TEACHING COMPUTER ARCHITECTURE THROUGH
SIMULATION

(A BRIEF EVALUATION OF CPU SIMULATORS)*

Timothy Stanley, PhD Computer and Network Sciences, Utah Valley University, Orem, Utah 84058, 801 863-8978, TStanley@uvu.edu

Vasu Chetty, Matthew Styles, Shin-Young Jung, Fabricio Duarte, Tin-Wai Joseph Lee, Michael Gunter, Computer and Information Sciences, Brigham Young University - Hawaii

Leslie Fife, PhD Dept. of Computer Science, LSUS, Shreveport, LA 71115

ABSTRACT

In our computer architecture class, we researched logic simulators for design of educational computers. We have regularly had students design, create and operate a simulated computer as part of this course. This has allowed students to understand the internal details of instruction decoding and data path control. In the past we have used Multimedia Logic (MML). MML is open source and free and it has an attractive user interface. However, a number of improvements have been made to Logisim and we felt a re-evaluation was appropriate. The machine we chose to build was Linda Null's sixteen bit MARIE computer. We implemented this computer in Multimedia Logic, Logisim, Cedar Logic and CPU Sim. The implementation in CPU Sim allowed us to add and test additional instructions to the MARIE instruction set and to use CPU Sim as an assembler for our other designs. This paper compares student designs and discusses the pedagogical value and ease of implementation of these designs.

INTRODUCTION

* Copyright © 2012 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.
To fully appreciate how microcomputers work, one needs to understand how an instruction decoder modifies the data path in the processor to accomplish the instructions in its architecture. Our experience is that this understanding and appreciation comes best through designing and implementing the data path and control circuitry of a custom processor design. This could be done using logic circuits in a lab, but we believe this approach is both unnecessarily expensive and needlessly complicated within the computer science curriculum.

There are several good open source logic simulation tools available, and you can use one of these to build the data path and control circuitry [1]. This avoids an expensive lab, broken parts, understanding how logic devices relate to the circuit inputs and outputs, and additional complications.

Our most recent computer architecture class was small, six students, but they were all excellent students. As is customary, we had a processor design project. We chose to design, layout and build the MARIE computer from the book Essentials of Computer Organization and Architecture by Null and Lobur [2]. MARIE is an acronym meaning "A Machine Architecture that is Really Intuitive and Easy. One reason for this choice was that all students were familiar with MARIE from their computer organization class.

One problem in using MARIE is the absence of indirect load and indirect save instructions. The indirect load can be provided by using two instructions: an accumulator clear followed by an indirect add. The only way we found to indirectly save was to perform a math operation on the save instruction, effectively modifying the code. Self-modifying code, while elegant is not good programming practice, so we chose to add an indirect load and an indirect save to the thirteen instructions defined by Null and Lobur.

The MARIE architecture is a Von-Neumann architecture consisting of a program counter, a memory address register, a memory data register, an instruction register, an accumulator, an input register, and an output register. Each register is sixteen bits except for the address register which is twelve bits, and the input/output registers which are eight bits. The thirteen defined instructions include add, add indirect, load, store, subtract, input, output, skip conditional, jump, store and jump, jump indirect, clear accumulator, and halt. Since four bits are allocated for operation codes a total of sixteen instructions could be defined.

During this course we also wanted to compare several simulation tools. The class was divided into four teams. One team of two designed and built MARIE in Cedar Logic and another team of two designed and built MARIE using Logisim. One student implemented MARIE in CPU Sim. The last student took an implementation of MARIE from a previous class designed in MML [6], and fixed the bugs in that implementation, made improvements to the I/O circuitry, and implemented the two additional instructions described. This earlier implementation of MARIE was described in [7]. Only portions of each design are shown.

IMPLEMENTATION IN CPU Sim

CPU Sim is a Java application written by Dale Skrien, which allows users to design a simple computer CPU at the microcode level and to run machine or assemble language
programs on those CPU’s through simulation [5]. The microcode level is where register transfers are defined or incrementing of the program counter occurs. These register transfer instructions are used to make up the machine instructions that comprise the instruction set architecture ISA of the CPU. While you need a complete understanding of the ISA to build the CPU in CPU Sim, you do not need to construct the data path or control circuitry to implement in CPU Sim. The CPU Sim implementation does not meet all of the goals for the design project, but it provided an assembler to use with the other implementations.

The first step in implementing MARIE in CPU sim is to create and name the hardware modules such as the registers and main memory. The next step is to create the register transfer instructions (called micro code in CPU Sim). The next step is to define the machines instruction fetch sequence. Finally the instructions are formed from the fetch sequence and the micro instructions, completing the definition of the machine. After the machine is defined it can be used to assemble instructions and test execution. Figure 1 shows the MARIE computer in CPUSim.

Figure 1 - MARIE in CPU Sim

IMPLEMENTATION IN LOGISIM

Recent Logisim implementations now include a clock, keyboard, and terminal. Logisim provides a great deal of flexibility in device configuration, wire bundles, and sub circuits [4]. Logisim also has the capability to generate logic circuits from truth tables. This capability was used to generate the instruction decoding circuitry. Logisim has the ability to define wires as multiple bit busses. Logisim has the ability to define and use custom circuits. Also, during simulation, Logisim updates memory and registers in real-time while the simulation is running. These features make for a single page machine that demonstrates very well. The only limitation found was the lack of "wireless connectors" available in the other simulators.
MARIE in Logisim is shown in the Figure 2. The main features shown in this figure include the Program Counter and Memory Address registers followed by the memory at the top of the figure. Below the registers are the ASCII display terminal and the keyboard interface. Below these from left to right are the Memory Buffer Register, the ALU, the Accumulator, and the Instruction Register. To the right of the Instruction Register is a custom circuit that does the comparisons that enable the skip on positive, negative or zero for the value of the accumulator. The balance of the circuitry is the clock and instruction decoder.
As this computer runs, the single bit control wires change color to indicate the logic level they are carrying. The multibit wire do not change color. The registers update their content and memory displays the address and data being accessed. By clicking on the memory, an ascii file can be loaded, giving the memory content. Figure 3 shows a portion of the custom logic circuit produced by the truth table for decoding instructions. Figure 4 shows the comparitor logic circuit. Figure 5 shows the ALU. The ALU defines additional functions beside the add and subtract of MARIE.

IMPLEMENTATION in MULTIMEDIA LOGIC

The original implementation of MARIE in MML was accomplished by Stanley, Prigmore, Mikolyski, Embrey and Fife and was presented in [7]. However, their design was incomplete. They only implemented skip on zero and non zero. So the first goal was to add skip on negative, zero, or positive. This was done by sensing the most significant bit of the accumulator to test for a negative number. The design from 2006 also only included ASCII I/O. To this were added decimal and hexadecimal inputs and outputs. Also added to the Input was an option to specify the input source through a parameter on the input command. Finally, the two new instructions were added. Implementing these required adding to the decoding matrix which was implemented in a read only memory. Figure 6 shows the Instruction Decoder and Control Circuitry for MARIE and Figure 7 shows the I/O screen in MML.

IMPLEMENTATION IN CEDAR LOGIC

Cedar Logic has some nice features, particularly for timing investigation and working with a Z80 CPU. But we were frustrated by the absence of any ALU components bigger than a four bit adder or comparator. Also the absence of an ASCII output limited the class of programs that could be directly demonstrated. To overcome this limitation an external program was written to convert hexadecimal into ASCII or Decimal. This limits the ability to demonstrate this design since the output is not real time.
Figure 8 shows the data path in Cedar Logic. This data path is relatively simple to design and layout. Figure 9 shows the decoder and clock circuitry. The complicated part of this circuit is obtaining the contents for the read only memory (ROM). The contents were generated in an excel spread sheet and then exported to the ROM. Figure 10 shows the skip conditional circuitry. Figure 11 is the ALU design.

Figure 8 - Data path for MARIE in Cedar Logic  
Figure 9 - Instruction decoder and clock for MARIE in Cedar Logic

Figure 10 - Skip conditional circuitry for MARIE in Cedar Logic  
Figure 11 - ALU for MARIE in Cedar Logic

COMPARISONS AND CONCLUSIONS

This was a nice exercise in comparison of various logic simulation tools in the context of a computer architecture class. The least frustration was in the design and implementation of MARIE in Logisim. Logisim fits nicely on one page and the dynamic display of registers and memory allows a very good demonstration of the operation of the CPU. Using only Logisim in this course would be a viable option.

The update of the previously produced MARIE in MML was quite straight forward, but the many pages in the MML layout limits its ability to provide meaningful
demonstrations. Careful layout could likely reduce the number of pages for this design, but not to the level of Cedar Logic or Logisim. The absence of the multi-bit wires in all but Logisim gives a significant advantage to layout in Logisim.

Cedar Logic gave a compact one page data path, and provides a real-time display of the memory contents. Cedar Logic also provides a nice timing facility which is useful in debugging timing issues. But the absence of sixteen bit ALU components and the absence of ASCII I/O capabilities are frustrating.

CPU Sim provides a nice facility to build an emulated machine for a custom instruction set and allows experimentation with microcode, but it does not provide experience with a data path and control circuitry.

These results and conclusions are summarized in the table below.

Table 1. Comparison of Architecture Simulation Tools

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Pages</th>
<th>Layout time</th>
<th>Problems Encountered</th>
<th>Pedagogical value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multimedia Logic</td>
<td>13</td>
<td>~100 hrs</td>
<td>Problems saving circuits</td>
<td>Limited by the large number of pages</td>
</tr>
<tr>
<td>Cedar Logic</td>
<td>4</td>
<td>~40 hrs</td>
<td>No ASCII I/O</td>
<td>Nice timing facility</td>
</tr>
<tr>
<td>Logisim</td>
<td>1*</td>
<td>~40 hrs</td>
<td>No wireless connectors</td>
<td>Displays very well, all registers &amp; memory are dynamic</td>
</tr>
<tr>
<td>CPU Sim</td>
<td>1**</td>
<td>~20 hrs</td>
<td>None</td>
<td>Does not actually have a data path and control circuits</td>
</tr>
</tbody>
</table>

Notes:
* The complete circuit is on one page, but includes four custom sub-circuits
** CPU Sim has a single run page, but many other pages are used to define the CPU

We found each of these tools to be valuable for learning, but the synergy of having teams working on each and comparing and discussing was even more valuable. However if we were to have a single tool for a Computer Architecture class, the clear choice is Logisim.

REFERENCES


