ATLAS – Advanced Tools and Languages for Microprocessor Architecture Simulation

Danko Basch and Mario Žagar

Elektrotehnički fakultet, Zagreb, Croatia

In this paper the digital system simulation program package ATLAS (Advanced Tools and Languages for microprocessor Architecture Simulation) is described. Different software modules, parts of ATLAS are developed: COMPAS – COnfigurable MicroProcessor Architecture Simulator, CONAS – CONfigurable ASsembler, ADEL – Assembler DEscription Language, COMDEL – COMponent DEscription Language and SYSDEL – SYStem DEscription Language. The simulator COMPAS and its implementation are explained in more details. The digital component description language features are given.

1. Introduction

The process of designing digital components and systems always has the same goal - achieving good characteristics of a system irrespective of its intended use. It means: high speed, high accuracy, high efficiency, high reliability, low price etc. The designer's scope of interest is limited to some of these characteristics, influenced by technology, architecture of components and architecture of the system. The designer can only choose one of the existing technologies, but the designer's knowledge and skills will have great impact concerning the architecture. For success on market, it is not enough to offer a good product, but also to offer it before other manufacturers. Different tools are used by designers to decrease design time and expenses.

The great complexity of digital systems and the behaviour dependent on events in its environment make behaviour prediction almost impossible. Analytical solution of these problems can be far too complex and, on the other hand, prototyping is expensive and inflexible. A widely accepted and compromise method is simulation. Simulation allows great modelling flexibility and later changes are easy to make. An experiment may be repeated numbers of times with the same or different parameters. Accuracy depends on how detailed model is made and, naturally, on the possibilities of the simulator itself. The main disadvantage is low simulation speed.

Many digital system architecture simulators have been developed. The architecture is described through special hardware description languages (HDL) which are also developed in great number (AYLOR et al. 86). Except for simulation, HDLs are used for other purposes: documentation, fault simulation, synthesis etc. HDLs are adapted for different purposes and their advantages and disadvantages descend from this fact. For example, ISPS supports a wide range of applications (BARBACCI 81), VHDL is simulation-oriented (LIPSETT et al. 86) etc. Some general purpose languages like Flat Concurrent Prolog (DOTAN et al. 90) can be used as HDLs. Today, VHDL is imposing itself as a standard (LIPSETT et al. 86).

The hardware can be described in two opposite ways: the structural description and the behavioral description (AYLOR et al. 86). Different HDLs can support only one approach or both of them or all levels between them. The scope of hardware can also be described on different levels — from the transistor level to the system-level. Many HDLs often allow a simultaneous description on different levels.

The main requirements on simulators are:
– simple usage from the user's point of view,
– great flexibility in hardware description,
– high simulation speed.

Simple usage means that the HDL should be readable, understandable etc. The simulator should have a user-friendly interface (windows, icons, graphical result representation etc.).

Flexibility is mostly dependent on the HDL and means that different levels of description should be allowed, together with the precise timing details, the user-defined data types etc. (AYLOR et al. 86) (KOSMAN et al. 85). Additionally, the language must enable new approaches in hardware design.

Simulation speed is very important, especially today, when hardware complexity constantly grows. Besides it is usually necessary to perform the simulation several times in order to balance different values in the system so that better performance is achieved.

Unfortunately, writing simulators is trade-off. For example, if we want a detailed description it is not possible to have a high simulation speed, nor a simple HDL.

One of the problems in the processor design is the selection of instruction set, registers and communication protocol of that processor. In designing other kinds of components, or the whole system, different elements will be relevant. Since the processor is the central and the most important component in the system, it must be designed very carefully.

Our intention was to concentrate on the processor simulation. Because processor is the most complex component, other components can be simulated automatically. The simulator is imagined as an auxiliary tool for designing digital system and its components, especially in the first stages of development and logical model testing. Performance measurement and system study are also supported. Designing special purpose processors is very specific. The problem solving algorithm must be effectively implemented in the microprocessor's architecture. This leads to unconventional solutions suitable for only one problem. Simulation could be of great help in that kind of design.

The conception and design of the simulator, together with the definition of the HDL were guided by our requirements and by the class of problems that we want to solve.

Impractical hardware description on the gate-level, due to the growing hardware complexity, was noticed long ago (ARMSTRONG et al. 80). Regardless of that fact, almost all HDLs allow the gate-level description. That is, of course, justified, because the internal fault simulation or the hardware synthesis demand the gate description level. For processor behaviour simulation, and that was our need, the description level is set to the definition of registers, pins and component's behaviour. The universal conception of connecting the components by buses is chosen as the basic model of the digital system.

Timing description and signal modelling do not need to be precise, like in the gate-level simulation (D'ABREU 85). In the extreme case, that kind of description is not necessary.

Two main methods are considered in the development of the simulator: the compiler-driven and the event-driven method. The latter is in use in more recent simulators. Arguments pro and con those methods are usually given for the gate-level simulation, because both methods are developed for it. For the processor-level simulation many of these arguments are not applicable. The main advantage of the event-driven technique is the evaluation of only active elements. In the gate-level model, the activity is about 10-15 percent (D'ABREU 85). The elements in the processor-level model have a different nature. Although activity could be the same as in the gate-level model, the inactive processor-level elements consume less time than active ones. The compiler-driven methods use very simple and fast simulation engine because fixed delay or zero delay is assumed. Regression in speed descends from evaluating all the elements in the model. The other deficiency is unsuitable modelling for the asynchronous design.

Following our requirements we have developed a method which contains some new elements together with the elements of both compiler-driven and event-driven technique. Similarly to compiler-driven simulation, all the components in the system are evaluated. For zero delay case, an operation similar to scheduling in event-driven simulation is performed. Inactive components enter the so-called wait state, and become active again when particular event on the bus occurs, or after specific time interval elapses.

One of the most important requirements on simulators, and its bottleneck, is the simulation speed. The speed could be increased in different ways — through software by different implementation of algorithms or by using better ones, or through hardware by using faster machines. Some authors have proposed that holding by some rules during the modelling could significantly contribute to the simulation speed (MICZO et al. 87). Another authors have proposed compiling techniques which will produce an architecture-specific simulator, based upon the HDL description. A step beyond is the inclusion of the program (which we want to execute on the model) in the compiling process. This results in not only an architecture- specific but also a program-specific simulator (SIEGELL et al. 87). For really complex systems, today, the only acceptable solution lies in the use of special purpose computers called hardware accelerators. They could achieve speed-up improvements of about 10,000 times over software simulators. In our simulator, a part of speed-up is accomplished by a chosen level of description which reduces the number of components for the description of the whole system from several millions to just a few of them. The speed-up, naturally, does not have the same proportion, because the components in the processor-level model are much more complicated than in the gate-level model.

ATLAS – Advanced Tools and Languages for microprocessor Architecture Simulation

The program package ATLAS is developed, with concern to the requests stated in the previous section. ATLAS is designed as a set of software tools, mainly oriented toward the simulation (see Fig. 1). ATLAS is developed in the C programming language on the UNIX operating system. The simulator COMPAS and the assembler CONAS are developed as ATLAS's tools. In the future, we plan to add some new tools to ATLAS (e.g. performance analysing tools).

CONAS (CONfigurable ASsembler) will be described briefly in this passage. CONAS is a meta-assembler and its configuration is defined by a special file written in ADEL (Assembler DEscription Language). ADEL file contains a mnemonic language description, together with the description of its translation in the machine code. Inputs to CONAS are ADEL file and source code which must be written according to the rules set in the ADEL file. Outputs from CONAS are print version of the source code and the absolute machine code. Machine code may be directly loaded into the COMPAS simulator.

COMPAS – COnfigurable MicroProcessor Architecture Simulator

COMPAS is digital system simulator on the processor-level, the memory level and other high complex components' levels. As it is mentioned before, the conception of connecting components by buses has been chosen. Each type of component is described with the COMDEL file (COMponent DEscription Language). COMDEL is a high level procedural language without concurrent facilities. The concurrent execution is achieved on the system level. The system is described with the SYSDEL file (SYStem Description Language). SYSDEL describes the connections between the components in form of a net-list. This is the only part of the description which contains structural data. The components are interconnected with the buses transferring data or signals.

There exist two versions of ATLAS – one for simple text-terminals and another for X Window System. Both versions have the same basic facilities, but due to the different working environment each version has some specific abilities. For example, in the text version it is possible to use both COMPAS and UNIX commands, and the other tools are executed like UNIX commands. So, operating with the simulator is similar to working with the UNIX shell. In the X11 version it is possible to run several ATLAS tools at the same time. Also, most of the COMPAS commands from the text version are still available, except those that are in contradiction with the GUI. Of course, in the X11 version all commands are intended to be given through menus, pop-up windows, buttons etc. X11 version has some additional possibilities (related to displaying of the model state during simulation) that are impossible in the text version.

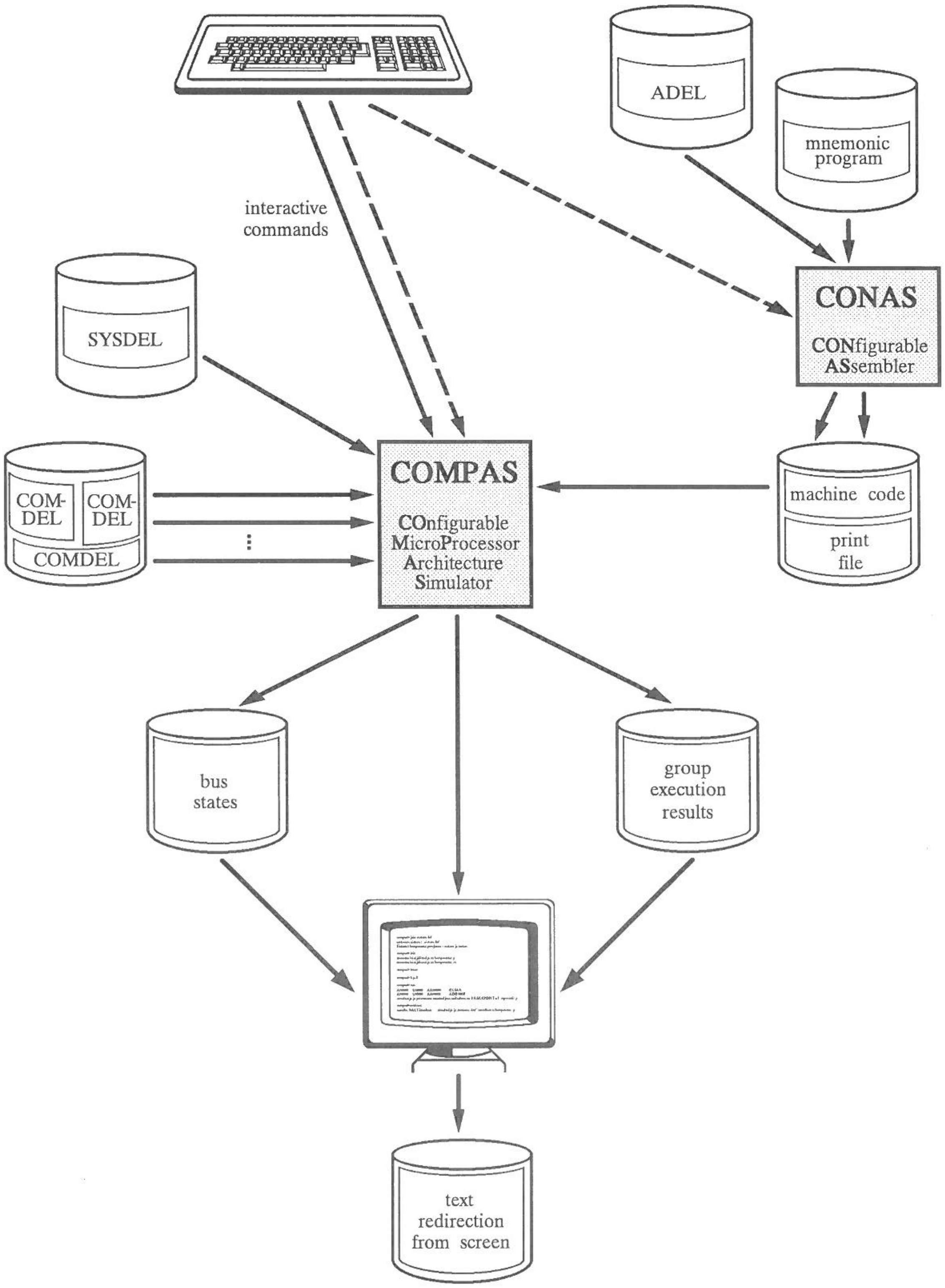


Fig. 1. Program package ATLAS

COMPAS is divided in the shell, used for communication with the user, COMDEL and SYSDEL analyzers, and the simulator itself (Fig. 2). The simulator consists of the simulation engine and the subroutines for the execution of the COMDEL statements. In fact, the simulation is performed by the interpretation of the COMDEL statements. Since the COMDEL statements describe the component behaviour, it would be necessary to analyze each statement before its interpretation. This procedure could reduce simulation speed, since the COMDEL statement level is very high. To avoid such problems during COMDEL analyses, each COMDEL statement is translated into one or more low level instructions (LLI). Therefore, COMDEL analysis is performed only once — when the model description is being loaded in the simulator. LLIs perform simple actions like take/put value from some location, condition evaluation, unconditional jump etc. LLIs are adapted for simulation engine so that the interpretation of LLIs can be direct and fast.

Although in a real system all its parts work in parallel, the simulation is performed on a uniprocessor computer. For that reason, all actions defined as concurrent are executed sequentially. To simulate parallelism, we make time discretisation: the smallest time unit is a quarter of clock period. We have called this time unit bus cycle (BC). At first sight, such discretisation seems to be insufficient, but in practice, for given description level, different commercial microprocessors are described with great accuracy.

The simulation engine takes care of pseudo-parallel simulation. Each component has its own behavioral description as the array of LLIs. The pseudo-parallel simulation is accomplished through the interpretation of LLIs for every component in every BC. The order of interpretation inside BC is expected to be random, theoretically. For convenience and higher speed this order is fixed and identical with the order of declaration of the particular components in the SYSDEL file.

The interpretation of one BC is performed in a number of steps:

1) All LLIs for all components are interpreted for the current BC.

2) Only the components in so-called wait state are evaluated in the second step. The component enters the wait state by executing COMDEL's wait statement equivalent in LLI. The component leaves the wait state when condition (e.g. change on bus etc.) connected to the COMDEL's wait statement is fulfilled. The second step determines which components will stay in the wait state, and which will leave it. For the components that leave the wait state, the remaining LLIs (if any) for the current BC are interpreted.

3) In the last step different flags are inspected. These flags signal the occurrence of error, the simulation interrupt request etc. The data in the bus descriptors are updated as well as the data connected with the statistic collection and time measurement. After this step the whole procedure (from step 1 to 3) is repeated for the next BC.

2. Data organization

All data about components, registers, variables, pins, buses, and other data necessary for the simulation are stored in separate data structures. Because the number and values of these data are not known in advance, all data are stored in linked lists. The elements of the lists are descriptors and different information (e.g. name, state, width in bits etc.) are stored in it. Each component, register etc. has its own descriptor. During the simulation, data fetching from lists is not performed by using any kind of list search. Because data fetch is a very frequent operation, lot of simulation time would be spent on searching. To avoid that, pointers are used so that data could be fetched directly. The values of pointers (addresses) are determined during the COMDEL and SYSDEL analyses.

COMDEL – COMponent DEscription Language

Each type of component that we use in the system (the model) must be described with a COMDEL file. COMDEL file is composed of several blocks: definition of registers, definition of variables, definition of pins, initialization procedure, user-defined procedures and run procedure. Formal syntax of COMDEL is given in the APPENDIX.

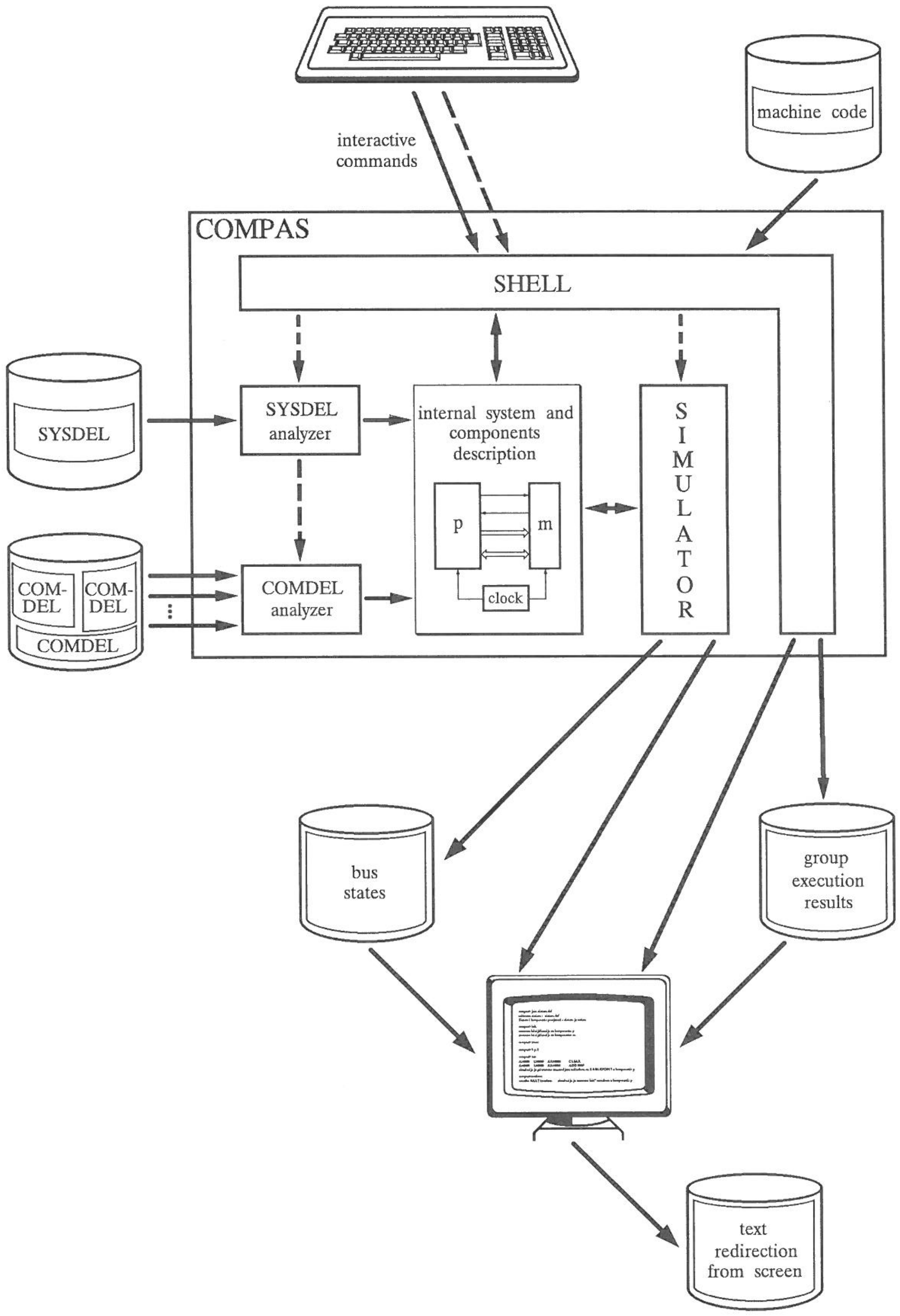


Fig. 2. COMPAS simulator

The register and variable definition blocks have the similar meaning as the definition of global variables in general purpose programming languages. The registers are intended for the description of data containers and buffers in a component, e.g. user-accessible registers of the processor, memory locations etc. The auxiliary locations like the internal registers, the temporary containers, the counters and the other locations that are used just for the description of a component rather than being its part, should be defined as variables. The COMDEL segment from example 1 shows possible register and variable definition blocks for some hypothetic processor.

The pin definition block is used for the component interface definition. The pins are used in the SYSDEL file to connect the components on buses. In the example 2 pin definition for the hypothetic processor from the first example is illustrated.

Previously described blocks are declarative and others are executable. The component initialization is described in the initialization procedure (the so-called init block). The init block will be executed when we give the init command from the COMPAS shell. The actions defined in the init block can be viewed as the component behaviour in the moment when power is connected to it. Typical init actions are: putting initial values in registers and variables, disconnecting or taking control over buses etc. The init block for the component from first two examples is shown in example 3.

The run procedure is the main block in COMDEL. When run command is given from COMPAS shell, simulation begins via interpretation of run

```
registers {

/***  Register names ADR, DATA, PC ... are arbitrary  ***/

    ADR   [ 4 ] /.16./,         // array of four 16-bit address registers
    DATA  [ 8 ] /.32./,         // array of eight 32-bit data registers
    PC /.16./,                  // 16-bit program counter
    SP    /.16./,               // 16-bit stack pointer
    STAT  /. 7./;               // 7-bit status register (flags)
}

variables {
    AR /.16./, DR /.32./,       // internal address (AR), data (DR)
    IR /.32./,                  // and instruction (IR) register
    counter_1  /.16./,          // auxiliary counter
    temporary  /.32./;          // temporary storage
}
```

Example 1. Register and variable blocks for hypothetic processor

```
pins {
/***  "Data" buses   ***/
    D  /.32./,                   // 32-bit data bus
    A  /.16./,                   // 16-bit address bus
/***  Signal lines (buses)  ***/
    AS,                          // address strobe
    DTACK,                       // data transfer acknowledge
    INT,                         // interrupt request
    INTACK;                      // interrupt acknowledge
}
```

Example 2. Pin block for hypothetic processor

```
init {
//***** Put 0 in DATA & ADR register arrays.
    let counter_1 = 0;
    while ( counter_1 !== 8 ) {
        let DATA [ counter_1 ] = 0;
        let ADR [ counter_1 /.0..1./ ] = 0;
        inc ( counter_1, counter_1 );
    }
//***** Initialize PC, SP & STAT
    let PC   = 0;
    let SP   = 65535;
    let STAT = 0;

//***** Put buses D, A, DTACK & INT in HIGH IMPEDANCE
    disable  D,A,DTACK,INT;
//***** Put buses AS & INTACK in the inactive (low or 0) state
    let AS = 0;
    let INTACK = 0;
}
```

Example 3. Possible init block for hypothetic processor

blocks in all components present in the system. Hence, statements in run block describe the component behaviour. For the component description, besides run block, user-defined procedures can be used. The procedure execution can be called from the run block or other procedures. The example 4 describes the typical run block of some processor.

Procedures have the same form as init or run block. The example for a procedure will be the description of a memory read bus protocol for the hypothetical processor from the previous examples. A simplified bus protocol is described in figure 3.

The processor controls the buses A and AS and the memory controls DTACK and D. All signals are active in the high level. The normal memory read cycle takes two clock periods (T0 and T1). For slow memories, wait states (Twait) may be inserted between T0 and T1. The buses are approached via the pins connected to them. In this example the pins and the buses are given the same names for convenience and simplicity. On the clock's falling edge the address is put

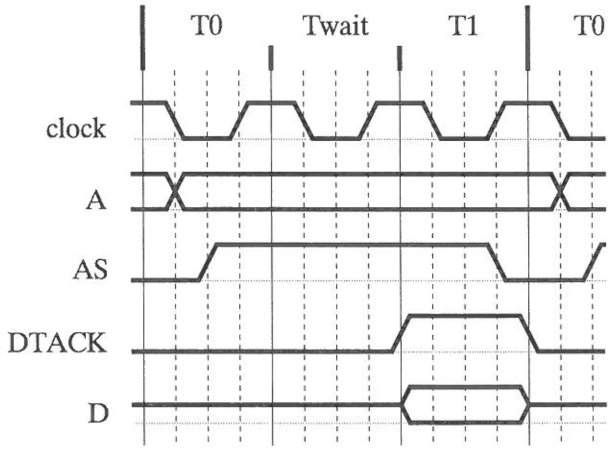


Fig. 3. Bus protocol for example 5

```
run {
   forever {
      call fetch_instr;  // call subroutine for instruction fetching
      call exec_instr;   // call subroutine for instruction executing
      call chk_int;      // call subroutine for interrupt checking
   }
}
```

Example 4. Typical run block for hypothetic processor on address bus A from internal address register AR. One BC later, the processor signalises to memory that the address on A is stable. This is accomplished by putting the signal AS in high level. The memory's task is to put the data on the data bus D and to signalize the valid data on D with the DTACK signal. The processor checks the DTACK signal on every next falling clock edge. If DTACK is inactive (low) Twait states are inserted. Otherwise, the current clock period is T1 and the data from D bus are loaded in the internal data register DR. On clock's rising edge in T1, the signal AS is deactivated and the next memory read/write cycle can start. The procedure for the described memory read bus protocol is given in the example 5.

About 80 statements and arithmetical-logical functions (ALF) are available for use in executable blocks. COMDEL has usual control flow statements (if, if-else, switch, while etc.) which allow a great freedom in component behaviour description. Another type of statements are the synchronization statements. They are used for internal and external synchronization of components. Some special statements connected with the simulation flow control are implemented, and also the statements used for trace mod execution, using breakpoints and collecting statistical data.

ALFs simulate the behaviour of ALU. Arithmetical functions (e.g. add, sub, neg, dec, inc etc.), logical functions (e.g. and, or, xor, not etc.) and shifting and data testing (test and parity) functions are implemented. If we want so, ALF can have influence on predefined flags (e.g. carry, overflow, sign etc.). These flags are defined in each component and can be used for describing the user-defined flags.

SYSDEL – SYStem DEscription Language

After describing the components with COMDEL files, we can describe the system with the SYSDEL file. In SYSDEL we can determine the clock frequency. At least one component must be declared and name is associated with each one of them. Each component has also its type which is determined with the COMDEL file name. The last part of SYSDEL is used for bus definition and for connecting components on buses. Because the SYSDEL file is relatively simple, the complete example is given (example 6). Formal syntax of SYSDEL is given in APPENDIX.

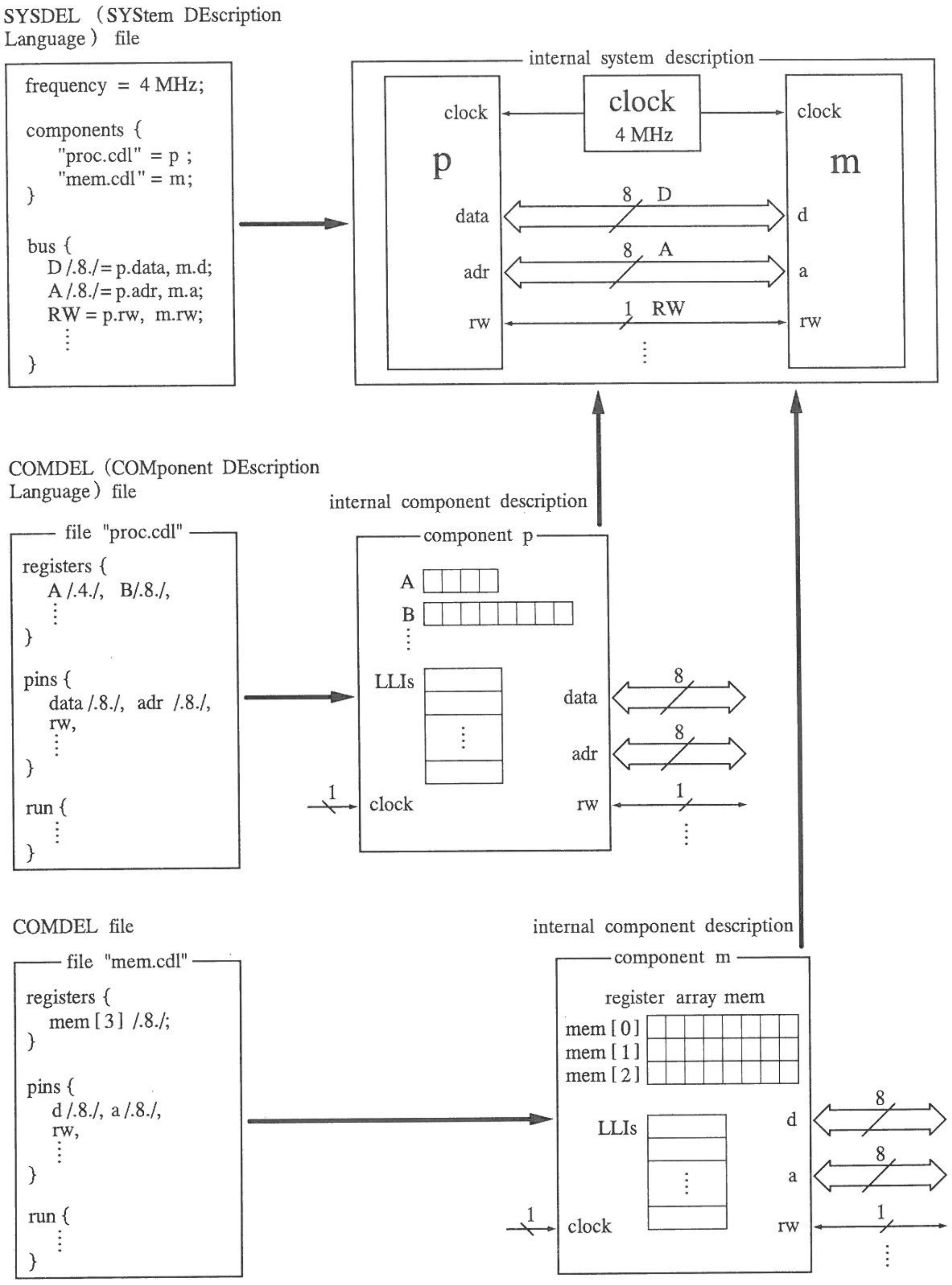
Suppose that the hypothetical processor from previous examples is described in the COMDEL file "processor.cdl" and that there are two more files describing the memory unit and the i/o unit. In SYSDEL we refer to COMDEL files in order to determine the type for each component and the instantiation of each component is accomplished by defining its name (p, io, m1 and m2 in this example). In bus definition we give the name and the net-list to each particular bus in the system. Bus names (A, D, DTACK, int, iack) and widths (/.32./ and /.16./) definitions are the same as pins definitions in COMDEL. The net-list is given in the form "component_name . pin_name". For example, on the signal bus iack,

```
read_mem {
    on ( fall( clock ) );      // wait for falling edge of clock
        let A = AR;
    on ( low( clock ) );       // wait for stable low level of clock
        let AS = 1;

// on every falling clock edge check if DTACK is active (high level)
    wait ( fall( clock )  and  DTACK == 1 );

    on ( low( clock ) );       // wait for stable low level of clock
        let DR = D;            // read data from D bus into DR
    on ( rise( clock ) );      // wait for rising edge of clock
        let AS = 0;
}
```

Example 5. Procedure for memory read bus protocol



SYSDEL (SYStem DEscription Language) file

```
frequency = 4 MHz;

components {
    "proc.cdl" = p ;
    "mem.cdl" = m;
}

bus {
  D /.8./= p.data, m.d;
  A /.8./= p.adr, m.a;
  RW = p.rw,  m.rw;
    ⋮
}
```

COMDEL (COMponent DEscription Language) file

file "proc.cdl"

```
registers {
   A /.4./,  B/.8./,
   ⋮
}

pins {
   data /.8./,  adr /.8./,
   rw,
   ⋮
}

run {
   ⋮
}
```

COMDEL file

file "mem.cdl"

```
registers {
   mem [ 3 ] /.8./;
}

pins {
   d /.8./, a /.8./,
   rw,
   ⋮
}

run {
   ⋮
}
```

Fig. 4. Connection between SYSDEL and COMDEL

```
frequency = 50 MHz;

components {
   "processor.cdl" =  p;       // Definition of component named p.
                               // It's type is described in COMDEL
                               // file "processor.cdl"

   "i_o_unit.cdl"  = io;
   "memory.cdl"    = m1, m2;   // Definition of components m1 and m2.
                               // Both of them have the same type.
}

bus {
   D /.32./ = p.D, m1.DATA, m2.DATA, io.d;
   A /.16./ = p.A, m1.ADR,  m2.ADR,  io.a;

   adrstrb = p.AS,     m1.ADRSTR,     m2.ADRSTR,     io.as;
   DTACK   = p.DTACK,  m1.DATA_VALID, m2.DATA_VALID, io.dtack;
   int     = p.INT,                                  io.int;
   iack    = p.INTACK,                               io.intack;
}
```

Example 6. SYSDEL file for simple digital system.

p component's pin INTACK and io component's pin intack are connected.

Two rules must be respected in net list definition:

1) The pin and the bus which are connected together must have the same type (either data or signal) and the same width in bits.

2) Each pin must be connected to exactly one bus.

The meaning of SYSDEL and COMDEL files and their internal representation in the simulator is shown in the figure 4.

3. Different levels and possibilities in component and system description

Before modelling we must decide how precisely and accurately model behaviour should represent the real (or hypothetical) system. We must also decide which aspect of the system behaviour is relevant to our requirements. For example, if we want to simulate a processor on the machine code execution level, it is not necessary to describe DMA and interrupt response. For a more accurate description, more effort in creating COMDEL files should be invested and the simulation speed will decrease. Sometimes it is possible to avoid the use of buses and more than one component. The normal approach is to define several components connected by a bus, but nothing can stop us to define only one component without pins. It is also possible to define several autonomous components and leave them unconnected. Whole computer behaviour may be defined on only one component (e.g. microprocessor, memory, i/o units etc.) and particular circuits (e.g. ALU, register, control unit etc.) as well.

The basic limits are the result of the chosen high description level. This level disables efficient description on lower levels (gate-level), accurate timing description (rise and fall times etc.). Another limitation is the maximal width of registers, variables, pins and buses set currently on 32 bits, because of the state of the available computers today. This will be changed in the future. However, in the moment this limitation can be solved, e.g. by defining two 20-bit registers instead of one 40-bit register.

The number of components, buses, registers, etc. is not restricted because the data about them are stored in lists whose elements are dynamically allocated during COMDEL and SYSDEL analysis. Therefore, these numbers are limited only by the computer memory. The same kind of limitation is related to the size of register arrays. A translation of one COMDEL procedure into LLI cannot exceed 32 K due to the use of 16 bit relative jumps inside one procedure in LLIs. This limits the COMDEL procedure size to approximately 16 K. For modelling component behaviour this is not a real limit because the number of procedures in COMDEL file is not restricted.

During COMPAS implementation we tried to get simulation speed as high as possible, although the interpretation is a limiting factor. A part of speed-up is gained by the high description level. Additional speed-up is accomplished by using three slightly different simulation engines. Which one will perform the simulation is automatically decided according to the requested simulation type. The bad influence of interpretation is reduced with partial analysis, translation and optimization of COMDEL statements. Besides, pointers are intensively used instead of searching or indexed table data access. The simulator performance measurement for simple system (Z80, memory, i/o unit, 4 MHz, no interrupt requests) gives simulation ratio of approximately 500 to 1. The simulation was run on SUN SPARC station 2 computer and measurement was made by using UNIX's time facility. Another important times — simulator initialization, loading system description, model initialization — for such simple system could be neglected (about 1 second each).

4. Conclusion

ATLAS fulfils its main purpose — to serve as a tool in the first stages of component and system design. Although great freedom in model description is given, limitations stem from the high description level. Because of the great complexity in today's components structures, description is almost purely behavioral. Besides its main purpose, there is another one. ATLAS is used in education, hence simulation is very practical for introducing students to basic microprocessor architectures. Principles of microprocessor behaviour can be easily explained on a simplified microprocessor model prepared by teaching staff.

ATLAS should not be viewed as the final product. Different improvements are possible, especially in COMDEL and SYSDEL, simulation speed etc. Therefore, the simulator COMPAS and the whole program package ATLAS are the first implementation and a good foundation for further development and extensions.

References

J. R. ARMSTRONG, G. W. WOODRUFF Chip-Level Simulation of Microprocessors, *IEEE Computer Magazine*, **13** (1) (1980), 94–100.

J. H. AYLOR, R. WAXMAN, C. SCARRATT VHDL-Feature Description and Analysis, *IEEE Design & Test of Computers*, **3** (2) (1986), 17–27.

M. R. BARBACCI Instruction Set Processor Specifications (ISPS): The Notation and Its Applications, *IEEE Transaction on Computers*, **30** (1) (1981), 24–40.

T. BLANK A Survey of Hardware Accelerators Used in Computer-Aided Design, *IEEE Design & Test of Computers*, **1** (3) (1984), 21–39.

M. A. D'ABREU Gate-Level Simulation, *IEEE Design & Test of Computers*, **2** (6) (1985), 63–71.

Y. DOTAN, B. ARAZI Concurrent Logic Programming as a Hardware Description Tool, *IEEE Transaction on Computers*, **39** (1) (1990), 72–88.

R. KOSMAN, A. SAVKAR, R. HARR, L. SUEN Interim Report on a Standard Language for System Design, *IEEE Design & Test of Computers*, **2** (5) (1985), 120–122.

R. LIPSETT, E. MARSCHNER, M. SHAHDAD VHDL-The Language, *IEEE Design & Test of Computers*, **3** (2) (1986), 28–41.

A. MICZO, D. MOHAPATRA, S. PERKINS, K. KAUFMAN, K. HUANG The Effects of Modelling on Simulator Performance, *IEEE Design & Test of Computers*, **4** (2) (1987), 46–54.

B. SIEGELL, T. GROSS Program-specific and Architecture-specific Simulators, 29–45, from *Computer Hardware Description Languages and their Applications*, edited by M. R. Barbacci and C. J. Koomen, Elseiver Science Publishers B.V. (North-Holland), 1987.

Appendix

Formal syntax of COMDEL and SYSDEL is given in BNF. Braces are elements of the languages and also of the BNF notation. To avoid ambiguity we enclosed braces in double quotes ("" and "") when they are a part of COMDEL and SYSDEL. Because of the limited space, syntax of COMDEL is not complete — some minor details are omitted. In the syntax description we use following symbols:

LETTER – any letter (capital or small) of the English alphabet or underscore _.

DIGIT – any digit from 0 to 9, including capital and small letters a, b,... f and x. Letter x means "don't care" in xdecode and xswitch statements.

CHAR – any printable character from ASCII set, except double quote ".

```
<empty> ::=

<name> ::= LETTER  { LETTER | DIGIT }

<quantity> ::= DIGIT { DIGIT }
<number> ::= <base> <sign> <quantity>
<base> ::= <empty> | % <base identifier>
<base identifier> ::= b | B | o | O | d | D | h | H | x | X
<sign> ::= <empty> | + | -

<string> ::= " { CHAR } "



<COMDEL program> ::= <registers block> <variables block> <pins block>
                     <init block> { <procedure block> } <run block>

<registers block> ::= registers "{" <registers definition> "}"
<variables block> ::= variables "{" <registers definition> "}"
<registers definition> ::= <register> { , <register> } ;
<register> ::= <name> <array size> / . <quantity>
<array size> ::= <empty>  |  [ <quantity> ]

<pins block> ::= pins "{" <pins definition> "}"

<pins definition> ::= <pin> { , <pin> } ;
<pin> ::= <name>  |  <name> /. <quantity> . /

<init block> ::= <empty>  |  init <statement block>
<procedure block> ::= <name> <statement block>
<run block> ::= run <statement block>

<statement block> ::= "{"  { <statement> }  "}"
<statement> ::= <statement block> | <assignment> | <decision> |
                <loop> | <simulation control> | <io> |
                <synchronization> | <al function> |
                call <name>; | return;

<assignment> ::= let <location> = <value> ;
<assignment> ::= letf <location> = <value> ;
<assignment> ::= swap <location> , <location> ;
```

```
<assignment> ::= disable ;  |  disable <name> { , <name> } ;

<decision> ::= if ( <condition> ) <statement> <else>
<else> ::= <empty>  |  else <statement>
<decision> ::= <multi-branch> ( <value> )
                "{"  { <case> } <default>  "}"
<multi-branch> ::= switch | xswitch | decode | xdecode
<case> ::=  <number> : <statement>
<default> ::= <empty> | default : <statement>

<loop> ::= while ( <condition> ) <statement>
<loop> ::= do <statement> while ( <condition> ) ;
<loop> ::= forever <statement>

<simulation control> ::= shell ;
<simulation control> ::= exit ;
<simulation control> ::= brkpt <value> ;
<simulation control> ::= count ;
<simulation control> ::= timer on ;  |  timer off ;
<simulation control> ::= trace . <trace number> ;
<trace number> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8

<io> ::= waitkey ;
<io> ::= read ( <base> <location> ) ;
<read base> ::= b | B | o | O | d | D | h | H
<io> ::= print ( <print data> { , <print data> } ) ;
<print data> ::= <string>  |  /  |  % <quantity> . <print base> <value>
<print base> ::= b | B | o | O | u | U | s | S |
                 h | H | x | X | c | C

<synchronization> ::= on ( <signal operator> ( clock ) );
<synchronization> ::= wait ( <condition> );
<synchronization> ::= delay <quantity> <delay unit> ;
<delay unit> ::= s | ms | us | ns | ps | bc

<al function> ::= <al 1> ( <value> ) ;
<al function> ::= <al 2> ( <value> , <location> ) ;
<al function> ::= <al 3> ( <value> , <value> , <location> ) ;

<al 1> ::= test | parity

<al 2> ::= inc | incf | dec | decf | neg | negf |
           not | notf

<al 3> ::= add | addf | addc | addcf | sub | subf | subc | subcf|
           and |  andf |  or |  orf |  xor |  xorf |
           nand | nandf | nor | norf | nxor | nxorf |
           shiftl0 | shiftr0 | shiftl0f | shiftr0f |
           shiftl1 | shiftr1 | shiftl1f | shiftr1f |
           shiftlh | shiftrh | shiftlhf | shiftrhf |
           shiftll | shiftrl | shiftllf | shiftrlf |
           shiftlx | shiftrx | shiftlxf | shiftrxf

<location> ::= <name> <offset> <bits>
<offset> ::= <empty>  |  [ <simple value> ]
<bits> ::= / . <simple value> . /
```

```
<bits> ::= / . <simple value> .. <simple value> . /

<value> ::= <sign extension> <simple value>
<sign extension> ::= empty  |  ( signed )  |  ( unsigned )
<simple value> ::= <location> | <number>

<condition> ::= ( <condition> )  |  <expression>  |
                { not } <condition>
<condition> ::= <condition> <logic operator> <condition>

<expression> ::= <signal operator> ( <name> )
<expression> ::= <value> <relational operator> <value>

<logic operator> ::= and | or | xor | nand | nor | nxor
<signal operator> ::= rise | fall | high | low | edge | change | stable
<relational operator> ::= <   |  >   |  <=   |  >=   |  =   |  != |
                          <<  |  >>  |  <<=  |  >>=  |  ==  |  !==

<SYSDEL program> ::= <frequency block> <components block> <bus block>

<frequency block> ::= frequency = <quantity> <frequency unit> ;
<frequency unit> ::=  kHz  |  MHz  |  GHz

<components block> ::= component
                       "{"  <component> ; { <component> ; }  "}"
<component> ::= <string> = <name> { , <name> } ;

<bus block> ::= empty  |  bus  "{"  <bus> ; { <bus> ; }  "}"
<bus> ::= <bus name> = <pin list> ;
<bus name> ::= <name>  |  <name> / . <quantity> . /
<pin list> ::= <name> . <name>  { , <name> . <name> }
```

Received: November, 1992
Accepted: October, 1993

Contact address:
Danko Basch, Mario Žagar
Elektrotehnički fakultet
41 000 Zagreb, Avenija Vukovar 39, Croatia
tel. (+38541) 62 96 17,
fax. (+38541) 62 97 79,
E-mail mario.zagar@rasip.etf.hr

DANKO BASCH was born on 12th October 1967 in Rijeka. He received B.Sc.EE degree from the Faculty of Electrical Engineering (ETF) at the University of Zagreb in 1991. Since 1992 he has been working as a researcher at the ETF. His main interests are in simulation of digital computers and configurable assemblers. He is coauthor of the book *Introduction to microcomputers.* At present, he is working on his M.Sc degree.

MARIO ŽAGAR received B.Sc.EE, M.Sc.CS and Ph.D.CS degrees, all from the University of Zagreb, Faculty of Electrical Engineering (ETF) in 1975, 1978, 1985 respectively. In 1977 he joined ETF where he is currently an Associate Professor. He received British Council fellowship (UMIST – Manchester, 1983) and Fulbright fellowship (UCSB Santa Barbara, 1983/84). His research interests include computer architectures, real-time microcomputer systems, operating systems, open computing environment and design automation. He has published over 80 papers and reports and is author and coauthor of the books *UNIX how to use it* and *Introduction to microcomputers.*