Project 22703: SYRF

Deliverable 6.1.1

VHDL-Verilog

Abstract: In this paper we present a methodology for designing complex hardware systems. This methodology is based on the synchronous data flow language SIGNAL which offers a formal framework to build executable specifications of hardware components. It can be applied to the dataflow synchronous common format DC^{+}. All design steps (i.e. refinements, verification, simulation, HDL generation, ...) are based on this unique formalism which allows to reduce product design cycle by decreasing communication problems between design phases. In this paper we emphasis on the verification process and the HDL generation which take advantages of the formal framework provided by SIGNAL.

1 Introduction

In most approaches to complex hardware system design (i.e. microprocessors, microcontrollers ...) multiple formalisms such as natural languages (i.e. English), standard programming languages (i.e. C), hardware description languages (i.e. Verilog [24] or VHDL [2]) and specific formalisms are used in the distinct specification steps. One of the consequences of design methods built on multiple formalisms is that the product design and checking is inherently long. Typically, from a specification written in a natural language a simulation oriented description (usually in C) and a synthesis oriented implementation (usually as a RTL HDL program) are derived. Such design methodology results in a very long cycle production time since communications between phases are difficult and error prone. For instance, when a design error (or an unexpected timing constraint) is discovered during the implementation, updating the design chain is very time consuming and may result in further errors. Furthermore the use of formal methods is very difficult in such a multi-formalisms framework.

In this paper, we present a design methodology for complex hardware systems where the synchronous language SIGNAL [19] is used as the unique formalism for all design steps. SIGNAL offers a very adequate formalism to hardware specification as it provides an implementation independent abstraction based on synchronous data flows.

One of the main advantage of SIGNAL is provided by its formal semantic that allows directly the application of formal verification methods. Correct by construction behavioral simulators and proved equivalent synthesizable HDL descriptions can also be automatically derived from a SIGNAL specification. An incremental development from a high level description to a very precise one is done through the specification of the implementation choices. These refinement steps are, as much as possible, proved to be correct and automatically performed.

This paper presents a preliminary description of the methodology for hardware design based on SIGNAL. We refer the reader to [20] for a presentation of the SIGNAL language. Section 2 describes the methodology for designing and deriving an application from an **executable specification**. In Section 3, we show how synthesizable HDL is derived from a SIGNAL specification. In particular, we emphasis on the concept of well clocked implementations that defines how abstract signals are mapped to implementable ones.

2 Methodology for Synchronous Hardware Design

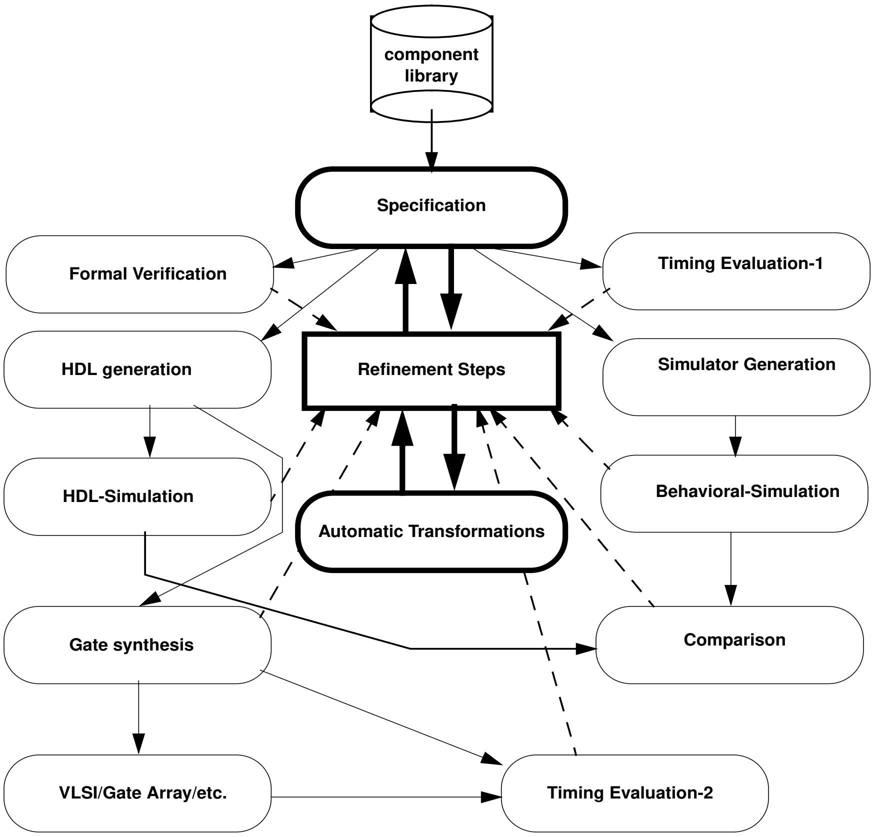


Figure 1: Methodology for hardware specification.

Figure 1 illustrates the methodology we propose for synchronous hardware design. It is based on refinements to achieve the final specification. We can distinguish two kinds of refinements: **behavioral refinements** which consist in modifications of the specification and **implementation refinements** which consist in adding the implementation details to the specification. Refinements may be done manually but also automatically (correctness is ensured in this case). Modification to the specification are made according to the information provided via various tools which are applied on the specifications:

Formal Verification: The first verification is provided by the compilation of the specification. Then other verification methods (model checking, theorem provers) are applied so properties of the specification are formally checked. This step consists essentially in two phases. In phase one, the formal properties to be verified by the component description are given. The second phase, mostly automatic, is the verification of the properties.

Component Library: Hardware components libraries can be taken into account into this process assuming that interface constraints and a behavioral model are provided. At any step in the design, the user can integrate a predefined library components.

Behavioral/HDL Simulations: Simulators can be provided either from the SIGNAL specification (via C code generation) or from the derived HDL description. As SIGNAL signals are comparable to HDL signals, the simulation results can be compared directly. Deriving the simulator directly from the SIGNAL specification can be done at any step of the design and allows to get, generally, faster simulation.

Timing Evaluations: The timing evaluations provide basic data to choose the implementation of the hardware components [14].

HDL Generation: When a SIGNAL description is a complete specification of a hardware component (i.e. input-driven, it has been successfully processed by the SIGNAL compiler), then it can always be fully compiled in some HDL language. However we are assuming that gate synthesis is performed through third party tools such as Synopsys.

```
process INT_MUX2=
{  ?  logical CTR; %CTR, V1, V2 are input signals%
   integer V1;
   integer V2
   !  integer VAL } %VAL is the output signal%
(| VAL := (V1 when (not CTR)) default (V2 when CTR)|)
end
```

Figure 2: Description of an integer multiplexer in SIGNAL.

Writing the specification of a component is performed in a top-down manner consisting first in the mapping of the input/output signals to SIGNAL and then in describing the relationship between these signals. For each signal, a type must be chosen to carry the data. In SIGNAL, among others the following types can be used:

event: An event can be used each time a signal has to trigger an action. For instance, any boolean signal active on true or false can be represented by an event.

logical: Any signal that must carry a logical value. For instance the signal command of a two entry multiplexer (shown figure 2).

integer: Most of the usage of integers is done as a preliminary step for any multi-bit values. As the specification is refined, integers can be replaced by a simpler types at the HDL implementation level.

arrays: arrays can be used to group signals, of the same type, if there are always used synchronously.

```
process COUNTER =
{  ?  event RESET,CLK,DEC,INC
   !  integer I }
(| LNI := ((0 when RESET)
         default (LI when INC when DEC)
         default ((LI+1) when INC)
         default ((LI-1) when DEC)
         default LI) cell CLK
   | LI:= LNI$1
   | I := LI
|)
where
   integer LI init 0, LNI init 0
end
```

Figure 3: Incr.-Decr. Counter in Signal

The component body is described as a set of operations (intrinsically a set of equations) on input signals (`?`) to produce the output signals (`!`). Figure 3 shows the specification of a simple resetable incrementer-decrementer counter. A second example is the description a simple finite state machine shown figure 5. It is described in SIGNAL using two integer signals, `S` denoting the next state, and `SZ` the current state. For each transition `T` a signal is created. This signal is an event that triggers the transition. For complex FSMs we provide a preprocessor that allows to write automaton description at a higher level.

One of the key for refinements is to allow to check that a refinement produces a specification that is "equivalent" to the previous one. In SIGNAL this is simply done, either because the refinement is automatically performed and proven correct (such as the transformation used for HDL generation, see section 3), or using formal verification techniques such as model checking (see [1]). For instance figure 4 shows how equivalence can be checked of a component part P and the refined one P'. E_1, E_2, E'_1 and E'_2 allow to express equivalence between I/O signals.

3 HDL Code Generation

The reader ie referred to the full version of this paper [1] for further details and a more formal presentation.

We provide here a first and direct translation path from SIGNAL specifications to HDL descriptions. The most fundamental requirement for the HDL code generation is that the resulting code should be functionally equivalent to the initial SIGNAL specification. In this section we first define the notion of implementation we use and then give a structural translation method, illustrated figure 6, to produce synthetisable HDL code.

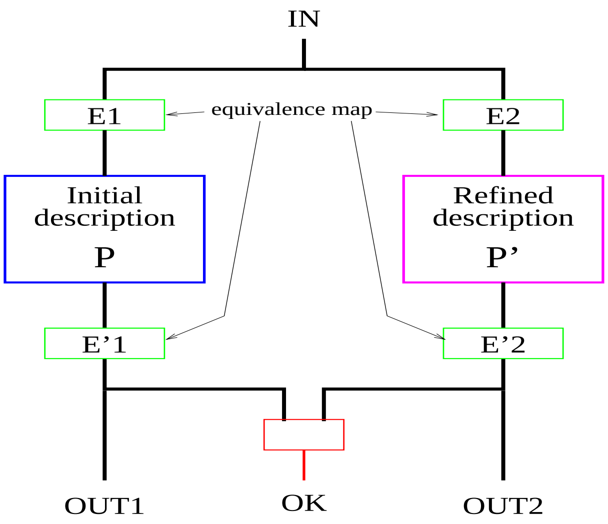


Figure 4: Refinement verification.

3.1 Well Clocked Implementations

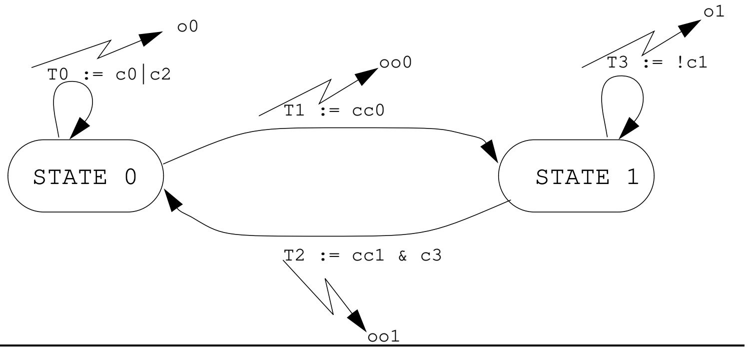
The well clocked implementation notion allows us to compare two SIGNAL processes, a specification P with a set A of I/O signals and an implementation P' with a greater or equal set A' of I/O signals. Roughly speaking, P' is a well clocked implementation of P if, the subset of I/O signals of P' that correspond to the I/O signals of P (a one to one correspondance σ) at the instants defined by a clock system of P (*definition of present*), then we assume that the implementation and the specification have the same behaviour. In other words, we assume that an implementation extends the event domain, but using strobe signals we are able to sample the events corresponding to the initial specification. In practice, this means that for each initial signal a we can add a strobed signal stb_a' and rename the initial signal into a', that both a' and stb_a' will be synchronized with the fastest clock clk, and if we look only at the signals a' of the new process, its behaviour is the same as the behaviour of initial process. Figure 7 illustrates this technique.

3.2 HDL Generation

The HDL generation is decomposed in two steps. A first transformation (I) modifies the SIGNAL specification in a new one that allows the HDL generation. This transformation must satisfy the property of well clocked implementation defined in previous section. The second step ($\mathcal{H}$) is a structural translation of the SIGNAL specification into the HDL target language (VHDL in our case). This is illustrated figure 8.

3.2.1 The Strobe Insertion Transformation $\mathcal{I}$

The main goal of $\mathcal{I}$ is to transform the SIGNAL program in such a way that it is composed of signals that can be implemented. In our case this consists in transforming all signals into strobed signals that are synchronous to a signal `clk` that represents the circuit main clock. In other words all signals `a` are transformed in two new signals `(a', strb_a')`. This transformation is illustrated figure 9.




```
process FSM=
{  ?  event c0,c1,c2,c3,cc0,cc1,CLK
   !  event o0,o1,oo0,oo }
(| S := (1 when T1) default (0 when T2) default SZ
   | synchro S, CLK
   | SZ := S$1
   | T0 := (c0 default c2) when (SZ = 0)
   | T1 := cc0 when (SZ = 0)
   | T2 := (cc1 when c3) when (SZ = 1)
   | T3 := when ((not c1) default CLK) when (SZ = 1)
   | o0 := when T0
   | o1 := when T3
   | oo0:= when T1
   | oo1:= when T2
|)
where
   event T0,T1,T2,T3;
   integer S, SZ init 0
end
```

Figure 5: An automaton description in Signal.

3.2.2 Producing HDL Code, the $\mathcal{H}$ Transformation

This transformation corresponds to the second step of the HDL generation process. Basically $\mathcal{H}$ applies a set of rules for a structural translation based on the chosen HDL target language. These rules translate operators, processes, and types of SIGNAL to the equivalent elements of the target HDL. For instance, when generating VHDL, the translation of the default operator is given figure 10. The mapping used for scalar types is the following: `event` is translated in BIT, `logical` is translated in BIT and `integer` is translated in INTEGER range MINVAL to MAXVAL.

Integers require a range to be provided for minimizing hardware. We envision a semi-automatic user-assisted process to extract such ranges. If a range is not known then default values have to be used depending on the intended internal/external bus widths.

A prototype of the VHDL code generator has been implemented. The output of the VHDL synthesis process, for the incrementer-decrementer counter given in figure 3, is

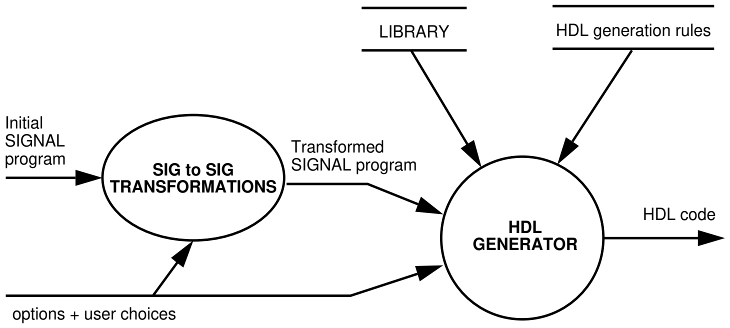


Figure 6: HDL generation process.

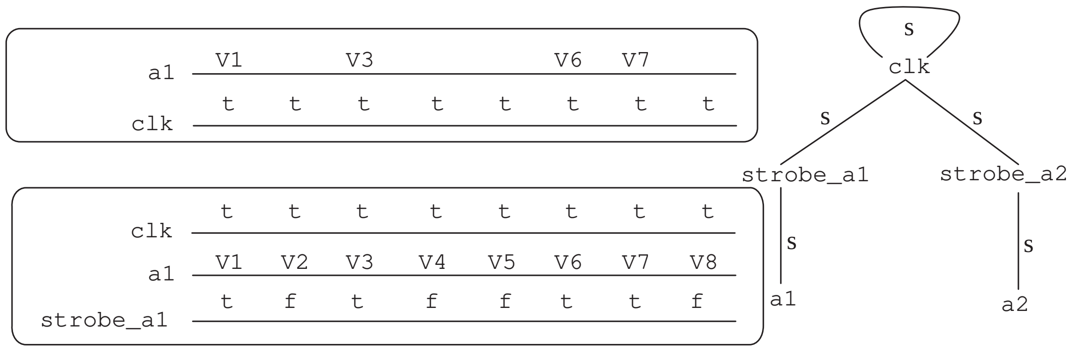


Figure 7: Well clocked implementation.

shown figure 11. The corresponding simulation is given figure 12.

Other circuits have been synthesized with our tool: an address generator (control protocol and data path), a GCD (based on euclid algorithm) and a eight words depth fifo. The results are summarized in Table 1. The areas are obtained using the *Class* library of the Synopsys [23] tool and after Synopsys optimization.

	SIGNAL	VHDL	gen. time	ports	nets	cells	Area		
							comb.	non comb.	total
Counter	17l	463l	0.24s	14	85	80	109	56	165
Address gen.	332l	1879l	0.31s	20	650	637	863	1024	1887
GCD	23l	607l	0.34s	29	175	155	213	112	325
FIFO	26l	763l	0.39s	22	646	634	798	952	1750

Table 1: Synthesis results

4 Related Works

Many approaches to hardware specification relies on hardware description language such as Verilog or VHDL [24, 2]. As these languages are conceptually similar to general purpose imperative languages such as C, ADA or Fortran they tend to the same drawbacks:

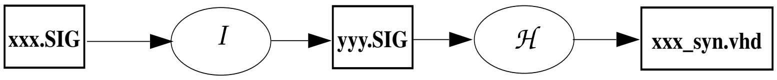


Figure 8: Transformations for HDL generation

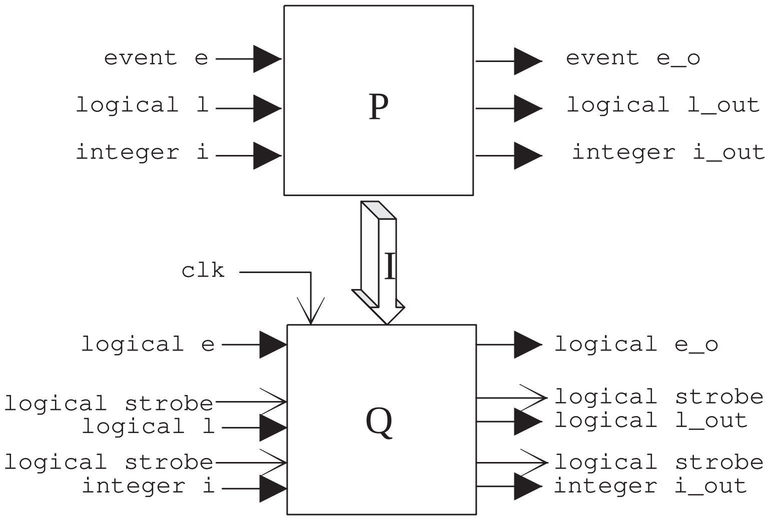


Figure 9: The strobe insertion transformation I.

1. HDL language descriptions are too close to implementations. The synchronous approach allows to write executable specifications which are more independent from any implementation.

2. In Signal there is no notion of a synthesizable subsets of the language which are generally obstacles to ensure portability between tools.

3. Model checking techniques are difficult to implement on HDL (especially on the full language) while Signal offers a formal model that already integrates clock calculus. Furthermore, it is possible to automatically transform programs into an equivalent form that improves accuracy of model checking. For instance, bounded integers can be transformed as a set of boolean variables for the purpose of model checking, while integers are used when generating the implementation or simulating it.

The methodology we are proposing can also be implemented on other synchronous languages than Signal [5, 10]. However, Signal offers a good compromise over other existing ones. Indeed, Signal is a declarative language that naturally provides an abstraction of hardware components. The clock calculus of Signal synthesizes the constraints and verifies their consistency (they admit a solution) and their completeness (they admit only one solution). For instance, compared to Statechart [12] we ensure the deterministic behavior of a component. For a more extensive comparison of synchronous languages the reader may refer to [11].

Hardware/software partitioning issues have not been considered so far, nevertheless our approach is also related to software/hardware co-design [21] environments for embedded system such as Polis [4], VULCAN [22], COSYMA [9], Chinook [7], etc. The closest to

```
process DEFAULT_SYN=
{ ? event CLK;
    logical STB_A;
    A;
    logical STB_B;
    B
  ! logical STB_C;
    C
}
(| synchro {CLK,A,STB_A,B,STB_B}
 | synchro {CLK,C,STB_C}
 | C := (A when STB_A) default B
 | AH := when STB_A
 | BH := when STB_B
 | CH := AH default BH
 | STB_C := (true when CH) default false
 |)
where
    event AH,BH,CH
end ;
```

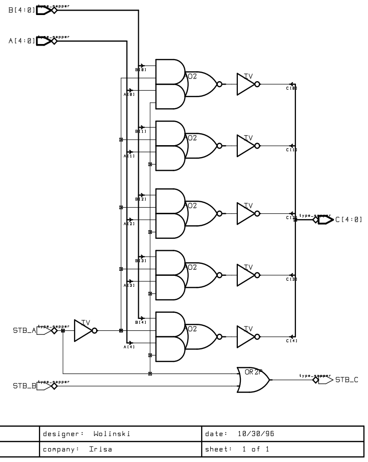


Figure 10: The default operator (left) and the synthesis result (right).

our approach is Polis. Others systems are either based on variations of the C language or on HDL language. Compared to Polis, rather than having multiple front-ends to a finite state machine language, our approach is based on a unique formalism for all specification steps. However, a SIGNAL interface to Polis (i.e. a translator from SIGNAL to the Polis CFSM language) could be easily implemented to provide a link between the tools.

5 Conclusion

We present in this paper a methodology based on the synchronous data flow SIGNAL language. We show that this language provides an adequate abstraction for the executable specification of complex hardware components. Generation of synthesizable HDL code can be done as well as model checking. This allows to integrate in a unique framework and many of the specification steps. However, this work still need to be extended to provide a very convenient tool. The first extension will be to enhance the SIGNAL language to allow more genericity in the description and also new data types. Part of future work will also consist in providing proven correct refinement transformations. None of these extensions are fundamental but they should make hardware specifications easier to write. The next direction of study is the integration of hardware components. In particular HDL to SIGNAL translation may be explored to use existing HDL libraries. Finally to improve the HDL code generation we will study optimizations based on the clock calculus, such as resource allocation to decrease hardware cost.

References

[1] M. Allemand, F. Bodin, A.A. Kountouris, P. Le Guernic, J-C. Le Lann, A. Seznec and C. Wolinski. A synchronous approach for hardware design. Research Report 1131, IRISA, Oct. 1997. http://www.irisa.fr/EXTERNE/bibli/pi/1131/1131.html (extended version of this paper)

[2] *IEEE standard VHDL: language reference manual.* IEEE, 1987. std 1076-1987.

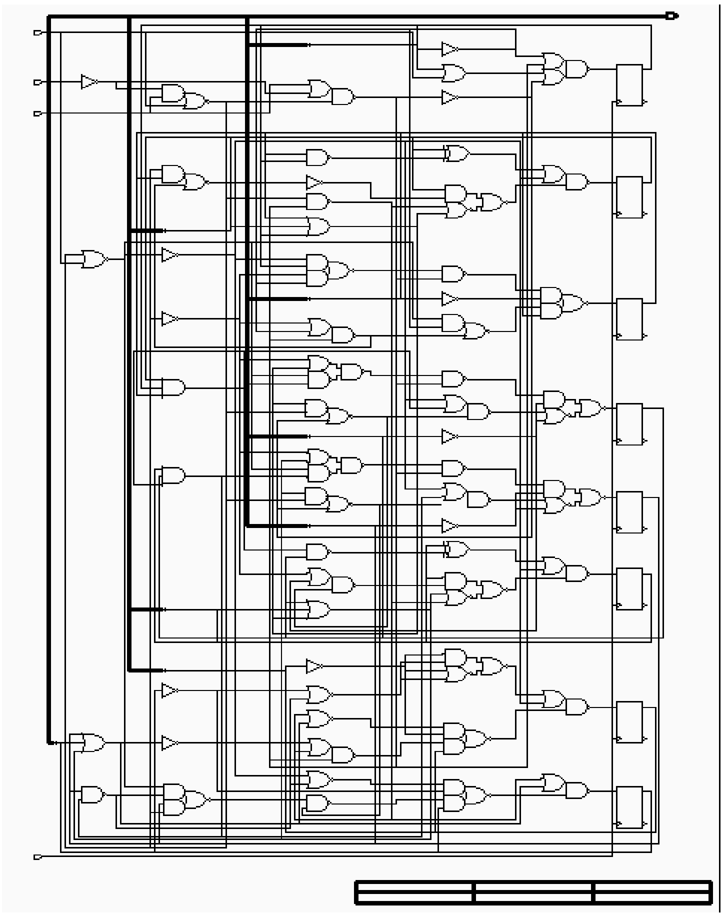


Figure 11: Synthesis of the Incr.-Decr. Counter.

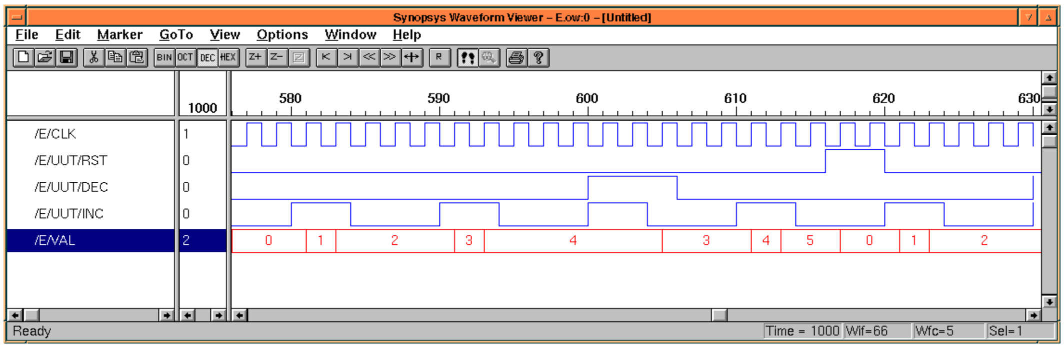


Figure 12: VHDL simulation of the Incr.-Decr. Counter.

[3] T.A. Amabegnon, L. Besnard, and P. Le Guernic. Arborescent Canonical form of Boolean Expressions. Technical Report 2290, INRIA, June 1994.

[4] F. Balarin, M. Chiodo, D. Engels, P. Giusto, and etc. Polis, A design environment for control-dominated embedded systems, Usr's Manual. Technical report, University of California, etc., December 1996.

[5] G. Berry, P. Couronné, and G. Gonthier. Synchronous Programming of Reactive Systems, an Introduction to ESTEREL. In *K. Fuchi abd M. Nivat, editors, Programming of Future Generation Computers.* Elsevier Science Publisher B.V. (North Holland), 1988.

[6] Patricia Bournai, Bruno Chéron, Thierry Gautier, Bernard Houssais, and Paul Le Guernic. Signal manual. Research Report 1969, INRIA, September 1993.

[7] Pai Chou, Ross Ortega, and Gaetano Borriello. The Chinook Hardware/Software Co-synthesis System. *Dept. of CS and Engineering, U. of Washington, Tech. Report 95-03-04*, March 1994.

[8] E. M. Clarke and Emerson A.A. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In Springer-Verlag, editor, *Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71, 1981.

[9] D.E.Thomas, J.K.Adams, and H.Schmitt. A Model and Methodology for Hardware-Software Codesign. *IEEE Design & Test of Computers*, pages 6–15, September 1993.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language LUSTRE. *Proceedings of the IEEE*, 79, September 1991.

[11] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[12] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8, 1987.

[13] G.E. Hughes and M.J. Creswell. *An Introduction to Modal Logic.* Methuen, 1977.

[14] Apostolos Kountouris and Paul Le Guernic. Profiling of signal programs and its application in the timing evaluation of design implementations. In *IEEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*, HP Labs Bristol UK, February 1996.

[15] M. Le Borgne, A. Benveniste, and P. Le Guernic. Dynamical Systems over Galois Fields and DEDS Control Problems. In *33th IEEE Conf. on Decision and Control*, volume 3, pages 1505–1509, 1991.

[16] M. Le Borgne, H. Marchand, E. Rutten, and M. Samaan. Formal Verification of SIGNAL Programs: Application to a Power Transformer Station Controller. In Springer, editor, *5th Conf. on Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 271–285, July 1996.

[17] Paul Le Guernic and Albert Benveniste. Real-time synchronous, data-flow programming : The language SIGNAL and its mathematical semantics. Internal publication 298, IRISA, Rennes, June 1986.

[18] Paul Le Guernic and Thierry Gautier. Data-flow to von neumann : the signal approach. Rapport de Recherche 1229, INRIA, May 1990.

[19] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, septembre 1991.

[20] P. LeGuernic, T. Gautier abd M. LeBorgne, and C. LeMaire. Programming real time applications with signal. *Proceedings of the IEEE*, 79, September 1991.

[21] R.K.Gupta and Giovanni De Micheli. Hardware-Software Cosynthesis for Digital Systems. *IEEE Design & Test of Computers*, pages 29–41, September 1993.

[22] Giovanni De Micheli R.K.Gupta. System Synthesis via Hardware-Software Codesign. *Computer Systems Laboratory Technical Report CSL-TR-92-548*, 1992.

[23] Inc. Synopsys, editor. *HDL Compiler for Verilog Reference Manual, Version 3.5.* February 1996.

[24] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1994.