MODELING AND SIMULATION OF A HARD REAL-TIME PROCESSOR

Vlado Glavinić, Stjepan Groš
University of Zagreb
Faculty of Electrical Engineering and Computing
Unska 3, **HR-10000 Zagreb**, Croatia
vlado.glavinic@fer.hr, sgros@zemris.fer.hr

Matjaž Colnarič
University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova 17, **SI-2000 Maribor**, Slovenia
colnaric@uni-mb.si

MODELING AND SIMULATION OF A HARD REAL-TIME PROCESSOR

Summary – Hard real-time systems are increasingly used in various areas of human activity justifying their implementation by means of specialized solutions, mostly of a suitable hardware/software combination. A frequently adopted approach to the realization of the hardware part is based on ASIC, usually a “general purpose” processor which operates according to real-time constraints. In this work the modeling of a processor for the hard real-time domain is described. It is structured as a collection of “task processors” being supervised by another one dedicated to the “kernel” functions. Specifically the behavior of the task processors is modeled using VHDL and subsequently simulated and tested. The paper also addresses the modeling process by determining the detailed requirements on the behavior of the task processor. The outcome of this step influenced the modeling process as the tools used were of restricted functionality and processor behavior enforced a particular decomposition. Because of a restricted VHDL subset available, it was necessary to model the task processor on the lowest level of behavioral abstraction. The task processor has been tested against chosen test programs written in an appropriate assembly language being specially developed for this purpose.

Keywords – hard-real time, task processor, modeling and simulation, VHDL

I. INTRODUCTION

Although the requirements for and the possibilities in the design of microprocessor based process control systems have significantly changed over the last decennia, the approach remained more or less the same since the early 1970s. Typically, there is a CISC microprocessor or a microcontroller receiving signals from, and communicating via computer internal standard peripheral interfaces with the environment.

It is in the very nature of computer control systems that certain actions must be performed within the specified time frames or else the action will fail with more or less severe consequences. In the usual applications this is assured by employing hopefully fast enough control systems and verified by testing of their temporal behaviour.

Common microprocessor architectures are mainly designed to meet the classical objective of maximizing average resource utilization. The application of the same guiding principle in hard real time systems design is based on one of the most common misconceptions pointed to by Stankovic (Stankovic, 1988) that real time systems were equal to fast systems. However, optimization of resource utilization and average performance usually contradicts the real time, viz., worst-case, requirements.

Thus, instead of computer speed, which cannot guarantee that specified timing requirements will be met, a different ultimate objective in designing consistent systems for embedded hard real-time applications was generally accepted:

predictability of temporal behaviour. While, for the systems usually employed in process control, testing of conformance to functional specifications is well established, temporal circumstances, being an equally important design aspect, are seldom verified consistently. Almost never it is proven at design time that such a system will meet its temporal requirements in every situation that it may encounter.

Determinism and predictability of process execution times, however, are unfortunately in conflict with the common design objective for universal microprocessor systems, viz., to optimize average performance. New paradigms are needed for the consistent design of control systems.

Finally, to ensure integrity of safety-critical hard real-time systems, they should be verified for their correctness. Complex architectures are unmanageable by state-of-the-art verification techniques and tools. E.g., exhaustive verification of even the simplest common microprocessor with exceptions and/or pipelining presents NP-complete problem. Hence, the design must be kept simple in order to allow for its verification.

Whereas academic research on the design of hard real-time systems mainly concentrates on issues such as scheduling, hardware architectures are widely neglected. They are taken for granted, and usually assumed to be operating deterministically. Further, hardware platforms must support the high level language constructs and operating systems' features dealing with time, which are characteristic for process control. Particularly in safety critical environments, apart from consistent designs regarding temporal circumstances, improved fault tolerance is required. It is necessary to intensify research in temporally deterministically behaving embedded computer control systems. In this paper design of a processor with fully predictable temporal behaviour is presented; it thus fits into the “layer-by-layer approach” (Stankovic and Ramamritham, 1990): predictability of each underlying level in the design of the real-time control system provides the necessary precondition for the predictability of the next subsequent layer.

In Section II the overall model of the control system will be briefly outlined. Its informal analysis and specification was given in (Colnarič, 1992). Specifically, the paper accounts for the VHDL-based modeling of a component within its architecture – the *task processor* (Groš, 1998) – as described in (Halang, 1988), (Colnarič, 1992), (Colnarič and Halang, 1993). As the processor's VHDL behavioral model can be parameterized and hence modified according to particular needs (Perry, 1990), it is suitable for hardware-software co-design (Wolf, 1994). Additionally, this behavioral model is technologically independent, thus leading to either FPGA or standard-cells implementations (Smith, 1997).

The rest of the paper is structured as follows. In Section III we describe the tools we have used for the design process and in Section IV the proper modeling of the processor . The simulation of the particular processor components is outlined in Section V. As testing of the processor as a whole represented an issue, the method used to overcome it is described in Section VI. Finally, concluding remarks and directions for future work are given in Section VII.

II. CONCEPT OF HARDWARE ARCHITECTURE

There are different obstacles in achieving predictability of temporal behaviour when designing a control system. One of the major causes of temporal non-determinism are interrupts. In our architecture they are dealt with by migrating the operating system kernel and the associated interrupt service from the application task processors to a dedicated OS kernel processor (Halang, 1988), (Colnarič, 1992), (Colnarič and Halang, 1993). Similar approaches were taken by several other groups (Ramamritham and Stankovic, 1989), (Lindh, 1989), (Roos, 1990), (Cooling, 1993), (Saez *et al*., 2000) as well.

The main idea can be illustrated by an everyday life example: the job of a manager's secretary is to relieve him from the routine work, so that he has more time left for more important tasks. The secretary also administers the manager's schedules and prevents him from being interrupted for unimportant reasons.

In classical computer architectures the operating systems are running on the same processor(s) as the application software. In response to any occurring event, the context is switched, system services are performed, and schedules are determined. Although it is rather likely that the same process will be resumed, quite a bit of computer capacity is wasted by superfluous overhead. This suggests employing a parallel processor to carry out the operating system services. Such an asymmetrical architecture depicted in Fig. 1 turns out to be advantageous since, by dedicating a special-purpose, multi-layer processor to the real-time operating system kernel, the user task processor(s) are relieved from any administrative overhead. The kernel processor and the task processors are connected point-to-point by serial links, thus avoiding collisions on common communication media as a possible source of non-determinism.

Fig. 1. Hard Real-Time Processor Model

The kernel processor (KP) is responsible for all operating system services. It maintains a real-time clock, and observes and handles all events related to it, to the external signals and to the accesses to the common variables and synchronizers, each of these conditions invoking assigned tasks into the ready state. It performs earliest-deadline-first scheduling on them and offers any other necessary system services.

External processes are controlled by tasks running in the task processors (TP) without being interrupted by the operating system functions. A running task is only pre-empted if, after re-scheduling caused by newly invoked tasks as a consequence of events, it is absolutely necessary to execute one of the arriving tasks immediately in order to allow for all tasks to meet their deadlines.

III. SELECTION OF DESIGN TOOLS

The design of a processor is a complex and computing intensive task. The input to the design process is usually an informal description of the processor, while the output is its technological description, e.g. (Smith, 1997). The latter depends on the chosen technology and eventually allows the manufacturing of the processor. Tools nowadays used in ASIC design process are highly sophisticated and quite capable of automating the design process in a great extent, thus reducing time to achieve a complete and

verified design. However, though automation saves time, it makes the design sub-optimal. A partial solution to this problem is to use more sophisticated tools, which will make the design closer to the optimal one. On the other hand, optimal design is only possible if it is done entirely by human designers, who are highly specialized for particular parts of the design process.

The first task, before starting the design process itself, is to find appropriate tools for that job. By "appropriate" it is meant that the tools cover the complete design process, support a high degree of design automation, are relatively easy to master and, finally, possess an attractive price/performance ratio. Three reasons made us to turn ourselves toward the Internet in search for an acceptable design toolkit. The first one is that the Internet has already become the source of many high quality applications, usually implemented as results of research projects within academia. As a general rule, these applications are not excessively user-friendly but are by all means of a state-of-the-art quality. The second reason is the immediate availability to which Internet has no match. The third reason is a purely economical one, since tools found on the Internet are generally either free of charge or are only nominally charged. After some search, mainly on well-known universities' sites and in newsgroups, all the tools found were grouped into three categories.

The first category consisted of many different and small design tools made as proof of a concept, and as such, highly specialized for only some particular tasks in the whole design process, like e.g. finite state machine (FSM) synthesizers. However, in spite of their great capabilities the idea of collecting these kind of tools was abandoned since integration seemed to represent a great problem. Also, learning all of them is exceedingly time consuming, because, as a rule, they do not conform to any standard. Nevertheless, it also might happen that some part of the design process remains uncovered by the appropriate tool.

Tools that, at least for educational purposes, have some nominal charge associated with them made the second category. The main representative to be mentioned here were tools offered by Berkeley University through its Industrial Liaison Program (Luehrmann and Rabaey, 2000). Although attractive, this alternative still meant purchase expenditure, even for tool evaluation purposes only, and was as well abandoned from further evaluation.

Tools that were found to satisfy all of our initial goals and to be also available for evaluation formed the third category. Here belong packages like *Alliance CAD* (Alliance, 1997), *Olympus Synthesis System* (De Micheli *et al.*, 1991), (Ku and De Micheli, 1990), and *Ocean* (Groeneveld and Stravers, 1993). After some evaluation of these tools we finally decided to use Alliance CAD since it was the most suitable for our purposes.

The Alliance CAD system is a complete set of tools for developing ASICs (Alliance, 1997). The system generates an output suitable for a standard cell or FPGA implementation and also offers an environment for model testing. Alliance has some restrictions though, especially with respect to the supported VHDL subset (IEEE, 1994). Further advantage of Alliance is the availability of different tools for generating behavioral, structural and physical models of different functional units. There are generators for adders, Booth multipliers, barrel shifters, RAM and ROM, as well as tools for generation of behavioral models of FSMs and the like.

IV. PROCESSOR MODELING

The design process should start with the behavioral modeling and simulation of the top-level entity, in our case the hard real-time processor. After this top-level entity is modeled and tested it is progressively decomposed into gradually smaller units up to the point where either the synthesis tools can be used, or structural modeling can be manually performed. This scenario, however, assumes an appropriate HDL, like e.g. VHDL (IEEE, 1994), which has to be supported within the design tool used. Unfortunately, the VHDL subset supported in Alliance CAD does not allow such an approach. Instead, the top-level entity description has to be entered in an already decomposed form.

As it is shown in Fig. 2, the following global units within the task processor can be clearly identified: (i) kernel interface, (ii) task control unit, and (iii) task process unit. These units are organized around the usual three busses global to a processor: the data bus, the address bus, and the control bus. Internal to the task process unit are additional busses – two data busses (`d0`, `d1`), and an address bus `a0`.

Fig. 2. Task Processor Model

1) Kernel Interface: this interface allows the interaction between the kernel processor performing operating system functions on the one side, and the task processors which control real-time processes, on the other. The respective specification is the responsibility of the kernel processor and is not pursued further. To provide for an adequate kernel processor interfacing, the internal busses are exposed in this interface.

2) Task Control Unit: This unit controls the global operation of the task processor. Three busses (data, address, and control) from Fig. 2 are also internal to this subunit. Namely, for efficiency reasons no internal task control unit busses are envisioned: as the task process unit makes use of private busses (`d0`, `d1` and `a0`) and the kernel interface is inactive during normal operation of the task processor, this is quite acceptable. Since Alliance CAD offers only simple VHDL constructs, the task control unit has to be further decomposed.

3) Task Process Unit: This unit performs actual transformations on data. It has its own internal busses, as stated in previous paragraphs. The number of busses, as well as their width and purpose, is determined by requirements placed upon the task processor during the specification phase. These requirements encompass execution of one instruction per cycle and parallel execution of data processing and flow control instructions. Since the execution of one instruction per cycle meant either some sort of pipelining or a considerable increase in processor complexity, some design compromises had to be done.

It was already stated that the task processor as a whole is too complex to be modeled with the available behavioral subset of Alliance's VHDL. Thus it has to be recursively decomposed into simpler subunits until a point is reached where some particular subunit is simple enough to be described with the available VHDL subset. As a consequence of this decomposition procedure it is necessary to provide for an appropriate subunits' interconnection. Within Alliance this is achieved by using the structural VHDL subset. In Fig. 3 the top-level structural model of the task processor is shown. As it can be seen, the respective primitives are the `COMPONENT`s. Each `COMPONENT` itself can be, and in our model they indeed are, composed of other

COMPONENTs. Eventually, at the leaf nodes of this decomposition behavioral COMPONENTs are to be given, because behavior is not described by structure, but has to be explicitly specified. In the following text it will be assumed that decomposable units possess both the structural description and, through the respective subunits, the behavioral model as well. Within this framework Fig. 4 shows a more detailed view of the task processor.

Fig. 3. Top-Level VHDL Structural Model of Task Processor

Fig. 4. Detailed Structure of Task Processor Model

Another point to be discussed, that is related to (sub)units' behavior is their mutual synchronization, which can be organized in two possible ways. In *strictly predefined synchronization* operation duration can be stated in advance. The other possible solution is *conditional synchronization* when a functional subunit will perform its operations as long as needed (and instructed so by an appropriate control signal). It was decided to implement the latter approach, mainly for a practical reason: there was no top-level task processor behavioral model available.

A. Task Control Unit

As the structure of this subunit was already described in (Colnarič, 1992) all that had to be done was to further refine it. This subunit consists of a program counter, program memory, stack, and control unit.

1) Program Counter: This is a relatively simple functional unit. The most difficult part in its modeling is implementing the contents increment because Alliance's VHDL has no arithmetic operators. After some simple analysis, the recursive formulae (1) and (2) were implemented, where n stands for the program counter weight:

$$new_pc_n = pc_n \oplus \left(pc_{n-1} \wedge \overline{new_pc_{n-1}}\right), \quad n \geq 1 \tag{1}$$

$$new_pc_0 = \overline{pc_0} \tag{2}$$

2) Stack: The stack is subdivided into two parts: stack memory and stack control. Since the memory is quite regular, has a simple structure, and its capacity is not known in advance, a small program in C is developed that automatically generates a VHDL behavioral RAM model with given capacity and data width. Stack pointer increment is modeled with the same specification code as for program counter increment; the one for stack pointer decrement is its slight modification. The code implements the recursive formulae (3) and (4):

$$dec_sp_n = sp_n \oplus \left(\overline{sp_{n-1}} \wedge dec_sp_{n-1}\right), \quad n \geq 1 \tag{3}$$

$$dec_sp_0 = \overline{sp_0} \tag{4}$$

3) Program Memory: The program memory is generated by a specially devised C program (using GNU's bison (Donnely and Stallman, 1995) and flex (Paxson, 1995) tools) that takes as input a program written in the task processor assembly language, and produces behavioral models of the program memory and of the local

memory that is part of the task process unit (to be described below). This approach is taken in order to overcome the need to run a number of small test programs for testing particular aspects of the task processor during the evaluation phase of the design (Glavinić *et al.*, 1999).

4) Control Unit: The control unit of the task control unit is implemented using Alliance CAD `syf` tool. This particular tool implements a subset of VHDL that is specialized for description of FSMs. However, `syf` imposes two restrictions: (i) any additional memory element as well as any additional functionality required by the control unit has to be implemented separately, and (ii) all signals are considered to be active with high logical value. Hence, the control unit is subdivided into three parts: the timer, the instruction register and the FSM.

B. Task Process Unit

After evaluation of required data processing instructions, and requirements such as one instruction per clock cycle as well as full predictability of processor behavior, few changes were made to the original design of the task process unit as described in (Colnarič, 1992). Mixed arithmetic instructions were removed and two new instructions introduced. The whole subunit is organized around five busses: two 8-bit address busses, two 32-bit data busses and a control bus. The subunit consists of the following functional blocks: a logical unit, an integer arithmetic unit, a floating-point arithmetic unit, two conversion units, a local register file, an external data access unit, and a control unit. Unlike the task control unit whose functional blocks are relatively simple, here we have some very complex functional blocks that have needed further decomposition.

During task process unit modeling some of Alliance CAD tools for behavioral model generation were extensively used accelerating in a great extent its development. These include tools like e.g. adder generator, multiplier generator and barrel shifter generator.

1) Logical Unit: The logical unit was relatively easy to model, its structure being determined by the use of the barrel shifter generator that is included in the Alliance CAD distribution. I.e. the whole logical unit was subdivided into a barrel shifter and a compound operator for the usual logical operations (`AND`, `NOR`, `XOR` and `AND_NOT`).

2) Integer Arithmetic Unit: The model of the integer arithmetic unit was almost completely generated with the help of Alliance CAD tools. The single parts of this unit so generated included the adder/subtractor as well as the multiplication unit, while the divider (except the subtracting subunit) was in part modeled manually. In the process of modeling the divider, two models representing disparate solutions in terms of time consumption per single division were devised: one that performs division in a single clock cycle (parallel subtraction) and the other that performs division in 32 cycles (serial subtraction). In either case, division has to be completed in a constant number of clock cycles to provide for predictability, although the structure of some operands allows for faster completion. In the final design the divider will usually be a compromise between the two extremes, the exact number of clock cycles necessary for completion of division being determined by the duration of the single clock cycle. For the present processor, however, no such requirements were stated.

3) Floating-Point Unit: The most complex functional block is the floating-point unit, which is further subdivided into subunits for addition/subtraction, multiplication and division, each of which is in turn further subdivided. Also, some common operations are separated into distinct units, hence, in addition to the aforementioned units, there is also one assigned the checking of correctness of input operands and of the output result. Multiplication was the easiest operation to model, basing on an adder for exponent and an integer multiplier for mantissa calculation. The divider is also not too complex, once the mantissa division is fixed, but it is the slowest unit in the task processor and might need to be re-designed, especially after performing timing analysis on the structural model. The same that was written for the integer divider holds also for the floating-point one. Apart from a divider for the mantissa, there is also a subtractor for the exponent. The adder and the subtractor are implemented in the usual way with the addition of an appropriate barrel shifter and exponent comparators for mantissa alignment.

4) Conversion Units: There are two conversion units in the task processing unit. One transforms the floating point numbers into integer ones and the other performs the inverse operation. These units are made of combinational logic only, but are quite complex since there are exceptions that need to be handled separately. The exceptions arise from the way floating-point numbers are coded. Also, care must be taken to check for some special values floating-point numbers can have, like NaN, before conversion (IEEE, 1985). These cases, as well as overflows and underflows are signaled thus enabling a proper processing.

5) Register File: This is a simple RAM local memory that resides on the task processor. Before execution of some process starts, the kernel processor pre-loads it with constants the process needs, and also, during context switch, saves temporarily the register file contents and restores it later when the process is restarted. However, as the kernel processor had not been available at simulation time, the program that generated the program memory (described in the previous subsection) was designed so that it could also generate VHDL models of local memory with constants prewritten into it, thus obviating kernel processor functions.

6) External Data Access Unit: This unit is the task processor interface to the external world. It is separated from the other units so that the task processor has a general interface, which can later be accommodated for particular standard busses, e.g. IIC (Philips, 1997). As some interfaces are asynchronous and some are synchronous the external data access is selected to have an asynchronous interface to the task processor thus serving both kinds of interfaces.

7) Control Unit: The control unit is structured as the one supervising the operation of the task control unit as described above. The VHDL subset for FSM description is used to specify a part of its functionality that is subsequently translated into the respective behavioral model using the `syf` tool.

V. SIMULATION

After modeling each block, its model was used to perform simulation testing. Simulation in essence applies some test inputs to a model and compares the outputs with the expected ones or simply outputs them to some file for later processing. This leads us to two types of simulations, which could be dubbed as "manual" and "automatic". The difference between the two is in the way the resulting outputs are

examined. Although it would be ideal to use automatic tests, there are situations that are not suitable for such an approach. For example, in the case of the floating-point unit it was necessary to use manual testing since it was both simpler and it offered the possibility to circumvent C rounding functions. In some other cases programming output calculations is simply too demanding and it is simpler to manually test the results.

Test inputs were prepared by using the `genpat` tool that is included into Alliance CAD distribution. `genpat` is actually a front-end to the C compiler and linker. The procedure for generating test inputs is straightforward; a simple C program is written, that, using macros offered by `genpat`, declares all input as well as internal and output signals of interest. After declaration, the program applies values on input signals, and optionally – depending on the type of tests (automatic or manual) – sets the expected values on the output and internal signals. After that, `genpat` is used to compile and run the C program, which as a result produces test patterns in special format files for Alliance's VHDL simulator `asimut`.

Testing the task processor' individual units for a correct operation does however not guarantee the correct operation of the processor as a whole. Thus, as the individual parts of the processor were modeled, they were gradually interconnected and tested.

Finally, as verification of all the parts of the processor was completed, the processor as a whole was tested through behavioral simulation. As the processor is a fairly complex unit, and simulations are process intensive, it was decided that many small different tests will be performed, each of which checks for a particular processor feature. Fig. 5 shows an example of a such a simple program that is translated into behavioral models for program and local memories, which are then interconnected with the other parts of the processor and simulated. Some characteristic processor signals during simulation are shown in Fig. 6.

Fig. 5. Simple Test Program

Fig. 6. Values of Some Internal Signals in Task Process Unit during Simulation

VI. MODELING PROGRAM AND DATA STORAGE

Both the program memory and the local memory are modeled as separate VHDL modules and shown as shaded blocks in Fig. 4. It should be noted that it is not very efficient to hand-code test programs and manually write the respective VHDL models: firstly, it is time consuming, and secondly, memories (in our case both program and local) have highly regular structures, which could be generated automatically. All this leads us to the conclusion that it is practical to define an assembler that translates programs written in the respective task processor assembly language into VHDL models for both the program and local memory.

A. Assembler Input and Output

The input to the assembler is a file containing a program written in the task processor assembly language. The outputs are two distinct VHDL behavioral models, one

containing the model of the program memory, and the other containing the model of the local memory with predefined constants.

B. Input File Syntax

The syntax of the input file is fairly simple: generally, it starts with variable definitions, continues with the program code and ends with the reserved word `end`. Blank lines and everything beginning with a hash (#) is ignored up to the end of the line. Variables are defined with the following syntax:

`variable_name type initial_value`,

where `variable_name` is any sequence of characters, `type` is one of real or integer, and `initial_value` is the starting value of the variable.

The original task processor instruction set (Colnarič, 1992) was slightly modified prior to behavioral modeling phase. Table 1 lists the available instruction mnemonics and assembler directives. In the input file each instruction is placed in its own input line with the following syntax:

`label: instruction`,

where `label` is used to mark the line if it is to be referenced; either `label` or `instruction` may be omitted.

Table 1. Available Instruction Mnemonics and Assembler Directives

C. Output VHDL Models

Output files, for program and for local memory, are VHDL models of respective modules within the task processor. Both of them have a regular structure built by regular repetition of basic cells. Also, both models start with the definition of the interface to the outside world.

Fig. 7 shows the VHDL model of the program memory with three-instruction contents. The model is easily extended to accommodate for larger programs.

Fig. 7. VHDL Model of Program Memory

Excerpts from the VHDL model of local memory are given in Figures 8 to 10. The interface to the outside world, as shown in Fig. 8, is the same for all the models. Fig. 9 shows the definition of some of memory locations (although not all of them are shown): there are all together 255 of them, along with two internal read busses. This section is also identical for all local memory models. Finally, Fig. 10 shows excerpts from the code that defines behavior of the local memory.

Fig. 8. Interface Definition for Local Memory

Fig. 9. Definition of Memory Locations and Internal Busses for Local Memory

Fig. 10. Local Memory: a) Writing Into; b) Reading From; c) Reset

D. Implementation

The assembler is implemented in C. Some of its parts are developed using the language construction tools `bison` (Donnely and Stallman, 1995) and `flex` (Paxson, 1995). In the first phase the assembler builds the internal representation of the program. Afterwards the postprocessor is called by the main function that fixes label addresses, and finally generates the output VHDL models.

VII. CONCLUSION AND FUTURE WORK

The goal of the paper was to explore the possibilities to implement consistently the design of a relatively simple processor for execution of hard real-time tasks, as proposed in previous works (Halang, 1988), (Colnarič, 1992), (Colnarič and Halang, 1993). The processor's ultimate requirement was thus temporal predictability of instruction execution. In the paper, it was shown constructively, that such a design is possible with simple and readily available means.

The fact that no kernel interface had been predefined influenced the present model of the task processor, which accordingly is not fully functional and its top-level structural model is far from being technologically implementable. After developing this interface, the next step in the design process would be full structural modeling (presently only partially done) and model mapping onto some technology. Structural modeling could be done with some tools already available in Alliance CAD which are capable of transforming behavioral descriptions into structural ones. However, these tools have some restrictions related to synthesis being applicable to simpler cases only; it would thus be necessary to rewrite some parts of the existing VHDL specification in order to allow its automatic synthesis. In the case that automatic synthesis is not possible, it might be needed to manually perform structural modeling. After mapping the model onto the selected technology the two important parameters – maximum clock rate and required number of transistors – can be established. However, it should be noted that limitations on the number of transistors exist thus possibly requiring processor redesign. Generally, more transistors means a more sophisticated architecture (as parallelism is possible), which would allow higher throughput, and a higher clock frequency as well. The same holds if the clock frequency resulting from the selected technology is lower than the required one.

ACKNOWLEDGEMENT

This work has been partly carried out within project 036033 *Architectural Elements for Regional Information Infrastructures*, funded jointly by the Ministry of Science and Technology of the Republic of Croatia and the Istrian County.

REFERENCES

[1] Alliance (1997) *Alliance: A Complete CAD System for VLSI Design*, Alliance 3.2b Distribution, Universitè Marie et Pierre Curie, Paris, December 1997.

[2] Colnarič, M. (1992) *Predictability of Temporal Behaviour of Hard Real-Time Systems*, Ph.D. Diss., Faculty of Technical Sciences, University of Maribor, Maribor, 1992.

[3] Colnarič, M., Halang, V (1993) "Architectural Support for Predictability in Hard Real-Time Systems", *Control Engineering Practice*, Vol. 1, No. 1, Pergamon Press, February 1993, pp. 51-59.

[4] Cooling, J. (1993) "Task scheduler for hard real-time embedded systems", *Proc. Int'l Workshop on Systems Engineering for Real-Time Applications*, pp. 196-201.

[5] De Micheli, G., Ku, D., Mailhot, F., Truong, T. (1991) *The Olympus Synthesis System for Digital Design*, `http://akebono.stanford.edu/users/cad/synthesis/olympus/doc/olympus.ps`, Computer System Laboratory, Stanford University, 1991.

[6] Donnely, C., Stallman, R. (1995) *Bison*, `ftp://ftp.zemris.fer.hr/pub/gnu/bison-1.25.tar.gz`, Bison 1.25 distribution, 1995.

[7] Glavinić, V., Groš, S., Colnarič, M. (1999) "Automated Generation of VHDL Behavioral Models for Testing Purposes", *Proc. 21st Int'l Conf. Information Technology Interfaces – ITI'99*, June, 15-18, 1999, Pula, Croatia, 249-254.

[8] Groeneveld, P., Stravers, P. (1993) *Ocean: the Sea-of-Gates Design System*, `ftp://metalab.unc.edu/pub/Linux/apps/circuits/ocean/ocean_docs.tar.gz`, Ocean Distribution, 1993.

[9] Groš, S. (1998) *Behavioral and Structural Modeling of a Real-Time Processor*, Diploma Work No. 1123, Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, 1998 (in Croatian).

[10] Halang, W. A. (1988) *Definition of an Auxiliary Processor Dedicated to Real-Time Operating System Kernels,* Technical Report No. UILU-ENG-88-2228 CSG-87, University of Illinois at Urbana Champaign, 1988.

[11] IEEE (1985) *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Std 754-1985, IEEE, New York, 1985.

[12] IEEE (1994) *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076-1993, IEEE, New York, NY, 1994.

[13] Ku, D., De Micheli, G. (1990) *HardwareC: A Language for Hardware Design*, Technical Report CSL-TR-90-419, `http://akebono.stanford.edu/users/cad/synthesis/olympus/doc/hardwarec.ps`, Computer System Laboratory, Stanford University, August 1990.

[14] Lindh, L. (1989) *Utilization of Hardware Parallelism in Realizing Real-Time Kernels*, PhD Thesis, Kungl Tekniska Hoegskolan, Sweden, 1989.

[15] Luehrmann, M., Rabaey, J. (2000) *About the Industrial Relations Office at UC Berkeley's EECS Department*, `http://buffy.eecs.berkeley.edu/IRO/about.iro.html`

[16] Paxson, V. (1995) *Flex, Version 2.5*, `ftp://ftp.zemris.fer.hr/pub/gnu/flex-2.5.4.tar.gz`, Flex 2.5a distribution, 1995.

[17] Perry, D. L. (1990) *VHDL*, McGraw-Hill, Inc., San Ramon, CA, 1990.

[18] Philips (1997) *PFC8524. I^2C-Bus Controller*, Philips Semiconductors, Product Specification, Document Order Number 9397-750-01554, 1997 March 19.

[19] Ramamritham, K., Stankovic, J. A. (1989) Overview of the SPRING project, *Real-Time Systems Newsletter*, Winter 1989, Vol. 5, No. 1, pp. 79-87.

[20] Roos, J. (1990) “The performance of a prototype coprocessor for Ada tasking”, *Annual International Conference on Ada, Proceedings of the Conference Tri-Ada*, ACM, December 3-6, 1990, Baltimore, MD USA, pp. 265-279.

[21] Saez, S., Vila J., Crespo A., Garcia, A. (2000) “A Hardware Architecture for Scheduling Complex Real-Time Task Sets”, *CIT–Journal of Computing and Information Technology*, this issue.

[22] Smith, M. J. S. (1997) *Application-Specific Integrated Circuits*, Addison-Wesley, Reading, MA, 1997.

[23] Stankovic, J. A. (1988) “Misconceptions About Real-Time Computing”, *IEEE Computer*, October 1988, Vol. 21, No. 10, pp. 10-19.

[24] Stankovic, J., Ramamritham, K. (1990) "What Is Predictability for Real-Time Systems?", *Real-Time Systems*, Vol. 2, No. 4, November 1990, pp. 247-254.

[25] Wolf, W. (1994) "Hardware-Software Co-design of Embedded Systems", *Proc. IEEE*, Vol. 82, No. 7, July 1994, 967-989.

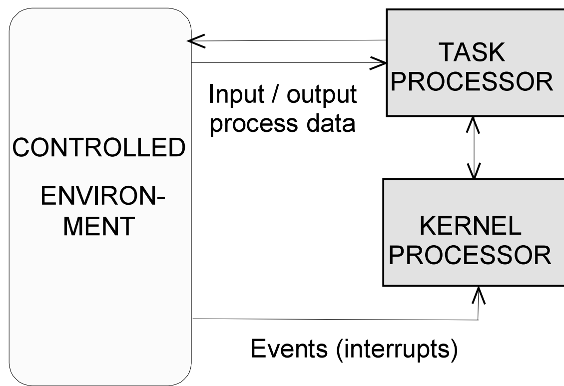

Fig. 1. Hard Real-Time Control Computer Model

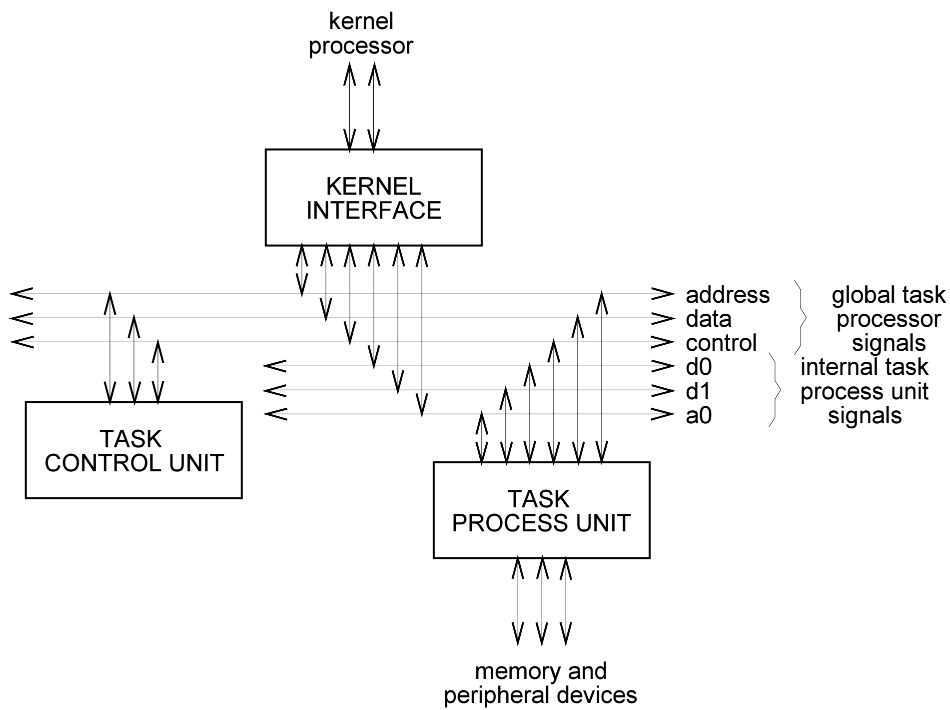

Fig. 2. Task Processor Model

```
ENTITY rttpi IS
 PORT (
  ck, reset, continuation, int, load_sp,
  save_sp, load_pc, save_pc: IN bit;
  ack: OUT bit;
  data_mem: INOUT mux_vector (31 DOWNTO 0) BUS;
  address_mem: OUT mux_vector (31 DOWNTO 0) BUS;
  oe_mem, wr_mem: OUT bit;
  ack_mem, vss, vdd: IN bit);
END rttpi;
ARCHITECTURE structural OF rttpi IS
 COMPONENT task_procesor_control
  PORT (
   ck, reset: IN bit;
   data: INOUT mux_vector (31 DOWNTO 0) BUS;
   continuation, int: IN bit; b_read: OUT bit;
   b_flag, sync_in: IN bit; sync_out: OUT bit;
   load_sp, save_sp, load_pc, save_pc: IN bit;
   ack: OUT bit; vss, vdd: IN bit);
 END COMPONENT;
 COMPONENT processing_unit
  PORT (
   ck, reset, sync_in: IN bit; sync_out: OUT bit;
   get_b_flag: IN bit;
   data: IN bit_vector (31 DOWNTO 0);
   b_bus: INOUT mux_bit BUS;
   data_mem: INOUT mux_vector (31 DOWNTO 0) BUS;
   address_mem:OUT mux_vector (31 DOWNTO 0) BUS;
   oe_mem, wr_mem: OUT bit;
  ack_mem, vdd, vss: IN bit);
 END COMPONENT;
SIGNAL data: mux_vector (31 DOWNTO 0) BUS;
 SIGNAL b_read: bit; SIGNAL b_bus: mux_bit BUS;
 SIGNAL sync_in, sync_out: bit;
BEGIN
 control: task_procesor_control
  PORT MAP (
   ck, reset, data, continuation, int, b_read,
   b_bus, sync_in, sync_out, load_sp, save_sp,
   load_pc, save_pc, ack, vss, vdd);
 processing_unit: processing_unit
  PORT MAP (
   ck, reset, sync_in, sync_out, get_b_flag, data,
   b_bus, data_mem, address_mem, oe_mem, wr_mem,
   ack_mem, vdd, vss);
END;
```

Fig. 3. Top-Level VHDL Structural Model of Task Processor

```
#  This simple test calculates the following expression:
#                      a * c - b
#

#  Binary code after translating is:
#
#       0x90010304
#       0x88040205
#       0x7FFFFFFF

a       real    1.0
b       real    -0.1
c       real    0.1e-5
temp    real    0.
result  real    0.

        org     0

        mul     a, c, temp
        sub     temp, b, result
        halt

        end
```

Fig. 4. Simple Test Program

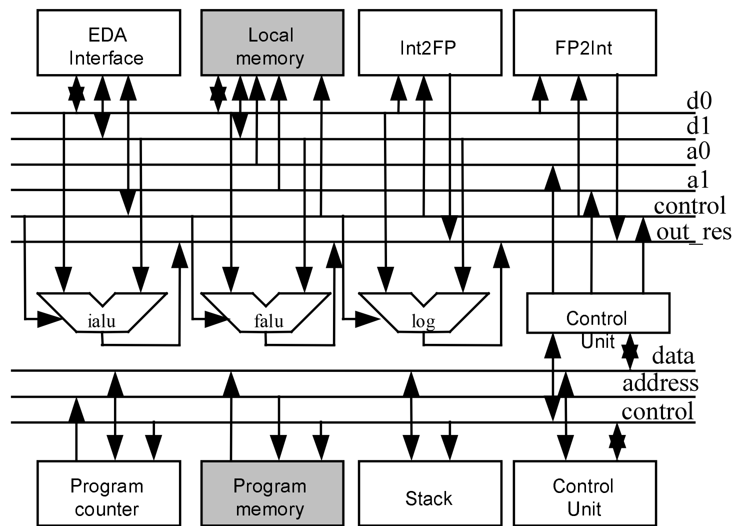

Fig. 5. Detailed Structure of Task Processor Model

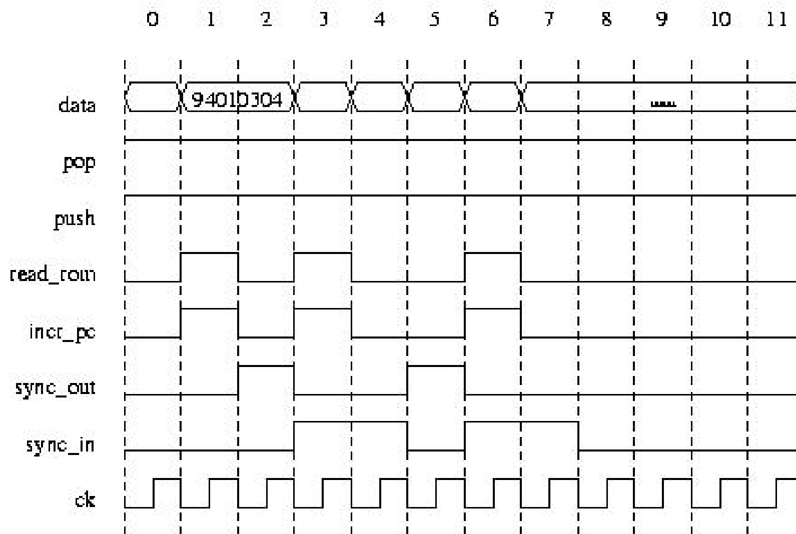

Fig. 6. Values of Some Internal Signals in Task Process Unit during Simulation

```
ENTITY program_memory IS
   PORT (
      read_rom: IN bit;
      address:  IN bit_vector (23 DOWNTO 0);
      data: OUT mux_vector (31 DOWNTO 0) BUS;
      vss, vdd: IN bit
   );
END program_memory;

ARCHITECTURE behavioural OF program_memory IS
   SIGNAL rom_out: BIT_VECTOR (31 downto 0);
BEGIN

   write_out: BLOCK (read_rom = '1')
   BEGIN
      data <= GUARDED rom_out;
   END BLOCK;

   WITH address (23 DOWNTO 0) SELECT
      rom_out <=   X"94010304" WHEN X"000000",
                   X"8C040205" WHEN X"000001",
                   X"7FFFFFFF" WHEN X"000002",
                   X"00000000" WHEN OTHERS;
END;
```

Fig. 7. VHDL Model of Program Memory

```
ENTITY local_memory IS
 PORT (
  reset: IN bit;
  d0: INOUT mux_vector (31 DOWNTO 0) BUS;
  d1: OUT mux_vector (31 DOWNTO 0) BUS;
  a0, a1: IN bit_vector (7 DOWNTO 0);
  oe0, oe1: IN bit;
  write: IN bit;
  vss, vdd: IN bit
 );
END local_memory;
```

Fig. 8. Interface Definition for Local Memory

```
ARCHITECTURE behavioural OF b IS
 SIGNAL cell_1: reg_vector (31 DOWNTO 0)           REGISTER;
 SIGNAL cell_2: reg_vector (31 DOWNTO 0)  REGISTER;
.................................................
SIGNAL cell_254: reg_vector (31 DOWNTO 0) REGISTER;
 SIGNAL cell_255: reg_vector (31 DOWNTO 0) REGISTER;

 SIGNAL read0: bit_vector (31 DOWNTO 0);
 SIGNAL read1: bit_vector (31 DOWNTO 0);
BEGIN
```

Fig. 9. Definition of Memory Locations and Internal Busses for Local Memory

Table 1. Available Instruction Mnemonics and Assembler Directives

Mnemonic/Directive	Mnemonic/Directive
jump <dest>	nor <source1>, <source2>, <dest>
jumpf <dest>	xor <source1>, <source2>, <dest>
jumpt <dest>	and_not <source1>, <source2>, <dest>
call <dest>	move <source>, <dest>
return	rol <source1>, <source2>, <dest>
wait <time>	rol <source1>, <source2>, <dest>
wait <time>, <dest>	ror <source1>, <source2>, <dest>
halt	shl <source1>, <source2>, <dest>
add <source1>, <source2>, <dest>	shr <source1>, <source2>, <dest>
sub <source1>, <source2>, <dest>	org <address>
mul <source1>, <source2>, <dest>	integer
div <source1>, <source2>, <dest>	real
and <source1>, <source2>, <dest>	

```
wr_1: BLOCK (write = '1' AND a0 = X"01")
BEGIN
 cell_1 <= GUARDED d0;
END BLOCK;
wr_2: BLOCK (write = '1' AND a0 = X"02")
BEGIN
 cell_2 <= GUARDED d0;
END BLOCK;
........................................
wr_255: BLOCK (write = '1' AND a0 = X"FF")
BEGIN
 cell_255 <= GUARDED d0;
END BLOCK;
```

a)

```
WITH a0(7 DOWNTO 0) SELECT
 read0 <= cell_1 WHEN X"01",
       cell_2 WHEN X"02",
       ..................
       cell_255 WHEN X"FF",
       X"00000000" WHEN OTHERS;
WITH a1(7 DOWNTO 0) SELECT
 read1 <= cell_1 WHEN X"01",
       cell_2 WHEN X"02",
       ..................
       cell_255 WHEN X"FF",
       X"00000000" WHEN OTHERS;
out_0: BLOCK (oe0 = '1')
BEGIN
 d0 <= GUARDED read0;
END BLOCK;
out_1: BLOCK (oe1 = '1')
BEGIN
 d1 <= GUARDED read1;
END BLOCK;
```

b)

```
reset_5: BLOCK (reset = '0')
BEGIN
 cell_5 <= GUARDED X"00000000";
END BLOCK;
reset_4: BLOCK (reset = '0')
BEGIN
 cell_4 <= GUARDED X"00000000";
END BLOCK;
..........
reset_255: BLOCK (reset = '0')
BEGIN
 cell_255 <= GUARDED X"00000000";
END BLOCK;
```

c)

Fig. 10. Local Memory: a) Writing Into; b) Reading From; c) Reset

Fig. 1. Hard Real-Time Processor Model

Fig. 2. Task Processor Model

Fig. 3. Top-Level VHDL Structural Model of Task Processor

Fig. 4. Detailed Structure of Task Processor Model

Fig. 5. Simple Test Program

Fig. 6. Values of Some Internal Signals in Task Process Unit during Simulation

Fig. 7. VHDL Model of Program Memory

Fig. 8. Interface Definition for Local Memory

Fig. 9. Definition of Memory Locations and Internal Busses for Local Memory

Fig. 10. Local Memory: a) Writing Into; b) Reading From; c) Reset

Table 1. Available Instruction Mnemonics and Assembler Directives