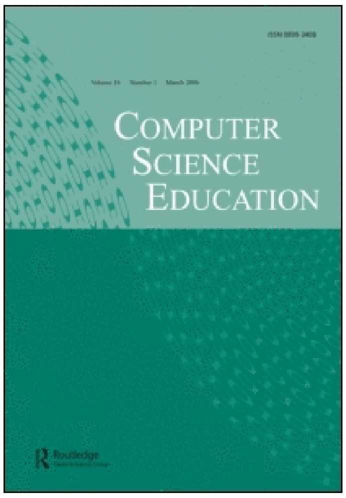

Computer Science Education

Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713734307

From Archi Torture to architecture: Undergraduate students design and implement computers using the Multimedia Logic emulator

Timothy D. Stanley [a]; Lap Kei Wong [a]; Daniel Prigmore [a]; Justin Benson [a]; Nathan Fishler [a]; Leslie Fife [a]; Don Colton [a]
[a] Brigham Young University Hawaii, USA

Online Publication Date: 01 June 2007

To cite this Article Stanley, Timothy D., Wong, Lap Kei, Prigmore, Daniel, Benson, Justin, Fishler, Nathan, Fife, Leslie and Colton, Don(2007)'From Archi Torture to architecture: Undergraduate students design and implement computers using the Multimedia Logic emulator',Computer Science Education,17:2,141 — 152

To link to this Article: DOI: 10.1080/08993400601165735

URL: http://dx.doi.org/10.1080/08993400601165735

Computer Science Education
Vol. 17, No. 2, June 2007, pp. 141–152

From Archi Torture to Architecture: Undergraduate students design and implement computers using the multimedia logic emulator

Timothy D. Stanley*, Lap Kei Wong, Daniel Prigmore, Justin Benson, Nathan Fishler, Leslie Fife and Don Colton
Brigham Young University Hawaii, USA

Students learn better when they both hear and do. In computer architecture courses "doing" can be difficult in small schools without hardware laboratories hosted by computer engineering, electrical engineering, or similar departments. Software solutions exist. Our success with George Mills' Multimedia Logic (MML) is the focus of this paper. MML provides a graphical computer architecture solution with convenient I/O support and the ability to build and emulate a variety of computer designs. It has proven highly motivational to upper division computer science students designing and constructing emulated computers. Student projects resulted in excellent student understanding of the detailed inner workings of computers. Students also developed better teamwork skills and produced useful training aids for the lower division computer organization class. Designs implemented include 8 bit and 16 bit von Neumann and Harvard architectures, from single cycle to 12 cycle instructions. Issues resolved during the learning process include timing, initialization, instruction set architecture, and I/O and assembler design. We discuss pedagogical issues involved in using MML to implement instructor and student computer designs. MML is compared to using a hardware-based Intel 8085 microprocessor basic systems course.

1. Introduction

Effective learning comes from doing, not just hearing. Computer architecture is an area where doing is a natural extension of the course material. There are many approaches to teaching a laboratory component for computer architecture. These include no laboratory at all, emulated hardware, actual hardware, and a mix of emulated and actual. Software tools range from register and memory level computer simulators to high level chip design languages. Between these extremes is George

*Corresponding author. School of Computing, Brigham Young University Hawaii, Laie, HI 96762, USA. E-mail: stanleyt@byuh.edu

ISSN 0899-3408 (print)/ISSN 1744-5175 (online)/07/020141-12
© 2007 Taylor & Francis
DOI: 10.1080/08993400601165735

Mills' Multimedia Logic (MML). We discuss the use of computer simulators and chip definition languages. We then introduce MML and our approach.

1.1. Computer/Logic Simulators

Many computer architecture classes use emulators. The popular textbook *Computer organization and design* (Paterson & Hennessy, 1994) features SPIM, a MIPS32 simulator, by James Larus (http://www.cs.wisc.edu/~larus/spim.html). It shows memory contents and registers and includes an assembler, but does not simulate the data path.

Moving closer to our goal, many computer architecture classes use the text by Null and Lobur (2003a) that introduced MARIE. They may also use MarieSim (Null & Lobur, 2003b). MARIE uses a von Neumann architecture with a simple instruction set. Using MarieSim students can write their own programs and watch them execute, seeing the effect these programs have on system state. MARIE has been implemented with a "data path simulator" that highlights registers and data paths visually. This shows the steps the processor goes through when running a program. However, students cannot build their own computers and simulate them.

At the other end of the complexity spectrum is implementation of a simulator of an entire real architecture. Clark, Czezowski, and Strazdins (2001) implemented a simulator for the Sparc V9. Even if it had a GUI, this is of limited usefulness for an undergraduate course in computer architecture. The level of complexity this embodies is not a good starting point for learning computer architecture. In addition, pedagogically, learning only one machine's architecture is limiting. The Alfa-1 simulator (Wainer, Daicz, Simoni, & Wassermann, 2001) also works specifically with the Sparc processor. This tool allows the user to experiment with the entire architecture, including extending it in some ways. However, the limited user interface and the restriction to a single architecture are limitations. The Alfa-1 was designed to replace tools that simulated obsolete architectures. This approach only ensures that an eventual replacement for Alfa-1 will be needed.

Some simulation tools address only a single problem. The KScalar simulator (Moure, Rexachs, & Luque, 2002) provides a tool for learning about microprocessors. This is, of course, only part of what must be covered in a typical computer architecture course. However, this might be a good choice for an advanced course on microprocessors or for a few laboratory assignments within a larger course. A similar approach was used by Holland, Harris, and Hauk (2003). They provide an incomplete processor and have students design the missing pieces. They can simulate any non-floating point instruction in their 8 instruction MIPS processor. This creates an opportunity for students to learn processor details. However, the student is still limited to only a single processor type and the absence of floating point. Other parts of the machine architecture must still be learned in some fashion. Another single purpose tool is SIMT (Tao, Schulz, & Karl, 2003). This simulator allows the evaluation of shared memory systems. While these tools may be very useful in specialized computer architecture courses, having to learn several unconnected single purpose tools has limited usefulness in a general computer architecture course.

1.2. Chip Design Languages

Several chip design languages are available.

Logisim (Burch, 2002) is another design and simulation tool. A basic package, Logisim allows the user to design circuits from basic logic components and some basic devices. The primary drawbacks are the lack of a clock for timing and the somewhat primitive I/O capabilities.

Logic Works 5 from Capilano Computing (http://www.capilano.com) is a wonderful package, but the emphasis is on detailed timing and simulation of circuits to be exported to silicon. It seems more suited to advanced students that already grasp computer architecture and are looking for the next step. It does not have the input and output devices available in MML.

DLSim by Matthew Leslie (http://urchin.earth.li/~mleslie/project.html) was developed while an undergraduate student with additional development funded by Cambridge. It is an open source Java program available from http://www.sourceforge.net/. DLSim has the capability to develop macros and can display logic states propagated through the circuits. However, the devices are blocks and the rich set of I/O devices available in MML is not available.

1.3. Multimedia Logic (MML)

The package we chose was MML, open source free software by George Mills (free download available at http://www.softronix.com). MML strikes a good depth of detail. We can design and implement logic at the gate level, but still use high level I/O operations. (This is parallel to the C programming language. C provides near-assembler access to the machine, but still includes a standard I/O library in the basic distribution.) We have seen how a variety of tools exist to simulate imaginary and real computers based on simple and complex instruction sets. Generally these tools allow the simulation of only a single architecture and they often hide many of the underlying details that we might like to reveal. You can write programs and see the results in various registers, but the computer itself remains a black box. Alternatively, some tools allow the student to access the logic gate level of design but have limited I/O capabilities.

We start our discussion of MML with a "Hello World" example, shown in Figure 1. This seems to be the first program shown in every programming language text, and it is fun to do it in an architecture course as well.

MML allows the student to define an instruction set and build the enabling architecture. This means the student is not limited to von Neumann computers or the instruction sets designed by others. With an excellent graphical user interface and ASCII I/O, the tool is easy to learn and use. Because MML is open source software, functions can be added and the tool recompiled. Despite being easy to use, it is

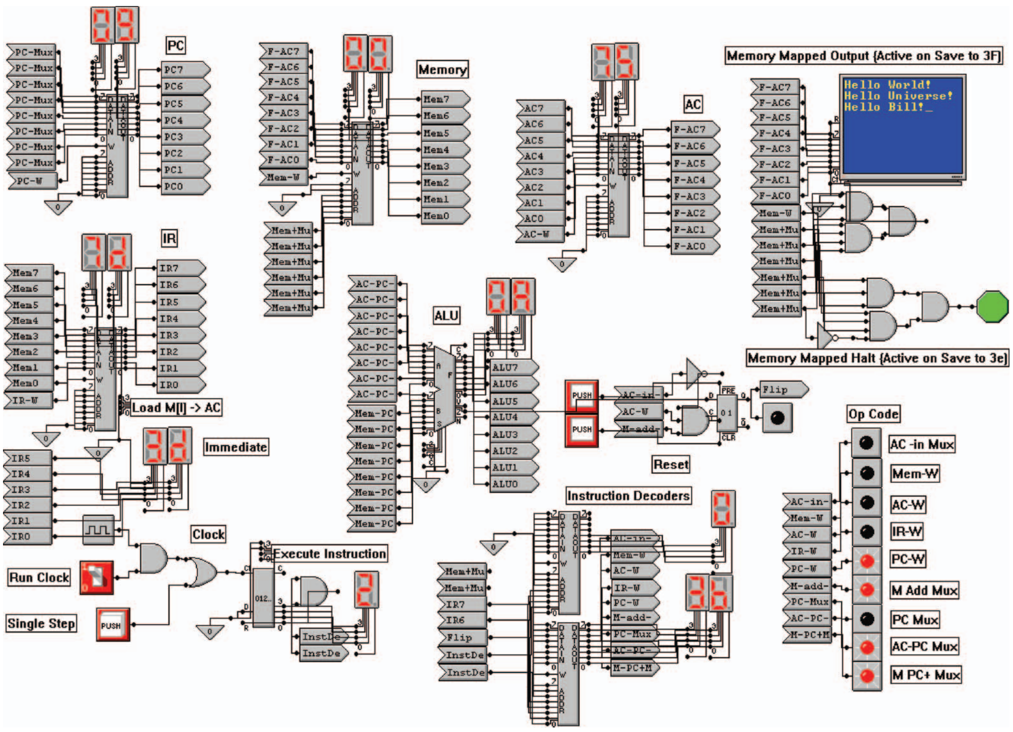

Figure 1. An example computer using an 8 bit von Neumann design

capable of sophisticated designs. For example, see the work of James Larson (http://www.dst-corp.com/james/MMLogic.html).

MML is a very decent environment for virtual computer construction. We were able to implement all of the necessary components in order to run programs written for MARIE. The benefits of MML for our project were extensive. The ease of use compared with physical hardware is obvious. In addition, MML uses abstraction to simplify the hardware components it offers and reduces the need to build all of one's own components, such as an arithmetic logic unit.

1.4. Architecture Course Design

Students are shown how to build a computer. Starting from an instruction set architecture and continuing through the building of emulated hardware to implement the instruction set, our students are required to design and build their own emulated computer. Students feel empowered and are highly motivated to participate in the laboratory assignments and survive the debugging process. They report having a great sense of accomplishment and achieving a profound understanding of how computers work at the logic level.

In the most recent semester to set the stage for this assignment the instructor provided as an example a 4 instruction, 8 bit, accumulator-based, von Neumann design that used three clock cycles to execute each instruction. The instructions are: load the accumulator from memory, save the accumulator to memory, add from a memory location to the accumulator, and jump if the last add produced a zero result. These four instructions were supplemented by two memory mapped commands, output on save to memory location x3F, and halt on save to memory location x3E. Figure 1 shows the main page of this computer running a "Hello World" program. A second page of multiplexers is not shown.

An alternative example also demonstrated was the single cycle, 8 bit, Harvard architecture, described in the *Proceedings of the Workshop on Computer Architecture Education* in 2005 (Stanley, 2005). After working through these example designs, students were challenged to invent an original design, including an instruction set, and the registers to support the instructions. Then they were asked to implement the design in MML.

2. Student Project Reports

Three team projects are the focus of this paper. Each team was comprised of three or four upper division undergraduate students. The first team implemented a 16 bit von Neumann architecture with a shared instruction and data memory system. The second team implemented a 16 bit Harvard architecture computer with separate memory for instructions and data. The third team implemented a computer in actual hardware using an Intel 8085 hardware trainer called the Micro-Mater MM-8000 from Elenco Electronics. Each project report is an edited version of what was written by the students.

2.1. 16 bit von Neumann Design

One three student team used as its basis the MARIE computer architecture described in Null and Lobur (2003a). MARIE stands for "machine architecture that is really intuitive and easy," although our group found that was not quite the case. It is a 16 bit von Neumann design with thirteen instructions and seven registers. The registers include: an accumulator (AC), a memory address register (MAR), a memory buffer register (MBR), a program counter (PC), an instruction register (IR), an input register (InREG), and an output register (OutREG).

The instructions are 16 bits, of which the most significant four constitute the op code and the least significant twelve the address. Instructions are variable in length and range from three to thirteen steps per instruction. The instructions include: load, store, add, subtract, input, output, halt, skip conditional, jump, store and jump, clear, add indirect, and jump indirect.

2.1.1. Design features. Our design uses two 16 bit ALUs. One is for incrementing the program counter and the other is for the add and subtract commands and logical tests for the skip conditional command. One novel simplifying device used is the MML node device. This device can be set to propagate a low state when it receives an unknown state. This capability was used to initialize the register arrays.

This design was implemented in twelve pages of MML. The pages include the memory address register, the memory buffer register, the accumulator with ALU, the input register, the output register, the instruction register, the memory, the program counter with incrementer and multiplexer, the bus, the master control module with clock and instruction decoder, the conditional skip logic, and the I/O module. The accumulator module is given in Figure 2. Figure 3 shows the master control module with instruction decoder and diagnostic LEDs.

2.1.2. Design challenges. One of the biggest challenges of this design was the instruction decoding. Some instructions, like clear, take four clock cycles, while others like save and jump take twelve clock cycles. Designing the decoder ROM with its interface to control the various multiplexers and logic devices was a difficult challenge.

2.1.3. Final product. This computer is able to run programs assembled in Linda Null's MARIE Java emulator (Null & Lobur, 2003b). The machine language instructions are copied into small text files that are loaded into memory when simulation starts. We tested a number of programs to verify operation of the design features.

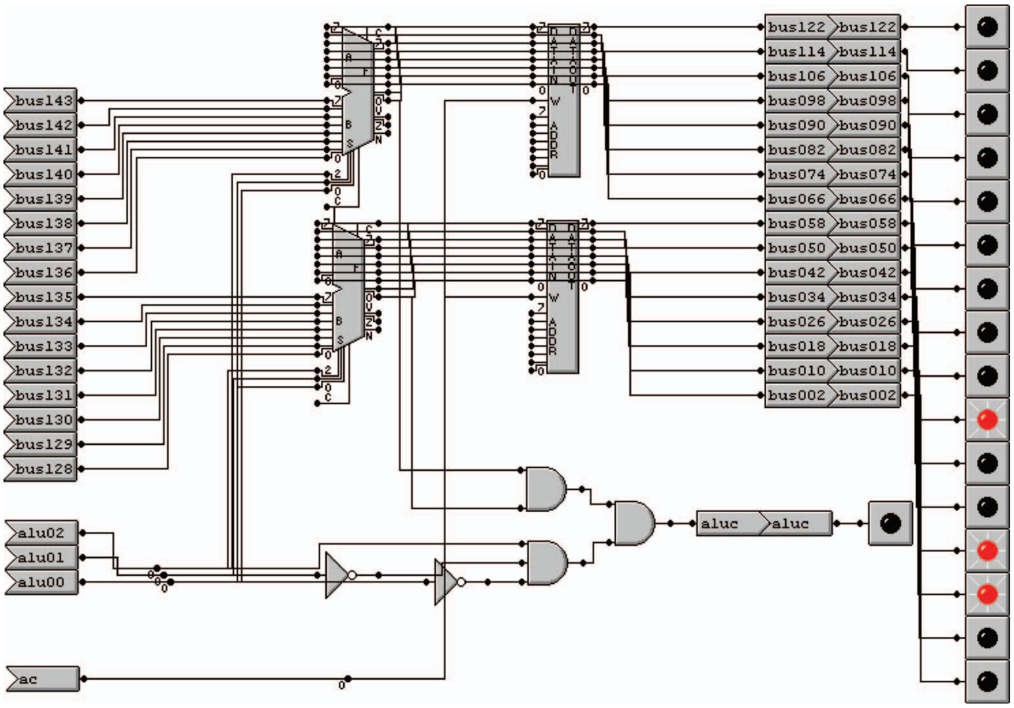

Figure 2. The accumulator module for the 16 bit von Neumann design

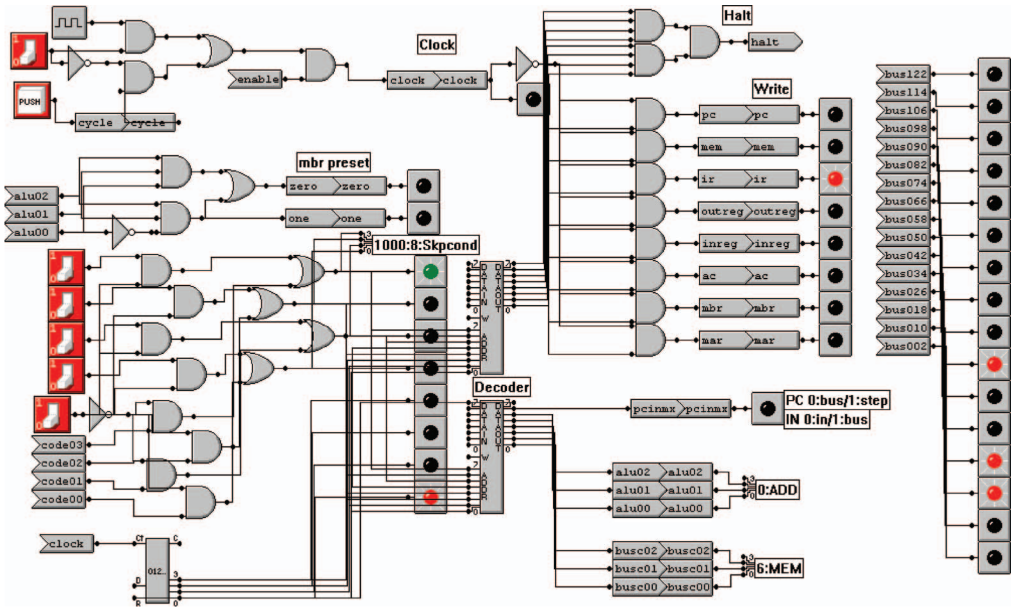

Figure 3. Control module for the 16 bit von Neumann design

2.1.4. Motivation to learn and excel. The motivation to succeed and excel at the task of building a computer was largely intrinsic in nature and was rooted in the computer science program at the university, as well as in the professor. It was certainly not the prospect of achieving a high grade in the course that inspired the hard work our group undertook. Many other and simpler ideas for computers could have been implemented to achieve the desired grade. Instead, the goal for each member of our small team was to understand how computers work, not necessarily how this particular machine worked. This approach made a huge difference for our project because an understanding of the various components that are used to build a computer allowed us to try different ways to accomplish a task when previous attempts failed. Additionally, this approach helped the group develop a machine that was as simple as possible while still being functional.

A motivating factor was the team approach, which ensured that another person was almost daily monitoring the effort you put into the project and lazy or sluggish work would be noticed and not be tolerated. This forced each member of the group to contribute and work hard.

Furthermore, at the outset of the project the professor demonstrated, step by step, how to build a program counter (PC). The program counter is a simple component, but the process of creating it gave the impression that building a computer was not as ominous a task as at first seemed. This instilled feelings of confidence about our abilities and boosted the morale of the group.

The fear of failure may have been another influencing factor in the success of the project because, in addition to building the computer, the class was also required to give a presentation to the class and other faculty members. With this presentation looming the group worked hard in order to complete the project on time and with a thorough understanding of the course material.

This project was very motivating for our team. We spent many hours ensuring that a quality project was built. We learned a lot through the design and assembly, as well as in the debugging. Our fellow students also learned from our presentation of this project and we know our instructor will be using it as a training aid in his computer organization class.

2.2. 16 Bit Harvard Architecture Design

Three students created an architecture they called "super computer," because it stretched the limits of MML. The group's goal was to make a 16 bit general function computer where all instructions took only a single clock cycle.

2.2.1. Design features. To create a general function computer we began with the instruction set: add, sub, multi, branchNE, branchLT, move, jump, load, print, input, halt. This required at least 4 bits for the op code, and thus up to 16 instructions could be supported. Eventually we added jumpI and regI for ease and write, div, and mod, because the instruction set did not fulfill the functionality intended. We made sixteen 16 bit registers so that each register address would use 4 bits, giving 4 bits for the op code and up to three 4 bit register addresses for an instruction. The immediate values were only able to be 12 bits. Most of these instructions are obvious as to their function, but a couple may need to be explained.

BranchLT is a conditional jump when the first register is less than the second. BranchNE is similar but the condition is 'not equal to'. The load and write instructions read and write to an address in the memory from a register. The regI instruction loads the immediate value, which is the last twelve bits of the instruction, into the default register, which is register 15 (from 0 to 15, thus 16 registers). The print and input instructions are more specific to MML. The team wanted general functionality for users so I/O was added into the instruction set.

This computer requires nine pages of multimedia logic to build. One of these, the user interface module, is shown in Figure 4.

2.2.2. Design challenges. One of the biggest design challenges was to develop the fully multiplexed 16 bit register, 16 register array so that any register can be used for any location in the instruction. An additional challenge was the multiplier. The ALU in MML provides only 8 bit results, so even though a multiply instruction is given, without a 16 bit result only a 4×4 bit multiply can be guaranteed to be accurate. Since we wanted the capability to do eight bit multiplies we built our own multiplier module.

Another problem the team fixed was with the load and write instructions and the memory. The way that memory works in MML is that there are three inputs, which are data in, address line, and write data, with one output, which is the contents of the

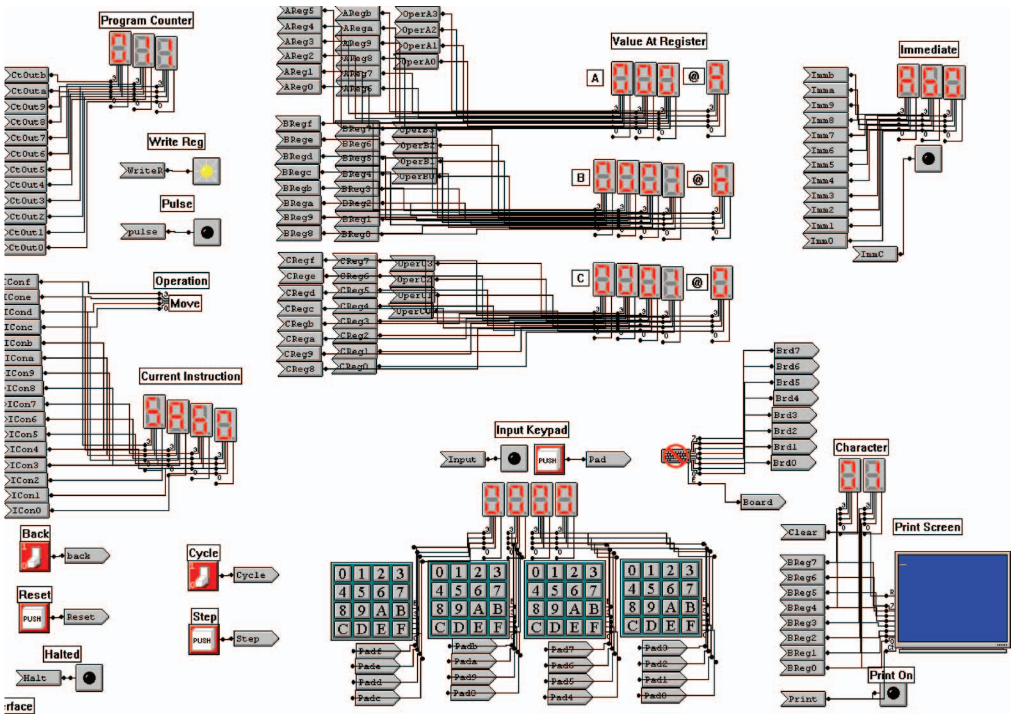

Figure 4. User interface for the Harvard architecture design

memory at the address line. As the program counter directed the memory to the current instruction the load instruction had to change the address to access the needed data, but this in turn would change the current instruction, which would cause unwanted results. To fix this we used the high and low cycle in the clock pulse so that on the high pulse for the load command the current instructions would be saved to a temporary buffer and the address of the memory would be changed to the needed load address and copied. Then on the low cycle the address would revert back to the program counter. This was like a two cycle process but only using one clock pulse.

2.2.3. Final product. This design features nine pages in MML. To assemble programs an assembler was developed in Microsoft Excel. The design works very well. To illustrate this design the user interface is included as Figure 4.

2.2.4. Learning outcomes. More important than the computer architecture we designed was the understanding of computer architecture we acquired. When the semester began we knew only how to make a basic ALU or increment through the contents of memory, but we were able to combine all of our previous knowledge to plan an instruction set and architecture, design the logical circuits, and implement the computer using MML. This approach made us comfortable with alternative architectures and implementations.

Although using MML avoids many difficulties that students may encounter when building real machines, we think we learned more than we would building a real machine. There are several reasons for this. First, we were not limited by the physical environment. For example, building a real 8 bit computer requires a lot of wiring. If we connected the wires incorrectly, it could take (waste) a lot of time to correct. The main purpose of building a computer is not to learn to be careful, but to understand the computer architecture.

We learned a lot by designing our 16 bit computer. Although the basic principle of building an 8 bit or 16 bit computer is the same, we actually understand more when we actually need to implement more instructions on our machine. By limiting our computers to a logical design most of the teams finished their projects. They enjoyed their completed products and were happy about their accomplishment. We were very proud of our design and product after our computer worked. Although we encountered a lot of latency problems from the software, we overcame them by changing our design, understanding more about our mistakes, putting more effort into designing rather than connecting physical wires, and greater discussion about our machines with our team mates. We believe that if we physically built 16 bit computers we would not be able to finish our projects within a semester.

2.3. Hardware Project Approach

One team of four students used a hardware kit to learn about computer architecture. This approach used an 8085 microprocessor basic systems course from Elenco Electronics Inc. The kit consisted of fourteen lessons to teach computer theory, construction, and programming. The hardware approach was initially very appealing to computer science students. Hardware development was a more tangible approach than that used by the other student groups: virtualization of circuitry using a computer program. They began to construct the system, soldering components together step by step. A short testing procedure followed each step involving soldering. As the computer began to form via completed steps they were required to finish the project by the creation of a program. It was during this period that they realized they had a problem.

During the initial creation of the computer there was no idea that the ROM had not been preprogrammed. When it came to the task of programming the ROM manually, frustration set in. Manual in this case meant using eight binary switches and translating hex into its binary value and writing it to the ROM in a three step process: enter the high order address bits, enter the low order address bits, and then enter the data. This turned out to be a total disaster due to the sheer amount of time it took and the need to go back and test every single instruction for functionality. So even though they were initially very motivated, they lost interest in the details of binary addressing and the data entry needed to build the console ROM. The level of detail and large number of instructions even with an 8 bit computer obscured some of the understanding desired for the students. A picture of this unit is given in Figure 5.

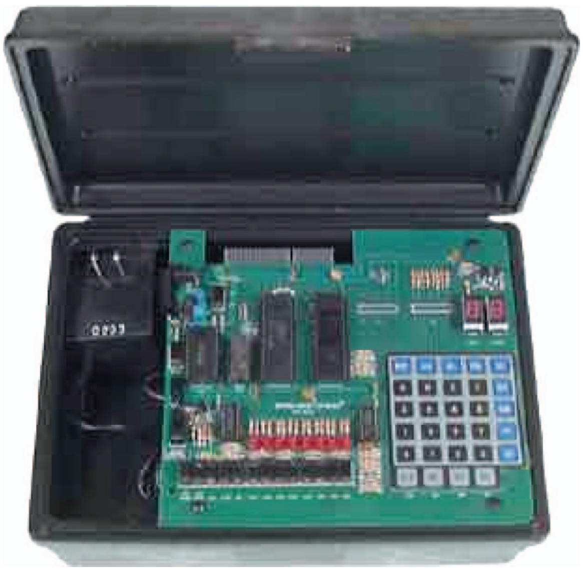

Figure 5. The 8085 hardware kit

3. Conclusions

These projects were done by student teams. The process of design, implementation, and debugging provided many learning opportunities, which proved to be highly motivational opportunities for the students involved. Students worked together closely in teams and produced excellent work. The final presentations showed a well-developed understanding of the principles of computer architecture. Patterson and Hennessy (1994) said "The processor comprises two components: data path and control. . . ." These students know what that means because they have built both.

Acknowledgements

Three students are co-authors of this paper. Other students involved in the two design projects featured included George Embrey, Scott Mikolyski, and Darren Salomons. These projects were team efforts and the other members not participating in authorship are gratefully acknowledged. The authors also thank Xuan Thanh Quach for redesign of the 8 bit Harvard architecture to use a multiplexer page. Other projects and teams in the class were the hardware team (Alex Daw, Wing Ho, and Masae Myoga), the Microcode Alarm Clock Team (Keith Radin and Deepak Shrestha), and the Matching Game Team (Willy Mariteragi and Samuel Raapoto). The authors gratefully acknowledge George Mills, the author of Multimedia Logic, for making his logic simulation package available without cost and including the source code on his web site (http://www.softronix.com/).

References

Burch, C. (2002). Logisim: A graphical system for logic circuit design and simulation. *ACM Journal of Educational Resources in Computing*, *2*(1), 5 – 16.

Clark, B., Czezowski, A., & Strazdins, P. (2001). Implementation aspects of a SPARC V9. *Proceedings of the 25th Australasian Computer Science Conference* (pp. 23 – 32). Darlinghurst, Australia: Australian Computer Society.

Holland, M., Harris, J., & Hauck, S. (2003). Harnessing FPGAs for computer architecture education. *Proceedings of the 2003 IEEE International Conference on Microelectronic Systems Education* (pp. 23 – 32). Washington, DC: IEEE Computer Society.

Moure, J., Rexachs, D., & Luque, E. (2002). The KScalar simulator. *ACM Journal of Educational Resources in Computing*, *2*(1), 73 – 116.

Null, L., & Lobur, J. (2003a). *The essentials of computer organization and architecture.* Sudbury, MA: Jones and Bartlett Computer Science.

Null, L., & Lobur, J. (2003b). MarieSim: The MARIE computer simulator. *ACM Journal of Educational Resources in Computing*, *3*(2), 1 – 29.

Paterson, D. A., & Hennessy, J. L. (1994). *Computer organization and design, the hardware/software interface.* San Francisco, CA: Morgan Kaufmann.

Stanley, T. (2005). An emulated computer with assembler for teaching undergraduate computer architecture. *Proceedings of the Workshop on Computer Architecture Education* (pp. 38 – 45). Washington, DC: IEEE Computer Society.

Tao, J., Schulz, M., & Karl, W. (2003). A simulation tool for evaluating shared memory systems. *Proceedings of the 36th Annual Simulation Symposium (ANNS'03)* (pp. 335 – 342). Washington, DC: IEEE Computer Society.

Wainer, G., Daicz, S., Simoni, L., & Wassermann, D. (2001). Using the Alfa-2 simulated processor for educational purposes. *ACM Journal of Educational Resources in Computing*, *1*(2), 111 – 151.