot of the visualizer displaying a table showing instruction arrival at various stages in the pipeline and reservation station occupancy of the Tomasulo-style machine shown in Figure 4. This table is dynamically updated with data from the running simulator.

In Figure 5 we can clearly see the `or` instruction that succeeds the jump instruction stall in the fetch stage starting at cycle 5 while the machine resolves the branch (recall that the sample Tomasulo-style machine does not support speculation). We can also see the `sll` instruction stuck in the reservation station awaiting operands during cycle 3. In cycle 4, we see the `sll` instruction issue to the EX stage but subseqently lose arbitration for the common data forcing a re-issue in cycle 5. The instruction once again loses arbitration and re-issues in cycle 6 and finally writes back in cycle 7.

The table is constructed by specifying the appropriate data collectors in the simulator description. The visualizer then monitors the output of these collectors to generate the table as the simulator runs. The table is completely generic and thus users of LSE may specify the column headings and how the table entries get filled via the specific messages emitted by the data collectors.

As discussed in the literature, the power of this kind of demonstration is invaluable since both the static machine structure and its dynamic behavior can be seen simultaneously [5]. With the Liberty Simulation Environment, these demonstrations can easily be constructed for many different types of machines so that students can easily understand the differences in the architectural techniques presented.

### 4.2. LSE for Student Exploration

As was described in Section 1, designing and implementing a machine with an RTL level description is too time consuming to do regularly throughout an architecture course. On the other hand, block diagrams do little to cement understanding of the dynamic elements of an architecture. The Liberty Simulation Environment, however, provides an excellent middle ground between hand-drawn hardware block diagrams and RTL level descriptions. Regular assignments can be given in which the student is required to modify an existing configuration to produce a new configuration. For example, students may be asked to modify a Tomasulo-style machine that does not execute loads and stores to one that does execute loads and stores in order, in 2 clock cycles. As the following example will demonstrate, this problem is certainly tractable for students in a week long assignment.

Figure 4, described earlier, shows a Tomasulo-style machine that does not execute loads and stores. Figure 6 shows that same machine with a load store unit added. Adding this load store unit is relatively straightforward.

First, the reservation station module needs to be augmented to force instructions to be issued in order. This augmented module will form the load-store issue queue (LSQ). This hierarchical module, shown in Figure 7 is built by taking the reservation station module and connecting it so that all the slots of the reservation station go to a `serializer` module, called `serialize` in the figure, followed by an `aligner` module, called `align`. The `serialize` and `align` instances, combined with the default control semantics in LSE, force instructions to come out of the `align` instance in order. Both the `serializer` and `aligner` are available in the standard LSE module library, and the reservation station is part of the original Tomasulo-style configuration.

Next, several additional module instances (created from modules in the library) are connected to the output of the LSQ and are used to extract the destination register (rd) from the data output by the reservation station, compute the load or store address, and generate the control signal that decides if the request will be a read or write. The specific fields that the module instance extracts and the function it performs are specified via algorithmic parameters (discussed in Section 2) provided by the modules.

The output of these module instances is then connected to a latch to end the first cycle of memory instruction execution. The output of this latch is then connected to the request ports of the data memory. Another module instance then combines the output of the data memory (generated for load requests) with the destination register field from the reservation station (arriving via the latch) and sends them off to the common data bus arbiter.

All of these modifications were performed in a few hours. Students moderately familiar with LSE should be able to complete such an assignment fairly easily. Furthermore, in the configuration just described, the default control semantics in LSE would allow students to vary the latency of the memory module (while keeping the initiation interval fixed at 1) and have the load-store logic stall waiting for the memory. Students could then explore the merits of their own design in the presence of different core-memory latencies with very little additional effort.

When used in this way, LSE enables instructors to give regular assignments that require students to build executable

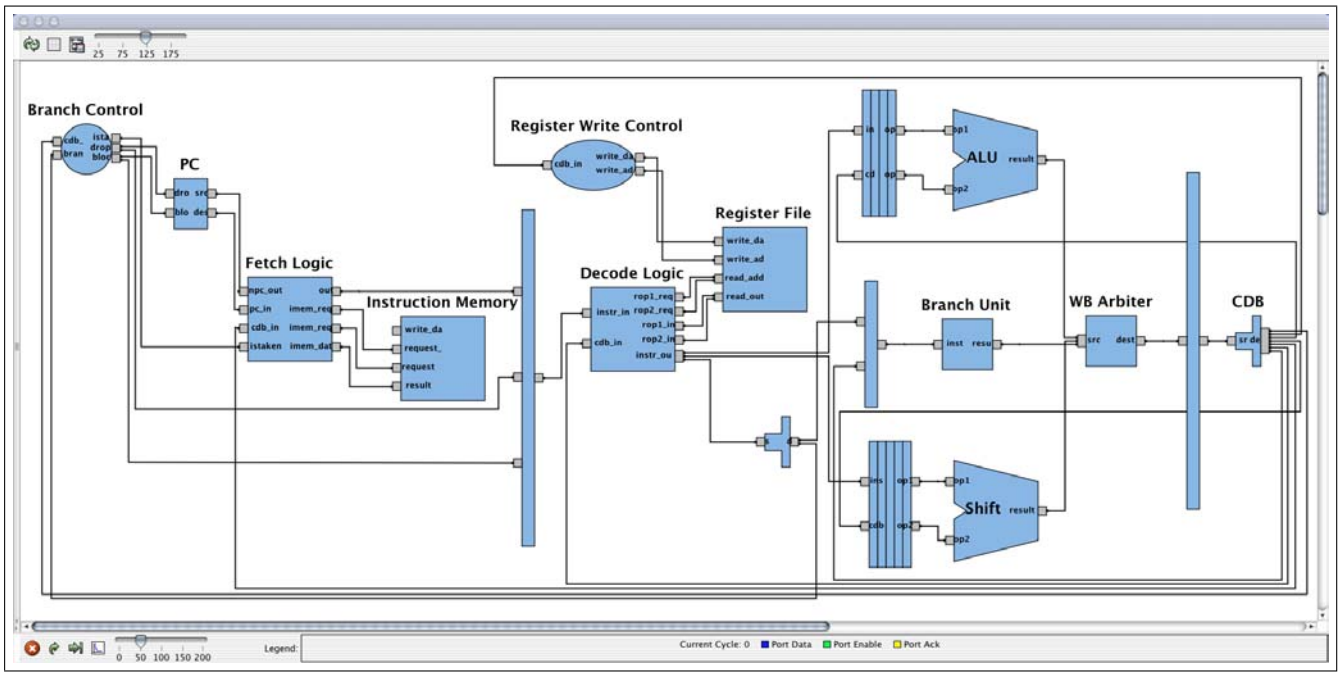


Figure 4: Simple Tomasulo-style pipeline that executes the DLX ISA.

| Cycle | IF | ID | ALU_res | Shifter_res | Branch_res | EX | WB |
|---|---|---|---|---|---|---|---|
| 0 | addi(0) | | | | | | |
| 1 | sll(1) | addi(0) | | | | | |
| 2 | add(2) | sll(1) | addi(0) | | | addi(0) | |
| 3 | lhi(3) | add(2) | | sll(1) | | | addi(0) |
| 4 | j(4) | lhi(3) | add(2) | sll(1) | | add(2) / sll(1) | |
| 5 | or(5) | j(4) | lhi(3) | sll(1) | | sll(1) / lhi(3) | add(2) |
| 6 | | | | sll(1) | j(4) | j(4) / sll(1) | lhi(3) |
| 7 | | | | | j(4) | j(4) | sll(1) |
| 8 | | | | | | | j(4) |

Figure 5: Table showing instruction arrival and reservation station occupancy in a Tomasulo-style DLX machine

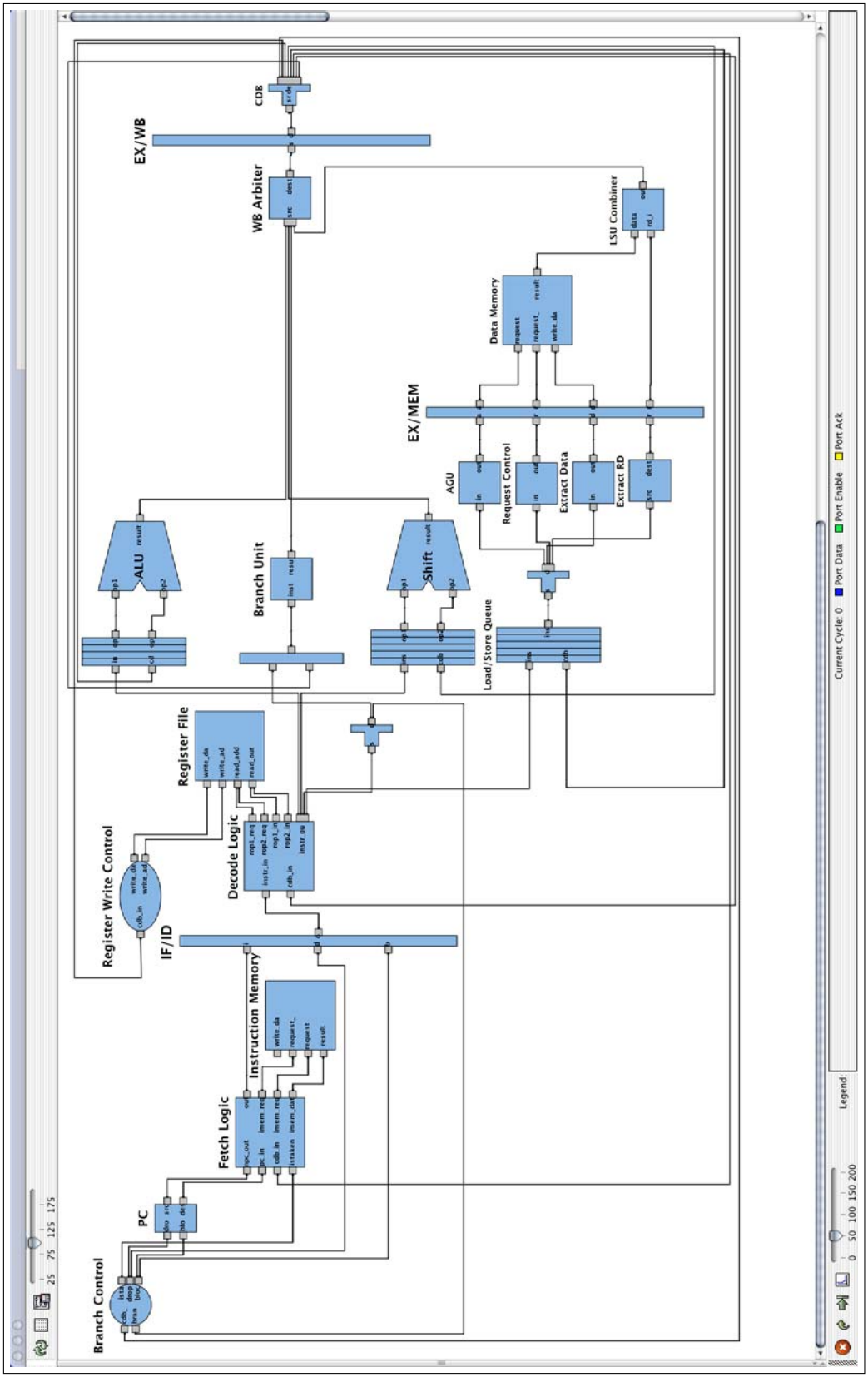


Figure 6: A Tomasulo-style DLX machine with a load-store capability.

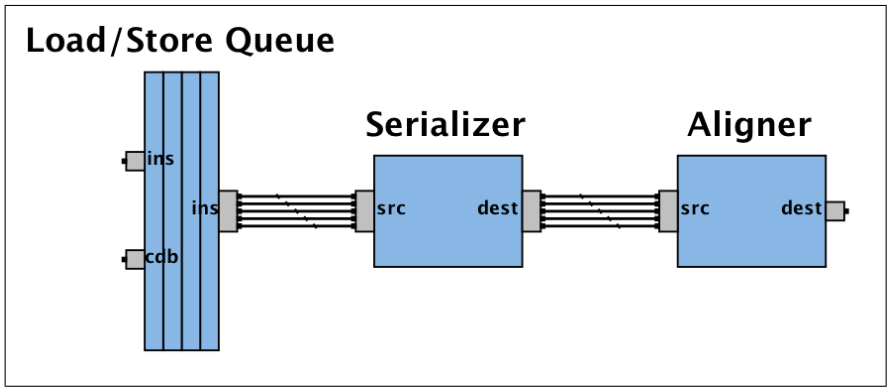


Figure 7: Load-Store issue queue.

models to verify that their understanding of an architectural concept is sufficient (i.e. the model runs programs correctly). The students can learn about most of the techniques presented in class with hands-on projects, instead of only a handful they would see by doing a single class project.

As a further example, students can explore machines not described in lecture by having them add architectural mechanisms to existing designs. For example, students could be asked to add rename logic to a scoreboarded machine before discussing advanced scoreboarded machines. In this way students can appreciate the relationship between renaming and WAR hazards and why scoreboards stall in circumstances where Tomasulo's machine does not. LSE makes this kind of exploration feasible in week long assignments.

## 5. Conclusion

To understand computer architecture, students must understand the *dynamic* interactions of all the hardware components in a microarchitecture. Unfortunately, conveying the many subtleties of this interaction during lecture is difficult. For many students, *static* illustrations and assignments do not build intuition about dynamic systems. Class projects which require students to build RTL simulation models are extremely useful, but the overhead in low-level model construction and modification often limits the scope of concepts explored. Modifying or writing a high-level simulator in a sequential language such as C allows students to avoid getting bogged down in irrelevant low-level hardware details, but it does so by obscuring the model. Students spend much of their time dealing with sequential language simulator issues rather than thinking about computer architecture.

In this paper, we have shown that the Liberty Simulation Environment (LSE) is an alternative to tools currently used in lecture and take-home assignments. LSE's simulator description resembles hardware, allowing students to think about hardware rather than simulator design issues. LSE descriptions are relatively easy to modify and use, allowing students to study the dynamic execution behavior of a wide range of machines. Furthermore, the LSE Visualizer improves LSE's use as a pedagogical tool by tying this all together with an easy to use dynamic and graphical visualization system.

## References

[1] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, "Microarchitectural exploration with Liberty," in *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.

[2] D. Penry and D. I. August, "Optimizations for a simulator construction system supporting reusable components," in *Proceedings of the 40th Design Automation Conference*, June 2003.

[3] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, January 1967.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantatative Approach*. San Francisco, CA: Morgan Kaufmann, 1996.

[5] C. T. Weaver, E. Larson, and T. Austin, "Effective support of simulation in computare architecture instruction," in *Proceedings of the 2002 Workshop on Computer Architecture Education (WCAE)*, May 2002.

# Multimedia components for the visualization of dynamic behavior in computer architectures

Peter Marwedel, Birgit Sirocic
Dept. of Computer Science,
University of Dortmund,
44221 Dortmund, Germany
peter.marwedel@udo.edu

## Abstract

*Understanding modern processors requires a good knowledge of the dynamic behavior of processors. Traditional media like books use text for describing the dynamic behavior of processors. Visualization of this behavior, however, is impossible, due to the static nature of books. In this paper, we describe multimedia components for visualizing the dynamic behavior of hardware structures, called RaVi (abbreviation for the German equivalent of "computer architecture visualization"). Available RaVi components[1] include models of a microcoded MIPS architecture, of a MIPS pipeline, of scoreboarding, Tomasulo's algorithm and the MESI multiprocessor cache protocol.*

## 1 Introduction

The presented project aims at facilitating understanding the dynamics of modern processor architectures, thereby overcoming an important limitation of books. Videos tapes and video distribution techniques have made it possible to show non-interactive media elements to students. However, video tapes have to be accepted by teachers and users on an "as-is" basis. It is not possible to use instruction streams other than those employed for the production of the video. Also, it is not possible to modify hardware structures in order to see the effect of hardware changes on the dynamic behavior. In short, videos are very inflexible and cannot provide interactiveness (except the simple type of interactiveness possible with DVDs).

Providing this interactiveness, however, is difficult, since it requires the simulation of hardware structures. This can be a challenging task which cannot be solved within the time-frame available for preparing a course. Why not just use available hardware simulators? These simulators are frequently designed for optimum simulation speed and complex design projects. Ease of use, excellent visualization and portability have normally not been top goals for simulator design. Also, powerful simulators are typically proprietary and come at high costs, preventing their widespread deployment to classrooms and into the hands of students.

Therefore, we tried to design RaVi models for the simulation-based visualization of the dynamic behavior of hardware architectures. In contrast to available models, emphasis is on visualization.

This paper is structured as follows: a short description of related work is provided in section 2. Section 3 describes the multimedia units developed so far. Section 4 discusses some design consideration regarding the availability and access-ability to various groups of students. Section 5 contains some of the results. The final section comprises a conclusion.

## 2 Related work

Simulators provide information about the dynamic behavior of computing systems. Plenty of simulators are available either commercially or in the form of public domain tools. They have been used for decades already. However, these simulators have hardly been designed for class room use. For such use, the limited resolution of screens must be taken into account. These days, it is also required that the simulators can be given into the hands of students. Expensive commercial simulators cannot be used for this reason. Also, feature-rich simulators requiring special training are not appropriate for this type of application. The target group for

[1]We gratefully acknowledge the funding of the RaVi-project (which is a subproject of the SIMBA-project) by the German ministry of research and development (BMBF).

this material includes first year students. It would be impossible to teach these students how to use a hardware description language: teaching the syntax of complex languages and to use tools such as Modelsim [6] would take too much time in a course which also has to cover a number of other important computer engineering subjects. The effort of generating models, for example in VHDL, should not be underestimated. Hence, most of the available simulators do not provide the required functionality. Notable exceptions include the JCachesim [1]. JCachesim is a simulator for cache architectures. However, JCachesim focusses on generating quantitative data (statistics etc.). In contrast, the work in this paper focusses of giving insight into how computer systems work.

## 3 Available multimedia units

### 3.1 Microcoded version of MIPS

Architectural models capable of executing a reasonable subset of some instruction set require a certain complexity of the model. The program counter, main memory, register file, ALU, control logic and a number of multiplexers all have to be included in the model. Otherwise, it would be impossible to demonstrate how instructions are executed. Many of these hardware components are connected. According to our observation, it is typically difficult for the students to understand how all the wires in a computer architecture are used. Also, the function of multiport memories seems to be a problem for students grown-up with von-Neumann languages.

Courses on computer architecture typically follow the sequence of Hennessy/Patterson's book for undergraduates [4]. Consequently, a microprogrammed version of the MIPS-machine is the first hardware structure which is introduced. It has to be introduced in such a way that students are able to comprehend how it works. We have therefore designed a multimedia unit highlighting the paths which are used during a certain micro-step (see fig. 1).

Just color-coding the values on the wires would lead to an abundance in color-coding and it would be difficult to find out, which of the lines are actually important. Therefore, only those paths leading to non-redundant inputs are marked. Lines printed in bold in fig. 1 correspond to the paths used in the final state of the store word instruction. The address input of memory Mem is driven by the computed effective address, as stored in temporary register T. The value stored is coming from the register file Reg. General simulators would typically not implement such a feature and would therefore create unnecessary barriers for the students.

Experimentation with this architecture is possible. For example, the contents of the register file Reg as well as the contents of the main memory Mem can be changed. A small dedicated assembler is provided such that assembly language programs can be assembled and loaded into the main memory.

Modifications of the wiring are possible, using the integrated schematic editor. A number of standard components are provided. These include multiplexers, registers, memories and ALUs. Modifications using these standard components can be done by the user without any programming. Adding new components not yet available in the library requires programming the behavior of these components in Java, however. This possibility is rarely used, except by the designer of the multimedia units.

### 3.2 MIPS-Pipeline

The operation of the MIPS-pipeline is described in the book by Hennessy and Patterson [4]. Several pages of the book are used for showing the different states the pipeline can be in. Nevertheless, this technique for explaining the operation of the pipeline has its limits: it is difficult to imagine, which situations arise for other code sequences. This is especially true for stall cycles. From available descriptions, it is difficult to understand which of the pipeline stages are stalled when.

Furthermore, it would be nice to use interactive elements in education. For example, students can be motivated to think about the behavior of the architectures by letting them "play around" with it.

All this is possible with RaVi models of the pipeline. There are essentially three models:

- The first model is a simple model without any bypassing. It can be used to demonstrate the wrong implementation of the instruction set.
- The second model includes bypassing, but does not have a separate adder for branches. This model can be used for demonstrating the advantage of bypassing.

  Also, this model implements two phase clocking. Using appropriate color coding, it can be shown that the register file is updated as a result of falling clock edges.

  The same model can be used to demonstrate the problems with branches if no special comparator for the instruction fetch stage and no special adder for calculating branch target addresses are added. Large branch delay penalties can be shown.

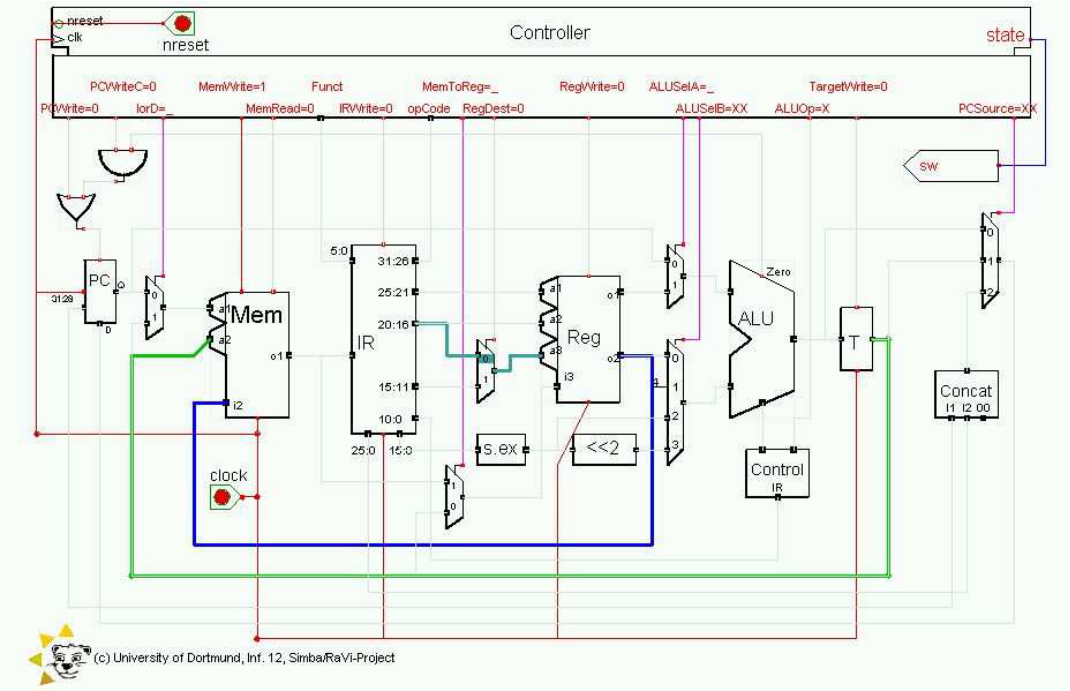


**Figure 1. Microprogrammed version of the MIPS machine (segment of a screenshot)**

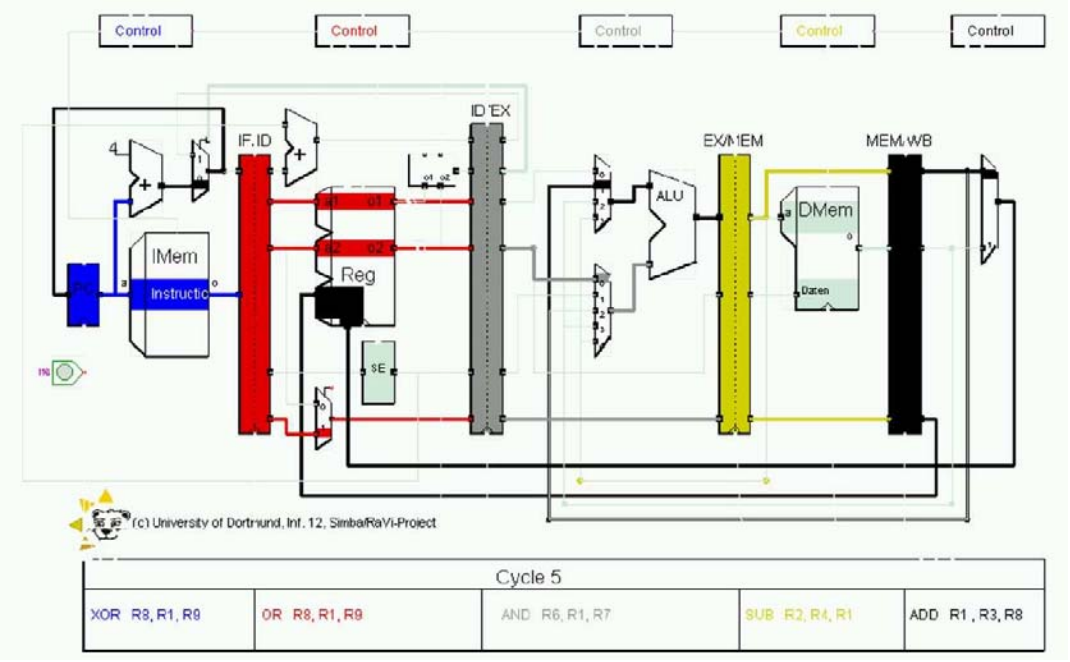


**Figure 2. Segment from screen-shot from pipeline unit**

- The third model includes the special hardware circuits for reducing branch delay penalties (see fig. 2).

All instructions are color-coded so that it is easy to see how instructions propagate down the pipeline. Implemented models support all major opcodes as well as mi-

nor opcodes in the register-to-register class (major opcode 0). A full implementation of all opcodes as well as exception handling is not consistent with the goal of keeping things simple so that students understand the models. Accordingly, function registers (like EPC) are not implemented. The same applies for special registers HI and LO. Irregular multiply instructions leaving their results in these registers (e.g. mult) have been replaced by their more regular pseudo instruction counterparts supported by the MIPS assembler (e.g. mul). Otherwise, too many hardware components would have to be on the screen.

### 3.3 MESI-protocol for a single cache block

The MESI protocol is typically included in the education of computer scientist in their third year. According to this protocol, single read requests for certain addresses cause the corresponding cache line to be in the exclusive state. Subsequent reads by other processors will cause the same cache line to be in the shared state. Writes in one processor will set the state in other caches to invalid. Due to its distributed nature, it is more difficult to understand than algorithms for mono-processors. We have therefore developed two multimedia units helping students to understand this protocol. The first unit shows the behavior of just a single cache block, of which copies may be available at four different machines (see fig. 3). The three state finite state machine used by Hennessy/Patterson [5] is replaced by the commonly used 4-state FSM.

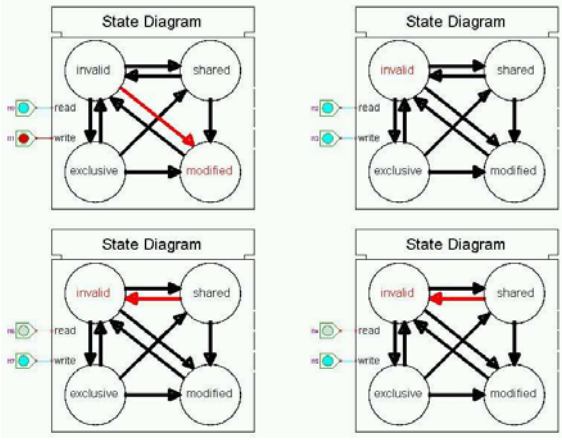


**Figure 3. RaVi visualization of the MESI protocol for a single block**

By generating read and write requests, the lecturer or the student can explain the behavior of these finite state machines. We found that students realized much faster that, once the shared state is reached, there is no way back to the exclusive state (in hardware, there is usually no signal which would allow going back to state exclusive except through state invalid).

### 3.4 MESI-protocol for the entire cache

After demonstrating the behavior of the four state MESI FSM for a single block, we are typically explaining the full MESI model for a number of cache blocks, also including tag bits. Fig. 4 shows a screen-shot of that model. Read and write requests can be generated interactively. Addresses and data for all read and write requests can be changed by using the context menue of the processors (shown at the top).

We found that students were surprised about the behavior of that model in case the same index bits but different tag bits are used in accesses to the different caches. Also, students did not expect the complexity of the operations on the bus.

### 3.5 Scoreboarding

Scoreboarding is known as one of the early techniques for increasing processor speeds. Due to the distributed nature of the algorithm, we found that students had problems with understanding the algorithm exactly. In order to change this situation, we have developed a multimedia unit for this algorithm as well. In order to let students make experiments with the model, different instruction streams can be used and the effect of the resulting parallelism can be studied. Fig. 5 shows a screen-shot.

We found that the resolution of currently available projection equipment puts a tight constraint on the level of detail that can be shown for this algorithm.

### 3.6 Tomasulo algorithm

The Tomasulo algorithm is a more advanced algorithm for speeding up processor architectures. The Tomasulo algorithm employs a more decentralized control, making it even more difficult to understand the overall behavior. The corresponding RaVi unit avoids this problem. Again, the students can “play” around with different instruction streams and observe the behavior of the architecture. Functional components can be deleted by the user (lecturer or student) and new components can be added. No programming is required as long as standard components are added.

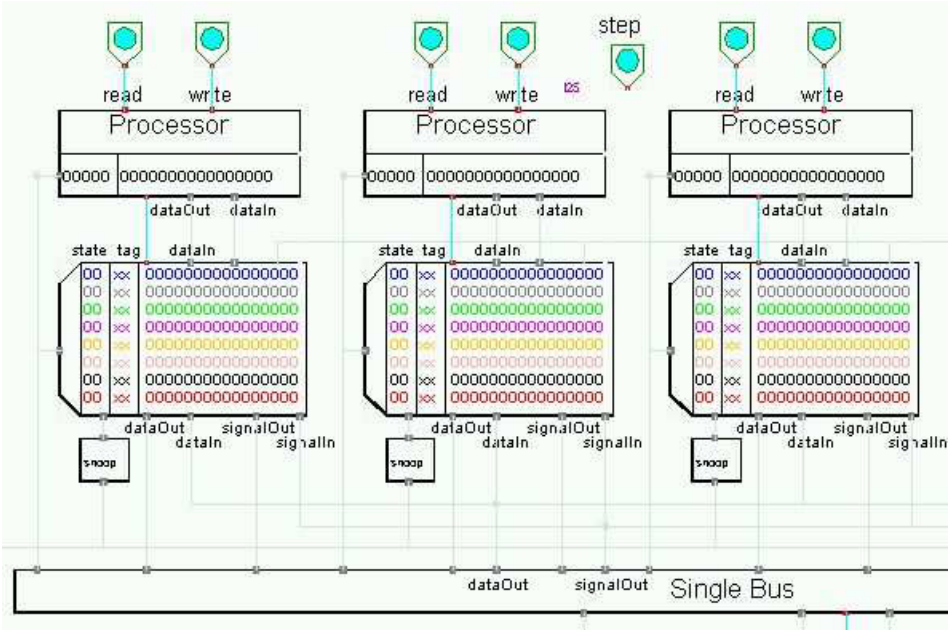


**Figure 4. Segment of a screen-shot from RaVi cache protocol unit (Memory omitted)**

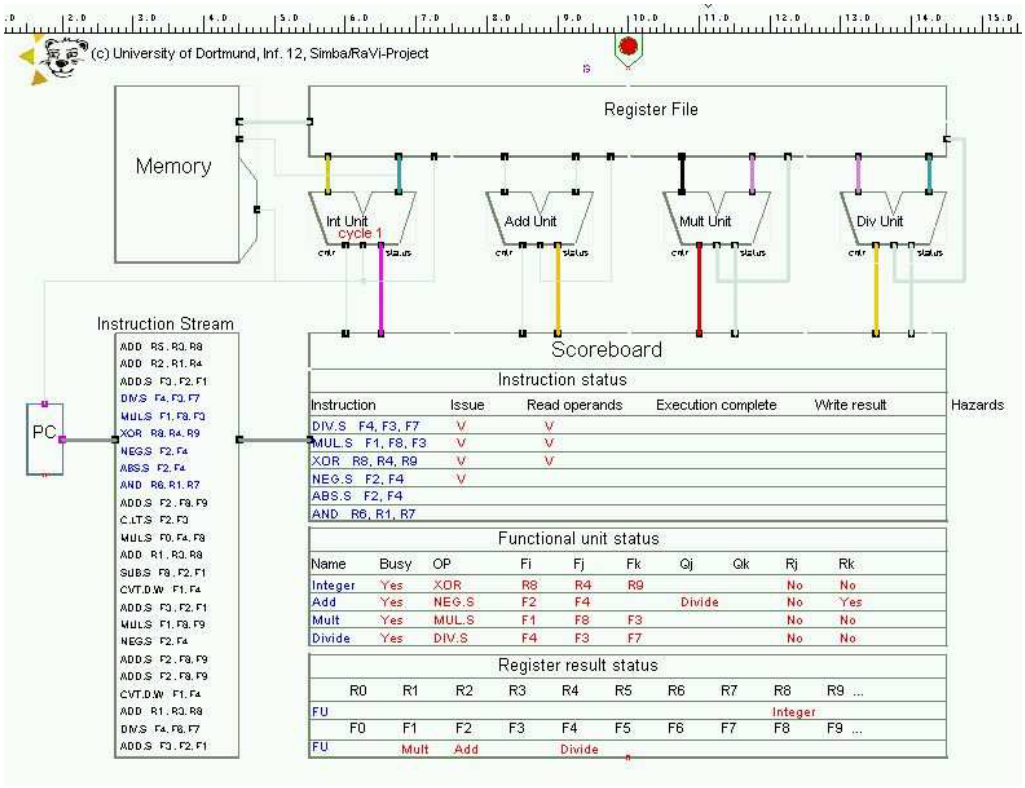


**Figure 5. Screen-shot from RaVi scoreboard unit**

## 4 Implementation aspects

### 4.1 Availability

The RaVi system is built on top of the HADES visualization framework for computer structures [3]. HADES is implemented in Java. The entire RaVi model follows the object-oriented paradigm. Every RaVi component is an instance of the corresponding hardware component class.

Due to being implemented in Java, RaVi can be used at a variety of platforms. We decided to make RaVi freely available on the Internet in order to promote its use. Initial versions of RaVi required a download of the software. Current versions are available as an applet and can be used without any software installation effort (provided Java is already installed). RaVi is available from //ls12.cs.uni-dortmund.de/ravi.

### 4.2 Gender-specific aspects

One of the goals of RaVi is to motivate also female students to study computer engineering. A number of considerations (see e.g. Fisher et al. [2]) have been taken into account during the design of RaVi:

- Before enterering the University, women typically have less hands-on-experience with computers in general and with computer engineering in particular, compared to most men. Therefore, a very careful definition of all technical terms must be used in the accompanying technical material.
- Educational material should avoid unjustified stereotypic views of computer users. For example, female computer users also include scientists and not only secretaries (in contrast, for example, to the cliparts provided by Microsoft).

### 4.3 Limitations

Simulation in the underlying HADES library is based on a VHDL-like two-phase simulation of synchronous architectures. Communication is based on explicit interconnections (which can be hidden on the screen). Simulation is less suited for applications in which explicit interconnections are difficult to use. Nevertheless, it was possible to use this simulation approach for demonstrating search in binary trees. Visualization is focussing on 2D models. 3D models are beyond the scope of the current approach.

## 5 Results

The RaVi project led to several results:

- We found that the generation of the multimedia units required significantly more time than expected. Due to using HADES, first versions could be designed rather quickly, requiring production efforts of a few weeks at most. However, the use of these units in the classroom led to requirements for improving the units. Only almost perfect units can be used in the classroom environments and given into the hands of students. Fine tuning of the units required as much work as their original design. A total of about 2 person years have been spent on the project so far.
- It is good scientific practice to try to measure by how much the quality of teaching can be improved by using the multimedia units. Following the advice by researchers from social sciences, we tried to get quantitative information on the level of understanding achieved through the use of these units. Even though we had two large groups of students (about 200 each) which could be compared, no quantitative conclusions could be drawn. A number of other effects (date and time of the teaching, characteristics of the students etc.) resulted in a wide variation of the results and prevented any meaningful conclusions. According to more recent advice from an expert in the area [7], attempts to quantitatively measure the effect of multimedia-based education are in fact bound to fail and a waste of time. One cannot expect more than just qualitative information on the improvements achieved. According to this qualitative information, the goals of the project have been reached.
- Students really like the presented units and appreciate their availability. They are typically highly motivated trying out these units at home and ask for download options. Also, colleagues typically comment very positively on the availability of these units. The most important argument is the added value of the units. While online-versions of static material provide only limited added value, if compared to books, visualization of dynamic properties adds a completely new quality.
- Visualization of the dynamic behavior has proven being indeed one of the key technologies for improving the teaching further and for exploiting modern equipment.
- Simulation based on the HADES simulation framework was found to be appropriate for various kinds

of digital circuits. While is was possible to use HADES for visualizing algorithms like tree-search, it is less appropriate of analog and time-continuous simulations. Simulation speed is sufficient even in applet-based versions of RaVi.

## 6 Conclusion

In the RaVi project, we have demonstrated how a deficiency of classical media for teaching computer architecture can be removed. We have shown that the visualization of computer architecture dynamics is appreciated by the students and helps them to understand the subjects. In general, RaVi units seem to improve the motivation of students. Unfortunately, it seems to be impossible to measure the effect of the new teaching aids on the student's success. In the future, we will be extended to approach to other areas of computer engineering. For example, we have started designing similar material to complement a book on embedded system design, which is currently being written at Dortmund.

## References


[1] I. Branovic, R. Giargi, and A. Prete. Web-based training on computer architecture: The case of jcachesim. *Proceedings of the workshop on computer architecture education*, pages 56–60, 2002.

[2] A. Fisher and J. Mangolis. Unlocking the clubhouse. *SIGCSE bulletin, Vol. 34, no. 2, Women and Computing*, pages 79–83, 2002.

[3] N. Hendrich. A Java-based framework for simulation and teaching. *Proceedings of the 3rd European Workshop on Microelectronics Education*, pages 285–288, 2000.

[4] J. L. Hennessy and D. A. Patterson. *Computer Organization – The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., 1995.

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1996.

[6] Model Technology. home page. *//www.model.com*, 2003.

[7] C. Moreau. Universite de Compiegne. *Oral communication*, 2003.

# Didactic Architectures and Simulator for Network Processor Learning

Henrique Cota de Freitas[1], Carlos Augusto P. S. Martins[2]
*Postgraduate Program in Electrical Engineering*
*Pontifical Catholic University of Minas Gerais, Brazil*
cota@pucminas.br[1], capsm@pucminas.br[2]
http://www.inf.pucminas.br/projetos/pad-r/r2np.html

**Abstract**

*In our university, we are developing a project about Reconfigurable Network Processors (RNP). There are four important results: Reconfigurable CISC Network Processor (RCNP) architecture, Reconfigurable RISC Network Processor (R2NP) architecture, Network Processor Simulator (NPSIM), and a performance analytical model for the ISA (Instruction Set Architecture). The architectures and the simulator are not commercial products, but conceptual models. This paper shows the main functionality of those four results and the their applicability on the Network Processor learning. As our Network Processor architectures and simulator are simpler than commercial products, their conceptual models can aid students to learn network processors concepts, as a first step to understand other complex architectures.*

*Keywords: Reconfigurable Network Processors, Didactic Architectures and Simulator, Learning Process.*

## 1. Introduction

Until the 1990's, network equipments used the traditional general-purpose processor (GPP) to process many types of packets and management services. The main advantage was flexibility. The software was capable to define many functions and applicability for the GPP's, but the performance was very harmed.

Another solution to solve the performance problem was the use of dedicated hardware. Using Application Specific Integrated Circuit (ASIC), the processing speed increases enough, but the flexibility was harmed.

So, there are two very important features: flexibility and performance. Nowadays, the Internet [28] is the main type of network, and the Quality of Service (QoS) is very important. For this reason, a best approach between those two features is necessary in network equipments. For example, a router is concentration point or a bottleneck in a network. The flexibility to process any kind of packets and the performance (processing time and throughput) are essential features that the Network Processor has to be capable to implement.

Network processors [19,20] appeared during the 1990's to replace some GPP's and ASIC's in network equipments. These processors were developed using architecture models like ASIP (Application Specific Processor) and SoC (System-on-Chip) [29] adding to RISC (Reduced Instruction Set Computing) [24] technique for a better computing performance. These processors have a dedicated ISA (Instruction Set Architecture) model for network operations. Thus, the instruction set and the architecture of Network Processors are specific to execute typical operations in a data communication network [28].

In Post-graduation Program in Electrical Engineering, we have a project called RNP (Reconfigurable Network Processor). The research goal is the development of a conceptual model of network processor using SoC and Reconfigurable Computing [17] techniques to improve computing performance and flexibility. The partial results are: Reconfigurable CISC Network Processor (RCNP) architecture [10], Reconfigurable RISC Network Processor (R2NP) architecture [11], Network Processor Simulator (NPSIM) [12,13] and the performance analytical model for the ISA (between RCNP and R2NP). During this paper, these four results will be presented.

Our main **objective** in this paper is to present didactic models of Network Processor architectures and a simulator to aid students to learn simple Network Processor architecture concepts. This is a first step to understand some details and complex features.

We searched for documents about words like learning and didactic models. However, nothing related with Network Processors was discovered. Thus, our **motivation** is present a simple way to learn the main features of Network Processors using didactic architecture models and a simulation tool.

## 2. Network Processors Overview

In this section we will describe the state-of-the-art of Network Processors. The main architecture features, the main functionalities, some related researches and companies.

The main logic blocks (figure 1) of a Network Processor are:

- ✓ Multiple RISC processors, co-processors or programmable ASIC's;
- ✓ Dedicated hardware for network operations;
- ✓ High speed of memory interface;
- ✓ High speed of I/O interface;
- ✓ General-purpose processors interface.

Each Network Processor has a typical architecture and uses some or all blocks showed. A Network Processor can use one RISC processor and co-processors like the packet processors, or only multiple RISC processors. If the SoC technique is used, possibly dedicated hardwares like switching fabric and memory can be in the architecture. However, GPP interface like PCI and I/O interface always appear. It's an important detail, because a Network Processor needs to communicate to other processors (to help in system management) and the network (the main of functionality).

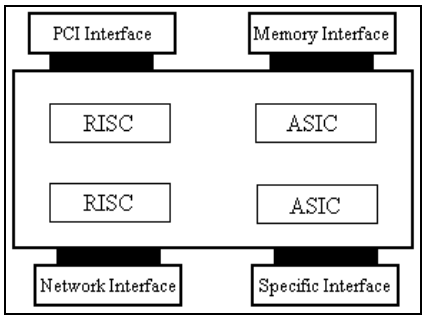

Figure 1 – Architecture Reference

The main functions of a Network Processor are:

- ✓ To analyze and classify the contents of head fields of a packet;
- ✓ To find in tables association rules related to head fields;
- ✓ To solve the destination path or QoS requirements;
- ✓ If necessary, to modify the packet (type of service or Diffserv, for example).

Nowadays, the active networks [6] are very important for QoS requirements, and equipments like active routers [27] appeared to improve performance and quality for Internet. The Network Processors have dedicated functionalities that provide flexibility and performance. For this reason, it has a large application in many network equipments.

During the developing process of Network Processors some companies joined among them we remark: Lucent / Agere [1], Motorola / C-Port [5] and Sitera / Vitesse [25]. Below, the main Network Processors and the companies are:

- ✓ IXP 1200 – Intel Corporation [16];
- ✓ NP4GS3 – IBM Corporation [15];
- ✓ C-5 Family – Motorola / C-Port [5];
- ✓ ASI/RSP/FPP – Lucent / Agere [1];
- ✓ IQ2000 Family – Sitera / Vitesse [25];
- ✓ AnyFlow 5400/5500 – MMC Networks [23];
- ✓ NP-1 – EZChip Technologies [8];
- ✓ NetVortex – Lexra Inc. [18];
- ✓ CS2000 – Chameleon Systems [3].

The Chameleon Systems was the first company to produce a Network Processor using the Reconfigurable Computing technique. This is an important characteristic. Reconfigurability is a technique that can be used in the NP architecture to improve flexibility.

Some researches about Reconfigurable Network Processors developed in universities are:

- ✓ "Reconfigurable Network Processors Based on Field Programmable System Level Integrated Circuits", University of Patras, Greece [22];
- ✓ "Design and Analysis of a Layer Seven Network Processor Accelerator Using Reconfigurable Logic", University of California, Los Angeles [9];
- ✓ "Design and Analysis of a Dynamically Reconfigurable Network Processor", University of Florida [14].

After section 1 and 2, it is possible to say that flexibility and performance are two important features during network processing. Thus, we conclude that some concepts are very important, and so students must know them before study a commercial Network Processors:

- ✓ CISC and RISC models;
- ✓ The concepts of ASIC's and ASIP's;
- ✓ The concepts of SoC's;
- ✓ The concepts of Reconfigurable Computing;
- ✓ The main logic blocks of Network Processor architecture;
- ✓ The main functions of Network Processor.

The next sections will present the didactic models of RNP project and the results that can aid students to understand the functioning of Network Processors, based on the features above.

## 3. Didactic Architecture Models

This section presents two architecture models: CISC model (RCNP) [10] and RISC model (R2NP) [11]. Both architectures were developed and simulated using

Reconfigurable Computing [17] and SoC [29] concepts and techniques to increase flexibility and performance.

Reconfigurable Computing: Input Ports and Crossbar has more flexibility in time execution. Buffer sizes and topologies can be created dynamically.

System-on-Chip: Functional blocks, that are found externally, as memory, I/O ports and switching fabric, are internally in the same chip. Like hierarchical memory, the proximity between functional blocks reduces processing time and increases performance.

The use of didactic architectures (RCNP and R2NP) is presented in section 5. Subsection 3.1 and 3.2 presents only technical features. We will implement both architectures using VHDL (VHSIC Hardware Description Language) [21] and FPGA (Field Programmable Gate Array) [21] in the future.

### 3.1 RCNP (Reconfigurable CISC Network Processor) Architecture

The basic features of RCNP [10] architecture are the following (figure 2):

- ✓ Eight input ports;
  - Temporary buffers (one static buffer for each port);
  - Permanent buffers (dynamic buffers, reconfigurable size buffers);
- ✓ Eight output ports;
- ✓ Reconfigurable Crossbar;
- ✓ Direct Access Memory (DMA);
- ✓ Eight general-purpose registers (8 bits);
- ✓ Data bus (8 bits) and address bus (24 bits);
- ✓ Maximum of size memory (16Mbytes)

The RCNP architecture was developed as a System-on-Chip. Memory, I/O ports and crossbar are placed internally.

The Reconfigurable Computing appears in Permanent Buffers and Crossbar. Thus, in execution time the size of the buffers and topologies (defined by crossbar) modifies dynamically.

The main features of instruction set are:

- ✓ General-purpose instruction set
  - Arithmetic instructions (Ex.: ADD and SUB);
  - Logic instructions (Ex.: AND and OR);
  - Memory access instructions (Ex.: LOD and STO);
  - Branch instructions (Ex.: JMP and JNZ);
- ✓ Dedicated network instructions
  - Input port reading (Ex.: ENT);
  - Output port writing (Ex.: SAI);
  - Crossbar control (Ex.: SEC);
  - Status register control (Ex.: SRS and LRS);

The RCNP architecture was not designed with pipeline technique. For this reason, all instructions are executed sequentially. In section 7, the performance analytical model for the ISA shows the impact of the architecture without pipeline.

Like all CISC projects, other instructions (different of load and store) access memory. The general-purpose instruction set of RCNP is not optimized. There are 256 instructions that can be found in the project homepage (http://www.inf.pucminas.br/projetos/pad-r/). The simulation tool (NPSIM) also has the instructions described in figure 9.

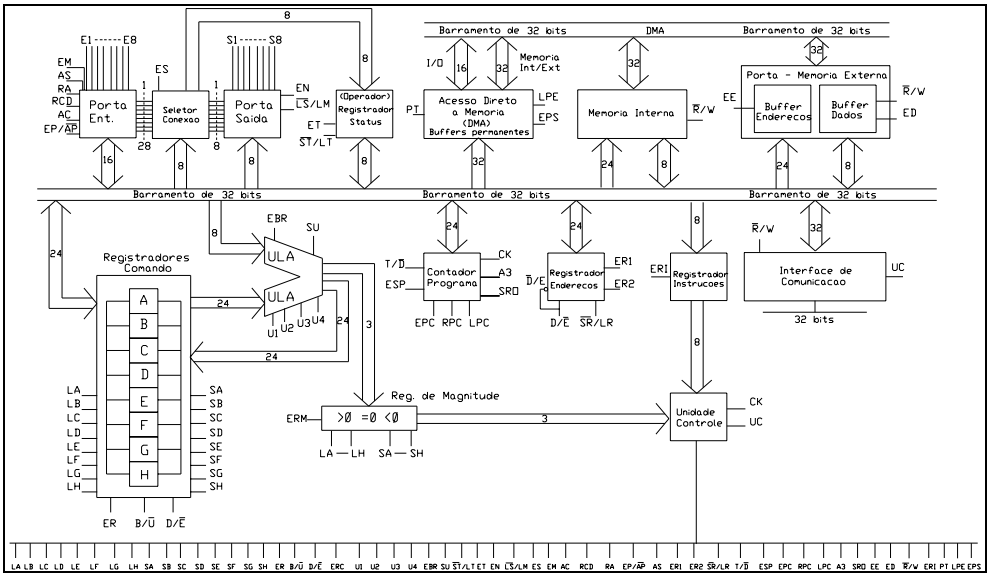

Figure 2 – RCNP Architecture

### 3.2 R2NP (Reconfigurable RISC Network Processor) Architecture

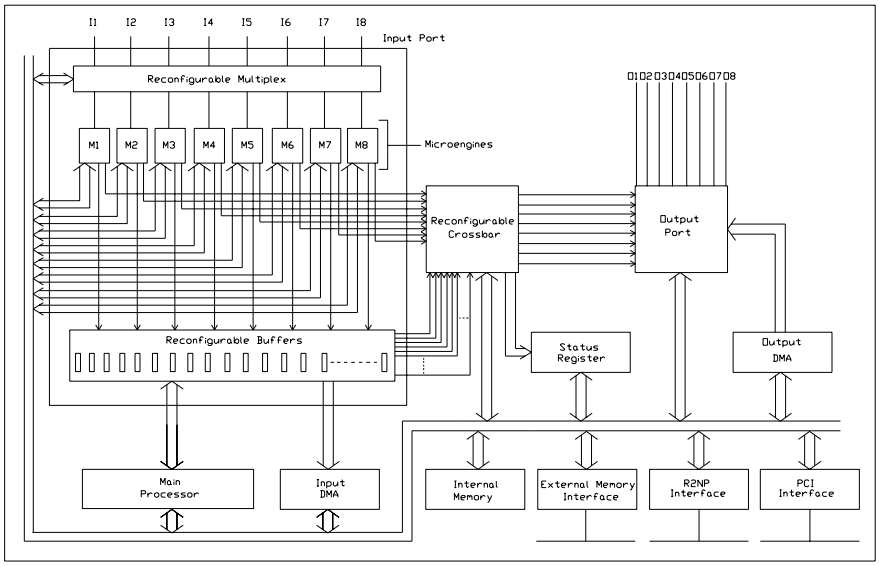


Figure 3 – R2NP Architecture

The evolution of RCNP is the R2NP [11]. This architecture uses the RISC model, pipeline and other reconfigurable blocks. The figure 3 shows the R2NP architecture.

The basic features of RCNP architecture are the follows:

- ✓ Eight input ports;
  - Reconfigurable Multiplex;
  - Programmable Microengines (one microengine for each port);
  - Permanent buffers (dynamic buffers, reconfigurable size buffers);
- ✓ Reconfigurable Crossbar;
- ✓ Eight output ports;
- ✓ Internal memory;
- ✓ Main RISC processor with data cache and instruction cache;
- ✓ Direct Access Memory (DMA), dedicated hardware;
- ✓ 256 registers (64 bits);
- ✓ Data bus (32 bits) and address bus (32 bits);
- ✓ Maximum of size memory (16Gbytes)

In R2NP project we add two important network blocks: Reconfigurable Multiplex and Microengines.

Microengines: Are responsible for the first analysis on the packet head. In this case the packet can be forward to output ports with no intermediary copies to buffers or memory.

Reconfigurable Multiplex: If you lost one microengine or need to use it in other function, the multiplex connects two or more ports to one microengine.

The instruction set of R2NP is more optimized than RCNP. Based on the RISC model, the instruction format is fixed and only load and store instructions access the memory. We present in table 1 the instruction set of R2NP.

Table 1 – Instruction Set of R2NP

| ***General-purpose*** | | | |
|---|---|---|---|
| *ADD A,B,C* | *MOV A,B* | *SPUSH A* | *CONV* |
| *SUB A,B,C* | *INC A* | *LPOP A* | ***Network*** |
| *MUL A,B,C* | *DEC A* | *JMP A* | *FCX A,B,C* |
| *DIV A,B,C* | *LOD A,End32* | *JZ A* | *LOB A* |
| *AND A,B,C* | *LDA A,End16* | *JMZ A* | *BRC A* |
| *OU A,B,C* | *LOX A,B* | *JMI A* | *SAI A,B* |
| *XOR A,B,C* | *LDI A,Imed16* | *JNZ A* | *LRS A* |
| *NEG A* | *STO End32,A* | *JNI A* | *SRS A* |
| *ROD A* | *STR End16,A* | *CALL A* | *SEC A,B* |
| *ROE A* | *STX A,B* | *RET* | *ENT A,B,C* |

There are two kinds of store and load instructions: with internal and with external memory access.

The internal memory is smaller and the instructions number 16 (*Ex.: LDI A,Imed16*) use sixteen bits to access the 64kbytes memory. The instructions number 32 (*Ex.: LOD A,End32*), access only the external memory. If the instruction has 32 bits of address, two fetch cycles will be necessary.

The network instructions (table 2) are very similar with the RCNP network set. However, by optimization, some differences appear in the instructions format.

The pipeline of R2NP is showed in figure 4. This is very similar with the conceptual model of pipeline [24], but the difference is the Buffer stage (together memory). One instruction that access buffer does not access memory.

| Stages | | | BF | |
|---|---|---|---|---|
| B | D | E | M | R |
| 1º | 2º | 3º | 4º | 5º |

Figure 4 – Pipeline Stages

The stages mean:

1. Fetch of instruction (B);
2. Decoding of instruction. Reading of register bases (D);
3. Execution of instruction (E);
4. Reading or writing in memory of reconfigurable buffers (M/BF);
5. Results. Writing in register bases (R);

The performance analytical model for the ISA, in section 7, will show how the pipeline project increases performance.

## 4. Didactic Network Processor Simulator

This tool [12,13] was constructed with C++, and the main interfaces are capable to aid and guide the student in the learning of Network Processor theory and functioning. The simulator has six interfaces and it simulates main logic blocks as memory, registers, buffers, crossbar switch, DMA, I/O ports and others that are responsible to store, process, receive and transmit data. The student can modify and visualize the status and movement of the data inside and between logic blocks. There are two edition boxes, a program assembler and an editor of network packets.

This tool simulates the RCNP architecture and is available to download (http://www.inf.pucminas.br/projetos/pad-r/r2np.html) in two languages (idioms): Portuguese and English. It makes functional tests in all logic blocks of the processor. Through this tool, it is possible to write and execute many algorithms (assembly programs) and visualize the execution and the results in objects like: registers, stacks and arrays represented in components of C++ Builder 5.0 (used to construct and compile the simulator).

The user interface has one main module and six other modules:

- Memory, Registers and Fast Access Buttons (compose the main module – figure 5)
- Assembler (assembly program window – figure 5)
- Permanent Buffers (reconfigurable buffers) – (*).
- Temporary Buffers (eight buffers for each input) – (*).
- Input Packets (network packet edition box – figure 6)
- Internal Crossbar (commutation array) – (*).
- DMA Registers (Direct Memory Access) – (*).

(*) These modules are not shown in this paper. In figure 5, the number 1 shows where other modules can be found. For more details see the references [12,13].

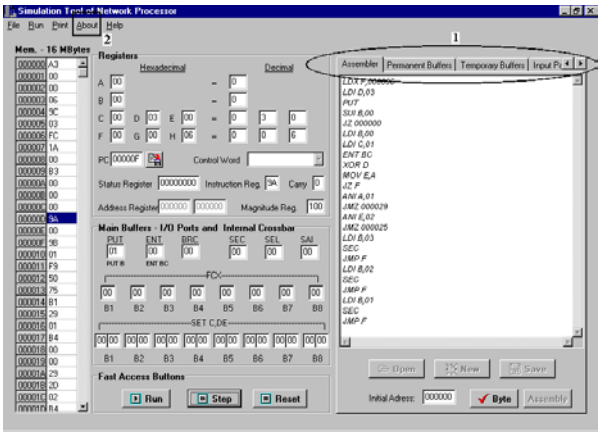


Figure 5 – Main Module (Assembler)

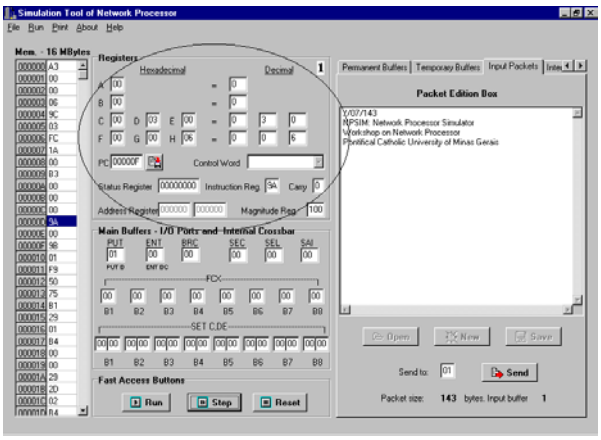


Figure 6 – Input Packets

In this section, we only describe the edition modules like the assembler and input packets. The figure 10 and 11, show the part of the Temporary Buffers and Crossbar modules, and its application as a way to learn Network Processor concepts.

With this simulator, it is possible to write and simulate routing algorithms (section 7), study the functioning of CISC network processor (RCNP) and understand the functioning of the Network Processors, described in section 5.

Other information about this simulator can be found in the papers: Simulation Tool of Network Processor for Learning Activities [12] and NPSIM: *Simulador de*

*Processador de Rede* (NPSIM: Network Processor Simulator) [13].

## 5. Using RNP Project to Learn NP Concepts and Functioning

This section will show how RNP project (didactic architectures and simulator) can help the students to learn network processor functioning and concepts.

The figures 7 and 8 show interfaces of NPSIM, which RCNP architecture and technical information. These interfaces aid students to understand the NPSIM modeling of RCNP proposal architecture. Basic features and important blocks as buffers and crossbar are described. Thus, concepts can be read and functional blocks visualized before the software execution. These interfaces can be found in the "About" option (figure 5, number 2).

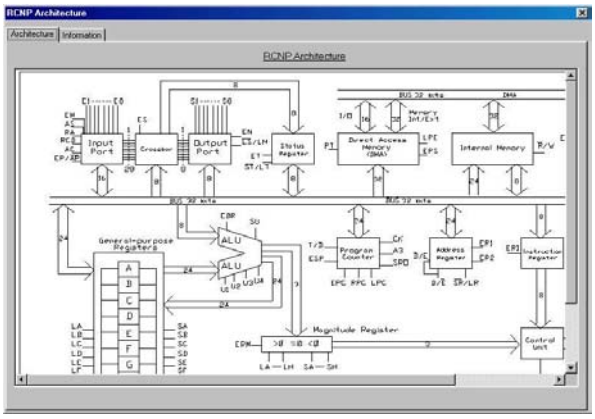


Figure 7 – RCNP Architecture

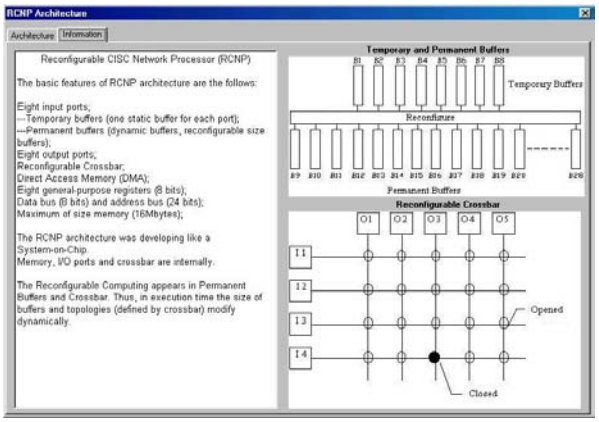


Figure 8 – Technical Information

The RCNP architecture has important blocks described in the architecture reference (section 2). Through the diagram architecture of RCNP we can visualize these functional blocks that represent important concepts of Network Processor, as flexibility (reconfiguration) [17] and performance (ASIC's). We present in table 2 these concepts:

Table 2 – Concepts of RCNP blocks

| Functional Blocks | RCNP Concepts |
|---|---|
| Input Ports | Reconfigurable ASIC (Buffers) |
| Output Ports | ASIC |
| Internal Crossbar | Reconfigurable ASIC |
| Internal Memory and External Memory Interface | ASIC |
| Communication Interface | ASIC |
| Typical and dedicated CISC processor blocks | ASIP |

The instruction set of RCNP can be visualized through the NPSIM interface. The figure 9 shows the instructions. This interface also can be found in the "About" option (figure 5, number 2).

The figure 10 shows interface modules that represent Temporary Buffers (inside Input Ports) and Internal Crossbar together Output Ports. A network packet sent by the Packet Interface (figure 6), arrives in Temporary Buffers and the behavior can be analyzed by students through control registers (figure 6, number 1). The behavior of routing algorithms and the destination of packets are showed through buffers, registers, crossbar, inputs and outputs. The buffers receive and allocate packets from inputs, the registers aid visualize the manipulation and data behavior, and the crossbar (figure 10) shows the way from input to output.

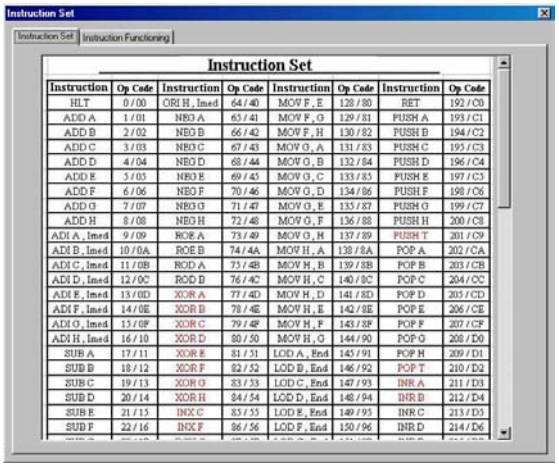


Figure 9 – Instruction Set of RCNP

These interfaces aid students to understand basic functions of Network Processors. Algorithms that execute functions described in section 2, as analysis and modification of contents, search for association rules, destination resolution, and QoS requirements can be found in NPSIM.

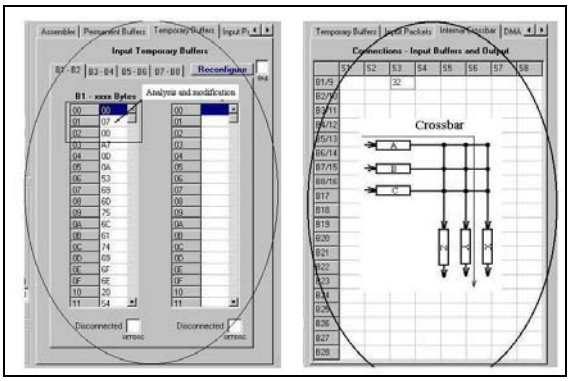


Figure 10 – Interface Modules

Although, the RCNP has many features of a Network Processor, one main characteristic does not exist, the RISC technique. RISC processors have better performance than CISC processors. Instruction format and pipeline are very important to increase processing speed and reduce processing time.

R2NP is the evolution of RCNP project. In this project, the goal is add RISC concepts and optimize the architecture and instruction set. The main difference is the pipeline and the instruction format. In section 7, the relation will be described.

The R2NP project is robuster than RCNP, and we present in table 3 some concepts related with Network Processors.

Table 3 – Concepts of R2NP blocks

| Functional Blocks | R2NP Features |
|---|---|
| Input Ports | Reconfigurable ASIC (Buffers) |
| | Reconfigurable ASIC (Microengines) |
| | Reconfigurable ASIC (Multiplex) |
| Output Ports | ASIC |
| Internal Crossbar | Reconfigurable ASIC |
| Internal Memory and External Memory Interface | ASIC |
| Direct Access Memory | ASIC |
| Communication Interfaces | ASIC |
| Typical and dedicated RISC processor blocks | ASIP |

The figure 11 shows the project evolution, based in hierarchical memory [24].

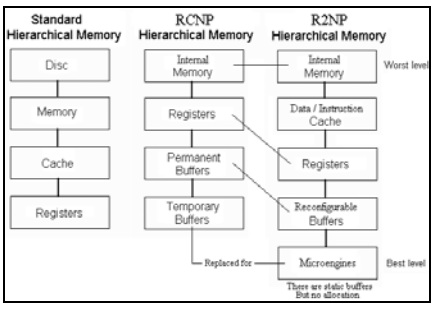


Figure 11 – Hierarchical Memory

The R2NP has data and instruction cache and RCNP has not. The Temporary Buffers were replaced by Microengines. In RCNP, packets could be store in Temporary Buffers, but in R2NP the microengines that also have static buffers, analyze and decide the destination of a packet, with no packet allocation. There are three routes: through crossbar and output ports, in the reconfigurable buffers or memory. In R2NP project, the main processor analyzes packets in buffers and memory.

## 6. Commercial Architectures of Network Processors

Some commercial architectures of Network Processors are presented and related with the reference architecture, also described in RNP project. The main features are numbered and appear in each figure. It's important to say that each commercial example represents details, features or concepts presented in our proposal, proving the real capability of RNP project as a didactic environment to learn Network Processors.

*NetVortex Architecture* [18] (figure 12): Each NetVortex is composed of many packet processors. It uses multi-threading in hardware and has the same instruction set of MIPS. Some features related to reference are:

1. Encryption Coprocessor: This architecture uses coprocessors for specific applications;
2. Crossbar Switch: Also uses dedicated hardware (ASIC) to increase performance;
3. Packet Processor: Specific processors for packet processing.

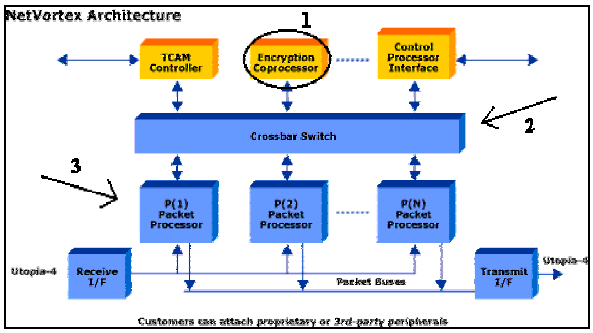


Figure 12 – NetVortex Architecture

*IQ2000 Architecture* [25] (figure 13): The IQ2000 Network Processor has four scale processors inside the chip. It has native support for MIPS, PowerPC and others RISC processors. There are specific coprocessors and hardware support for Quality of Service (QoS). Some features related to reference are:

1. Multiples CPU´s: The IQ2000 uses parallel processing based in multiples processors;

2. QoS Engine: Dedicated hardware (ASIC) to increase performance.

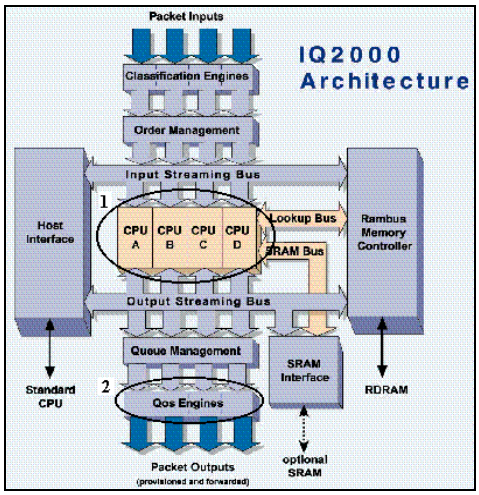


Figure 13 – IQ2000 Architecture

*IXP1200* [16] (figure 14): The IXP1200 is composed of seven RISC processors. The first processor (StrongARM) is responsible for managing the network and for complex processing. The other six processors (the microengines) are responsible for processing and routing packets. Some features related to reference:

1. StrongARM Processor: The main processor responsible for complex processing.
2. PCI Unit: Dedicated communication hardware (ASIC).
3. Multiples microengines: For parallel processing of network packets.

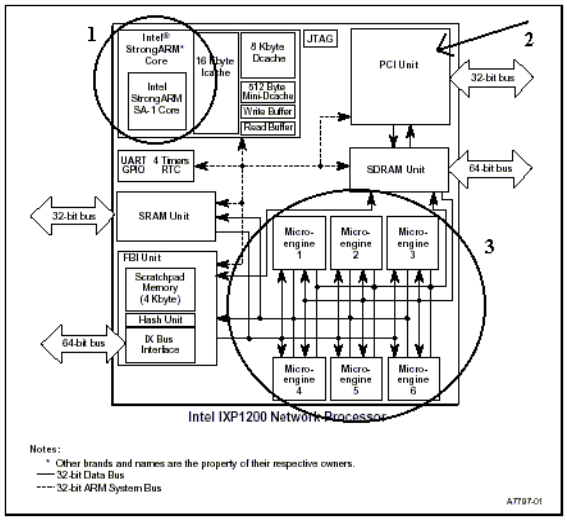


Figure 14 – IXP1200 Architecture

*Reconfigurable Fabric of CS2000* [3] (figure 15): This figure shows the reconfigurable block of CS 2000. This block is divided in four slices with three blocks. Each block has Datapath Units, Local Memories, Multipliers and Control Logic Unit.

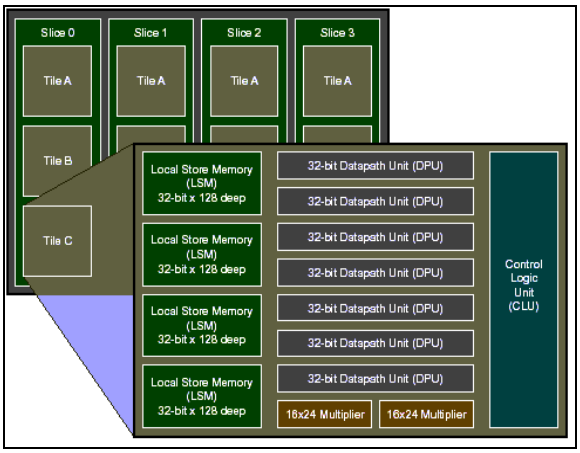


Figure 15 – Reconfigurable Fabric of CS2000

This section presented some features of RNP project, how to learn Network Processors using it and four commercial Network Processors. During all the description we related the features of RNP project and commercial NP to the architecture reference. The next section will present the experimental results from simulations and analytical model that contribute to validate the evolution of RNP project.

## 7. Experimental Results using NPSIM

The simulation results and the performance analytical model for the ISA were based in three topologies. For each topology was written three algorithms. These simulations were very important to validate the concept of RCNP. Using the results, an analytical model was proposed to verify the performance between ISA of RCNP and ISA of R2NP. The topologies (figure 16) are the follow:

Hypercube topology: Where each vertex is a simulated network processor. The routing algorithm is based in different bit resulted by XOR operation between source address and destination address.

Unidirectional ring topology: Constructed using the internal crossbar. The unidirectional ring program shows how internal crossbar can construct a topology.

Balanced Tree: Where each vertex is a simulated network processor. In this case the routing program uses the standard address by each vertex. The addresses grow from left to right.

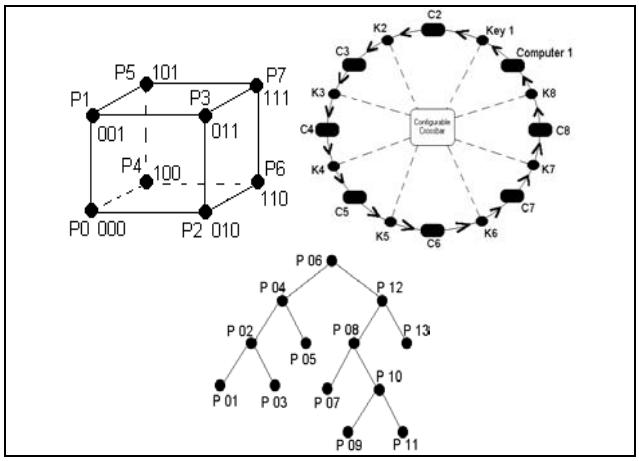

Figure 16 – Simulated Topologies

The metrics defined to analyze performance for ISA are:

Cf → Clock Frequency (Hz)
Tp → Time of Processor
Ncpc → Number of cycles of program clock
Cpi → Cycles per instructions
Npi → Number of program instructions
Pf → Performance factor

We can related through these equations:

Cpi = Ncpc / Npi
Tp = Npi * Cpi / Cf

It's important to say that the RCNP model does not use pipeline, the instructions are executed sequentially. The RCNP and R2NP does not exist physically, for this reason, the clock frequency is 500Mhz for definition.

For the Unidirectional Ring topology, the results are the follows:

*RCNP proposal*
Npi = 45, Ncpc = 191, Cpi = 191 / 45 = 4,244
Tp = 191 / 500 10E6 = 0,382 µs
*R2NP proposal*
Npi = 38, Ncpc = 43, Cpi = 43 / 38 = 1,131
Tp = 43 / 500 10E6 = 0,086 µs
Pf = 0,382 / 0,086 = 4,44
The R2NP is 4,44 faster than RCNP for this simulation.

For the Hypercube topology, the results are the follows:

*RCNP proposal*
Npi = 17, Ncpc = 73, Cpi = 73 / 17 = 4,294
Tp = 73 / 500 10E6 = 0,146 µs
*R2NP proposal*
Npi = 17, Ncpc = 21, Cpi = 21 / 17 = 1,235
Tp = 21 / 500 10E6 = 0,042 µs
Pf = 0,146 / 0,042 = 3,47
The R2NP is 3,47 faster than RCNP for this simulation.

For the Balanced Tree topology, the results are the follows:

*RCNP proposal*
Npi = 12, Ncpc = 56, Cpi = 56 / 12 = 4,666
Tp = 4,666 / 500 10E6 = 0,112 µs
*R2NP proposal*
Npi = 15, Ncpc = 19, Cpi = 19 / 15 = 1,266
Tp = 19 / 500 10E6 = 0,038 µs
Pf = 0,112 / 0,038 = 2,94
The R2NP is 2,94 faster than RCNP for this simulation.

Through this analytical model, the students can understand how RISC processor can be faster than CISC processors, using project techniques as pipeline, for example.

## 8. Conclusions

Nowadays, there is a great need of high performance in the data communication network [2,28]. The study of various equipments [4,27] and their functions, influenced in the project, and development of dedicated processors, that can supply the need of performance and quality. Thus, the Network Processors were initially developed to contribute with the increase of speed and quality of service in the communication systems.

In this paper we presented a project called Reconfigurable Network Processor (RNP) that has a main goal: to aid students to learn and know initial basic concepts of Network Processors, as a first step to understand commercial products.

RCNP [10] and R2NP [11] architectures, and NPSIM [12,13] simulator were described with many options to understand the reference architecture and the basic network processors functioning. In section 5, the architectures and NPSIM were shown for the student to compare the learning possibilities. The same features in reference architecture appear in RNP project. Using these didactic proposals it is possible to learn the basic concepts. Four commercial architectures were presented and related with the reference to show the use of didactic models before the studying of commercial Network Processors.

Didactic models and simulator were looked for, but they were not found anywhere. However, one correlate paper was presented in NP1, "A Methodology and Simulator for the Study of Network Processors" [7]. But they have different goals. That paper describes a model of the Cisco Toaster architecture and show simulated performance results of a Diffserv implementation. It describes a commercial product and simulates performance. Our goals in this paper are present a didactic model to introduce the main concepts of Network Processors before the study of complex architectures. A paper or research with didactic features for NP's, were not found.

The main presented results of our research, are the RCNP architecture, R2NP architecture, NPSIM

simulator and experimental results. Those results validated our goals and showed how conceptual models can aid students to understand complex architectures of Network Processors.

Thus, our main contribution, in this paper, is present didactic architectures and simulator for beginning process of Network Processors learning.

Our future works are: To simulate R2NP using Rconf_KMT (Reconfigurable Simulation Tool) [26] and VHDL (VHSIC Hardware Description Language) [21], prototype using FPGA (Field Programmable Gate Array) [21], simulate it in a real network system [2,28], and to develop didactic environment to learn Network Processors.

## 9. References


[1] Agere System, Fast Pattern Processor (FPP) Product Brief, April 2001, http://www.agere.com

[2] Buya, R., High Performance Cluster Computing, Volume 1, Prentice Hall, 1999

[3] Chameleon Systems, "CS2000 Reconfigurable Communications Processor", Family Product Brief, 2000

[4] Cisco Systems White Paper, "The Evolution of high-end Router Architectures-Basic Scalability and Performance Considerations for Evaluating Large-Scale Router Designs", 2001, http://www.cisco.com

[5] C-Port, C5e Network Processor Product Brief, January 2002, http://www.motorola.com

[6] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, G. J. Minden, "A Survey of Active Network Research", IEEE Communications Magazine, Volume 35, Nº 1, pp.80-86, 1997

[7] D. Suryanarayanan, G. T. Byrd and J. Marshall, "A Methodology and Simulator for the Study of Network Processors", Workshop on Network Processor (NP1 at HPCA 8), Cambridge Massachusetts, February 2-6, 2002

[8] EZChip Network Processors, http://www.ezchip.com

[9] G. Memik, S. O. Memik, W. H. Mangione-Smith, "Design and Analysis of a Layer Seven Network Processor Accelerator Using Reconfigurable Logic", The 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines FCCM'02, Napa, California, 21-24 April, 2002

[10] H. C. Freitas, C. A. P. S. Martins, "Processador de Rede com Suporte a Multi-protocolo e Topologias Dinâmicas", II Workshop de Sistemas Computacionais de Alto Desempenho, WSCAD'2001, Pirenópolis - GO, Brasil, pp.31-38 (in portuguese)

[11] H. C. Freitas, C. A. P. S. Martins, "R2NP: Processador de Rede RISC Reconfigurável", III Workshop de Sistemas Computacionais de Alto Desempenho, WSCAD'2002, Vitória, ES, Brasil, pp. 60-67 (in portuguese)

[12] H. C. Freitas, C. A. P. S. Martins, "Simulation Tool of Network Processor for Learning Activities". Frontiers in Education Conference (FIE 2002), Boston, MA, USA, November 2002, Session S2F, pp.1-6

[13] H. C. Freitas, C. A. P. S. Martins, "NPSIM: Simulador de Processador de Rede". XXVIII Latin-American Conference on Informatics, CLEI'2002, Montevideo, Uruguay, November 2002 (in portuguese)

[14] I. A. Troxel, A. D. George, S. Oral, "Design and Analysis of a Dynamically Reconfigurable Network Processor", IEEE Conference on Local Computer Networks (LCN), Tampa, Florida, November 6-8, 2002

[15] IBM PowerNP NP4GS3 Databook, http://www.ibm.com

[16] Intel, "IXP 1200 - Network Processor", Datasheet, May 2000, http://www.intel.com

[17] K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", ACM Computing Surveys, Vol. 34, No. 2, June 2002, pp. 171-210

[18] Lexra, NetVortex Network Communications System Multiprocessor NPU, http://www.lexra.com

[19] Lucent Technologies, Building for Next Generation Network Processors, September 1999

[20] Lucent Technologies, The Challenge for Next Generation Network Processors, September 10, 1999

[21] M. Glesner, A. Kirschbaum, "State-of-the-Art in Rapid Prototyping", XI Brazilian Symposium on Integrated Circuit Design, SBCCI'98, Búzios, Rio de Janeiro, 1998, pp.60-65

[22] M. Iliopoulos, T. Antonakopoulos, "Reconfigurable Network Processors Based on Field Programmable System Level Integrated Circuits", Field-Programmable Logic and Applications, The Roadmap to Reconfigurable Computing, 10th International Workshop, FPL 2000, Villach, Austria, August 27-30, 2000, pp. 39-47

[23] MMC Networks, "EPIF-105, EPIF-200, GPIF-207, XPIF-300, Packet Processors", http://www.mmcnet.com

[24] Patterson, D. A., J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufmann Publisher, 1997

[25] Sitera IQ2000, Network Processor Product Brief, http://www.sitera.com

[26] T. H. Medeiros, C. A. P. S. Martins, "Reconf_KMT, Uma Ferramenta Reconfigurável para a Simulação de Microprocessadores", III Workshop de Sistemas Computacionais de Alto Desempenho, WSCAD'2002, Vitória, ES, Brasil, pp. 32-38 (in portuguese)

[27] T. Wolf and J. Turner, "Design Issues for High Performance Active Routers", International Zurich Seminar on Broadband Communications, Zurich, Switzerland, 2000, pp. 199-205

[28] Tanembaum, A. S., Computer Networks, Prentice-Hall, 1996

[29] W. D. Mensch. Jr. and D. A. Silage, "System-on-chip Design Methodology in Engineering Education", International Conference on Engineering Education, ICEE2000 (IEEE/CS), Taipei, Taiwan, August 2000, pp. 224-228

# On the Introduction of Reconfigurable Hardware into Computer Architecture Education

Ross Brennan and Michael Manzke
Trinity College Dublin
Ireland
rbrennan@tcd.ie, michael.manzke@cs.tcd.ie

## Abstract

*The introduction of reconfigurable logic devices as teaching-aids, into undergraduate and graduate education, enables the students to conduct experiments they could otherwise not perform. Furthermore, this approach gives instructors the freedom to choose architectures that are dictated by the underlying hardware to a lesser extent.*

*Today's Field Programmable Gate Arrays (FPGA) can implement integer units as complex as the SPARC V8 at a reasonable hardware price. Educational boards that replace conventional CPUs with reconfigurable logic devices can be integrated into an existing syllabus with legacy hardware requirements without disruption, as long as the* soft-CPU *core on the reconfigurable logic device provides opcode compatibility with the superseded processing unit.*

## 1 Introduction

For almost 20 years computer science students and computer engineering students at Trinity College Dublin have been asked to construct a complete microprocessor system as part of the coursework. The students perform this tasked by wire-wrapping integrated circuit (IC) components to a Motorola MC68008 microprocessor. The 8 bit data bus keeps the required amount of wire wrapping to an acceptable level. The CPU has no caches and therefore allows the observation of all memory accesses on the system bus. The CPU is clocked at a modest 7.5MHz. Despite its age the processor is very suitable for this task but it has become increasingly difficult to source these CPUs. We had to look for an alternative that is equally suitable for the application.

Replacing the MC68008 with a more modern CPU that exhibits similar properties is a feasible design option. We compared this option with an approach that utilises soft-CPUs that execute on reconfigurable logic devices. FPGAs that are large enough to hold a complex CPU can be bought at a competitive price.

It was obvious that an FPGA based solution would not only allow us to replicate the properties of the exiting board but also to increase the scope to an extent that it could be useful as an educational-aid for many more undergraduate and postgraduate courses.

In this paper, we argue for the migration from standard CPUs on computer architecture lab boards, to an FPGA based approach that employs *soft-CPUs*. These FPGA based boards should allow the student to download a choice of *soft-CPU* cores into the SRAM based FPGA. A number of configuration PROMs can be used to hold a range of these *soft-CPU* cores, that may be individually selected through a *PROM-select-switch* before the board is powered on. This allows the utilization of the boards for a variety of teaching objectives, from a wire-wrapped microprocessor project to CPU design projects.

We propose the following *soft-CPU* options:

- A simple instruction set processor that can be implemented by a second-year student
- An open-source, fully configurable SPARC V8 architecture conforming to the IEEE-P1754 standard
- Legacy architectures that simplify the transition to FPGA-based boards. At Trinity College Dublin (TCD) this requires a Motorola MC68008 compatible core.

All of the possible *soft-CPU* options must use a common system-bus interface. In particular, the open source SPARC core gives students access to a HDL model of a 'state-of-the-art', industry standard processor. The ability to configure the HDL sources

enables the students to activate and deactivate components in the designs in order to perform specific experiments. This will be of interest to students in advanced undergraduate and postgraduate courses.

In lower undergraduate courses, the board may be used by second year students to wire-wrap a complete microprocessor system and to design, test and synthesise a simple instruction set computer.

As a first step to evaluate these design objective we prototyped an FPGA based board that executes an open-source SPARC core (the LEON P-1754) [1] in a Xilinx XC2V1000 Virtex-II FPGA. This core was written as a highly configurable, fully synthesisable VHDL model and was originally developed by the European Space Agency (ESA) with the intention of being used in future space missions. The HDL model is configured with an 8-bit data bus and is clocked at 6MHz to enable students to wire-wrap a complete microprocessor system. The processor is configured to run with the internal caches disabled, which allows students to observe all of the bus transactions using a logic state analyser connected to the external system bus. This facilitates debugging of the system and allows the development of a rudimentary operating system.

## 2 Background and Related Work

Using custom hardware and reconfigurable logic devices to aid in the teaching of computer architecture is not a new concept and its success has already been demonstrated through several different projects. These projects focus mainly on teaching the students using a fixed architecture with a fixed project board configuration and thus do not take full advantage of the reconfigurable nature of the underlying hardware system.

Work has been carried out to design and implement custom hardware and simulation tools at the University of Waikato [2]. This involved developing and implementing a computer architecture solely for the purposes of teaching and then designing a project board and simulation tools to complement it.

Several efforts have been made to design processors that are not compatible with commercial Instruction Set Architectures (ISAs), such as the work carried out at Harvard [3]. Their *Ant-32* processor architecture was developed with the intention of providing a simple yet functional platform through which to introduce students to the operation of a basic microprocessor system.

A project kit based around a dedicated soft-CPU architecture has also been developed by Gray Research [4], with the intention of teaching basic computer architecture concepts to undergraduate students.

Without major updates to the processor models, these implementations do not allow the system to grow in complexity as the students understanding of system design concepts increases. By using fully functional, configurable processor models, the complexity of operation of the microprocessor system may be tailored to suit the needs of different student groups.

Currently computer architecture education in the Computer Science degree at TCD accounts for 35% of the overall teaching effort. In the first year of their studies, students are educated in Assembly Language Programming, Digital Logic Design and Electrotechnology.

These courses serve as the foundation for the Computer Architecture and Digital Electronics courses in the second year. The first semester of this Computer Architecture course requires students to construct a working microprocessor system from integrated circuit (IC) components. Students work in small groups and use wire wrapping to interconnect the required circuitry. Figure 1 shows a lab board that is currently used by students for their microprocessor project and Figure 2 provides a schematic for the same project. A basic operating system that enables communication with a host system and the execution of application code is also developed. In the subsequent semester students are introduced to VHDL and learn about register transfers, the design of a datapath, sequencing and control. As part of their coursework, students are asked to design a Multiple-Cycle Microprogrammed Instruction Set Computer. This design is implemented in VHDL.

More advanced aspects of computer architecture are covered in Computer Architecture II and Computer Engineering in third year. This includes CISC and RISC architectures, memory hierarchy, I/O subsystems, high performance processing systems and VLSI Design.

In their final year students may select two computer engineering related subjects, Computer Architecture and Integrated and Systems Design. The option is also available to students to choose a final year project with a computer architecture content.

A similar syllabus is taught to computer engineering students. Engineering students may choose to join the degree programme in computer engineering, following the first two years that are common to all engineering students. These students receive

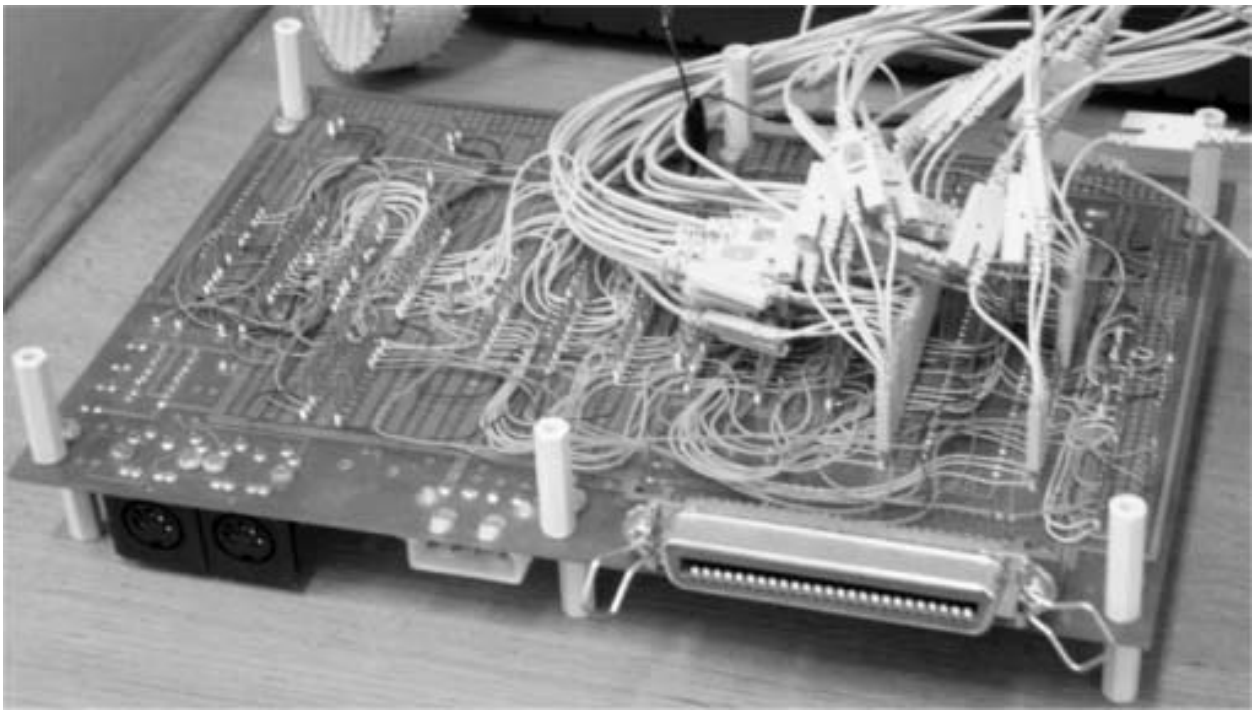

Figure 1: Photo of the current wire-wrapping lab board

the same computer architecture lectures that computer science students receive but in a compressed form over the last two years of their degree.

The proposed lab boards should be suitable for use in most of the computer architecture related subjects for both computer science and computer engineering students.

It is envisioned that the custom built Motorola 68k microprocessor system will be replaced with the new FPGA based solution. Currently the 68k microprocessor system is use to teach first year computer science students and third year computer engineering students 68k assembly language. The FPGA boards could be used in conjunction with a 68k HDL model and a daughter board that substitutes the wire-wrapped part of the board. The daughter board could be populated with more sophisticated integrated circuit components that also take advantage of a full 32bit bus width.

The daughter board could also be used with a version of the LEON core that is configured for this purpose. The core would operate with a 32bit system bus and caches would be enabled. Furthermore, there is scope for I/O capabilities such as PCI and Ethernet. This configuration is suitable for more advanced classes.

The proposed board would also allow students to construct a microprocessor system through wire wrapping by removing the daughter board and using an 8-bit version of the LEON core, the 68k compatible model or a student-built Multiple-Cycle Microprogrammed Instruction Set computer with an 8-bit system bus interface.

The current design project is based around the 16-bit Motorola MC68008 microprocessor, which was released in 1982. The main features of this processor are its CISC architecture and 8-bit external data bus. This processor is becoming too outdated to reflect modern computer architecture trends and so a new alternative had to be sought, which was more in line with current technological developments.

The microprocessor project requires each of the student groups to complete the following tasks in order to obtain a fully functional microprocessor system:

- The first task is to verify that the project board's internal clock circuitry is operational and to generate the correct 16MHz signal.
- A clock divider is then implemented using one of the GAL devices. An 8MHz signal is used as the CPU clock frequency and a 1MHz signal is used as a reference signal for the serial port baud rate generation circuitry.
- The reset circuitry is then designed and implemented in order to debounce the reset-button

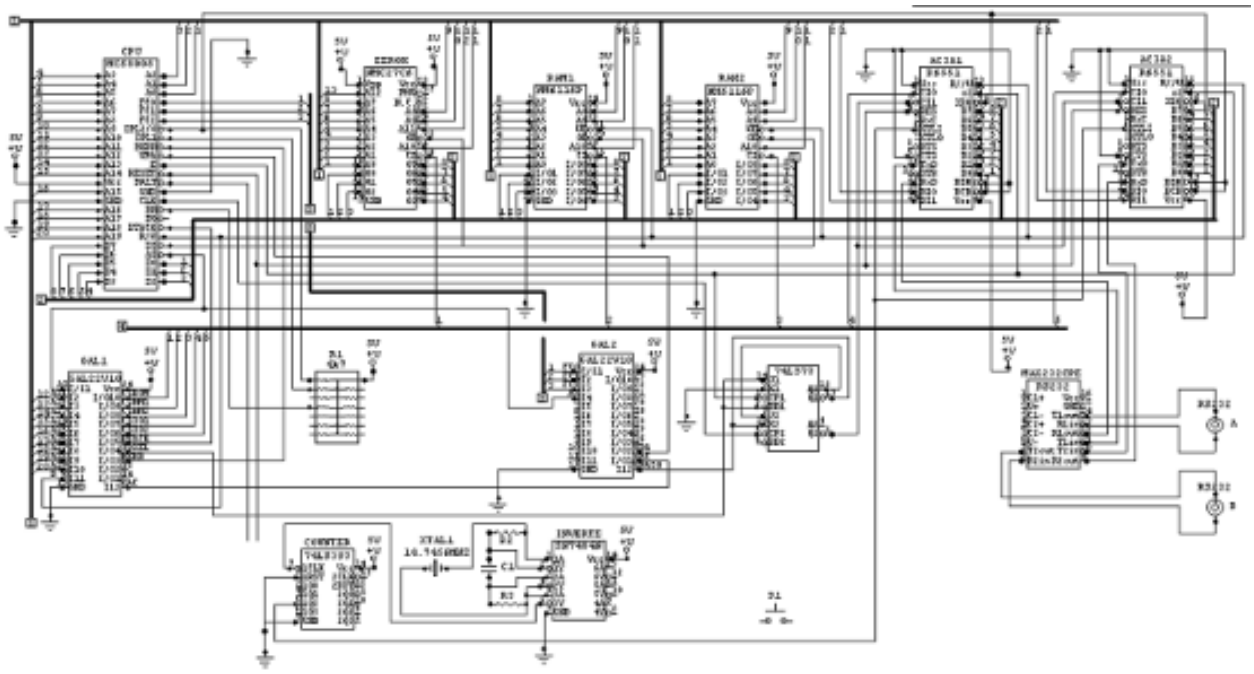

Figure 2: Schematic of the current microprocessor project

and hold the reset signal low for at least three clock cycles in order to allow the processor and peripherals to reset correctly.

- The address space is then segmented into regions where the ROM, RAMs and UARTs are mapped, using logic implemented within the GALs, in order to control the peripheral select signals according to the device memory mappings.

- The EPROM is then programmed with a test program, which allows the students to verify that the core "glue-logic" has been implemented and is functioning correctly.

- Two serial ports are then interfaced with the processor and their functionality tested using a transparent link program, which allows two separate consoles to connect to each other using the microprocessor system.

- At this stage, the project hardware has been completed and a monitor program is now written in assembly code and downloaded onto the EPROM. This is the final task in the completion of the microprocessor design project.

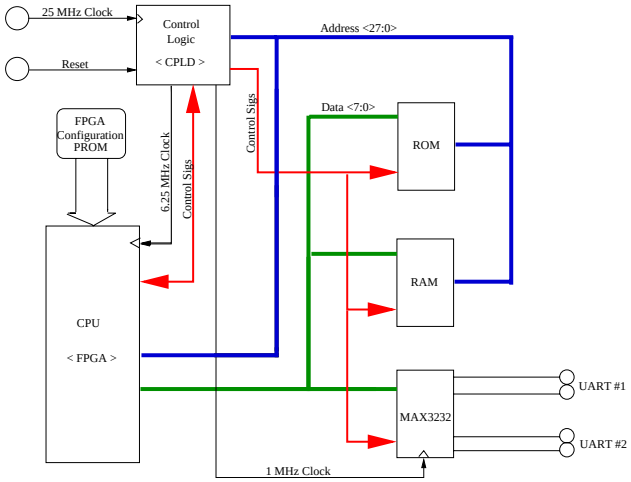


Figure 3: Block Diagram of the Prototype Design

## 3 The Prototype Project

The design for the new project was created using reconfigurable logic devices to implement the CPU and control logic, in conjunction with standard ROM and RAM chips, as shown in Figure 3.

For the purposes of prototyping the design, only one configuration PROM was used to store the soft-CPU. As soon as the board is powered-on, the FPGA is automatically programmed with the contents of

the PROM. This could easily be extended to provide several different PROM configurations, each containing a different soft-CPU, which could allow the user to choose the CPU option that was to be programmed into the FPGA on power-up. This could be accomplished using a switch to select between the different PROM configurations before power was applied to the project board.

The prototype hardware was designed with the intention of providing as much compatibility with the current design project hardware, in terms of layout and components, while taking advantage of the advances in technology since the advent of the original project. This influenced several design decisions, such as using LVTTL components throughout the design.

## 3.1 The Project Board

The hardware platform chosen to be used in prototyping the new project design was the VirtexII FPGA [5] [6] used in conjunction with the VirtexII prototyping board [7] [8]. All of the prototype project circuitry was implemented on the breakout area of the board using LVTTL components which were wire-wrapped together. One ROM chip [9], one RAM chip [10] and two serial ports were added as external devices to prove the operability of the CPU core.

All of the control logic was implemented on the same FPGA that held the CPU core, in order to prove that the design would work. Students undertaking the project, however, will implement the required control logic in a separate FPGA or CPLD to the one containing the CPU.

A standardised bus interface was developed in order to ensure that the hardware design would allow different soft-CPU options to be run, without having to change any physical wiring on the project board. Any processor that implemented this interface could be programmed onto the FPGA and run on the hardware system. It was decided to base the bus interface on that of the Motorola MC68008 processor as this was already in use in the current design project and would facilitate the migration to the new project design.

## 3.2 The LEON Model

The LEON-P1754 microprocessor [1] is a VHDL model of a 32-bit RISC processor conforming to the IEEE-1754 standard, which is fully compatible with the SPARC V8 reference architecture [11]. The model is fully synthesisable and can be implemented on both ASICs and FPGAs. It incorporates

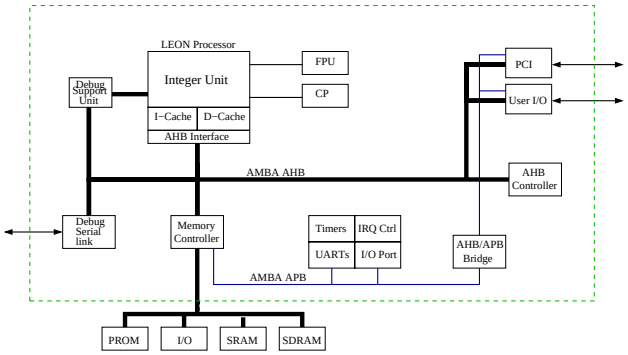


Figure 4: Block Diagram of the LEON core

an integer unit, seperate instruction and data caches and peripheral modules, which are all interconnected via a full implementation of an AMBA APB/AHB bus[12], as shown in Figure 4.

The LEON processor was originally designed by Jiri Gaisler while working for the European Space Agency and is currently under development by Gaisler Research. The first release of the LEON core was made available in October 1999 and has been continuously upgraded and enhanced since then. In order to promote the SPARC standard and enable development of system-on-chip (SOC) devices using SPARC cores, the ESA made the full source code for the processor freely available under the GNU-LGPL license.

This processor[1] was chosen as an alternative to the Motorola MC68008 due to its high level of configurability and proven operability. The processor model, as well as all of its supporting software tools, are freely available from Gaisler Research [2].

The primary method of configuring the LEON core is through the provided graphical configuration utility, which is based around the Linux kernel configuration utility. This method provides options to configure most of the model functionality. However, several major alterations were made to the model source code in order to introduce greater configurability into the model. This was important as the model had to be altered in such a way as to make it compatible with the layout of the current design project.

- The internal data and instruction caches were made removable
- The two internal UARTs were made removable

[1] LEON2-1.0.10-xst was used in the project
[2] http://www.gaisler.com

- The internal reset signal generator was made removable
- The internal memory map was altered to allow the removal of the pre-defined ROM and RAM device mappings
- The internal generation of the bus transaction signalling was altered so that the signals could be suppressed
- A new bus transaction protocol, modelled on the MC68008 protocol, was developed and implemented as a configurable option

All of these changes were implemented in such a way so that their functionality could be enabled or disabled using the graphical configuration utility. In this way, it was possible to obtain the original functionality of the model by setting the appropriate configuration options. This would allow the core to be tailored to specific levels of complexity, depending on the target group of students.

With the core configured to its most basic level, it would be suitable for use as a direct replacement for the Motorola chip in the design project. It would also be possible to include more advanced options in the core, such as the internal UARTs and caches, if a higher level of complexity was required. This means that the processor could be introduced to students at a first and second year level and could be increased in complexity in order to follow them through third and fourth year as new concepts in computer architecture, such as caching, are introduced to them.

### 3.3 Software Tools

Software tools for use with the LEON core are freely provided by Gaisler Research. These include a simulator, debugger and compiler, which can be used in conjunction with the LEON processor.

LECCS is the cross-compiler system for LEON. It can be used to compile C or assembly programs into a format suitable for running on the LEON core.

TSIM is a simulator that emulates the LEON core, which is useful for teaching the students about the SPARCV8 RISC assembly language and allowing them to compile and run programs on a standard pc.

DSUMON is the Debug Support Unit MONitor available from Gaisler Research, suitable for use with the LEON core. It allows access to the contents of the LEON on-chip registers when the DSU option is enabled in the processor and communicates with the board using a dedicated UART.

## 4 Conclusions

The prototype design successfully proved that the modified LEON core is suitable for use as a soft-CPU option on a reconfigurable hardware platform. Due to the implementation of a standardized bus interface within the LEON core, the soft-CPU was successfully able to run on the prototyped board using test programs that had been written in SPARC assembler and compiled using the LECCs compiler for LEON. The resulting hardware system is fully reconfigurable in nature and as a result gives rise to the possibility of implementing and running several different soft-CPU options on the prototype board, without having to make any modifications to the hardware.

The LEON core used as the soft-CPU could be synthesised with many different configurations, allowing a wide variety of hardware options to be tested on the board. Benchmark programs would allow the comparison of different processor configurations and would be useful for highlighting performance differences due to the various architectural features, such as whether cacheing is enabled or not, cache size, pipelining, etc. These physical comparisons would be of use to different student groups, depending on their level of knowledge about computer architectural concepts.

The "hands-on" nature of the project allows students to experiment with different processor architectures and configurations, aiding in the teaching process by adding a practical side to the theory and concepts being taught.

## 5 Future Work

The work described in this paper established the basic hardware design and component requirements of a reconfigurable hardware board using a synthesisable VHDL model of a microprocessor. The next task leading on from this work is to design and build a dedicated hardware board based on the prototype project design. This board could include the option to store either multiple configurations of the LEON core (with varying degrees of complexity) or several different processor architectures (implementing the standardized bus interface protocol) on the one board, using multiple user configuration PROMs.

Several different processor architectures could also be evaluated on the hardware system, for example, a synthesisable VHDL model of the MC68008 could be designed and implemented. The only constraints for implementing a processor architecture on the reconfigurable hardware board are that they implement

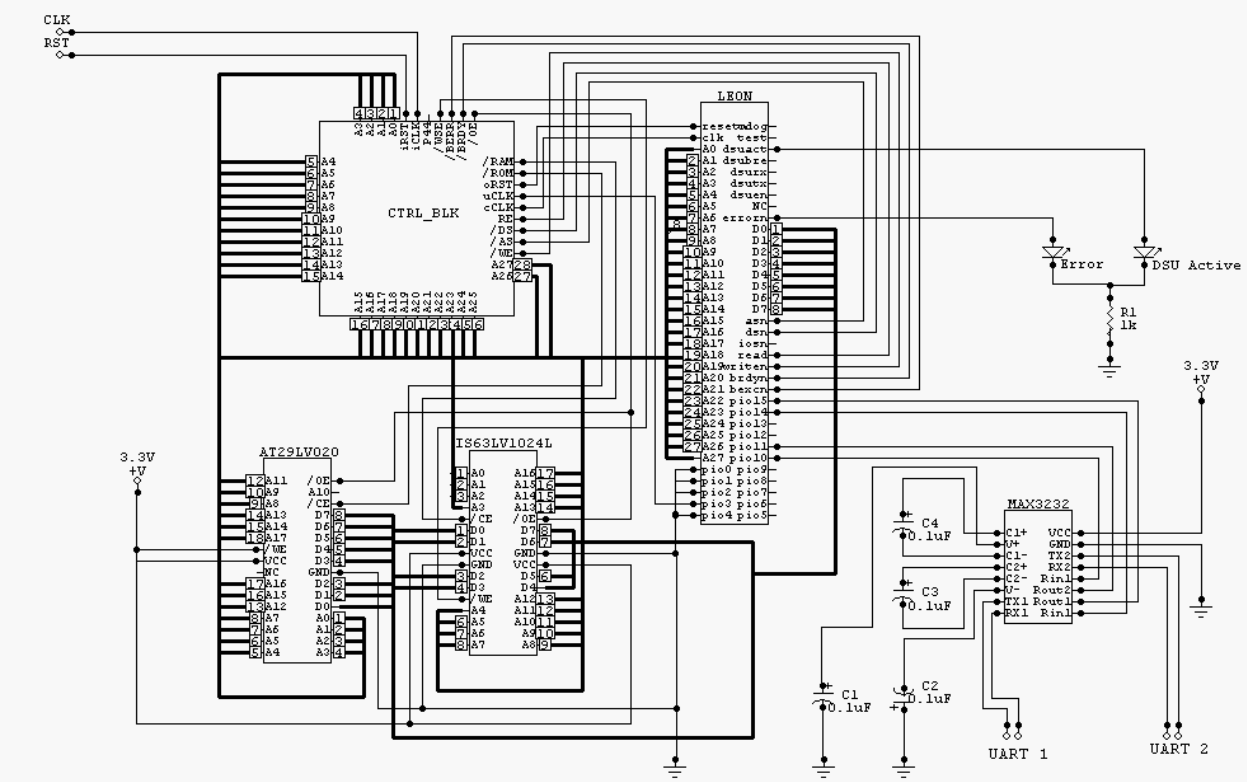


Figure 5: Schematic of the Prototype Design Project

the standardized bus interface.

## References


[1] J. Gaisler, *Leon2-1.0.10 Users Guide.* Gaisler Research, December 2002. http://www.gaisler.com.

[2] Murray Pearson, Dean Armstrong, Tony McGregor, "Using Custom Hardware and Simulation to Support Computer Systems Teaching," *Workshop on Computer Architecture Education*, 2002.

[3] Daniel Ellard, Daivd Holland, Nicholas Murphy, Margo Seltzer, "On the Design of a New CPU Architecture for Pedagogical Purposes," *Workshop on Computer Architecture Education*, 2002.

[4] J. Gray, "Hands-on Computer Architecture - Teaching Processor and Integrated Systems Design with FPGAs," 2001.

[5] Xilinx, *Introduction to the VirtexII Product Family.* Xilinx Inc, December 2001. http://www.xilinx.com.

[6] Xilinx, *VirtexII Advance Product Specification.* Xilinx Inc, September 2002. http://www.xilinx.com.

[7] Xilinx, *VirtexII Prototype Platform Users Guide.* Xilinx Inc, June 2001. http://www.xilinx.com.

[8] Xillinx, *VirtexII Platform FPGA Handbook.* Xilinx Inc, December 2001. http://www.xilinx.com.

[9] Atmel, "Atmel AT29LV020 Flash Memory," tech. rep., Atmel Corporation, May 2002. http://www.atmel.com.

[10] ISSI, "ISSI IS63LV1024L Static RAM," tech. rep., Integrated Silicon Solutions Inc, July 2002. http://www.issi.com.

[11] SPARC, *SPARC V8 Manual.* SPARC International Inc, January 1992. http://www.sparc.org.

[12] ARM, *AMBA Specification V2.0.* ARM Limited, May 1999. http://www.arm.com.

# Use of HDLs in Teaching of Computer Hardware Courses

Zvonko Vranesic and Stephen Brown
University of Toronto
{zvonko@eecg.toronto.edu}

## Abstract

A modern treatment of an introductory course on the design of logic circuits should include an early introduction of a hardware description language (HDL). This can done without sacrificing the emphasis on fundamental concepts of logic circuit design. An example of how this may be achieved is given.

## 1 Introduction

This presentation focuses on the use of hardware description languages and design automation tools in the teaching of courses on logic circuits and computer architecture. It is based on the experience gained at the University of Toronto, which involved courses in Computer Engineering, Electrical Engineering, and Computer Science programs.

While there has been considerable debate about the optimal way of structuring the courses that teach the concepts of computer hardware, a traditional sequence based on an introductory course in the design of logic circuits and a subsequent course in computer organization (architecture) is still a very attractive option. It is even better if these courses are followed by a more advanced course in computer architecture. This is the structure at the University of Toronto. Each course is accompanied by a laboratory in which students develop a real understanding of the key concepts and the various ways in which they may be implemented in practice. For the purposes of this discussion, we will assume just the basic two-course sequence.

## 2 Logic Circuits

A course in logic circuits can be taught effectively as soon as the students have acquired an understanding of some high-level programming language and have learned the fundamentals of good programming practices. The course should emphasize the important concepts which include the notions of implementability, cost, optimization, timing, stability, and performance. The amount of material that can be covered depends on the length of the course, the ability of the students, and the quality of the supporting facilities comprising laboratories and CAD tools.

While everybody agrees that the students must learn about logic functions and their implementation, arithmetic circuits, multiplexers, decoders, flip-flops, counters, finite-state machines, and other standard circuits, there is less agreement about the means used to expose students to this material. In particular, when and how should the students discover CAD tools, and what laboratory exercises provide the best learning experience?

Today it is highly advisable to introduce a hardware description language (HDL) as soon as possible. Without an HDL it is impossible to exploit properly the capabilities of CAD tools and FPGA-based laboratory equipment. A prudent choice of HDL is either Verilog or VHDL. It should be noted that Verilog is winning the battle in the industrial environment of North America, so it is likely that it will gain greater favor with academics in the near future.

Our experience shows that the HDL can be introduced surprisingly early. Moreover, the instructor need not spend an inordinate amount of time teaching the intricacies of the language. Students are keen to learn and use the HDL because of its obvious practical value, hence they are willing to learn on their own many details that are illustrated in examples given in the textbook. During lectures, the instructor has to focus on explaining the important differences between the HDL and computer programming languages that students are familiar with. For example, explaining the key differences between Verilog and C can lead to fascinating lectures. Since the HDL will be used in laboratory exercises, which requires a certain amount of homework preparation, the material that should be taught in the classroom may be covered in as little as three to four lectures. This approach is particularly viable if the textbook integrates efficiently the discussion of logic circuit concepts and their possible HDL descriptions.

## 3 Introducing HDL - A Practical Approach

A good understanding of computer hardware must be based on a good understanding of underlying logic circuits. An HDL, particularly when used at the behavioral level, can mask many important aspects of logic circuits. Therefore, it is important to find a good balance between teaching the students the essence of

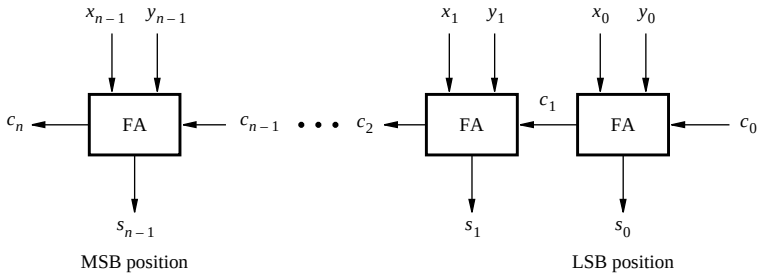


Figure 1: An $n$-bit ripple-carry adder.

circuits and the efficiency of design using the HDL and CAD tools. It is particularly important that using the HDL does not obscure the existence of fundamental logic blocks such as gates and flip-flops. To illustrate this notion, we will consider an example based on a ripple-carry adder, using Verilog as the HDL [1].

Figure 1 shows the general structure of an $n$-bit ripple-carry adder, comprising a cascade of full-adder circuits. Knowing that the full-adder is defined by the logic expressions

$$\begin{aligned} s &= x \oplus y \oplus Cin \\ Cout &= x \cdot y + x \cdot Cin + y \cdot Cin \end{aligned}$$

it is easy to visualize the functionality of the cascaded circuit, as well as the propagation delay due to the rippling of the carry. Using these expressions, the full-adder can be defined in Verilog as shown in Figure 2.

```
module  fulladd (Cin, x, y, s, Cout);
   input  Cin, x, y;
   output  s, Cout;

   assign  s = x ^ y ^ Cin;
   assign  Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Figure 2: Verilog code for the full-adder.

Now, we can specify a ripple-carry adder structurally as indicated in Figure 3. To keep the example simple, this specification defines the inputs $X$ and $Y$, as well as the sum $S$, as four-bit vectors. The internal carries are defined as a three-bit vector $C$. The structure of the resulting circuit is the same as in Figure 1.

This would be an awkward way of describing a larger $n$-bit adder, so we can use a generic specification instead. The ripple-carry adder in Figure 1 can be described using the expressions

$$\begin{aligned} s_k &= x_k \oplus y_k \oplus c_k \\ c_{k+1} &= x_k y_k + x_k c_k + y_k c_k \end{aligned}$$

```
module  adder4 (carryin, X, Y, S, carryout);
   input  carryin;
   input  [3:0] X, Y;
   output  [3:0] S;
   output  carryout;
   wire  [3:1] C;

   fulladd  stage0  (carryin, X[0], Y[0], S[0], C[1]);
   fulladd  stage1  (C[1], X[1], Y[1], S[1], C[2]);
   fulladd  stage2  (C[2], X[2], Y[2], S[2], C[3]);
   fulladd  stage3  (C[3], X[3], Y[3], S[3], carryout);

endmodule
```

Figure 3: A four-bit adder.

for $k = 0, 1, \ldots, n-1$. A Verilog specification of the adder based on these expressions is given in Figure 4. It is apparent that this approach also implements the cascaded adder structure. Next, the students should learn that Verilog includes higher-level constructs for specification of commonly used circuits. One such construct uses the arithmetic assignment statement, which allows the adder to be specified as shown in Figure 5. We can add to this circuit the capability to produce the carry-out and arithmetic overflow signals as presented in Figure 6. The expressions for these two signals are

$$\begin{aligned} carryout &= x_{n-1}y_{n-1} + x_{n-1}\overline{s}_{n-1} + y_{n-1}\overline{s}_{n-1} \\ overflow &= carryout \oplus x_{n-1} \oplus y_{n-1} \oplus s_{n-1} \end{aligned}$$

They can be derived as a useful exercise.

A more elegant way of specifying the same circuit is given in Figure 7. It uses an $(n+1)$-bit vector named *Sum*. The extra bit, *Sum*$[n]$, becomes the carry-out from bit position $n-1$ in the adder. To make the addends $n+1$ bits long, the vectors $X$ and $Y$ have a zero concatenated on the left side. This conveniently introduces the Verilog concatenate operator. It also ensures that the students see $X$ and $Y$ as bits in a circuit rather than just numbers.

Having introduced the idea of concatenation in Verilog, our circuit can be defined more compactly as shown in Figure 8. Finally, at this point the students can see that even the full-adder circuit can be defined behaviorally as depicted in Figure 9.

The progressive sequence of showing the students different ways in which Verilog code can specify an $n$-bit adder teaches a number of important aspects of Verilog. It shows the difference between structural and behavioral approaches in defining circuits. It illustrates how a **for** loop generates a cascade by replicating a subcircuit $n$ times. It indicates that powerful statements, such as the arithmetic assignment statement, exist which lead to simple and easily understandable code. It also shows how clever ideas, exploiting the notion of concatenation in our example, can be used to good effect.

It is important to understand that a behavioral specification will not necessarily lead to a circuit structured in the form that the designer may envisage, perhaps as learned from a textbook. When the Verilog compiler of a given CAD tool encounters a construct used to define a commonly used circuit, it will attempt to use a predefined module from a library of parameterized modules provided with the CAD tool. Moreover, if the designed circuit is to be implemented in a technology such as an FPGA, then the final implementation will be in the form of logic elements used in a particular FPGA device. Thus, the implementation may not involve the basic gates that one saw in the lectures!

# 4 Computer Organization

A good laboratory is essential for conveying the essence of the various structures found in computer systems. The students should learn what a computer looks like through the eyes of a programmer interested primarily in using the machine and a designer intent on developing the hardware needed to build systems. At the University of Toronto we use the Ultragizmo board, which is a custom board containing a Motorola 68000-based microcontroller device, an FPGA device and a variety of interfaces that allow the board to be connected to our main laboratory system which includes a full networking capability. We also use Altera's UP-1 boards, which include programmable logic devices, to provide additional capability.

Typical experiments include investigation of simple I/O using parallel and serial ports, interrupts, adding SRAM chips, DMA controllers, design of arithmetic circuits, A/D and D/A interfaces, and various applications such as processing of sound and controlling simple Lego-implemented robots. At the end of our courses, we tend to have a three-week project for which students may implement anything that the instructor deems interesting. The project may entail even designing a simple processor, or building an interesting system based on an existing soft-core processor that may be instantiated in the FPGA.

The FPGA device makes it possible to quickly implement relatively complex circuits needed in specific experiments. Of course, these circuits have to be specified in the HDL. This raises an interesting question about the necessary competence of students in terms of the HDL use. More specifically, is the knowledge gained in the introductory logic course sufficient to deal successfully with more ambitious designs needed in this laboratory.

Instructors habitually complain that the students do not know as much as they should. Specifically, given a rather limited exposure to the HDL in the logic course, they cannot immediately tackle more demanding tasks in the computer organization laboratory. Indeed, this may be true, particularly in the case of weaker students. However, with just a single refresher tutorial on DOs and DON'Ts of the HDL the students can be brought to a level where they can handle the requirements of the laboratory. As with most subjects, an early exposure to the HDL, followed by a short review and subsequent intensive use in the follow-on course, will leave the students with reasonable competence and a great deal of self-satisfaction.

The described approach has been successful in our practice. It has led to the development of three books [1-3]. The feedback from our students has been very positive.

# 5 References


1. S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with Verilog Design*, McGraw-Hill, 2002.

2. S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, McGraw-Hill, 2000.

3. V.C. Hamacher, Z. Vranesic and S. Zaky, *Computer Organization, ed. 5*, McGraw-Hill, 2002.

```
module  addern (carryin, X, Y, S, carryout);
   parameter  n = 32;
   input  carryin;
   input  [n−1:0] X, Y;
   output  [n−1:0] S;
   output  carryout;
   reg  [n−1:0] S;
   reg  carryout;
   reg  [n:0] C;
   integer  k;

   always @(X or Y or carryin)
   begin
      C[0] = carryin;
      for (k = 0; k < n; k = k+1)
      begin
         S[k] = X[k] ^ Y[k] ^ C[k];
         C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
      end
      carryout = C[n];
   end

endmodule
```

Figure 4: A generic specification of a ripple-carry adder.

```
module  addern (carryin, X, Y, S);
   parameter  n = 32;
   input  carryin;
   input  [n−1:0] X, Y;
   output  [n−1:0] S;
   reg  [n−1:0] S;

   always @(X or Y or carryin)
      S = X + Y + carryin;

endmodule
```

Figure 5: Specification of an $n$-bit adder using arithmetic assignment.

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n−1:0] X, Y;
   output [n−1:0] S;
   output carryout, overflow;
   reg [n−1:0] S;
   reg carryout, overflow;

   always @(X or Y or carryin)
   begin
      S = X + Y + carryin;
      carryout = (X[n−1] & Y[n−1]) | (X[n−1] & ∼S[n−1]) | (Y[n−1] & ∼S[n−1]);
      overflow = carryout ^ X[n−1] ^ Y[n−1] ^ S[n−1];
   end

endmodule
```

Figure 6: An $n$-bit adder with carry-out and overflow signals.

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n−1:0] X, Y;
   output [n−1:0] S;
   output carryout, overflow;
   reg [n−1:0] S;
   reg carryout, overflow;
   reg [n:0] Sum;

   always @(X or Y or carryin)
   begin
      Sum = {1'b0, X} + {1'b0, Y} + carryin;
      S = Sum[n−1:0];
      carryout = Sum[n];
      overflow = carryout ^ X[n−1] ^ Y[n−1] ^ S[n−1];
   end

endmodule
```

Figure 7: A different specification of $n$-bit adder.

```
module  addern (carryin, X, Y, S, carryout, overflow);
   parameter  n = 32;
   input  carryin;
   input  [n−1:0] X, Y;
   output  [n−1:0] S;
   output  carryout, overflow;
   reg  [n−1:0] S;
   reg  carryout, overflow;

   always @(X or Y or carryin)
   begin
      {carryout, S} = X + Y + carryin;
      overflow = carryout ^ X[n−1] ^ Y[n−1] ^ S[n−1];
   end

endmodule
```

Figure 8: Simplified complete specification of an $n$-bit adder.

```
module  fulladd (Cin, x, y, s, Cout);
   input  Cin, x, y;
   output  s, Cout;
   reg  s, Cout;

   always @(x or y or Cin)
      {Cout, s} = x + y + Cin;

endmodule
```

Figure 9: Behavioral specification of a full-adder.