

Timing Preserving Interface Transformations for the Synthesis of Behavioural VHDL *

P. Gutberlet¹ W. Rosenstiel²

¹Forschungszentrum Informatik (FZI)
Haid-und-Neu-Str. 10-14
D-76171 Karlsruhe

²FZI and University of Tuebingen

Abstract

As VHDL spreads widely, its usage for abstract modeling and synthesis is limited by the simulation semantics, which necessitates the specification of the interface signal transitions at bit level with exact timing. This paper shows a methodology to model the interface of a behavioural description suited for high level synthesis where different abstraction levels are separated. It shows the transformations to generate a RT data path while holding the exact simulation semantics at the interface.

1 Introduction

Within the algorithmic specification timing is often specified as the timing for the execution of syntactic blocks or groups of statements. In [Nest87] the interface behaviour is specified as the minimum or maximum delay between the single interface assignments. These constraints are specified in terms of clock cycles and are directly considered during the scheduling. [KuMi90] provide an ASAP scheduling method for minimum/maximum constrained CDFGs based on the specification language "Hardware C". [TaWo92] are using a hierarchical FSM based timing model with minimum/maximum constraints, to allow the specification at different hierarchy levels. These constraints are represented in a graph model.

To allow also the specification of asynchronous dependencies in complex protocols, more general models like Petri nets or event graphs [Borr89] are used. The interaction of all communication partners can be modelled at a very low level, e. g. in [AmBS91] the Intel Multibus protocol is represented as a graph with about 40 nodes. So these exact specifications are only suitable for special parts of a circuit.

With the increasing availability of the simulation language VHDL [IEEE88] recent approaches are using the timing concepts of VHDL to specify the behaviour. In [CaST91] a restricted subset for the purpose of synthesis was proposed. Most restrictions are dealing with the timing concepts of VHDL because only synchronous hardware is generated. In [StDu92] the timing of the circuit interface is exactly specified at the top level in terms of clock cycles. But these strongly synchronous modeling styles for VHDL are not able to consider more general dependencies with a higher granularity like asynchronous dependencies or dependencies independent from the clock level.

This paper presents a method to separate the algorithmic specification from the specification of the protocol level allowing a hierarchical design and the usage of suitable tools for the different parts. During synthesis the exact VHDL timing model is used allowing the validation of the design by simulation.

In section 2 the VHDL subset is defined and a target architecture is given to link the different parts into one synchronous data path. Also details of the timing of the interface part are given. Section 3 describes the extraction of the interface part as structural components. Section 4 describes the synthesis results.

2 Interface specification

2.1 VHDL subset definition

[GuRo93] specifies a VHDL subset used by the synthesis system CADDY. Here only a summary is given. In VHDL the specification of a circuit starts with an *entity declaration*. The circuit communicates within its environment via the signals specified there. As an example a simple processor with a bus interface with asynchronous handshake signals is used (Fig. 1).

To allow simulation or synthesis an *architecture body* has to be defined for this *entity*. The architecture may

* This work has been partially supported by DFG under RO1030 and BMFT under 01M2897B.

specify different ‘views’ (e. g. structural or behavioural) of the circuit. For the high level synthesis *processes* are used. A *process* specifies a performed algorithm with *sequential statements* like common programming languages.

The used subset is restricted to the following statements:

- *assertion statements* and *null statements* (ignored during synthesis)
- variable assignment statements
- *procedure call statements* (no recursion allowed)
- *if statements* and *case statements*
- *loop statements* (restricted to loops with one single exit point like while and for loops)
- only synchronous *wait statements* (*wait for* <multiples of the cycle time>)

Not allowed in the algorithmic part are:

- signal assignments
- *next statements* and multiple *exit statements*
- asynchronous *wait statements* (*wait on*, *wait until* and asynchronous *wait for*)

```
entity processor is
  port (
    -- clock signal
    clock      : in    BIT;
    -- signals for sample bus protocol
    bus1_req   : out   BIT; -- bus request
    bus1_ack   : in    BIT; -- bus acknowl.
    bus1_rd    : out   BIT; -- read strobe
    bus1_wr    : out   BIT; -- write strobe
    bus1_data  : inout BIT_VECTOR(0 to 15);
    bus1_addr  : out   BIT_VECTOR(0 to 15)
  );
end processor;
```

Fig. 1: example entity

2.2 Specification style

The main restriction concerning the interface is the prohibition of direct signal accesses. This restriction is leading to an interface specification style where the access to the interface must be hidden in procedures.

This formal separation allows now the use of suitable synthesis tools for the specific parts, where only the main algorithm is the direct domain of the behavioural synthesis. The interface procedures are not restricted to the described subset allowing more general dependencies, which can not be considered directly by a synchronous data path. Typical interface procedures are therefore invocations of a specific protocol e. g. a complete bus access.

2.3 Timing specification at the algorithmic level

According to the VHDL semantics the algorithm executes in zero time and only at *wait statements* time is

consumed. *wait on* and *wait until statements* are used for external synchronizations e. g. within protocols, so without the access to interface signals in the algorithmic part, only *wait for statements* are useful and allowed. Using these *wait for statements* (e. g. *wait for 150ns*), the performance is specified independently from the clock frequency allowing the automatic optimization of this parameter. These *wait statements* are treated as synchronous, that means that the clock cycle time has to be selected as an integer divisor of all occurring time values where the *wait statements* may now be interpreted as a certain number of control steps. The interface behaviour at this level is fully determined by the control steps in which the interface procedures are called. This is now a scheduling problem which can be handled by the high level synthesis.

Replacing the absolute timing value in the *wait for statements* by generic parameters allows the specification of indetermined timing. These generic parameters can be used as timing constraints and provide also a method of backannotation after synthesis. This additional feature is also described in detail in [GuRo93].

2.4 Specification of interface procedures

The access to the interface signals is only allowed in the interface procedures, so the circuit communicates with its environment by calling these procedures.

Fig. 2 lists a typical interface procedure which may be suited for a processor to access global interface signals (*bus1_req*, *bus1_ack*, *bus1_addr*, *bus1_data*) and to handle a complete access to an external RAM. To allow access to the global signals the procedure must be declared within the *block* containing the *process body*. As an alternate style the interface signals to use may be declared as formal parameters which is suited to store the procedure in a library.

```
procedure bus1_read (
  addr: in    integer;
  data: out   integer ) is
begin
  bus1_req <= '1';
  wait until (bus1_ack = '1');
  bus1_addr <= int2bitvec(addr);
  wait for 5ns;
  bus1_rd <= '1';
  wait for 30ns;
  data := bitvec2int(bus1_data);
  bus1_rd <= '0';
  bus1_req <= '0';
  wait until (clock = '1');
end bus1_read;
```

Fig. 2: example interface procedure

Within these procedures the timing is not necessarily bound to control steps, so protocols with a finer granularity than clock cycles or asynchronous dependencies may be specified. As this procedure is not synthesized directly during the high level synthesis, it is not restricted to the defined subset. But one necessary condition is that all interface procedures must terminate with the statement "wait until (clock = '1');" to keep the global time in the process body synchronized with the clock signal.

2.5 Target architecture

Fig. 3 shows a typical data path as the result of the high level synthesis. The specified operations (additions, subtractions) are executed on hardware components (adders, ALUs). Variables are stored in registers and the values are transported from their origin to their destination via bus drivers, buses and multiplexors. Multiplexors, registers, bus drivers and ALUs are controlled by a synchronous FSM which starts the correct operations with the correct data in the correct clock cycle under the correct conditions.

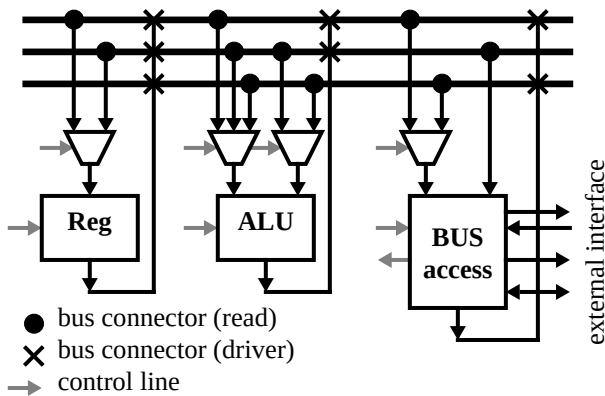


Fig. 3: target architecture

For the interface synthesis also the calls of the interface procedures are mapped on hardware elements, which are referred as interface components. Interface components are connected with the data path like the other components. But they are also connected to the external environment and are building the connection between the internal data path and the external pins of the circuit. An interface component has therefore two parts (Fig. 4).

The first part contains the interface to the data path and is synchronous. Only this interface must be considered by the high level synthesis system. It consists of the ports `data_in` and `data_out` which are the corresponding input and output parameters of the interface procedure. To allow the necessary synchronization there are two additional control signals. The `start` signal is provided by the controller of the data path to initiate an operation. If the

completion time of the operation is due to external synchronization not fixed, the `ends` signal is used to synchronize the controller of the circuit with the operation. If the component executes different procedures (e.g. `bus_read` and `bus_write` using the same interface signals) a mode signal is provided to select the correct procedure.

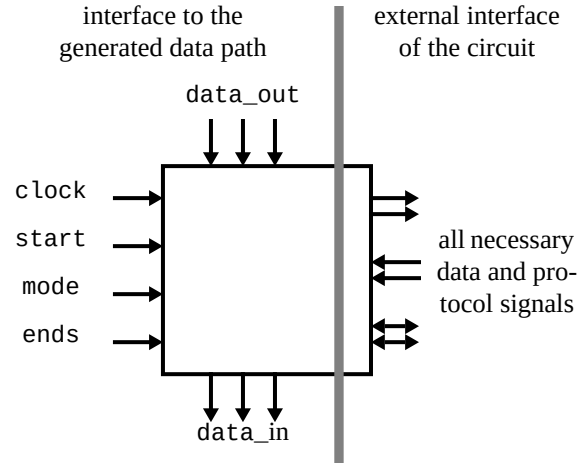


Fig. 4: interface component

The exact timing of these signals can be taken from (Fig. 5). The `start`, `mode`, and the `data_out` signals are provided by the data path and its controller and are synchronous signals. This means that they are only valid a certain time before the positive clock transition. The `mode` and `data_out` signals are further only valid if the operation is started. The `ends` and `data_in` signals are provided by the interface component and must be generated according to the clock. Because the `ends` signal is used by the circuit's controller it must be stable a certain time before the clock transition. If the `ends` signal is set, the `data_in` signals must be stable, so they may be loaded with the next positive clock transition into data path registers.

The second part contains the external interface of the circuit. It contains all signals for the specific protocol. The interface component handles the complete protocol without further interaction with the data path. So the interface component may contain internal registers and a local controller.

3 Interface synthesis

The interface synthesis consists of two parts: the synthesis of the data path with the specified cycle by cycle behaviour [GuRo93], and the extraction of the interface components which is described here.

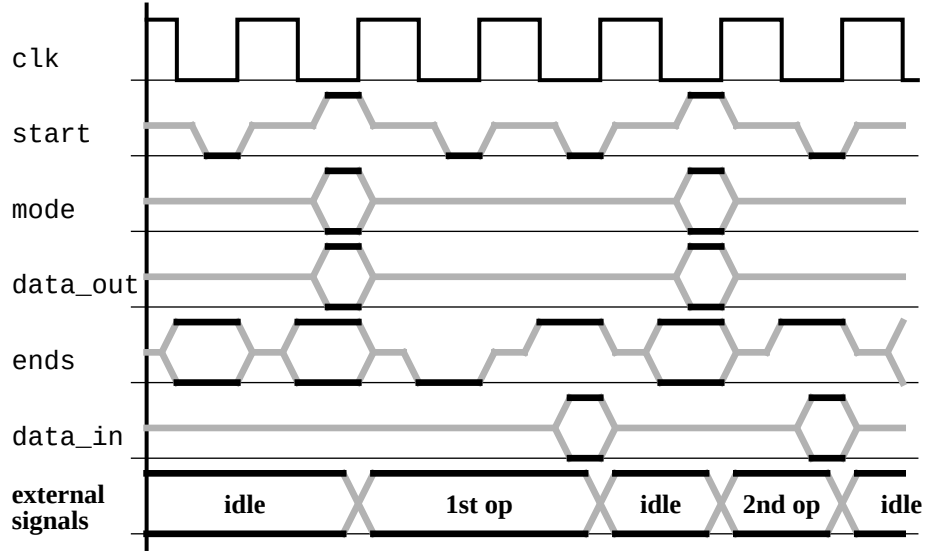


Fig. 5: timing diagram of an interface component

3.1 Allocation of interface component

In the allocation a set of interface components has to be selected to perform the required procedures. One interface component is associated with one logical port of the circuit, where a logical port is a set of interface signals which is used by a common protocol. If multiple operations are using the same port like read and write operations on a common bus, they are mapped to one component. Such a component is modelled as a multifunctional unit and used like an ALU type by the synthesis system.

In the first step signals which occur in a procedure parameter list are expanded. This means that the signals in the parameter list are deleted and their occurrence in the procedure body is replaced by the actual parameters of the procedure call. If a procedure is called with different sets of actual signal parameters, multiple instances for the procedures are generated.

After this step the procedures are accessing only global signals. Then for every procedure the set of accessed signals is extracted. Let Proc_p be the single interface procedures ($p = 1, \dots, P$) and let S_p be the set of all interface signals accessed by procedure Proc_p . Then we define an relation “ $\sim$ ” between the procedures with (F1) and (F2).

$$\text{Proc}_p \approx \text{Proc}_q : \Leftrightarrow S_p \cap S_q \neq \emptyset \quad (\text{F1})$$

$$\text{Proc}_p \sim \text{Proc}_q : \Leftrightarrow \text{transitive closure of (F1)} \quad (\text{F2})$$

The interface components can now be associated with the equivalence classes of the relation “ $\sim$ ” and an interface component executes exactly the procedures in its associated class.

3.2 Entity generation

Now the ports of the interface component are generated as a VHDL *entity*. It consists of the signal clock, the control signals start, mode, and ends, the data signals which are the parameters of all interface procedures executed by the component, and the external signals which are the interface signals accessed by these procedures. (Fig. 6) shows the *entity* generated for the example.

```

type bus1_comp_modes is (call_bus1_read,
...);

entity bus1_comp is
port(
    clock      : in    BIT;
    -- synchronization signals
    start      : in    BIT;
    mode       : in    bus1_comp_modes;
    ends       : out   BIT;
    -- data signals
    addr       : in    BIT_VECTOR(0 to 15);
    data       : out   BIT_VECTOR(0 to 15);
    -- external signals
    bus1_req   : out   BIT;
    bus1_ack   : in    BIT;
    bus1_rd    : out   BIT;
    bus1_wr    : out   BIT;
    bus1_data  : inout  BIT_VECTOR(0 to 15);
    bus1_addr  : out   BIT_VECTOR(0 to 15)
);
end entity;
```

Fig. 6: generated component entity

This *entity* is then used as a data path component. Within the data path (Fig. 3) the control signals are connected to the controller, the data signals are connected to the data path and the external signals are connected to the data path's interface.

3.3 Architecture generation

For simulation of the resulting data path and for further synthesis, a VHDL architecture specification for the interface components is generated. This is done based on the procedure specifications. In the procedural specification the synchronization is performed with the VHDL *procedure call* mechanism and the data flow is specified by the parameter mechanism. In the data path the interface component is now a separate VHDL *entity* where the synchronization has to be replaced by the protocol with the start, mode, and ends signal (Fig. 5). The parameter mechanism has to be replaced by the synchronous communication in the RT-structure.

To use widely the procedure code each component is specified as a separate VHDL *process*. The process frame in (Fig. 7) performs the necessary synchronization.

```
architecture behaviour of bus1_comp is
begin
process
begin
wait until clock'event AND
(clock = '1') AND (start = '1');
ends <= '0';
case mode is
when code_bus1_read =>
-- bus1_read(addr, data);
-- the procedure code
-- is expanded here
when ...
end case;
ends <= '1';
end process;

end behaviour;
```

Fig. 7: generated component architecture

First the procedure call mechanism is translated into the first *wait statement*. Notice that the process starts only if the start signal is set during an rising clock transition. After this *wait statement* the mode signal and all used data signals (addr) are valid. If a component executes different procedures a unique code is generated for every procedure during the component assignment. This code is then applied as mode signal which is used to switch between different branches in a *case statement*. In every case branch the appropriate interface procedure is copied where the last

wait statement of the procedure specification is left out. This code handles the interface access in the same way as the original procedure code. After it is completed the signal ends is set and the interface component is now ready to start the next operation.

At the next positive clock transition the data path controller leaves a wait state and resumes the tasks of the main algorithm. This is exactly the behaviour of the interface procedure including the last *wait statement*.

4 Results

At this stage the RT structure of the data path generated by the high level synthesis together with the generated process description of the interface components are building a complete specification which may be simulated and synthesized at lower levels. (Fig. 9) shows two simulation charts for the clock rates 25 MHz and 50 MHz based on the specification in (Fig. 8). Each chart describe the simulation result of the specification as well as the data path. The example shows one complete and the start of the second calculation.

```
architecture behaviour of gcd is
begin
process
begin
... -- set up adr1, adr2, adr3
bus1_read(adr1, d1);
bus1_read(adr2, d2);
while d1 /= d2 loop
... -- calculations
wait for 40 ns;
end loop;
wait for 40 ns;
bus1_write(adr3, d1);
end process;

end behaviour;
```

Fig. 8: specification gcd

Because 40 ns is a multiple of both clock rates, the main algorithm keeps synchronized with the clock and this behaviour is preserved in the synchronous data path. Only within the interface procedures/components the time is not necessarily bound to clock transitions.

The two data paths may be quite different and have different numbers of states. But the overall performance is nearly equal, because during the scheduling exactly the number of states is generated to fit the specification. The slight differences in the two charts are caused by the resynchronization after the completions of the interface procedures/components.

To show that the specification and the synthesized data path have exactly the same timing behaviour at the interface two steps have to be considered. In the first step it can be shown that the interface procedure calls in the specification occur at the same cycle as the start of the interface components, which is a solved scheduling problem. In the second step it can be shown that the interface components behave exactly like the procedures in the specification, because the procedure body has been copied into the component's *process specification* by the construction method of the interface components.

Finally the interface components have to be synthesized based on their process specification. For this part special tools with asynchronous techniques may be used. The subset for the specification depends then on the used synthesis system.

In the current system, interface components for some bus protocols which were designed manually together with the procedures are held in an interface library.

5 Conclusion

A methodology to specify the interface behaviour of VHDL *processes* was presented. This allows now the integration of interface parts with complex protocols into the high level synthesis and extends the application domain to the synthesis of embedded systems.

The exact timing behaviour of the VHDL specification is bewared during the high-level synthesis allowing the validation of the behaviour and timing by simulation in early design steps.

6 References

- [AmBS91] T. Amon, G. Borriello, C. Séquin, "Operation / Event Graphs: A Design Representation for Timing Behavior", Proc. of the 10th International Symposium on Computer Hardware Description Languages and their Applications, 1991
- [Borr88] G. Borriello, "Combining Event and Data-Flow Graphs in Behavioral Synthesis", Proc. of the ICCAD, 1988
- [CaST91] R. Camposano, L. F. Saunders, R. M. Tabet, "VHDL as Input for High-Level Synthesis", IEEE Design & Test of Computers, March 1991
- [GuRo93] P. Gutberlet, W. Rosenstiel, "Interface Specification and Synthesis of VHDL Processes", Proc. of the 2nd EURO-DAC, 1993
- [IEEE89] "IEEE Standard VHDL Language Reference Manual", IEEE Std 1076-1987, March 1988
- [KuMi90] D. Ku, G. De Micheli, "Relative Scheduling under Timing Constraints", Proc. of the 27th DAC, 1990
- [Nest87] J. A. Nestor, "Specification and Synthesis of Digital Systems with Interfaces", Research Report No. CMUCAD-87-10, Carnegie-Mellon University, 1987
- [StDu92] A. Stoll, P. Duzy, "High-Level synthesis from VHDL with exact timing constraints", Proc. of the 29th DAC, 1992

[TaWo92] A. R. Takach, W. Wolf, "Modular Scheduling Constraint Specification", Proc. of ISCAS, 1992

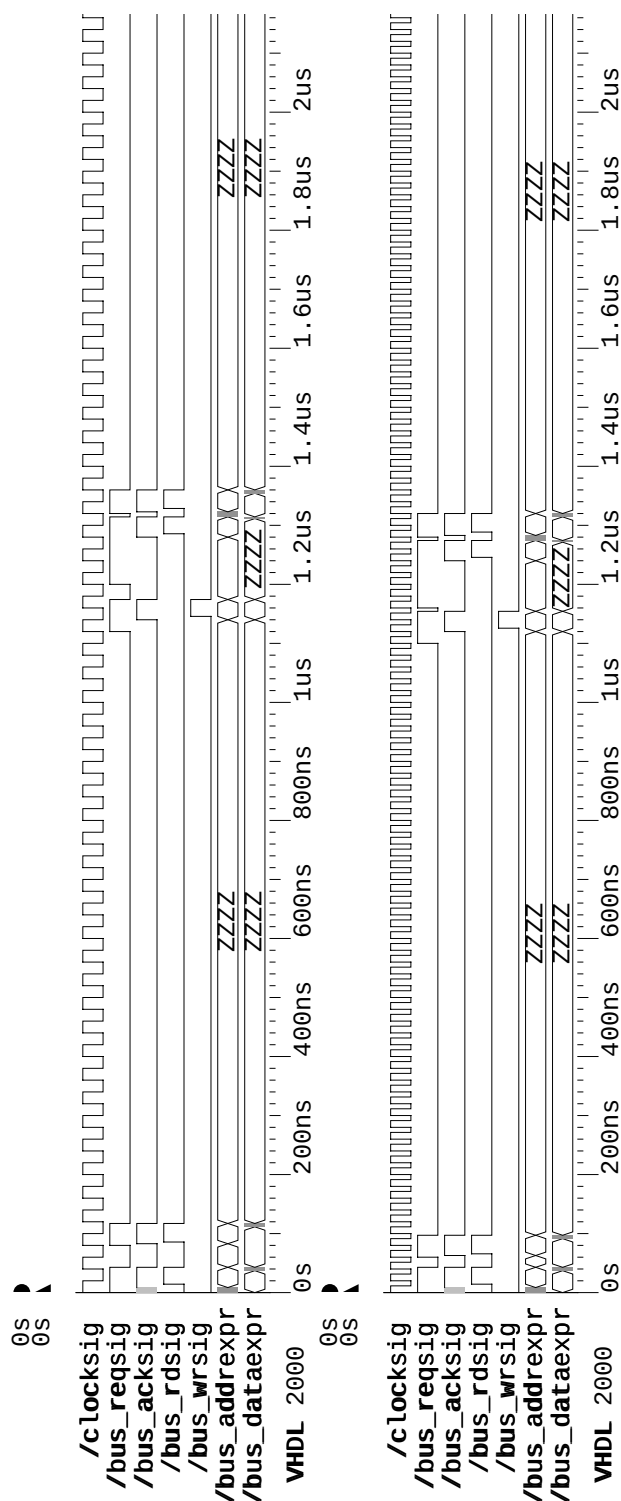


Fig. 9: Simulation Results at 25MHz and 50MHz