An Emulated Computer with Assembler for Teaching Undergraduate Computer Architecture

Timothy Daryl Stanley, PhD
Brigham Young University Hawaii, #1854
55-220 Kulanui Street
Laie, Hawaii 96762-1294
(808) 293-3388
stanleyt@byuh.edu

Mu Wang
Brigham Young University Hawaii, #1854
55-220 Kulanui Street
Laie, Hawaii 96762-1294
mw024@byuh.edu

Abstract
An eight-bit computer has been designed using an open source logic emulation package called "Multimedia Logic" from www.softronix.com. The intent of the project was to make clear to computer science students how the data path and control lines work to provide computer functionality.

This computer is an excellent teaching aid because:

1. All registers, ALU outputs, control lines, and memory outputs are instrumented.
2. Instructions can be executed with a single step switch or run with a clock.
3. The architecture is quite simple, with separate memory devices for data and instructions.
4. It is supported with an assembler patterned after the MIPS assembler used with the SPIM simulator.
5. An ASCII output display is available.

The instruction set designed for this computer includes: Add from memory, Add immediate, Load from memory to the input register, save from the output register to memory, jump to the address given by the immediate, jump to the address given by the immediate if the last add produced a zero result, and halt.

The design includes an instruction format of three bits of operation code followed by five bits of immediate.

Using this design as a launching point, students have been encouraged to design their own computers. Some excellent designs have been submitted. These include an elaborate multi-cycle 16-bit design, and many application specific designs.

This paper provides details of this computer design, assembler and example programs as well as descriptions of designs submitted by students.

Categories and Subject Descriptors
B.6 Hardware / Logic Design / Simulation.

C.1.1 Computer Systems Organization / Computer Architectures

General Terms
Design, Human Factors, Theory

Keywords
Logic Simulation, Computer Design, Binary Visualization, Multimedia Logic

1. Introduction

The concepts of computer architecture are some times very difficult for beginning computer science students to visualize because the action is all happening at the electron level in microscopic circuits. By building on knowledge from other courses students may be able to visualize what is happening in circuits, but many layers of abstraction are involved. For example, if one builds a computer with TTL circuits, there is a level of abstraction in the relation ship between circuit pin outs and logic elements. There is also a complex chain of detail between circuits that is visible only with logic probes or additional expensive instrumentations. Also when a student has spent the time to understand and master the breadboard circuit the semester is over, the circuit is disassembled and used for the next class.

The emulated logic approach the authors have developed overcomes these limitations in understanding the details of computer architecture. The circuits are designed by "wiring" up logic elements with all data inputs coming in on the left, control signals coming in from the bottom, and

outputs exiting from the right. The high level devices like memory circuits and ALU's look like the devices in schematic diagrams, making these devices easier to visualize. By designing simple circuits the operation of the individual components can be understood. At the completion of the class the students can take the design with them.

While the focus of this paper is an emulated computer for teaching architecture, a series of introductory circuits used to develop an understanding of the components that make up a digital computer are also provided. Many of the concepts of digital logic are difficult to grasp without practical experience. Some use prototyping boards with small scale digital circuits to design and build examples of digital devices [1]. Others use a hardware design language, like Verilog, to illustrate and teach digital logic concepts [5]. One school even uses students actors to emulate instruction flow in a computer [6].

The 8-bit computer will be thoroughly documented starting in section three.

2. Component Learning Projects and Outcomes

A number of projects built and demonstrated by students will be given in this section. We will start with simple projects and advance to more complex designs. Each design will be demonstrated with the presentation of this paper at the conference.

2.1 Calculator with Binary and Hexadecimal Outputs

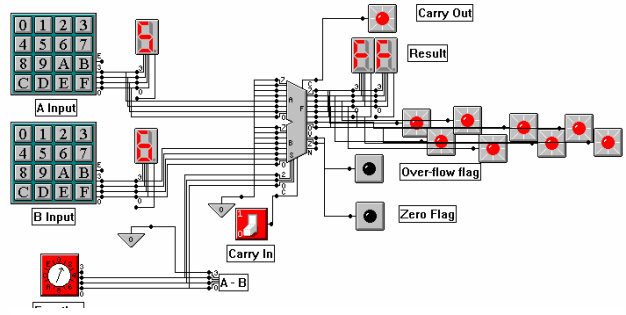

Figure 1. Calculator with Binary and Hexadecimal Outputs

The first project, illustrated in fig 1, is a calculator that takes two four-bit inputs, from hexadecimal keypads, and provides an output in both binary and hexadecimal, based on a function selected. The function is selected with the selector switch. The functions available in the ALU are: addition, subtraction, multiplication, division, equal, less than, shift right and shift left.

This is a nice project to start with as it builds on the ALU device example that is provided with Multimedia Logic.

The learning outcomes of this project are: familiarity with the ALU, comparing hexadecimal and binary, exploring properties of binary numbers under operations like the 5-6 operation shown in figure 1 to see the two's complement binary notation of a negative one.

2.2 Scanned Memory to Output Display

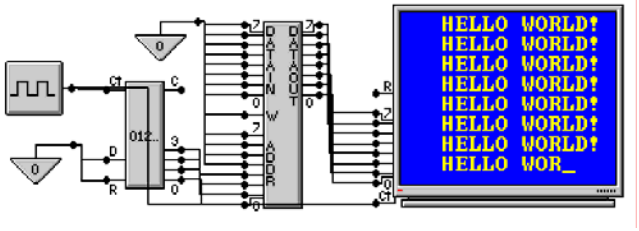

Figure 2. Scanned Memory to Output Display

This project, shown figure 2, connects the output of a memory device to an ANSII display device. Then by sequentially scanning the memory addresses with a counter connected to a clock, the content of the memory is sent to the display. In this case the content of the memory is " HELLO WORLD! ". For this project, only the first sixteen locations in memory are used, however, with an 8-bit counter, 256 locations could be used.

In Multimedia Logic the memory contents can be read from a "text" file or written to during the simulation. In this case the memory contents are loaded from a file and the memory is treated as a read-only memory (ROM).

Learning outcomes from this project are: an understanding of the relationship between memory address lines and data output lines, understanding counters, and clock oscillators, and synchronous data transfer from memory to display.

2.3 Programmable Calculator

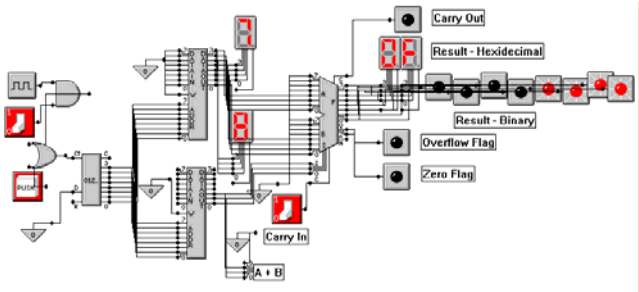

Figure 3. Programmable Calculator

This project, shown in figure 3, is a combination of the first two, using scanned memory to provide functions and data to an ALU. This project begins the comparison to a real

computer, with the upper memory serving as data memory, the lower memory which provides functions to the ALU as a program memory, and the counter as a program counter.

The learning outcomes of this design are observation of the different things that a series of binary lines can be, from instructions to data to addresses, to clock pulses. This is where we also learn about data paths and control paths.

2.4 Four-Bit Adder

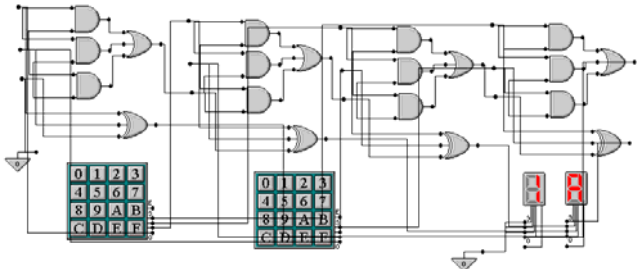

Figure 4. Four-bit Adder

These next two projects are designed to understand the inner workings of an ALU. The first, shown in figure 4, is a ripple carry binary adder. Two four-bit values are provided on the hexadecimal key pads and the results of the addition are displayed on the seven segment displays. By inverting the B inputs and making the C input for the first stage one the adder can be converted to a subtraction unit, illustrating the algorithm for converting a binary number to its two's complement negative.

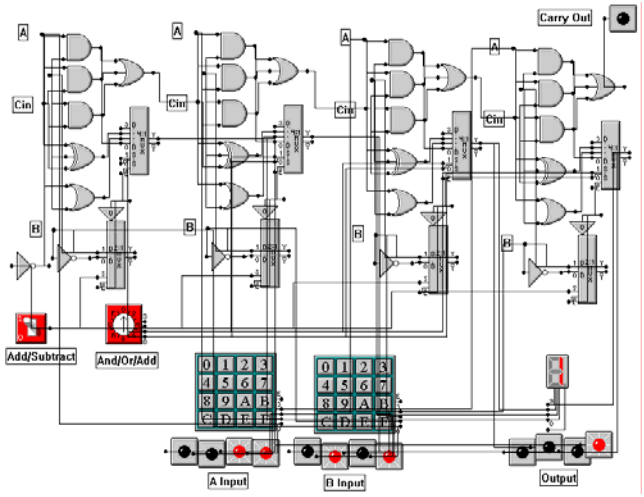

Figure 5. Four-bit ALU

The most important learning outcome of this design is an appreciation for how logic circuits can perform the kinds of operations we see computers perform.

2.5 Four-Bit ALU

This project, shown in figure 5, illustrates the complexity in the design of an ALU. This ALU, designed after the one-bit ALU from Patterson (Figure 6), can And, Or, Add, and Subtract. It is very useful for illustrating the bitwise operations of And and Or. For example the output illustrated above is the bit wise And of 3 and 5.

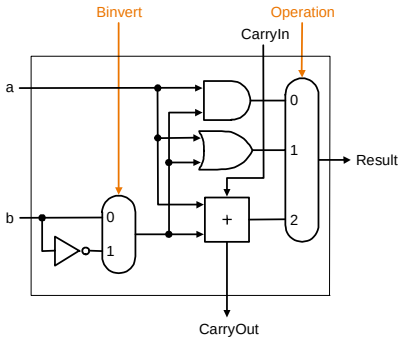

Figure 6. One-bit ALU from Patterson [4]

Learning outcomes of this project include an appreciation of how multiplexers make possible the control path in a computer—and again, an appreciation of how gates can be combined to produce computer functions.

3. An Emulated Computer for Teaching Computer Architecture

Providing a computer that is very well instrumented, visible on one page and easy to demonstrate, was the main goal of this design effort. In my computer architecture classes I ask my students to design an emulated computer. This design was one I produced to illustrate what I wanted from my students. I suggested they start with an instruction set and register design and build a computer from this foundation. For this eight-bit computer, an instruction format of three bits of operation code and five bits of immediate was chosen. This instruction format provides for eight instructions. These with mnemonics are:

1. adi - Add the immediate value to the input register and place in the output register,
2. adm - Add memory location addressed by the immediate to the input register and place in the output register,
3. lmi - Load the contents of the memory location addressed by the immediate value to the input register,
4. som - Save the output register to the memory location addressed by the immediate value,
5. ji - Jump to the location given in the immediate,

6. jzi - Jump to the location given by the immediate if the result of the last addition was zero,
7. om - Output the data from the memory location addressed by the immediate to the output display device,
8. hlt - Halt operation.

The physical architecture was to use two separate memories, to hold the data and program. This parallels the MIPS emulator PC SPIM which has a ".data" segment of memory holding constant data and a ".text" segment that contains the machine instructions. This construction simplifies the data path of the computer, but limits the capability to do recursion. The design includes an input register and an output register.

This design is a complete eight-bit, single cycle, stored program computer. The data paths are connected at the start of the clock cycle at then at the clock transition registers and memory are writing enabled. This enables demonstration of the inputs to commands being set up and then the operation being executed.

One non-physical device available in the logic emulator used is a binary controlled text display. This device can be seen just below the vertical column of control line indicators. This display shows one of sixteen lines of text, depending on the binary inputs to the device. In this case the device is used to show the operation being set up in the computer.

The memory devices can be used as read-only devices reading content from an underlying file, or they can be initialized with a file and altered dynamically during program execution. For registers memory devices with all address lines grounded are used.

One limitation of this emulation package is the absence of a 2-by-8 multiplexer. As a result the multiplexers are assembled by stacking a series of 1-by-2 multiplexers

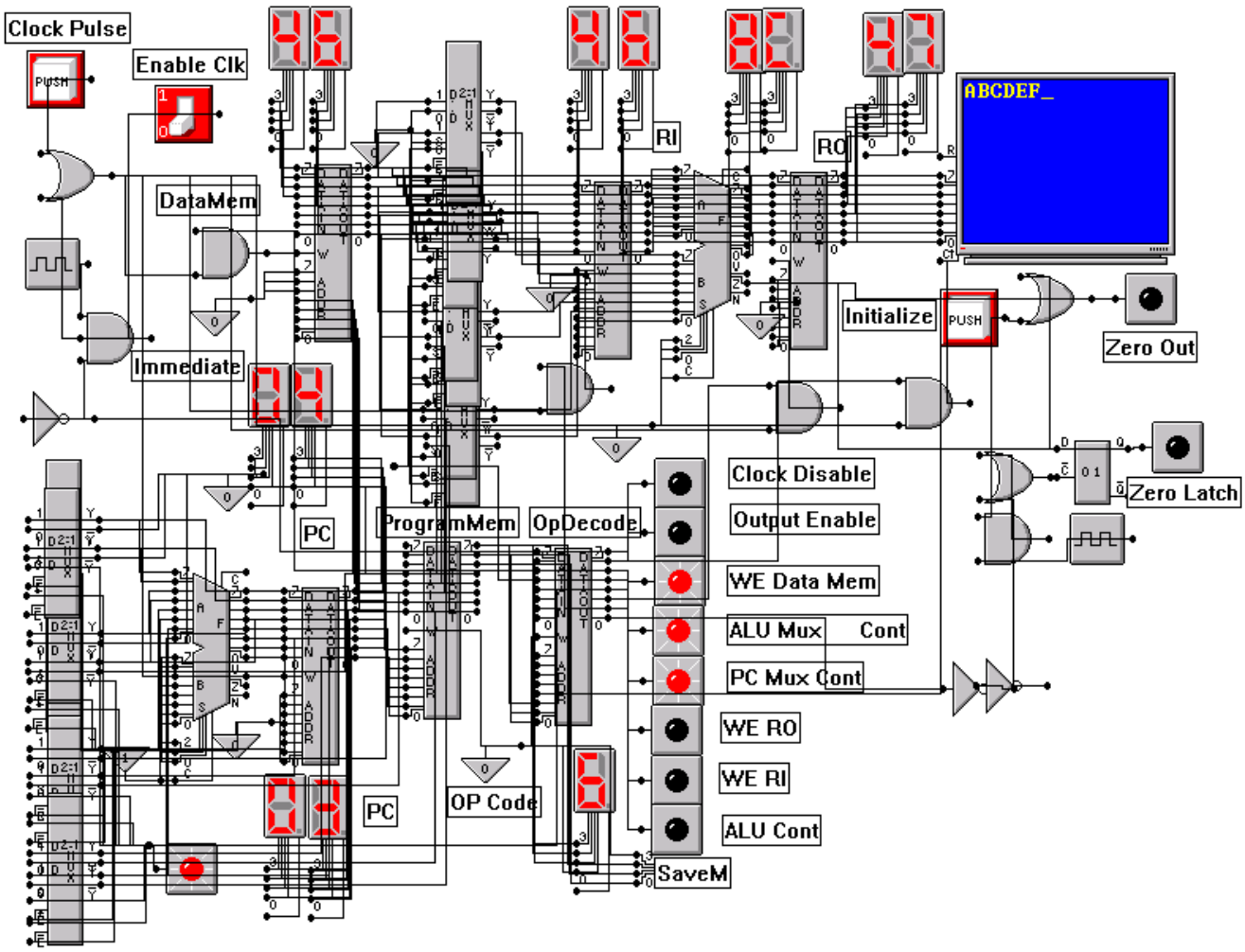

Figure 7. Eight-bit teaching computer design implemented in multimedia logic

partially overlapping one another. Since this emulator is published with its source code, I have built versions of this computer using a version of the software with a modified ALU that has an A out and a B out instruction. Then the multiplexer stacks can be replaced with ALUs. I have not included this design because it uses ALUs in a non-standard way and because the design could not be used with the emulator down loaded from the emulator's web site.

4. Sample Programs for the Computer

With this set of instructions a number of demonstration programs have been written. The file underlying the memory has a format that includes two hexadecimal digits that are the memory content for each line. The memory ignores any additional information on the line. So following the operation code or data a comment can be given. This allows instruction documentation information to be included with each line. These include a program to send a string in data memory to the output display device, a program with an up counting loop and a down counting loop to display the letters of the alphabet and halt at Z, and a program to display various size boxes on the display.

The design includes two ALUs, one incrementing the program counter and one performing the additions. Memory devices include a data memory, program memory, input register, output register, program counter, and an operation decode ROM. The nicest feature of implementing a computer design this way, rather that in a breadboard, is the much greater instrumentation of registers, and data lines. One can see each value as the computer steps through the program.

Three sample programs are included in this section.

4.1 Sample Program 1, ABCs.

This first program was designed to be simple but use all eight of the operations of this computer. It consists of a loop that counts up one memory location from ASCII A to ASCII Z, and counts down in another location to halt the computer after 26 letters. To implement this program the memory contents in the following tables are place into the data and program memories. Note that in these tables that the two hex digits in each line are the actual output from the memory device and the rest of the line is a comment. Data and program memory files are shown if tables 1 and 2 below. The output is shown with figure 7 above.

00	zero (not used)
19	Hex for character count in alphabet
ff	Twos complement negative one
fe	Twos complement negative two (not used)
41	ASCII code for letter A
41	(not used)
41	"
00	"
00	"
00	"

Table 1. Data memory content for program 1

c4	Output from memory location 04
44	Load input register from memory location 4
01	Add I (01) to input register
64	Save output register in memory location 04
c4	Output from memory location 04
41	load input register from memory location 01
22	Add from memory location 02
61	Save output register to memory location 02
aa	Jump if last calculation result was zero to 0a
80	Jump to memory location 00(+1)
e0	Halt execution

Table 2. Program memory content for program 1

4.2 Sample Program 2, Hello World.

The second program was to be the simplest possible, like the "Hello World" used to introduce all programming languages. For this program a string in the data memory is sent character by character to the output screen and then the program loops back to the beginning. The lack of instructions to update program memory based on calculations prevents the use of simple iteration to implement this program. The data and program memory files are show in tables 3 and 4 and the output is shown in figure 8.

```
20  Space
20  Space
48 H
45 E
4c L
4c L
4f  O
20  Space
57 W
4f O
52 R
4c L
44 D
21 !
0d  New Line
```

Table 3. Data memory content for program 2

```
c0    Output from memory location 00
c1    Output from memory location 01
c2    Output from memory location 02
c3    Output from memory location 03
c4    Output from memory location 04
c5    Output from memory location 05
c6    Output from memory location 06
c7    Output from memory location 07
c8    Output from memory location 08
c9    Output from memory location 09
ca    Output from memory location 0A
cb    Output from memory location 0B
cc    Output from memory location 0C
cd    Output from memory location 0D
ce    Output from memory location 0e
80    Jump to Zero (+1)
```

Table 4. Program memory content for program 2

4.3 Sample Program 3, Triangle.

This program was written to test the assembler discussed in the next section. It uses two nested loops to print a triangle on the output screen. Data and program memory files are given below and the output is shown in figure 8.

```
06    column (size of triangle)
03     row  (not used in program)
00    column step
00     row step
ff     negative one (allows decrementing  )
00     zero
2a     symbol "*"
0d     new line
```

Table 5. Data memory content for program 3

```
45          Load input register from memory location 5 (zero)
20          Add memory location 0 (column)  to input register
62          Save result in memory location 2 (column step)
63          Save result in memory location 3 (row step)
c6  lp1:  Output from memory location 6 (symbol "*")
44            Load input register from memory location 4 (neg
one)
22          Add from memory location 2 (column step)
62          Save result in memory location 2 (colmn step)
a9          Jump on zero to lp2:
83          Jump to lp1:
23   lp2:  Add from memory location 3 (row step)
63          Save result in memory location 3 (row step)
b6          Jump on zero to :hlt
c7          Output from memory location 7 (new line)
45          Load input register from memory location 5 (zero)
20          Add memory location 0 (column)  to input register
44            Load input register from memory location 4 (neg
one)
20          Add memory location 0 (column)  to input register
60          Save result in memory location 0 (column)
45            Load input register from memory location 4 (neg
one)
20          Add memory location 0 (column)  to input register
62          Save result in memory location 2 (column step)
83          Jump to lp1:
eo  hlt:    Halt
```

Table 6. Program memory content for program 3

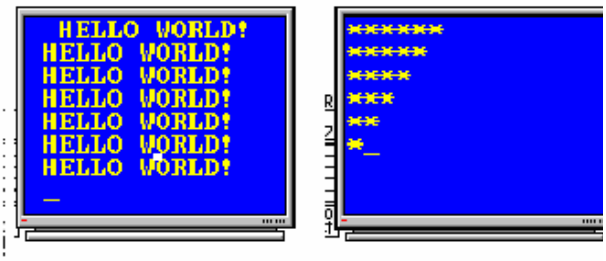

Figure 8. Output screens for programs 2 and 3

5. The Assembler in PERL

To add to the utility of this computer, an assembler was designed in the PERL language. As the assembler runs it generates text files that can be loaded into the data and program memory in the simulated computer. The assembler allows symbolic linking between the data and the program and allows symbolic naming of jump locations. The assembler was patterned after the assembler imbedded in the MIPS emulator PC SPIM.

This assembler starts execution by asking the user for data and program memory file names. Then the user sees the screen from the table below which gives a review of the instruction set of this computer and then provides a sample input file to show the syntax that must be used. When the line with the stop command is given, the program closes the files and returns.

6. Student Computer Designs

Using this computer and its design process as an example, computer architecture students have been required to design a computer of their own from the registers and instruction set to layout and implementation with example programs. The first design from a student team was an elaborate 16-bit design that used eight cycles to decode and execute each instruction with the idea of demonstrating a pipeline implementation. This computer consisted of eight pages of logic. While this computer represents a great deal of effort on the part of the students involved, it is not as useful for demonstration because parts of the display are on separate pages and can not be viewed simultaneously.

Some students had difficulty designing a computer starting with operations and layout. For these students the approach that seemed to work best was to start with an application they would like to demonstrate on their computer and then design a computer to meet that requirement. Some examples of the application-motivated designs were for an electronic door lock and a "Whack a Mole" game.

```
**********       Operation Code      **********
*********************************************
*********   adi- Add Immediate        *********
*********   adm- Add Memory           *********
*********   lmi- Load Mem -> Ri       *********
*********   som- Save Ro-> Mem        *********
*********   ji- Jump Immediate        *********
*********   jzi- J on z Im            *********
*********   om- Out Mem Im            *********
*********   Hlt- Halt                 *********
*********************************************

************ Sample  Input **************
*********************************************
.data
(PLease Input Data for DataMem.)
numlet:26d
negone:ffh
acode:41h
.text
(PLease Input Data for ProgramMem.)
omi acode
start:lmi acode
adi 01d
som acode
om acode
lmi negone
adm numlet
som numlet
jzi stop
ji start
stop:hlt
*********************************************
```

Table 7. Assembler Output

7. Comments from Students

In this section, student's comments are provided to show the value of this approach to teaching the inner-workings of a computer. One student, Daniel McCallum, wrote in an email [2] after completing Computer Organization:

"Multimedia Logic has helped me a lot to comprehend many of the complex ideas behind the workings of a computer. It helps me see things visually and can look at things one step at a time. For example how an ALU works made a lot more sense when I could put it together and take it apart myself, using Multimedia Logic. Another big aspect of Multimedia Logic was that I can see all the different switches, gates, etc. visually and have come to understand how basically a computer does what it does."

Several students commented that they now understood how circuits make computers and how computer functions can be made from simple switching logic devices. Students that previously used breadboard devices commented that understanding what was going on was much easier in the emulated environment because each register can be instrumented individually.

8. Limitations of Multimedia Logic

One difficulty encountered with Multimedia Logic is the unexplained dropping of wires from saved files. This occurs the first time a new file is saved and seems to be a problem with overlapping components. For example a horizontal row of eight light-emitting diodes will lose connection to every other light when saved, if they are placed adjacent to each other and are vertically lined up. The "work-around" for this problem is to stagger the lights slightly in the vertical direction. This vertical staggering can be seen in figures 1, 3 and 5.

9. Summary

A number of designs built in Multimedia Logic have shown to be useful to students in gaining an understanding the inner workings of a computer and related technology. Students in computer architecture classes have successfully used this tool to design many eight-bit and even two sixteen-bit computers, most with single cycle designs, but two with multi-cycle designs. Through this experience the details of how switches can make computers becomes very clear.

10. Acknowledgments

Thanks to George Mills of www.softronix.com, who has graciously made his product, Multimedia Logic, available for free download and included the source code.

And a special thanks to the faculty and students of our computer science department who have encouraged me in this effort by their enthusiastic support.

11. References

[1] Hoffman, Mark E., *The Case for More Digital Logic in Computer Architecture*, Conferences in Research and Practice in Information Technology, Vol. 30.

[2] McCallum, Daniel, Email of 6/25/2004.

[3] Mills, George, www.softronix.com, Multimedia Logic download kit and source kit.

[4] Patterson, David A., and Hennessy, John L. *Computer Organization and Design, the Hardware/Software Interface, 2nd Edition*, Morgan Kaufmann Publishers .

[5] Patterson, David A., and Hennessy, John L. *Computer Organization and Design, the Hardware/Software Interface, 3nd Edition*, Morgan Kaufmann Publishers .

[6] Powers, Kris D., "Teaching Computer Architecture in Introductory Computing: Why? And How?" Sixth Australasian Computing Education Conference (ACE2004), Dunedin.

[7] Wolffe, Greg, Yurcik, William, Osborne, Hugh, and Holliday, Mark, "Teaching Computer Organization/Architecture With Limited Resources Using Simulators", SIGCSE 2002, ACM Press, Northern Kentucky USA, Feb/March 2002.