TECHNICAL ARTICLE

SIMULATION 75:2, 82-90

ISSN 0037-5497/00
Printed in the United States of America

Microprocessor Simulation and Program Assembling Using Spreadsheets

Ali El-Hajj, Karim Y. Kabalan, Maher Mneimneh, and **Feras Karablieh**
Electrical and Computer Engineering Department
Faculty of Engineering and Architecture
American University of Beirut
P.O. Box 11-0236, Beirut, Lebanon

In this paper, a method for simulating microprocessor systems using modern spreadsheet programs is presented. The microprocessor architecture is depicted in one sheet and used to simulate the result of a step-by-step program execution. This simulation allows the user to write a program on a separate sheet from an instruction set available on two separate sheets. This program will then be assembled using the instruction information sheets and executed on the microprocessor architecture sheet. Additional sheets are also accessible for listing memory contents and input/output ports. This simulation has been illustratively applied to the Z80 microprocessor.

Keywords: Spreadsheets, microprocessors, assemblers, Excel

1. Introduction

Spreadsheets have been used to solve some electrical engineering problems [1, 2], and to simulate many devices and systems such as logic networks [4–7], control systems [8–10], analog computers [11], and some basic operations of microprocessor systems [12, 13].

In simulating logic and control systems, a system is divided into basic building blocks. Each block is represented by a multi-input/multi-output box labeled with its name. Two blocks are connected when the input of one is assigned a formula equal to the output address of the other. In the case of logic circuits, the basic blocks simulated are logic gates, flip-flops and some MSI circuits. In addition, a clock is simulated for use in synchronous networks. It is then possible to simulate combinational, sequential, synchronous, and asynchronous networks. In the case of a linear control system, the basic block device is the integrator. Connecting together a number of integrators with appropriate scaling coefficients can simulate a transfer function in the s-domain. These simulated transfer functions are connected, as required, to construct the block diagram of a linear control system. In the case of nonlinear control systems, the basic building blocks are the nonlinear elements. These blocks can be incorporated

in a large control system and the response of this system to any input can be obtained. In the case of sampled data control systems, the delay z^{-1} forms the basic block. Connections using a number of z^{-1} blocks allow the representation of any transfer function or control system described in the z domain.

The same simulation method is used to simulate a one-bit microprocessor (Motorola 14500) [12]. This microprocessor was represented as a box showing its inputs and outputs. Formulas are used to calculate the outputs as functions of the data and control inputs. A different approach was also used in [13] to simulate a simple ALU with limited memory, registers, and instruction sets.

This paper presents an improved method for microprocessor simulation using the Excel™ 97 spreadsheet program. This improved method makes use of many advanced features available in this program. It adds many advantages to the previous method, making it more attractive in an educational environment and for spreadsheet users. In addition to the flexibility and availability of the spreadsheet programs, this method has the following characteristics:

1. Writing an assembly program is done on a separate sheet, making use of the spreadsheet editing facilities.
2. A program can be assembled. The resulting code is stored on a separate sheet that can be examined by the user.
3. It allows a step-by-step execution of a program showing how register contents and flags change after every instruction.
4. A code can be directly loaded into memory for execution.

This method has many advantages when compared to existing software development tools that allow a user to write an assembly language source code, assemble the source code to machine code, and debug the resulting file. Some of these advantages are:

1. The method makes use of spreadsheets that are basic software packages widely available. It inherits from spreadsheets many of their characteristics, such as low cost, simplicity, and flexibility.
2. It is possible for the user to define her/his own microprocessor and assembly language. This is particularly suitable for educational purposes where simple microprocessors may be used at an introductory level. It is also suitable for users who have no immediate access to assemblers for some machine languages.
3. It employs an improved interface that allows a user to navigate between the source code, the machine code, the instruction set, and the runtime environment, as simple as navigating between adjacent worksheets.

To illustrate this method, the simulation tools are introduced in Section 2, the program assembling is presented in Section 3, and the program execution is outlined in Section 4. The method is discussed in Section 5, and concluding remarks are given in Section 6. The Z80 microprocessor [14] is used for illustration as having small instructions bit-size, a wide range of instruction sets, and many different addressing modes. The method is also applicable to the Pentium microprocessor and to other types of microprocessors [15].

2. The Simulation Tools

In order to make the simulation possible, many spreadsheet features are used. Some of these features are:

1. **The Binary and Hexadecimal operations**: These operations allow the conversion between decimal, hexadecimal, binary, and octal representations of numbers. When used, arithmetic integer operation can be applied on hexadecimal operands by converting these operands to decimals, applying the operation, and converting the result back to hexadecimal. This method is used in calculating addresses of instructions in the case when the address of an instruction is equal to the size of the previous instruction that is added to its address.
2. **String operations**: These operations, when used, allow the user to concatenate code, to extract parts of some code, or to do other tasks. For example, in some cases an instruction code is obtained by concatenating the operation code with the operand values.
3. **Lookup functions**: These functions are used in assembling or executing a program. In assembling a program, an instruction code is obtained by a lookup in a table that contains all instruction mnemonics and their corresponding codes. In executing an instruction, a register value is calculated by a lookup in a table that contains all instructions and the registers that are affected by the required operation to get their new values.
4. **Macros**: Most of the simulation is done using the tools described above. In a few cases, a set of tasks must be automated and some codes must be written using macros. In order to minimize the programming effort, a macro that partially or completely performs a certain task is first recorded. The macro code is then edited by eliminating unnecessary code that is often added by the macro recorder, or by adding more code that completes a certain task.

In order to make the simulation easy to use, different information is stored in different workbook sheets. Some of these workbooks are used in the assembling process of a program, with the others being used during the execution process. In assembling a program, three sheets are used: The Program sheet, the Instruction Set sheet, and the Code sheet. The Program sheet

is used to input a program, the Instruction Set sheet is used to store all instructions and their code, and the Code sheet is used to store the executable form of a program. The assembling process reads the program in the Program sheet and uses information from the Instruction Set sheet to produce the executable form in the Code sheet. This assembling process will be described in detail in Section 3. In executing a program, the Code sheet is used as the input to the execution process. This process makes use of two additional sheets: The Instruction Operation sheet and the Architecture sheet. The Instruction Operation sheet is used to store all instructions, the registers, and the flags affected, along with the required operation to get their new values. The Architecture sheet shows all registers with their contents. The execution process executes an instruction by looking at its operation in the Instruction Operation sheet and updating the affected registers in the Architecture sheet. This execution process will be described in detail in Section 4.

3. Program Assembling

Assembler design techniques using a high-level programming language are discussed in many textbooks [16]. In order to use these techniques in a spreadsheet environment, the assembler tables and outputs are stored in worksheets, and the assembler algorithms are modified to fit in the new environment and are implemented using functions and macros.

In assembling a program, three sheets are used: the Program sheet, the Instruction Set sheet, and the Code sheet. These sheets contain the following information:

1. The Instruction Set Description sheet (InstrSet): This sheet contains all microprocessor instructions and their codes as shown in Figure 1. It is prepared once for each type of microprocessor. The information is stored in different columns as follows:
   a. Column C: Contains all instruction mnemonics.
   b. Columns D and E: These two columns are used to calculate and store the operand values of the instruction that is currently assembled. As an example, consider the instruction LDA,n at cell C45. The operand n is obtained using a lookup at the corresponding instruction in the program currently in execution. This is done by writing at cell D45 the instruction:

      =VLOOKUP("LDA??H";Program;5;FALSE)

      In this instruction, "Program" is the name of the four cells in the row that contains the current instruction to assemble. In the case of the program in Figure 2, when Program consists of the instruction at row 6, the value of the calculated operand in Figure 1 is 24H.
   c. Column F: This column is used to combine the instruction mnemonic with its operands by using the string concatenation spreadsheet operator. This leads to the concatenated instruction form used for lookup purposes. As an example, the concatenated form of the instruction LDA,24H is obtained by writing at cell F45 the formula: ="LDA"&D45.

	C	D	E	F	G	H	I	J	K	L	M
1	InstructionSet										
2	*Instruction*	*Operand1*	*Operand2*	*Instruc*	*Byte1*	*Byte2*	*Byte3*	*Byte4*	*FullOpcode*	*No. of Bytes*	*No. of Opcodes*
34	LD H,B			LDHB	01100000				01100000	1	1
35	LD H,C			LDHC	01100001				01100001	1	1
36	LD H,D			LDHD	01100010				01100010	1	1
37	LD H,E			LDHE	01100011				01100011	1	1
38	LD H,L			LDHL	01100101				01100101	1	1
39	LD L,A			LDLA	01101111				01101111	1	1
40	LD L,B			LDLB	01101000				01101000	1	1
41	LD L,C			LDLC	01101001				01101001	1	1
42	LD L,D			LDLD	01101010				01101010	1	1
43	LD L,E			LDLE	01101011				01101011	1	1
44	LD L,H			LDLH	01101100				01101100	1	1
45	LD A,n	24H		LDA24H	00111110	00100100			00111110	2	1
46	LD B,n	#N/A		#N/A	00000110	#N/A			00000110	2	1
47	LD C,n	#N/A		#N/A	00001110	#N/A			00001110	2	1
48	LD D,n	#N/A		#N/A	00010110	#N/A			00010110	2	1
49	LD E,n	#N/A		#N/A	00011110	#N/A			00011110	2	1
50	LD H,n	#N/A		#N/A	00100110	#N/A			00100110	2	1
51	LD L,n	#N/A		#N/A	00101110	#N/A			00101110	2	1
52	LD A,(HL)			LDA(HL)	01111110				01111110	1	1
53	LD B,(HL)			LDB(HL)	01000110				01000110	1	1
54	LD C,(HL)			LDC(HL)	01001110				01001110	1	1

Figure 1. The Instruction Set description sheet

d. Columns G to J are used to store, one byte per column, the binary instruction code. Column G contains the operation code obtained from the microprocessor instruction manual. The other columns contain operation codes or operands. The operands are sometimes calculated values using formulas. As an example, the operand of the instruction LDA,24H is calculated by writing at cell H45 the formula:

=HEX2BIN(MID(D45;1;LEN(D45)-1);8)

In this formula, the letter H is eliminated and the operand is converted to binary. For some instructions, the corresponding H cell contains a second operation code byte, and operand calculation starts at column I.

e. Column K is used to calculate the total operation code (operands not included). This calculation uses the instruction format obtained from column M.

f. Columns L and M are used to indicate the instruction format. The instruction length (operation code + operands) is stored in column L, and the operation code size is stored in column M. It follows that the instruction code calculated in column K uses the formula:

K2: =IF(M2>1;G2&H2;G2)

This formula is copied to the other cells in column K.

2. The Program sheet is shown in Figure 2. This sheet is used to enter programs in assembly language as follows:

a. Column B is used for labels.

b. Column C is used for the instruction mnemonic.

c. Columns D and E are used for the first and second operand.

d. The mnemonic and the operands are concatenated to obtain the same form as in column F of InstrucSet sheet. The resulting concatenated instruction is stored in column A. This is done by writing in cell A4 the formula:

=C4&D4&E4

This formula is then copied to the other cells in column A. For convenience in the display and protection, column A is formatted as hidden.

e. Column F contains the relative memory instruction address. An address is obtained by adding the address of the previous instruction to the size of the current one. This size is obtained after a lookup at column M in sheet InstrSet. This is done by writing in cell F5 the formula:

DEC2HEX(HEX2DEC(F4)+VLOOKUP (A4;InstrucRange;7;FALSE);4)

In this formula, InstrucRange is the name of the lookup area in sheet InstrSet. The addition of the hexadecimal values is done in this formula, as explained in the previous section. Addressing starts at memory address zero that is stored at cell F4. Column F is formatted as hidden for convenience in display and protection. To start at an address different from zero, column F is unhidden and the starting address is stored at cell F4. As an alternative, the directive ORG can be used, but the formula that calculates addresses becomes more complicated.

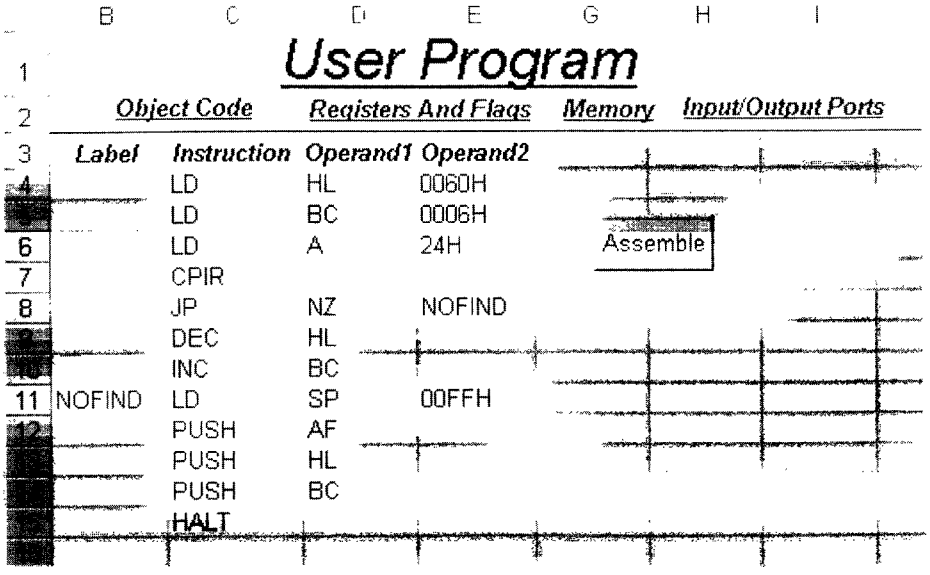


Figure 2. The Program sheet

3. The Code sheet: This sheet contains the executable code of the program written by the assembling process. The content of this sheet is written at assembly time and used as input to the execution process. As shown in Figure 3, this sheet contains the following columns:
   a. Column A for memory addresses.
   b. Column B for the operation code.
   c. Columns C and D for the operand values.

The assembling process reads the program in the Program sheet and uses information from the Instruction Set sheet to produce the executable form in the Code sheet. In order to assemble the program, two passes are needed.

In a first pass, and after necessary initializations, the following steps are done for each instruction:

1. The range containing the current instruction is named "Program."
2. The matching row in InstrSet is located.
3. The cells of the located rows in columns D, E, and F in InstrSet are calculated.
4. The address of the current instruction in column F in the program sheet is calculated, as indicated in Section 3, Item 2.e.

In a second pass over the program, the following steps are done for each instruction:

1. Labels used as operands are looked up in column B and replaced by their absolute or relative values obtained using column F.
2. The matching row in InstrSet is located.
3. The cells of the located rows in columns D, E, F, H, I, and J in InstrSet are calculated.
4. The current instruction code obtained by lookup at columns G, H, I, and J in the Instruction sheet is stored in the Code sheet.

The actions in the two passes are automated in a macro that is attached to the button "Assemble" as shown in Figure 2. It is important to note that formulae are already written in Columns A and F of the Program sheet and columns D, F, H, I, and J in the InstrSet sheet. The assembling code only causes a spreadsheet calculation of the cells in these columns.

4. Program Execution

The execution process uses the content of the code sheet as its input. It makes use of three additional sheets: The Instruction Operation sheet (InstOperation), the Architecture sheet, and the Memory sheet. These sheets contain the following information:

1. The Instruction Operation sheet: As shown in Figure 4, this sheet contains a description of instruction operations with affected registers and flags. This is accomplished by including all instructions, and the registers and the flags affected, with the operation required to obtain their new values. The execution process executes an instruction by looking at its entry in this sheet, and by updating in the Architecture sheet the affected registers and flags and in the Memory sheet the appropriate memory addresses. This sheet is organized as follows:
   a. Column A: This column contains some intermediate calculations used by some instructions. It is formatted as hidden for convenience in the display.
   b. Column B: Contains instructions mnemonics.
   c. Column C: Contains instructions operation codes.
   d. Columns D and E contain the operands. The operands are obtained using a lookup in the Code

	A	B	C	D	E	F
1		Object Code				
2		User Program	Registers And Flags	Memory	Input/Output Ports	
3	Address	Opcode	Operands			
4	0000	00100001	01100000	00000000		
5	0003	00000001	00000110	00000000		
6	0006	00111110	00100100			
7	0008	1110110110110001				
8	000A	11000010	00001111	00000000		
9	000D	00101011				
10	000E	00000011				
11	000F	00110001	11111111	00000000		
12	0012	11110101				
13	0013	11100101				
14	0014	11000101				
15	0015	01110110				
16						

Figure 3. The Code sheet

sheet. For example, the operand of the instruction LDA,24H is obtained at cell D45 using the formula:

=VLOOKUP(C45;Execute;2;FALSE)

In this instruction, Execute is the name of the four cells starting at column B in the row that contains the current instruction to execute. In the case of the program in Figure 3, when Execute consists of the instruction at row 6, the value of the calculated operand in Figure 4 is 00100100.

e. Column F: Contains the register affected by the instruction. For example, the first register affected by the instruction LDA,24H is register A. Cell F45 is then loaded with the string expression regA1 in which the variable regA1 is the name of a cell located in the Architecture sheet and used to store the accumulator value.

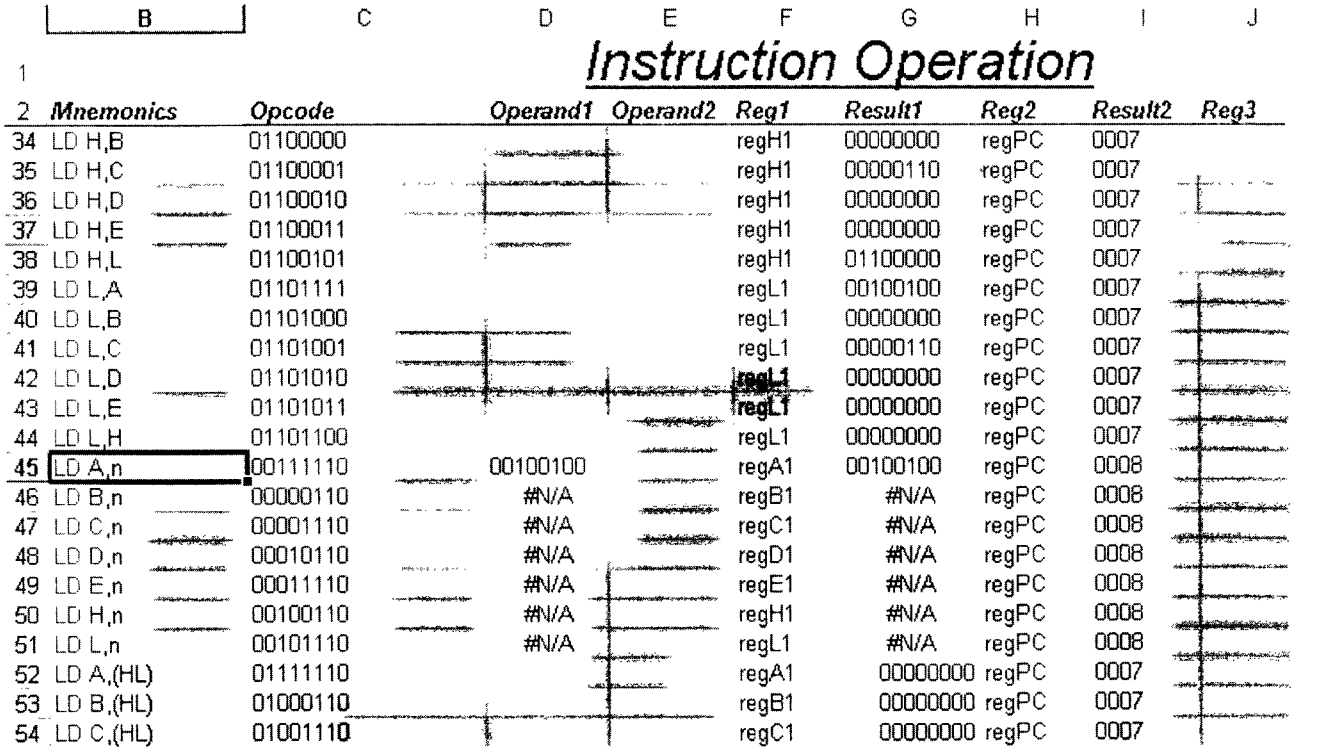
f. Column G contains the new value to store in the register affected and whose name is in the previous column. This value is calculated using different methods for different instructions. For example, the value loaded into register A by the instruction LDA,24H is obtained by writing at cell G45 using the formula: =D45.

g. The information in columns F and G is repeated in successive columns since more than one register may be affected by an instruction. The Program Counter (PC) is treated as a register affected by all the instructions. In the case of the instruction LDA,24H, Column H45 contains the string expression regPC indicating that the PC is affected. The new value of the PC is equal to its old value added to the size of the current instruction. This is done by writing in cell I45 the formula:

=DEC2HEX(HEX2DEC(regPC)+
VLOOKUP(C45;FullOpcode;2;FALSE);4)

In this formula, the size of the instruction in cell C45 is obtained by a lookup at a defined subrange called FullOpcode in the InstrSet sheet. The variable regPC is the name of the cell used to store the PC value located in the Architecture sheet. Since the PC is affected by all instructions, this formula is copied to many cells (jump instructions contain different formulas) in column I and cells in column H containing the regPC value. Additional columns are used to indicate that other registers, flags, memory locations, input/outputs, and interrupt devices have been affected.

2. The Architecture sheet: This sheet is shown in Figure 5. It shows the last executed instruction and the updated contents of all registers and flags. In the case of the Instruction LDA,24H, it is shown that the register A and the program counters are loaded with the right values. This sheet can be used to show the normal code execution or a step-by-step execution by clicking at the right button in the range



Instruction Operation

	B	Opcode	Operand1	Operand2	Reg1	Result1	Reg2	Result2	Reg3
34	LD H,B	01100000			regH1	00000000	regPC	0007	
35	LD H,C	01100001			regH1	00000110	regPC	0007	
36	LD H,D	01100010			regH1	00000000	regPC	0007	
37	LD H,E	01100011			regH1	00000000	regPC	0007	
38	LD H,L	01100101			regH1	01100000	regPC	0007	
39	LD L,A	01101111			regL1	00100100	regPC	0007	
40	LD L,B	01101000			regL1	00000000	regPC	0007	
41	LD L,C	01101001			regL1	00000110	regPC	0007	
42	LD L,D	01101010			regL1	00000000	regPC	0007	
43	LD L,E	01101011			regL1	00000000	regPC	0007	
44	LD L,H	01101100			regL1	00000000	regPC	0007	
45	LD A,n	00111110	00100100		regA1	00100100	regPC	0008	
46	LD B,n	00000110	#N/A		regB1	#N/A	regPC	0008	
47	LD C,n	00001110	#N/A		regC1	#N/A	regPC	0008	
48	LD D,n	00010110	#N/A		regD1	#N/A	regPC	0008	
49	LD E,n	00011110	#N/A		regE1	#N/A	regPC	0008	
50	LD H,n	00100110	#N/A		regH1	#N/A	regPC	0008	
51	LD L,n	00101110	#N/A		regL1	#N/A	regPC	0008	
52	LD A,(HL)	01111110			regA1	00000000	regPC	0007	
53	LD B,(HL)	01000110			regB1	00000000	regPC	0007	
54	LD C,(HL)	01001110			regC1	00000000	regPC	0007	

Figure 4. The Instruction Operation sheet

J5:K10 on the sheet. Clicking the reset button is required before starting any program execution. The simulation is extended to support interrupts. The user during a step execution may signal an interrupt by inputting the value 1 at cell J19. This causes saving of the Program Counter and starting the execution of an interrupt handler whose location depends on the interrupt mode set by the program at cell J17. This requires that the program using cells J23 and K23 enables the interrupts. The interrupt acknowledgement is indicated using cell J21.

The execution process performs a single pass over the code in the Code sheet. For each instruction, the matching row in the Instruction operation sheet is located and calculated. The resulting values of the affected registers and flags are stored in their representative cells in the Architecture sheet. The next instruction to execute is obtained using the calculated value of the Program Counter. This allows the handling of all Jump instructions by an appropriate calculation of the next Program Counter value for each of these instructions and locating the next instruction to execute in the Code sheet. This execution process is done using a macro code.

The Memory sheet is shown in Figure 6 for the program of Figure 2 with code in Figure 3. In this sheet, every row is designed to contain 16 bytes. A memory address is obtained by masking the row address with the column address. The content of the Memory sheet is loaded at assembly time by a special macro. It is updated during execution when an instruction that affects memory is reached.

An Input/Output sheet is used in the case when input/output devices participate in a program. This sheet gives all input/output ports that are updated during execution. It is organized in a similar way to the Memory sheet.

5. Discussion

The method presented has its advantages and limitations when compared with other methods for assembler design [16]. Its main advantage is in using spreadsheets

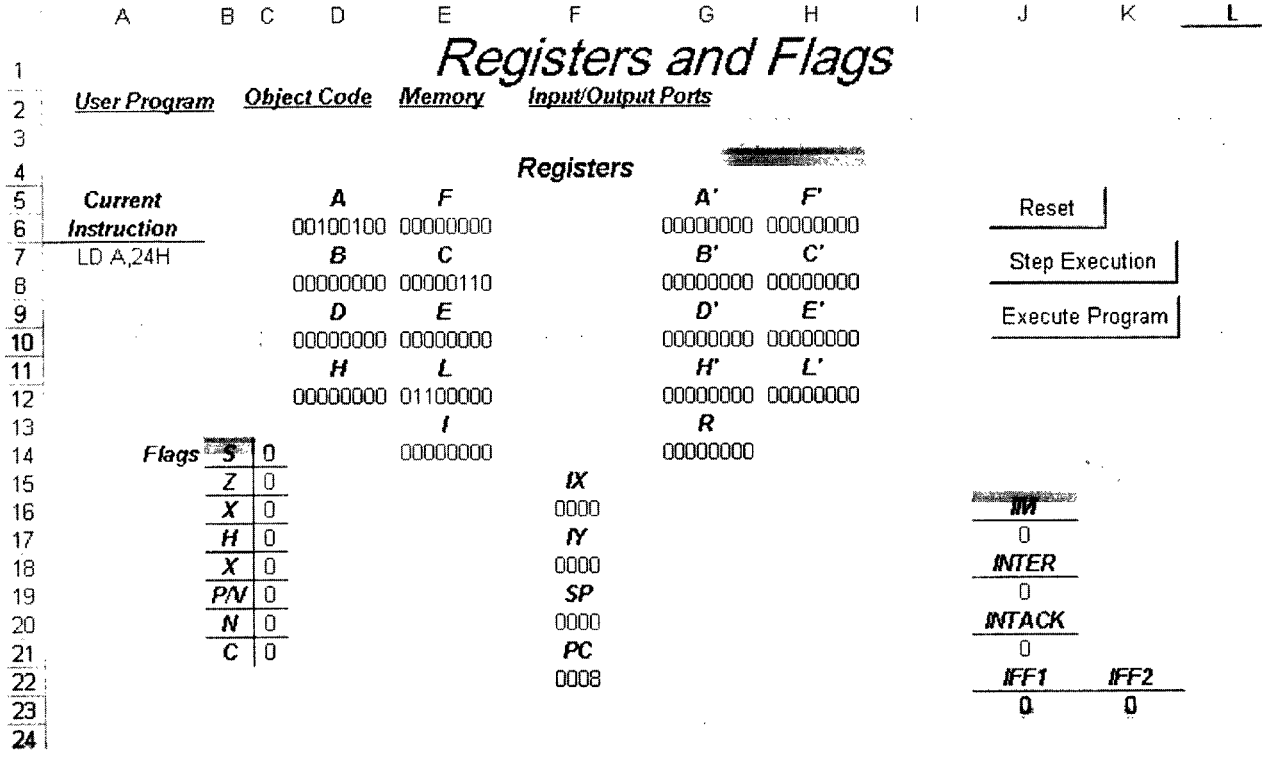


Figure 5. The Architecture sheet

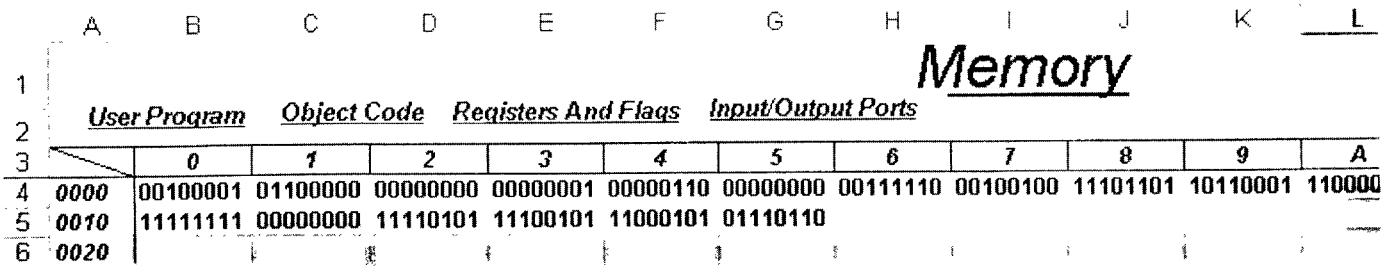


Figure 6. The Memory sheet

that are widely available on most computers, and familiar to a large class of computer users. Due to our familiarity with spreadsheets, it was possible to develop such an application using macros and formulas. This approach reduces considerably the programming effort (Section 2) that is usually required to develop similar applications using a high-level programming language.

In order to illustrate this method, the Z80 microprocessor [14] was used for the following reasons:

1. Most Z80 microprocessor instructions are coded on 8 or 16 bits. Illustrating the method in this case is much easier than using other microprocessors with larger instructions sizes [14].
2. The Z80 microprocessor has a broad instruction set that can be classified into five main categories: data transfer, data processing, test and branch, input/output, and control instructions.
3. The Z80 microprocessor supports many addressing modes: implied addressing, immediate addressing, absolute addressing, relative addressing, indexed addressing, and register indirect addressing.

More information about the Z80 microprocessor can be found in [14]. The same simulation method can be applied to more advanced microprocessors such as the Pentium [15]. This microprocessor has the same instruction categories as the Z80, in addition to the "Strings" category. It has some more advanced addressing modes such as based addressing and string addressing. This leads, in many cases, to more complicated instructions that are not found in the Z80. However, the same simulation method can be used for the Pentium since the additional complexity in the instructions affects mainly the spreadsheet formulas used in the implementation. On the other hand, this method does not show the timing characteristics of microprocessors. It can only be used to write an assembly program and execute it.

In summary, this method is of low cost and is good for quick simulation when the user doesn't have the time or the means to write or access a sophisticated microprocessor simulation package. It is very useful for educational purposes where budgets and resources are sometimes limited, and lends itself to easy implementation in an undergraduate project. It provides introductory instructional tools allowing the definition of simplified microprocessors. It also permits the simulation of many microprocessors and the execution of programs using their own assembly languages. This feature makes the method different from many available software programs that assemble a program and generate code that runs only on one machine. The method is not suitable for studying timing and electronics microprocessors features.

6. Conclusion

A method for microprocessor simulation using modern spreadsheet programs has been presented. This method uses many spreadsheet features such as formulas, macros, control buttons, and others. The method uses a basic software package familiar to a large class of computer users to perform a very complicated task with no need to write a sophisticated program or access a sophisticated package. It is also very useful for educational purposes since for each execution step of the program, the contents of the registers and memory can be recorded, and input/output values can be easily read. This method has been illustrated using the Z80 microprocessor. This method is general and can be applied to simulate other microprocessors.

7. Acknowledgments

The authors are grateful to Professor P. Azoury for reviewing the manuscript and to Mr. K. Mazloum for his assistance.

8. References

[1] Huelsman, L.P. "Electrical Engineering Applications of Microcomputer Spreadsheet Analysis Programs." *IEEE Transactions on Education*, Vol. E-27, No. 2, pp 86-92, 1984.

[2] Rao, N.D. "Typical Applications of Microcomputer Spreadsheets to Electrical Engineering Problems." *IEEE Transactions on Education*, Vol. E-27, No. 4, pp 237-242, 1984.

[3] Saul, P.H. "The Spreadsheet as a High Level Mixed Mode Simulation Tool." *Electronic Engineering*, pp 49-55, 1991.

[4] El-Hajj, A. and Kabalan, K.Y. "A Spreadsheet Simulation of Logic Networks." *IEEE Transactions on Education*, Vol. E-34, pp 43-46, 1991.

[5] El-Hajj, A., Kabalan, K.Y and Yehia, M. "Simulation of a Class of Integrated Circuits Using Spreadsheets." *IEE Proceedings-G*, Vol. 139, No. 5, pp 607-610, 1992.

[6] El-Hajj, A. and Hazim, M. "On Using Spreadsheets for Logic Networks Simulation." *IEEE Transactions on Education*, Vol. 41, No. 4, pp 311-319, 1998.

[7] El-Hajj, A., Diab, H. and Hazim, M. "An Improved MSI Logic Circuits Simulation Method Using Spreadsheets." *IASTED International Conference on Modeling and Simulation 1999*, pp 491-494, Philadelphia, 1999.

[8] El-Hajj, A. and Kabalan, K.Y. "Time Domain Analysis of Linear Control Systems using Spreadsheets." *IEEE Transactions on Education*, Vol. 38, No. 4, pp 317-320, 1995.

[9] Kabalan, K.Y., El-Hajj, A. and Smari, W. "Nonlinear and Sampled Data Control Systems Analysis Using Spreadsheets." *IEE Proceedings-A*, Vol. 143, No. 1, pp 52-56, 1996.

[10] El-Hajj, A. "Functional Simulation Using Spreadsheets." *SIMULATION*, Vol. 73, No. 2, pp 80-90, August 1999.

[11] Kabalan, K.Y., El-Hajj, A., Diab, H. and Fakhreddine, S. "Analog Computer Simulation Using Spreadsheets." *SIMULATION*, Vol. 68, No. 2, pp 101-106, 1997.

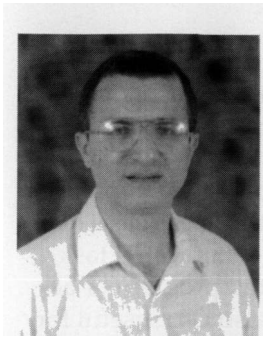
[12] El-Hajj, A., Kabalan, K.Y. and Chehab, A. "A Microprocessor Simulation Using Spreadsheets." *International Conference on Microelectronics*, pp 284-287, Cairo, Egypt, 1991.

[13] El-Hajj, A., Kabalan, K.Y. and Diab, H. "A Spreadsheet Educational Tool for Microprocessor Systems." *Journal of Computer Applications in Engineering Education*, Vol. 3, No. 3, pp 205-211, 1995.

[14] Zaks, R. *Programming the Z80*, Third Edition, Sybex, 1982.

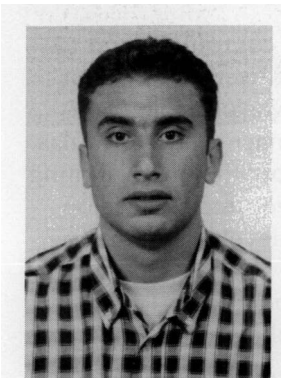
[15] Antonakos, J.L. *The Pentium Microprocessor*, Prentice-Hall International, 1997.

[16] Beck, L.L. *System Software*, Third Edition, Addison Wesley, 1997.

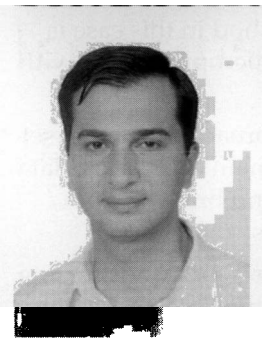


Ali El-Hajj was born in Aramta, Lebanon. He received a BS degree in Physics from the Lebanese University, Lebanon, in 1979, the degree of "Ingenieur" from L'Ecole Superieure d'Electricite, France, in 1981, and the "Docteur Ingenieur" degree from the University of Rennes I, France, in 1983. From 1983 to 1987, he was with the Electrical Engineering Department at the Lebanese University. In 1987, he joined the American University of Beirut where he is currently Professor of Electrical and Computer Engineering. His research interests are numerical solution of electromagnetic field problems and engineering education.

Karim Y. Kabalan was born in Jbeil, Lebanon. He received a BS degree in Physics from the Lebanese University in 1979, and MS and PhD degrees in Electrical Engineering from Syracuse University in 1983 and 1985, respectively. During the 1986 fall semester, he was a Visiting Assistant Professor of Electrical Engineering at Syracuse University. Currently he is a Professor of Electrical Engineering with the Electrical and Computer Engineering Department, Faculty of Engineering and Architecture, American University of Beirut. His research interests are numerical solution of electromagnetic field problems and software development.



Maher Mneimneh received his BE in Computer and Communications Engineering from the American University of Beirut in 1999. He is currently pursuing his MS and PhD degrees in Computer Science and Engineering at the University of Michigan in Ann Arbor. His research interests include computer architecture and microarchitecutre and the development of hardware verification technologies for microprocessors.



Feras Karablieh was born in 1978 in Beirut, Lebanon. He received his BE in Computer and Communications Engineering from the American University of Beirut in 1999. He is currently pursuing his PhD in Computer Science and Engineering at Arizona State University. His research interests include distributed programming, software security and application level fault tolerance.