ALPHA-0: A SIMULATED COMPUTER AS TOOL FOR COMPUTER ORGANIZATION COURSES

Gabriel A. Wainer.
gabrielw@dc.uba.ar

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Pabellón I - Planta Baja - Ciudad Universitaria
(1428) Buenos Aires. Argentina.

ABSTRACT

The study of computer organization in Computer Sciences courses leaves the students with an uncomplete knowledge of the different subsystems studied. Alpha-0 is a set of simulation tools built to study the different levels of a computer system. The tools showed to help in the fully comprehension of these complex systems. It also allowed to make empirical comparison and performance studies in the first years of undergraduate studies.

Keywords: Computer organization, computer system levels, architectures, performance evaluation.

1. INTRODUCTION.

The theoretical study of computer architecture and organization usually give the students an incomplete and sometime erroneous view of how a computer system works. Computer organization bibliography [1, 2, 3, 4] usually emphasizes the basic behavior of the logical subsystems of a computer. The lack of practical experience can make that the underlying complexity of the subsystems, and their interaction could not be completely understood. The main problems are related with the existence of several levels used when studying computer organization. The levels usually analyzed include assembly language, instruction set, microprogramming and digital logic. The introduction of higher levels (programming languages, Operating Systems) makes the task even more complex.

The inter-level interaction can make the the system execution as a whole to be confused. Also, the detailed behavior for each of the subsystems can be complex to analyze. The existence of hardware or software tools with educational purposes in this area is reduced, making difficult to make the concepts clear through practice.

This work was partially supported by the project SECYT EX-003 of the Universidad de Buenos Aires.

In this work we propose to analyze these complex systems by using simulation to build a simple computer. Alpha-0 is a simulated computer built with academic purposes. It allows to understand the behavior of a computer system from the architectural point of view, also permitting to make performance analysis of the subsystems. Each of the system's levels was simulated individually, so a complete panorama of them can be obtained. It also allows to understand the complete behavior of the system and the interaction between levels. Different tools were built for each of the levels, and they will be analyzed in the following sections.

2. A DIGITAL LOGIC LIBRARY.

To understand the behavior of the basic circuits of a computer [1], a Digital Logic Library was built. The library was implemented as a C++ class library, allowing hierarchical construction of complex circuits [5]. The idea was to build complex circuits based on primitive components: the logical gates AND, NOT, OR, NOR and XOR. (the last two were derived from the first ones). Lower physical levels (transistor level and their connection) were not considered. Using the basic logical gates, higher level circuits were built. The following were included:

. **Comparator:** it simulates a circuit comparing two inputs (Figure 1a), and determining which one is different from the other (including also inequality comparators).

. **Multiplexor**: input lines are detected, choosing one of them to see if its status. This value is output, ignoring the values of the rest (Figure 1b).

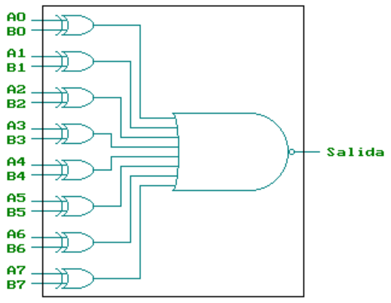
. **Decoder**: it activates the output line (2^n outputs) corresponding to the number composed by the input lines (n inputs - Figure 1c).

. **D-latch**: these circuits simulate the storage of information into the processor. The d-latch stores one bit, and it is driven by the pulse of a clock (Figure 1d).

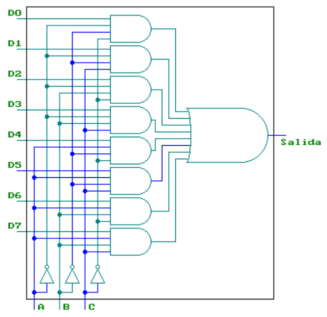
. **Adder**: it adds two bits by considering a third input representing the carry bit. The result of the addition and the propagation of the carry are returned (Figure 1e).

. **Shifter**: it shifts a set of bits one bit to the left or the right, filling the empty places with zeros (Figure 1f).

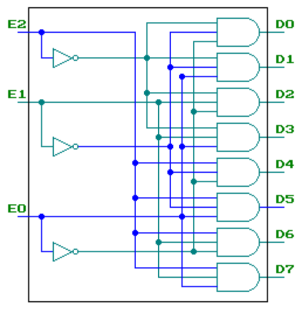
. **Register**: it is built by connecting several d-latches (Figure 1g).



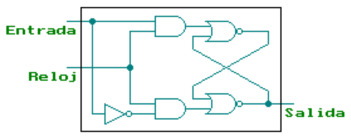
(a)



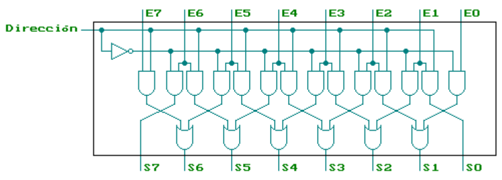
(b)



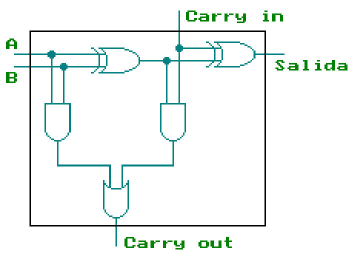
(c)



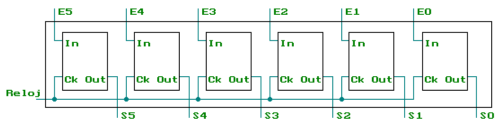
(d)



(e)



(f)



(g)

Figure 1. Library Circuits (a)Comparator(b)Multiplexor (c)Decoder (dD-Latch (e)Shifter (f)Adder (g) Register.

. **One-bit ALU**: the unit has only four one-bit operations: ADD, AND, OR and NOT. The class was built using the decoder and adder units, showing of the use of simple circuits used to build complex ones (Figure 2).

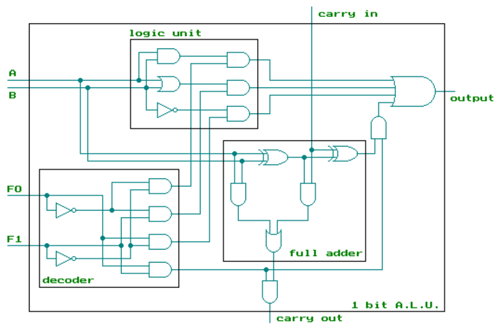


Figure 2. One-bit ALU.

. **N-bits ALU**: it was built by connecting several one-bit ALUs. A row of one-bit ALUs should be connected, linking the carry-out of each of them with the carry-in of the next.

. **Memory**: a simple 4x3 bits' memory was implemented. It is too specific and has little flexibility, but the purpose is to show how a complex circuit can be built from d-latches or registers and it can help to analyze the behavior of a simple memory. It was not included in the library, therefore, it cannot integrate other models. It use input address, and q Read/Write line to indicate the desired operation. An output line is used to show the present value in the actual address.

The size of the circuits is dynamic, and the library also has been provided with a graphical interface allowing to see the circuit basic schemes (Figures 1 and 2 were generated using the library). The interface also shows the changes in the input lines' values, allowing to study the detailed behavior for each circuit.

Performance checking routines were implemented. Their goal was to compare the speed of the simulated circuits with that of the real circuits. Each circuit was measured using 10.000 operations, using 16 bits circuits and variables (real and simulated). The were the following:

. Left shifts using a simulated and a real shifter:

SHIFTER tests (seconds)			
	Total	Avg. per op.	Std. dev.
Real	0,002	2,083x10^{-7}	3.62x10^{-7}
Simulated	0,237	2,374x10^{-5}	3.97x10^{-7}
Simulated (GUI)	0,289	2,899x10^{-5}	4.12x10^{-7}

. Assignment to a 16 bit register.

REGISTER tests (seconds)			
	Total	Avg. per op.	Std. dev.
Real	0.002	1.579x10^{-7}	6.12x10^{-7}
Simulated	0.503	5.033x10^{-5}	1.89x10^{-7}
Simulated (GUI)	0.525	5.254x10^{-5}	3.90x10^{-7}

. Two values' addition using the real and simulated ALUs.

ADD tests (seconds)			
	Total time	Avg. per op.	Std. dev.
Real	0.00197	1.971 x 10^{-7}	3.554x10^{-7}
Simulated	8.66677	0.0008666	2.058x10^{-6}

. Bitwise OR for 16 bits operands (ALU).

OR tests (seconds)			
	Total time	Avg. per op.	Std. dev.
Real	0.001785	1.786 x 10^{-7}	7.029x10^{-7}
Simulated	8.80792	0.00088079	2.089x10^{-6}

3. MICROARCHITECTURE LEVEL

As a second step, the circuits in the digital library were used to simulate the execution of a microprogrammed processor [6]. The microarchitecture components are supposed to be connected by a single local bus. The Control Unit executes a microprogram for each instruction, using a predefined language that allows to define the input and output flow between the processor's components, and its communication through the local bus.

The main goal of this level is to define the behavior for each of the instructions defined for the processor. Though instruction set was based in that of the SPARC processor (a RISC processor whose Control Unit should be hardwired), the processor is microprogrammed with instructional purposes.

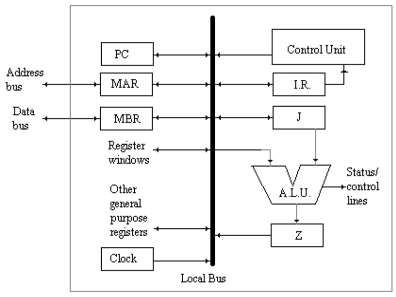


Figure 3. Structure of the simulated microarchitecture.

It is supposed that the memory, processor and input/output subsystems are connected by synchronic buses. The delays for each microoperation were also specified, so the total execution time for each instruction can be computed. Each microinstruction can be traced, showing the status of the local bus and registers and the data path.

A cache memory with 64 bytes' cache was also simulated [7]. It has 32 words divided in 8 blocks of 8 bytes each. Several algorithms were tested, including Direct, Associative (FIFO, LFU, Random, LRU), and Set Associative Mappings. Various tests were executed, comparing the execution time of the microcode operations using the original simulator and the ones with cache memory.

	Test 1	Test 2	Test 3	Test 4
Non-cached	3602	2157	2466	1691
DM	3285	1975	2272	1597
AM (FIFO)	3278	1969	2274	1585
AM (LFU)	3331	2008	2318	1619
AM (Rand.)	3285	1978	2286	1585
AM (LRU)	3279	1969	2274	1582
SAM (Rand.)	3200	1949	2235	1567
SAM (FIFO)	3179	1955	2229	1545
SAM (LFU)	3183	1958	2246	1562
SAM (LRU)	3182	1955	2240	1545

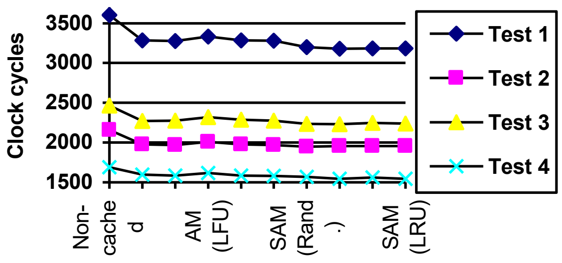


Figure 4. Test results of caching with different policies.

4. INSTRUCTION SET LEVEL ARCHITECTURE

The next level to be defined was the instruction set. As stated earlier, a RISC computer was simulated, allowing to use a RISC platform in low cost processors. The SPARC architecture was chosen as reference to build the simulator. It executes a subset of the SPARC instructions and allows to follow the execution flow of a program. The changes of values in registers and memory can be studied step by step. The complexity of the simulator was reduced by restraining the complexity of the SPARC architecture. The simulated architecture has the following features [8]:

. It executes a subset of the SPARC instructions.
. It addresses one word at a time.
. It does not access to memory using delays.
. Delayed jumps are not implemented.
. The jumps use absolute addresses.
. The instruction set interpreter simulates the existence of a data and instruction memories.

The simulator uses this basic organization, also based on the SPARC architecture. There are 520 registers of 32 bits each, organized in overlapping windows. In any instant there are only available 32 of the registers. The available registers are shown in the Table 1, and are organized in four groups.

Name	Group
%G0 .. %G7	Global registers.
%I0 .. %I7	Input parameters registers
%L0 .. %L7	Local registers
%O0 .. %O7	Output parameters registers

Table 1. General use registers.

The Table 2 includes the special purpose registers chosen from the SPARC architecture:

Name	Use
%G0	Always zero. No value can be assigned.
%I7	Return address to be executed when the present procedure is finished.
%O7	Used to pass the return address.

Table 2. Special purpose registers.

As stated earlier, the instruction set was reduced. The chosen subset includes the following instructions (further information can be found on [8]): LD, ST, ADD, ADD, SUB, SUBCC, BEQ, BNE, BLE, BLT, BGE, BGT, B, CALL, RET.

There are two registers not available to the programmer. One is the Program Counter (PC), containing the address of the next instruction to be executed. The other is the Program Status Word (PSW). The execution flow can be monitored by using the user interface defined in the Appendix.

5. ASSEMBLY LANGUAGE LEVEL

An assembly language was also provided for the Alpha-0 processor. The goal is to build user programs to test each of the components. The assembly language was built using a two-pass translator. Each program is composed of Assembly language directives, data definition directives, instructions, labels, constants, and comments. The directives indicate to the translator how to manage the following program lines. There are three directives: #DATA, #PROGRAM, #END. The #DATA pseudo-instruction is used to tell that the following lines are used to define data in memory. Three kinds of data can be defined: 32 bit words, word arrays, and character arrays.

Instruction lines cannot be included before the #DATA directive, that can be used to define the memory size. #PROGRAM tells that the following lines include valid instructions of a program. The program execution will start by the first instruction found after this directive. The sentence also admits an optional parameter indicating the size for the instruction memory, expressed as an instruction size. The default value is of 512 instructions, and the sizes can be adjusted to execute larger or smaller programs. Finally, #END indicates the program area's end.

The translation is divided in two phases. First, it parses each line and if a label is found, it stores it into a symbol table together with its location. If the line contains an instruction whose operand is a previously defined label, it verifies if is stored in the symbol table. If not, it stores it and it also records this value into an auxiliary table (to verify if it is external). The second pass reads each program line and it stores the value for each label (detecting external references, that remains unsolved and added into a symbol table). Finally, it codes the instruction. The symbol table is used to make a linking pass, parsing the symbol table and relating the parts. The table contains the label for the symbol, and its relative address. All the information recorded into the symbol table is loaded into memory. Using this information an external references table is created. The linker routine loads all the executable code into an array, taking one object file at a time. The executable code is copied, and the unsolved external references are replaced (using the symbol table).

6. CONCLUSION

In this work a complete set of tools used to simulate a simple computer was presented. The tools can be used in computer organization courses to analyze and understand the basic behavior of the different levels of a computer system. The interaction between levels can be studied, and experimental evaluation of the system can be done.

A basic instruction set and assembly language were built. Both of them are based on the SPARC architecture, allowing to study the main features of this processor, analyzing features not existing in simpler processors. Several basic circuits were also implemented, allowing to build the computer by using them. Finally, both levels were connected by using microprogramming.

The use of this set of tools allowed the students to obtain a complete understanding of computer organization. A tow-down or a bottom-up approach can be used to build a part or a complete computer system. Experiences with the top-down analysis allowed to obtain significant results. For instance, some students groups discovered microprogramming concepts before its conceptual definition in the course.

At present, the set of tools is being completed by including a complete input/output subsystem set. The main input/output devices, the input/output interfaces, DMA controllers and channels will be simulated. Different transference techniques (polling, interrupts, DMA) will be considered to implement the input/output electronics. The tools are public domain, and can be obtained at "http://www.dc.uba.ar/people/materias/oc1".

7. REFERENCES

[1] TANENBAUM, A. S., *Structured Computer Organization*, 4th. edition, Prentice Hall, New Jersey, 1996.

[2] VAN DE GOOR, A. *Computer Architecture and Design*. Addison-Wesley. 1989.

[3] STALLINGS, W. *Computer Organization and Architecture*. Macmillan, New York. 4th. Edition. 1996.

[4] HENNESY, J.; PATTERSON, D. *Computer Architecture: a quantitative approach.* Prentice Hall International. 1994.

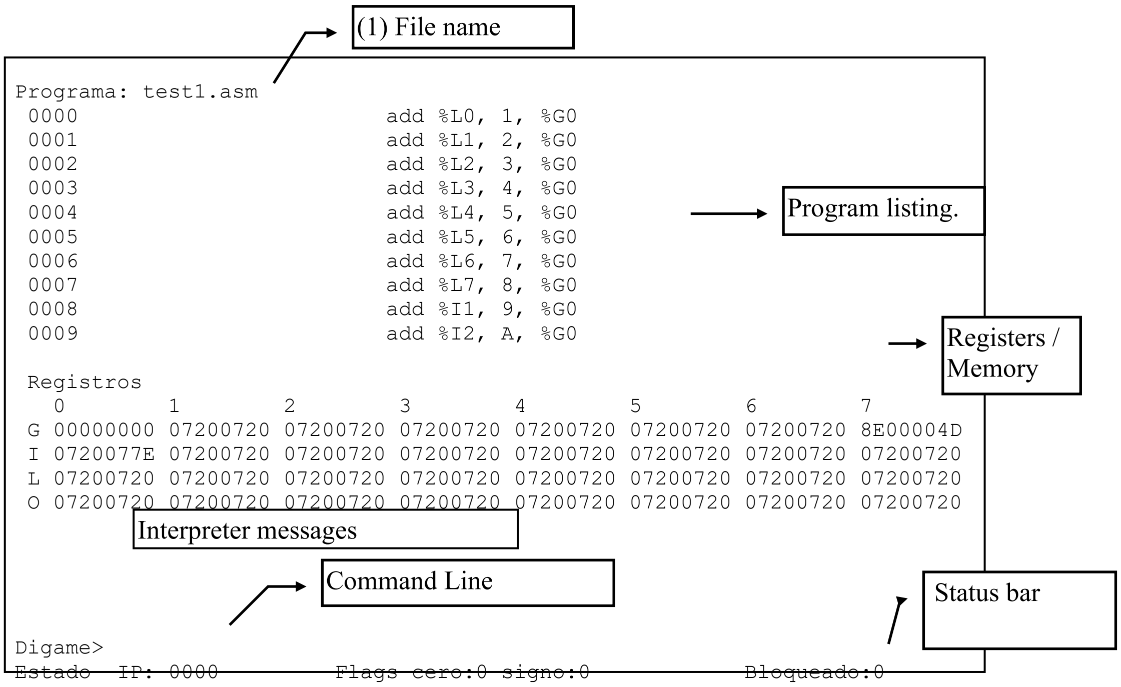
[5] FERRARI, A.; ROMANO, S.; WAINER, G. "Implementation of a digital logic library". (in Spanish). *Proceedings of INFOCOM ARGENTINA '98*. 1998.

[6] BEVILACQUA, R.; PATARO, G.; WAINER, G. et al. *Computer architecture and Operating Systems* (in Spanish). To be published by Nueva Librería S.A. 1996.

[7] ROMERO ZALIZ, L.; WAINER, G. "Implementation of a simulated cache memory for the Alpha-0 processor". (in Spanish). *Internal report, Departamento de Computación, FCEN/UBA*. 1998.

[8] TROCCOLI, A.; WAINER, G. "CRAPS: a simulator for the SPARC processor " (in Spanish). *Proceedings of INFOCOM ARGENTINA '98*. 1998.

APPENDIX



```
Programa: test1.asm
 0000                        add %L0, 1, %G0
 0001                        add %L1, 2, %G0
 0002                        add %L2, 3, %G0
 0003                        add %L3, 4, %G0
 0004                        add %L4, 5, %G0
 0005                        add %L5, 6, %G0
 0006                        add %L6, 7, %G0
 0007                        add %L7, 8, %G0
 0008                        add %I1, 9, %G0
 0009                        add %I2, A, %G0

 Registros
   0        1        2        3        4        5        6        7
 G 00000000 07200720 07200720 07200720 07200720 07200720 07200720 8E00004D
 I 0720077E 07200720 07200720 07200720 07200720 07200720 07200720 07200720
 L 07200720 07200720 07200720 07200720 07200720 07200720 07200720 07200720
 O 07200720 07200720 07200720 07200720 07200720 07200720 07200720 07200720




Digame>
Estado  IP: 0000         Flags cero:0 signo:0            Bloqueado:0
```

Figure 5. User interface organization.