Int. J. Engng Ed. Vol. 20, No. 6, pp. 000–000, 2004
Printed in Great Britain.

0949-149X/91 $3.00+0.00

Spreadsheet-based Logic Controller for Teaching Fundamentals of Requirements Engineering*

HANANIA T. SALZER and ILYA LEVIN
Tel Aviv University, School of Education, Tel Aviv, Israel. E-mail: salzerha@post.tau.ac.il

The paper introduces a new idea of using spreadsheets for teaching basics of the discipline called requirements engineering. The discipline is familiar to those dealing with design of complex systems. To this purpose, the authors propose utilizing a specific spreadsheet-based logical simulator. For simulating a computer system on the level of its initial specifications the authors developed a dedicated so-called requirements simulator (RS). For building the simulator, students have to separate the system requirements into a number of abstraction levels. The paper presents a real example of designing RS for the control part of the well-known PARWAN microprocessor using a spreadsheet.

INTRODUCTION

THE PRESENT DECADE may be characterized as a period of triumph for the spreadsheet technology in education. While ten years ago a spreadsheet was perceived just as one of the possible calculation means, and only a small group of enthusiasts believed in its educational potential, currently it goes without saying that spreadsheets have turned into a classic learning environment for exploring the idea of simulation.

On the one hand, it is still too early to summarize the role of spreadsheets in education, because new concepts, ideas and applications continue appearing both in the technological and in the pedagogical arenas. On the other hand, some subject matters and applications have acquired sufficient experience that permits us not only to give practical recommendations but also to formulate some conceptual ideas.

One such field is computer design, which is genetically predestined for simulation. A widely used hardware description language, VHDL, is an example of a tool used for simulation of computer systems operation.

The simulation-based techniques offer a very promising direction from both the practical and the research points of view. It is enough to mention works [1, 2] that study various methods for constructing the simulation-based learning environments. In [3, 4] the authors investigate specific human-machine interaction issues in simulation-based learning, while [5–7] focus on computer architecture. The most interesting works in the context of our paper are [8–11]. They study the use of spreadsheet simulations for teaching computer architecture and other similar subjects.

* Accepted 14 July 2004.

Advantages of using spreadsheet simulations for teaching microprocessors are well recognized in the field, especially by experts in engineering education. We are not re-inventing the wheel by convincing the reader that spreadsheet simulation can be successfully used in a computer architecture class. We have a different goal. To the best of our knowledge, in all the above-mentioned works the authors present simulations of the processor (computer) structure. They never discuss or develop simulations of requirements (specifications) by using spreadsheets. We propose developing a spreadsheet simulation of a microprocessor (computer) at the level of requirements, which would allow verification of internal logic of the requirements before designing and even simulating the microprocessor structure. We address the present paper precisely to this issue.

REQUIREMENTS ENGINEERING

Requirement specifications provide textual models of system components. The traditional scope of requirements engineering (RE) is concerned with the real-world goals for functions of and constraints on systems [13] as has been specified by clients, customers and users. On the other hand, the definition of [14], 'Requirements engineering maps problems from a problem domain into a proposed solution from the solution domain', implies that RE is applicable to any level of abstraction. The broader definition of requirements is not limited by the role of the person who introduced it or who can comprehend it; the specifications of a particular function or component, at any abstraction level, are that function's or component's requirements.

Documenting requirement specifications is useful

for communicating goals and constraints, because they reflect, to a reasonable extent, the writer's mental model. For that reason, student-written specifications provide teachers with a tool to evaluate their students' comprehension of that object's functionality, and enable students to carry out reflection on their own understanding.

Although students learn an important system design tool, teachers should be careful to ensure that students do not confuse the technique of design documentation with the art of designing. Design is a creative process, inventing new specifications at a certain abstraction level, with the aim to fulfill the requirement specifications laid down at a higher abstraction level. The learning activity in this paper does not require students to invent controllers, but to learn from a text-book the functionality of a specific controller design, and to apply state-of-the-art requirements engineering techniques to document its specifications.

Next, we present the significance of positioning requirements within the context of abstraction levels, and the significance of atomizing them.

Abstraction levels

Abstraction is a functional specification of something, corresponding to many different, alternative implementations [15, 16]. Abstraction levels are the, sometimes arbitrary, layers in a hierarchy of functional specifications. Looking from a higher abstraction level down at a lower one, the latter abstracts away any particular implementation detail. Looking from a lower abstraction level up, the higher abstraction level provides high-level requirement specifications that the lower one is expected to implement. Consequently, every specification statement is a requirement specification ('what') relative to the abstraction level with which it is associated, and in the same time it is also the specific internal design specification ('how') of a higher abstraction level [16–18, 19 (pp. 161–162)]. Because of this duality, all specifications are called in this paper—requirement specifications.

The duality of abstraction ('what'), on the one hand, and the selected implementation ('how'), on the other hand, can materialize in any number of abstraction levels in a system's specification hierarchy. Classic user requirements are associated with the highest abstraction level—the system. Usually, four levels of abstraction are of interest for computer architecture courses to describe the behavior of digital computers. Going from a high abstraction level to the lower one, they are [2]:

- assembly language;
- binary-code instruction set;
- state transitions and micro-instructions;
- the micro-operations.

A micro-instruction is the output of a state transition. A micro-operation is a single variable in the transition output, that is, a micro-instruction is comprised of one or more micro-operations.

A specification is of any meaning only when it is associated with a system-function, with a system component, or with both, at a certain abstraction level.

Atomic requirements (ATRs)

The notion of atomic requirement (ATR) specification [20] extends the definition of a well-formed requirement [21]. An ATR is defined as a requirement or design specification that is:

(a) associated with a system functionality or component;
(b) is well formed;
(c) consists of a condition and of a corresponding operation;
(d) the condition and the operation are indivisible (atomic) at the abstraction level at which the specification is being considered.

No standard languages with precise semantics exist for specifying requirements; hence mainstream industrial RE practice still uses natural language descriptions for requirements [22]. In [23] and elsewhere, there is a reluctance and inability of practitioners to use formal methods on formal specifications or to master the skills of their use. The high cost of training is one reason. Unlike a formal specification, an ATR's formalism is not in its language but in its atomicity and in its association with a single abstraction level.

Three rules of thumb may help students in composing ATRs:

- First, the specification must be associated with a known abstraction level. It may only use the lexicon of that abstraction level or higher ones. For example, the specifications of an assembly instruction may not refer to the data bus.
- Second, in a conditional specification, the condition must be indivisible. For example, the specification starting with the phrase 'when the instruction is either LDA, or ADD or AND or SUB . . . ' is not atomic; it must be split into four.
- Third, the operation must be indivisible in such a way that when the implementation is tested against the ATR, only two results are possible—either the implementation has completely passed the test, or it has completely failed it [20].

The contribution of atomicity to facilitating communication of intent between people has been reported in [20]. We expect to find similar benefits in written and oral communication using ATRs within the educational context.

ATRs of standards

The notion *ATRs of standards* and the method for their use has been introduced in [20]. The set of ATRs that are common to several constituents of an abstraction level comprise that abstraction level's standards.

For example, the two ATRs shown below are common to all state transitions in the PARWAN microprocessor's controller. Therefore these ATRs

are listed at the micro-instruction abstraction level; consequently, there is no need to repeat them for every state transition. Note that combining the two into a single specification would violate atomization.

- Change state at every falling edge of the clock.
- Change state only at the falling edge of the clock.

SPECIFYING THE MICROPROCESSOR CONTROLLER

This section provides example ATRs, regular and standard, at three different abstraction levels of the PARWAN controller—assembly language, micro-instruction and micro-operation.

The students need to be advised that any ATR at a certain abstraction level could be the subject of decomposition into several ATRs at a lower abstraction level. Each ATR is atomic only in respect to the abstraction level with which it is associated, but not in respect to lower levels. This process of successive decomposition introduces at each level a wave of new information that was not specified at the higher abstraction level; hence, decomposition itself is the creative process of design.

In this activity the students extract from [12] or from an equivalent source the PARWAN controller's specifications. They concentrate on identifying abstraction levels of interest and on atomizing the requirements. Specifying requirements from *de novo*, that is, inventing a controller, is beyond this paper's scope.

Specifying the abstraction level of assembly

At the assembly abstraction level a portion of the Von Neumann model's schematic structure is introduced. The lexicon of this level includes the accumulator, the status register and the memory array. Other components, such as buses, are hidden in lower abstraction levels.

This abstraction level's specifications are the collection of all assembly instructions' functional specifications. Figure 1 lists a few examples.

Specifying the abstraction level of micro-instructions

The micro-instruction abstraction level reveals the controller's interface: the input and output

Standards for this abstraction level:

- With *direct* addressing the value of the target address is attached to the command.
- With *direct* addressing the target address is in the same memory page as the command.
- With *indirect* addressing the target address' page is the page portion of the address attached to the command.
- With *indirect* addressing the target address' offset is at the address whose value is attached to the command.
- With *indirect* addressing the target address may be at any memory page.
- With all addressing schemes (full and page) the target address' offset is attached to the command.
- With *full* addressing the target address' page is attached to the command.
- With *page* addressing the target address is in the same page where the command occurs.
- The content at a memory address has the size of one byte.
- Status flag zero is set when the contents of the shifter unit (SHU) is zero.

LDA

- Load into the accumulator the target address' contents.
- Addressing can be indirect.

ADD

- Add to the accumulator's contents the target address' contents.
- Addressing can be indirect.

BRA_Z

- If the zero-status flag is on then go to the target address.
- Addressing is direct.

Fig. 1. Example ATRs at the assembly abstraction level.

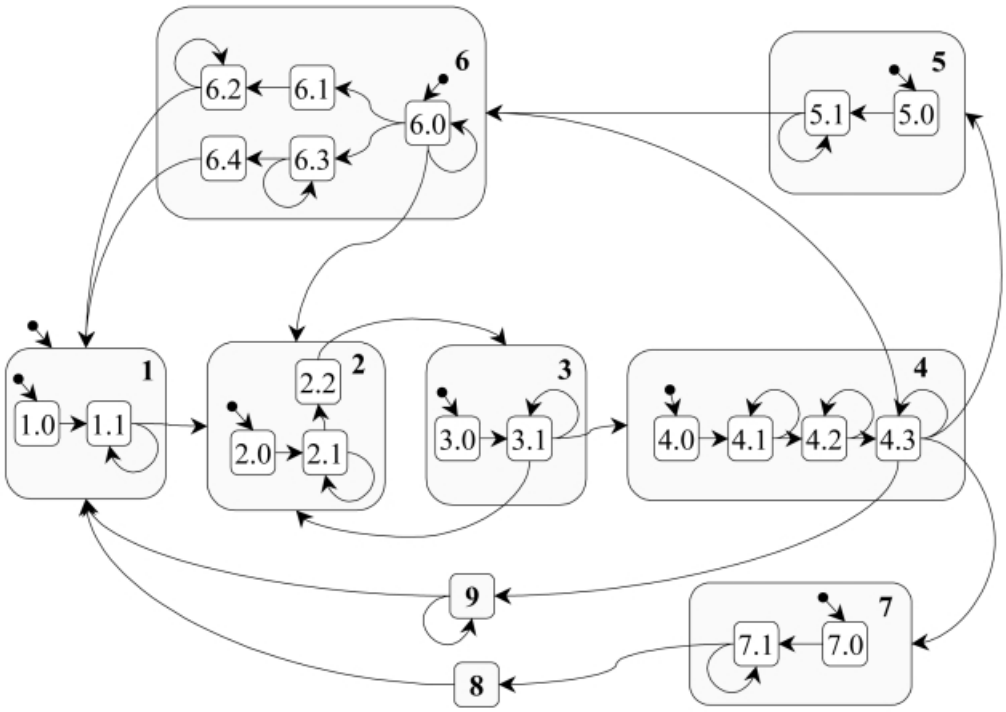

Fig. 2. State transition diagram of the PARWAN controller; edge weights are not presented.

signals [12 (p. 281)], and the rules governing its behavior as seen from the outside, while hiding its internal construction [24]. Indeed, this is the controller's behavioral design. We recommend discussing with the students that the design of the registers, memory and buses is at the same abstraction level as the design of the controller.

The full extent of the Von Neumann model is revealed at this abstraction level. The appropriate lexicon includes all registers, the register's operations repertoire and the buses as well as the memory array, but not the bus gates. Accordingly, the ATRs at this level refer to data moved between components, but do not mention the controller signals involved.

The controller is modeled as a finite state machine (FSM); therefore it is appropriate to describe its functionality with a state transition diagram, as in Fig. 2, or with a state transition table (STT), as it is constructed in the spreadsheet

State Number	State Definition
1	Ready to fetch an instruction.
2	The memory address of the next instruction is in MAR.
3	The current instruction is in IR.
4	The current instruction includes two-bytes. (The next memory byte belongs to the current instruction.)
5	The current instruction is a full address one with indirect addressing. MAR contains the address of the operand address.
6	The current instruction is a full address one with direct addressing. MAR contains the complete operand address.
7	The current instruction is JSR.
8	Ready to fetch first instruction of a subroutine.
9	The current instruction is BRA, and the branch address is in MAR.

Fig. 3. The PARWAN controller's main states.

Standards for this abstraction level:

- A bus shall get data from only one source at a time.
- Only one operation shall be sent to a register at any point of time.

- State 1.0 to 1.1: next instruction to be fetched from memory is where the PC points.
- State 1.1 to 1.1: reset the microprocessor when requested.
- State 1.1 to 2.0: if no request to reset – do nothing.
- State 3.1 to 2.0: for NOP – do nothing.
- State 3.1 to 4.0: for a two-byte instruction – the operand to be fetched from memory is where the PC points.
- State 6.0 to 6.1: for STA – prepare for writing into memory at the location specified by MAR.
- State 6.0 to 6.3: when the operation is on the content of the accumulator – read the operand from memory at the address specified by MAR.
- State 6.0 to 2.0: for JMP – move from MAR to PC the address of the location where the program should jump.

Fig. 4. Example ATRs for the micro-instruction abstraction level.

simulator in Fig. 6 and Fig. 7. The diagram follows the State Chart conventions [25]. The eight main states preserve the numbering of the eight states in [12 (pp. 300–308)]. The state transition diagram's purpose is to provide a high-level visual aid while working with the fully detailed state transition table.

Figure 4 lists example ATRs for a few state transitions to illustrate the micro-instruction abstraction level. In a STT, the current state is one of the conditions for a transition to take place, and the target state is one of the operations. We reduce the complexity of the ATRs' text considerably by moving the references to the current and target states into internal headers, as shown in Fig. 4.

Specifying the abstraction level of micro-operations

The controller's micro-operations are its output binary signals. This is the lowest abstraction level with which we deal in this paper.

The lexicon of the micro-operations abstraction level includes the gates on the buses and the three bits that code the arithmetic logic unit (ALU) operations. The signals that affect the other registers (e.g., *load_ir*) are not new to this abstraction level because the natural language of ATRs refers to them in the same way in both abstraction levels—the micro-instruction and the micro-operation.

At this abstraction level each micro-operation is an independent component that is specified separately from the other micro-operations. For this reason, the specifications at this level are simply a list of the signals, each one specified by a rather banal ATR. Figure 5 shows a few examples.

CONSTRUCTING THE CONTROLLER SIMULATOR

We propose to construct the requirement simulator of the microprocessor controller as programmable logical array (PLA) described in [26]. There are three spreadsheets that have the same number of rows and columns, taking advantage of the software's capability to directly reference from a cell in one spreadsheet the cell located in the same row and column of another spreadsheet.

One spreadsheet comprises a state transition table, with the columns on the left side representing the current state and the conditions for input signals, and the columns on the right representing the next state and the output signals corresponding to the conditions. The students fill in this spreadsheet with a combination of 1s, 0s and spaces. The second spreadsheet translates the 1s and 0s in the first spreadsheet to = true and = false functions, respectively, in the second one.

The third spreadsheet accepts the controller input signals (in row number 2) and generates the

- When *load_ac* is on – load the *dbus* data into the *accumulator*.
- When *zero_ac* is on – zero all bits of the *accumulator*.
- When *dbus_on_databus* is on – let the data from the *dbus* propagate to the *data bus*.

Fig. 5. Example ATRs for the controller output signals (micro-operations).

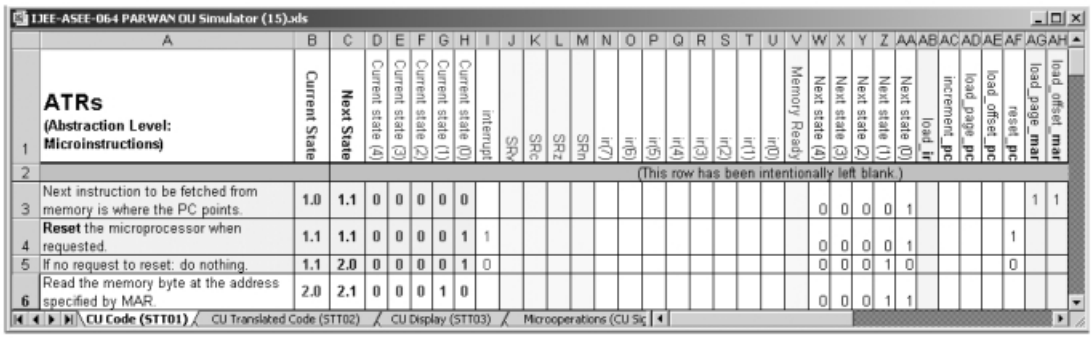

Fig. 6. The spreadsheet 'CU Code (STT01)' showing the input-signals' columns (D through V).

output signals (in the bottom row). This spreadsheet contains the mechanism that fits the input signals to a row of conditions. If the first spreadsheet is correctly set up, that is, all conditions are orthogonal and together they are complete, then any input will fit to one, and only one, row of conditions. The output generated at the bottom row is a set of cells, each one simulates an output signal having the value of either = true or = false.

Once the students have filled in the first spreadsheet, the running simulator does not change this spreadsheet or the second one. Of course, the students are encouraged to keep modifying the first spreadsheet as part of the learning activity. By that they can experiment with alternative control logic designs and can debug their design.

Students can follow this simulator's behavior at a resolution of single clock ticks. Each press on the Step button advances the process one tick of the clock. Before another press on the Step button the student may journey around the workbook's spreadsheets and examine the input and output signals (FALSE or TRUE) and the state of the controller (state number). If the rest of the simulator (all registers, buses, and the memory) are included in the setup then they can be examined as well.

Constructing the spreadsheets

This section describes step-by-step the controller construction in spreadsheets that simulate a state transition table (STT). The STT covers two abstraction levels in two spreadsheet dimensions; the micro-instructions comprise the rows, and the micro-operations the columns.

The STT is implemented within a set of three MS Excel spreadsheets labeled 'CU Code (STT01)', 'CU Translated Code (STT02)' and 'CU Display (STT03)', as shown in Fig. 6 and Fig. 7. (CU stands for 'control unit', which is another term for 'controller'.) The students start off with three blank spreadsheets. They code the controller by filling in data only in one sheet, 'CU Code (STT01)'. In the other two sheets they enter only a few formulae.

Fill in the 'CU Code (STT01)' worksheet cells as follows:

- Cell A1 through H1: enter the text as shown in Fig. 6.
- Cells I1 through V1: enter the names of the controller's input signals.
- Cells W1 through AA1: enter the text as shown in Fig. 6.
- Cells AB1 through BB1 enter the names of the controller's output signals.
- Leave row 2 empty.
- Select the range A1:BB100, and define for it the name STT01.

Enter the micro-instruction level ATRs into the *ATRs* column (column A) of the 'CU Code (STT01)' worksheet, and fill in the source and target states in the columns *Current State* and *Next State* (columns B and C), respectively. Define for each state a unique binary code. Enter the binary codes for the current and next states into columns D:H and W:AA, respectively. It is

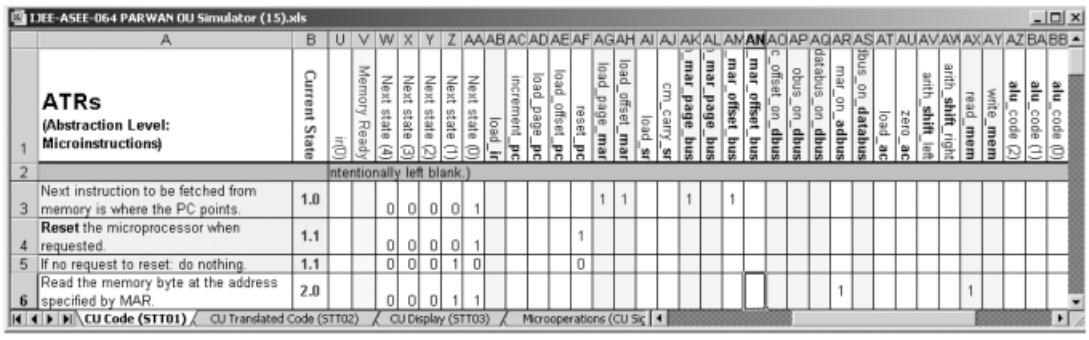

Fig. 7. The spreadsheet 'CU Code (STT01)' showing the output-signals' columns (W through BB).

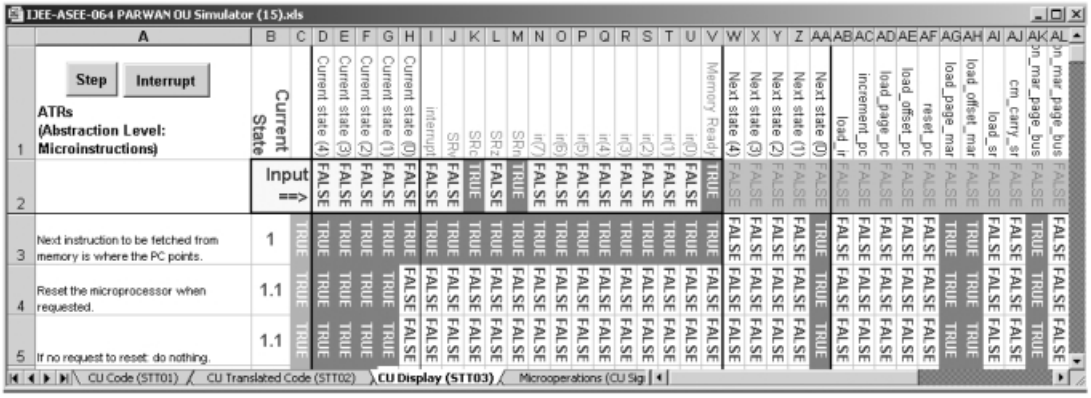

Fig. 8. Example input in the 'CU Display (STT03)' worksheet.

not necessary to use up all the rows down to row 100; empty rows have no effect on the simulator.

For each row, examine the ATR, and enter the input and output signals. For an input signal that must be on: enter 1. For an input signal that must be off: enter 0. Leave all the other input signals blank. For an output signal that must be on: enter 1. Leave all the other output signals blank. Look at the 'CU Translated Code (STT02)' spreadsheet to see that your codes have been translated to TRUEs and FALSEs.

Add rows to ensure completeness of the STT even where there is no such explicit ATR. For example, the ATR '*State 2.1 to 2.2: When the instruction, which has been read from memory, is available: put it into the ALU's A side*' implies that while the data is not available: do nothing. While the implied specification is superfluous in the ATRs' list, it is mandatory for the STT to be complete; the STT must take care of every possible combination of input signals.

Fill in the 'CU Translated Code (STT02)' worksheet cells as follows:

- Cell A1: enter the formula =STT01. Copy the formula from cell A1 to the ranges A1:BB1 and A1:A100.
- Cell D3: enter the formula =IF(STT01="","", IF(STT01=0,FALSE,TRUE)). Copy cell D3 to the range E3:BB100.
- Select the range A1:BB100, and define for it the name STT02.

Fill in the 'CU Display (STT03)' worksheet cells as follows:

- Cell A1: enter the formula =STT01. Copy the formula from cell A1 to the ranges A1:BB1 and A1:A100.
- Cell C3: enter the formula =TRUE. Copy the formula from cell C3 to the range C4:C100.
- Cell D3: enter the formula =AND(C3, OR(STT02="",STT02=D$2)). Copy cell D3 to the range E3:V100. To ensure that relative cell references are adjusted, first select the range E3:V3, and press CTRL+R, and then, select the range E3:V100, and press CTRL+D.
- Cell W2: enter the formula =FALSE. Copy the formula from cell W2 to the range W2:BB2.
- Cell W3: enter the formula =OR(W2,AND ($V3,STT02=TRUE)). Copy cell W3 to the range W3:BB100. To ensure that relative cell references are adjusted, first select the range W3:BB3, and press CTRL+R, and then, select the range W3:BB100, and press CTRL+D.
- Cell W101: enter the formula =W100. Copy the formula from cell W101 to the range W101:BB101. To ensure that relative cell references are adjusted, select the range W101:BB101, and press CTRL+R.

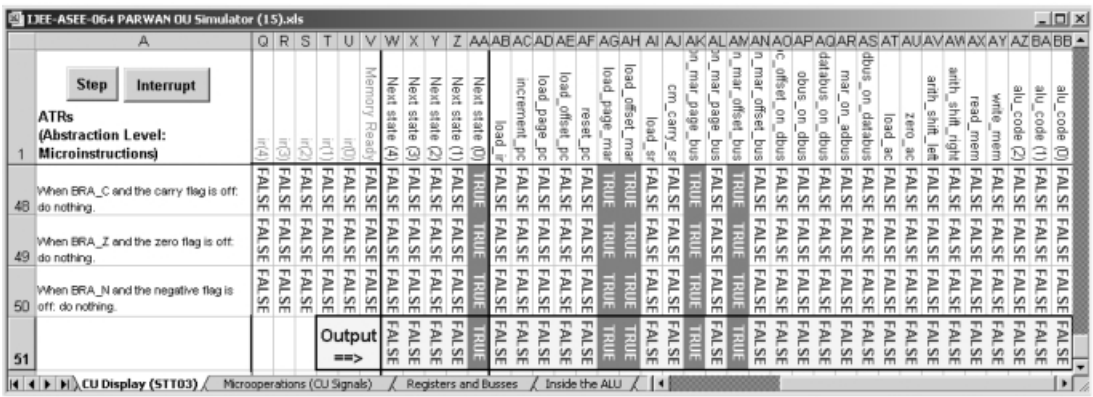

Fig. 9. Example output in the 'CU Display (STT03)' worksheet.

At this time the spreadsheet-simulation of the controller can be run manually, even without connecting it to the spreadsheet-simulation of the rest of the PARWAN microprocessor. Open the 'CU Display (STT03)' worksheet, and enter an input of some combination of =TRUE and =FALSE values into the input area at cells D2:V2, as shown in Fig. 8. Go to the bottom-right corner of the spreadsheet, and see the output signals in cells W101:BB101, as shown in Fig. 9.

To run the simulated controller within the environment of the simulated PARWAN microprocessor, the students should download the Excel workbook from [27]. Copy your three controller worksheets (described above) into the downloaded workbook and follow the instructions that accompany the workbook to create the necessary connections. Defining names for certain cells in the 'CU Display (STT03)' worksheet and copying two buttons into it accomplish the necessary connections.

Running the simulator

To program the microprocessor, enter a binary code program into the 'Memory' worksheet, such as in Fig. 10.

To run the simulator, repeatedly press the Step button, which is available at the top of most spreadsheets. Each press on the Step button advances the simulation by one tick of its internal clock. The simulator has only one clock. You may go at any time to any other worksheet, and continue the process pressing the Step button on that worksheet. This way, it is possible to examine at any point of time the contents of all registers, buses, memory, signals that enter the controller, micro-commands, and a very detailed view of the controller's internal state.

Press the Interrupt toggle-button to interrupt the program. The PARWAN microprocessor senses the interrupt when it reaches state 1.1. Press again on the Interrupt toggle-button to let the microprocessor run the program that starts at memory address 0:000.

REQUIREMENTS VERIFICATION BY STUDENTS

The requirements-based foundation of our approach facilitates specification verification as well as solid requirements-based verification. The latter means that for each test the student predicts the expected result from the ATR in question. Furthermore, the finite list of ATRs provides students and teacher with an objective reference to determine at any point of time what fraction of the specification has been verified and what fraction of the functionality has been already covered by the tests. Students should verify the correctness of their designs and the correctness of their implementations, and they should analyze and correct the discrepancies.

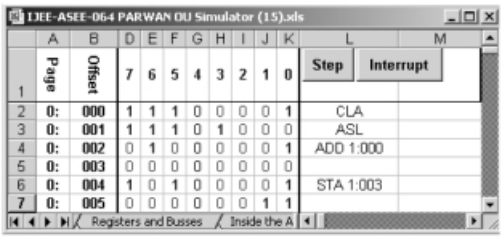

Fig. 10. A section of the 'Memory' worksheet.

A good starting point is comparing the student' implementation in the simulator's state transition table (STT) against the ATRs of standards. For example, they may verify all ATRs' compliance with the ATR of this standard: '*Only one gate may be open on a bus at any point of time*'.

An obvious test is the comparison of the controller's behavior against the student-written ATRs. In this kind of test exactly one STT row in the spreadsheet is tested against the ATR written in the first column of that row.

A test of a different scope verifies the correct execution of each assembly instruction against the ATRs that they wrote for these instructions. The two tests test at two different abstraction levels.

The ease by which the 1s and 0s can be changed in the STT is a fundamental characteristic of spreadsheets. Students should be encouraged to play around with different possibilities. However, when identifying bugs, they should not hurry to change the 1s and 0s in the STT. First they must decide whether the bug has resulted from an incorrect implementation of the said row's ATR, or, maybe the ATR itself is incorrect. Bugs in specifications are more frequent than many people would like to believe [28]. For this reason, and because it precisely zeroes in on the student's misunderstanding, revealing a bug in an ATR has great educational value.

CONCLUSION

Among numerous works in the field of spreadsheet-based simulation of computer architecture, none describe spreadsheet simulations on the level of requirement specifications. We have proposed a way to fill this vacuum.

In our paper, we presented a specific method for the spreadsheet simulation of system requirements (specifications). The method is based on two main ideas:

1. Separating system specifications into a number of levels of abstraction. Only when strictly distinguishing different levels of abstraction students are able to develop a proper requirement simulator (RS).
2. Using the matrix logical simulator [26] within the spreadsheet as the core of RS. This specific matrix logical simulator enables very simple and flexible implementation of the computer controller specifications by the students.

The paper demonstrates an RS implementation for the PARWAN microprocessor [12] widely used for

educational needs. First results of using this method in the undergraduate computer architecture class are promising.

The newly proposed simulation raises new questions, which are as follows.

- Does the spreadsheet provide an appropriate environment for simulating requirements for a computer system?
- Does this simulation allow verification of the requirements more easily than by using a regular VHDL-based simulation?
- Are there any advantages in performing the spreadsheet simulation of the system requirements before the VHDL-based simulation?

These questions constitute our current research agenda.

REFERENCES

1. H. Diab and I. Demashkieh, A computer-aided teaching package for microprocessor systems education, *IEEE Trans. Educ.,* **34**(2), 1991, pp. 179–183.
2. G. A. Wainer, S. Daicz, L. F. de Simoni and D. Wasserman, Using the ALFA-1 simulated processor for educational purposes, *ACM J. Educational Resources in Computing*, **1**(4), 2001, pp. 111–151.
3. C. Yehezkel, W. Yurcik, M. Pearson and D. Armstrong, Three simulator tools for teaching computer architecture: EasyCPU, Little Man Computer, and RTLSim, *ACM J. Educational Resources in Computing*, **1**(4), 2001, pp. 60–80.
4. C. Yehezkel, A taxonomy of computer architecture visualizations, *ITiCSE'02*, Aarhus, Denmark, pp. 175–177.
5. R. Yen and Y. Kim, Development and implementation of an educational simulator software package for a specific microprogramming architecture, *IEEE Trans. Educ*., **29**(1), 1986.
6. M. R. Smith, A microprogrammable microprocessor simulator and development system, *IEEE Trans. Educ*., **27**, 1984, pp. 93–100.
7. M. Cutler and R. Eckert, A microprogrammed computer simulator, *IEEE Trans. Educ*., **30**(3), 1987, pp. 135–141.
8. P. H. Saul, The spreadsheet as a high level mixed mode simulation tool, *Electronic Engineering*, 1991, pp. 549–555.
9. A. El-Hajj, K.Y. Kabalan and H. Diab, A spreadsheet educational tool for microprocessor systems, *Computer Applications in Engineering Education*, **3**(3), 1995, pp. 205–211.
10. A. El-Hajj, K. Y. Kabalan, M. Mneimneh and F. Karablieh, Microprocessor simulation and program assembling using spreadsheets, *Simulation*, **75**(2), 2000, pp. 82–90.
11. I. Levin, Behavioral simulation of an arithmetic unit using the spreadsheet, *Int. J. Electrical Eng. Educ.*, **31**, 1994, pp. 334–341.
12. Z. Navabi*, VHDL: Analysis and Modeling of Digital Systems,* 2nd ed., McGraw-Hill Professional (1997).
13. P. Zave, Classification of research efforts in requirements engineering, *ACM Computing Surveys*, **29**(4), 1997, pp. 315–321.
14. A. M. Davis and A. M. Hickey, Requirements researchers: do we practice what we preach? *Requirements Engineering*, **7**(2), 2002, pp. 107–111.
15. D. L. Parnas, Software aspects of strategic defense systems, *Communications of the ACM*, **28**(12), 1985, pp. 1326–1335.
16. N. Nisan and S. Schocken, *The Elements of Computing Systems*, MIT Press (2004). Draft is available at: http://www1.idc.ac.il/csd/book/index.htm
17. R. Harwell, E. Aslaksen, I. Hooks, R. Mengot and K. Ptack, What is a requirement? *Proc. Third Int. Symp. NCOSE INCOSE*, (1993).
18. H. Kilov and J. Ross, *Information Modeling: An Object-oriented Approach*, Prentice-Hall (1994), pp. 28–32.
19. C. Ghezzi, M. Jazayeri and M. Mandrioli, *Fundamentals of Software Engineering*, 2nd Ed., Prentice-Hall, Upper Saddle River, New Jersey (2003).
20. H. Salzer, ATRs (atomic requirements) used throughout development lifecycle, *12th Int. Software Quality Week (QW99)*, 1(6S1), San Jose, CA (1999).
21. IEEE Std 1233, 1998 edition, Guide for developing system requirements specifications, *IEEE Standards, Software Engineering, Vol. One, Customer and Terminology Standards,* IEEE, Computer Society (1998).
22. H. Kaindl, S. Brinkkemper, J. A. Bubenko Jr., B. Farbey, S. J. Greenspan, C. L. Heitmeyer, J. C. S. Leite, N. R. Mead, J. Mylopoulos and J. Siddiqi, Requirements engineering and technology transfer: obstacles, incentives and improvement agenda, *Requirements Engineering*, **7**(3), 2002, pp. 113–123.
23. M. Archer, C. Heitmeyer and E. Riccobene, Proving invariants of I/O automata with TAME, *Automated Software Engineering*, **9**, 2002, pp. 201–232.
24. D. L. Parnas, Information distribution aspects of design methodology, *Proc. 1971 IFIP Congress*, pp. 339–344.
25. D. Harel, On visual formalism, in *Diagrammatic Reasoning: Cognitive and Computational Perspectives,* J. Glasgow, N. H. Narayanan and B. Chandrasekaran (eds.), AAAI Press, Menlo Park, CA (1995).
26. I. Levin, Matrix model of logical simulator within spreadsheet, *Int. J. Elect. Enging. Educ.,* **30**(3), 1993, pp. 216–223.
27. http://www.tau.ac.il/~salzerha/mypubs/CUsimulator.xls.
28. B. W. Boehm, R. K. McClean and D. B. Urfrig, Some experience with automated aids to the design of large-scale reliable software, *IEEE Trans. Software Engineering* **1**(1), 1975, pp. 125–133.

Ilya Levin received the M.Sc. Degree in Electrical Engineering (Cum Laude) in Leningrad Transport Engineering University and Ph.D. degree in Computer Engineering from the Latvian Academy of Science in 1976 and 1987 respectively. During 1985–1990 he was the Head of the Computer Science Department in the Leningrad Institute of New Technologies (Russia). During 1993–1996 he was the Head of the Computer Systems Department of the Center for Technological Education, Holon (Israel). Being presently a faculty of the School of Education of Tel Aviv University, he is a supervisor of Engineering Education program. He is an author of more then 50 papers both in Design Automation and in Engineering Education fields.

Hanania Salzer is a Ph.D. student at the School of Education in the Tel-Aviv University. For the last 20 years he worked in the software industry. He earned his M.Sc. in Zoology and B.Sc. in Biology at the Tel-Aviv University.