Protocol Merging: A VHDL-Based Method for Clock Cycle Minimizing and Protocol Preserving Scheduling of IO-Operations

W. Ecker, M. Glesner, A. Vombach
Corporate Research and Development – Institut für Datentechnik
Siemens AG – TH Darmstadt, Germany

Abstract

This paper presents a new algorithm for clock cycle minimizing and protocol preserving scheduling of input and output operations. The algorithm relies on the possibility to overlap protocols at different ports in time without changing the behavior. It merges the operations required for complete protocols and allows thus for a compact schedule of a set of correlated protocols. Both, input and scheduled output for subsequent High-Level- and RT-Synthesis use VHDL for description.

Introduction

Hardware consists in general of a set of concurrent interacting processes. The interaction is based on protocol-based communication and synchronization. In an early design phase, eg. on system-level, abstract protocols are used, which specify causality only. Hence, a temporal overlap of protocols can not be specified at this point in the design process.

When synthesizing the early specification to RT-level, a clock scheme has to be introduced to allow for a clock-related timing specification. Also, process interaction has to be mapped on concrete clock-based protocols. A simple replacement of the abstract protocols by clock-based protocols does not destroy the sequence of the operations and thus does not allow for overlapping of protocols. Thus, a solution resulting from this design steps is not optimal, since a temporal overlap of protocols allows to reduce the number of clock cycles in the resulting hardware.

Methods as implemented in high-level synthesis tools (see eg. [1, 2, 3, 4]) typically do not attack this problem. A simple export of the protocols in concurrently executed statements or processes also does not solve the problem. This would not allow for overlapping protocols in a cyclic way or would also require additional protocol-based synchronization between communicating, synchronized processes and other processes.

The presented scheduling algorithm, named protocol merging, solves this problem and generates a schedule with overlapping protocols. Due to the fact, that high-level and RT-level synthesis tools, which support in most cases VHDL (see [5]) as input language, are used to reach the gate level, VHDL is used for both as input language and as basis for the merging tool. The schedule is then performed modifying the order of sequential VHDL-statements.

The paper is organized as follows: The first section illustrates the idea of protocol merging by an example and shows an approach for the algorithm. A pseudo code specification of the merging algorithm and an overview over the program structure are shown in Section 2. Features of the algorithm are discussed and compared with other approaches afterwards. The application of the algorithm is shown in Section 4, and results are presented in Section 5.

1 The Merging Mechanism

1.1 A Handshake Protocol

In order to introduce the basic mechanism, we use a transmitter which receives and sends data via parallel data transmission and which protects the data via handshake. The VHDL source code of the transmitter is printed in Listing 1. The sequence of statements shown in Listing 1 reflects the order of the more complex operators `send` and `receive`. This sequence results either from the transformations of a system-level specification or from the inline expansion of subroutines (see Listing 6). Like in all behavioral RT-level VHDL descriptions, in the VHDL description in Listing 1 controlflow-related wait statements are used to specify the clock-related timing.

The waveform for the transmission of one datum is shown in Figure 1. Here, after the second clock edge a datum lies at the port `data_in`. This is signalled by the value `'1'` at port `ok_in`. Some (in this case one) clock cycle(s) later, port `ack_in` is set to `'1'` by

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission. (c) 1994 ACM 0-89791-687-5/94/0009 3.50

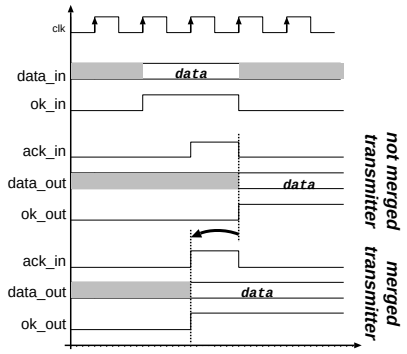

Figure 1: Waveform of a Transmitter

the transmitter, signalling that the datum has been received correctly. One clock cycle later the transmitter sets port `ack_in` to `'0'` which finishes the `receive` operation at the ports belonging to the transmission channel `in`.

The `send` operation starts immediately after the `receive` operation by putting the received data to port `data_out` and setting port `data_ok` to `'1'`. Then Port `ok_out` is polled until it gets the value `'1'`. This finishes the `send` operation.

```
transmitter: process
   variable buf : bit_vector( 7 downto 0 );
begin
   loop                                -- receive
      exit when ok_in = '1';
      wait until clk = '1' ;
   end loop ;
   buf := data_in;
   ack_in <= '1';
   wait until clk = '1';               -- receive
   ack_in <= '0';                      -- e n d
   ok_out <= '1';                      -- send
   data_out <= buf;
   wait until clk = '1';
   loop
      exit when ack_out = '1';
      wait until clk = '1';
   end loop ;
   ok_out <= '0';
end process ;
```

Listing 1: The Transmitter before Merging

1.2 The Basic Idea

The schedule of the transmitter could be improved by one clock cycle, if acknowledgment is given on the transmission channel `in` and data are put to transmission channel `out` in the same clock cycle. This can be achieved by *merging* the tail of the `receive` operation and the head of the `send` operation as shown in Listing 2.

```
transmitter: process
   variable buf : bit_vector( 7 downto 0 );
begin
   loop                                -- receive
      exit when ok_in = '1';           -- b e g i n
      wait until clk = '1' ;
   end loop ;
   buf := data_in;
   ack_in <= '1';
   ok_out <= '1';                      --< merged
   data_out <= buf;                    --< send
   wait until clk = '1';               --< a n d
   ack_in <= '0';                      --< receive
   loop                                -- send
      exit when ack_out = '1';         -- r e s t
      wait until clk = '1';
   end loop ;
   ok_out <= '0';
end process ;
```

Listing 2: The Merged Transmitter.

In this case, merging is performed by moving the tail statements of the receive protocol (here: `ack_in <= '0';`) behind the next wait statement. The wait statement, which is part of the tail of the receive protocol is deleted.

Moving statements and deleting wait statement is the basic mechanism underlying the merging algorithm presented in this paper.

1.3 Refinements

Protocol merging can be refined, if conditionally executed wait statements are also taken into account. It has to be considered, however, that the statements, which are placed behind a conditionally executed wait statement are also executed conditionally. Hence, the statements of the tail of a protocol, which are placed behind a conditionally executed wait statement are executed conditionally. This implies, that the tail of the protocol is executed conditionally and thus that a protocol may not be finished completely. A mechanism has to be introduced to solve this problem.

Since a protocol must be finished before it is executed again, further statements must be inserted before the protocol. These statements, however, may be executed only if the protocol has not finished. Hence, a flag is introduced to control the execution of these additional statements. This flag is set when the protocol starts and reset when the protocol has already finished during the execution of a wait statement. This is shown for the receive operation in Listing 3. Here, `sel='1'` represents any not constant condition, whatever.

```
transmitter: process
   variable buf : bit_vector( 7 downto 0 );
   variable receive_ready : boolean := TRUE;
begin
   if receive_ready = FALSE then    -- force to finish
      wait until clk = '1'          -- receive
```

```
      ack_in <= '0';                -- conditionally
   end if;
   loop                             -- receive
      exit when ok_in = '1';
      wait until clk = '1' ;
   end loop;
   buf := data_in;
   ack_in <= '1';
   receive_ready := FALSE;
   if sel = '1' then                -- finish
      wait until clk = '1';         -- receive
      ack_in <= '0';                -- conditionally
      receive_ready := TRUE;
   end if;
   -- ...
end process;
```

Listing 3: Conditionally Executed Statements

Conditionally executed tail statements of a protocol, like the tail of the send operation shown in Listing 1, can be merged similarly. Moreover, the tail of a protocol can be merged in this way, if it requires more than one cycle. To accomplish this, a counter is introduced instead of the flag, which signals how many cycles of the end of the protocol still have to be executed.

Applying this refinement, the scheduling algorithm is able to schedule protocols into branches and loops as well.

2 The Merging Algorithm

2.1 Program Structure

The source code transformation required for protocol merging is not performed on the source code directly but on an intermediate format (see [6] and 2.2), which is generated by a VHDL analyzer from a VHDL source code file. A generator allows to rebuild a VHDL source code file from the intermediate format. The merge utility consists of three tasks performed on a copy of the data structure.

- In the first step all procedure calls are inline expanded. This flattens the description, makes all wait statements visible and allows to merge protocols encapsulated in procedures.
- Merging is performed as second task. The algorithm is described in Section 2.3 in more detail.
- The final task is wait folding (see [7]) with an extension to allow for straight-forward false-path analysis. It compiles a multiple wait-statement description in a single wait-statement description without changing the behavior.

A VHDL description is finally generated from the intermediate format produced by wait folding. This VHDL description can be used for both High-Level and RT-synthesis as well as for validation of the scheduled protocols and timing constraints.

2.2 Data Structure

A textual VHDL process description is presented by a directed, cyclic, hierarchical control flow graph $CFG(V, E)$ with $V = V_L \cup V_P \cup V_C \cup V_W \cup V_S$ and $E = E_C \cup E_V$.

Since VHDL'87 does not allow for setting labels on all sequential statements, a set of predefined dummy procedures are used as labeling statements. V_L is the set of nodes representing labeling statements (see 4.1). They are necessary to specify the heads and tails of a protocols' CFG representation. Nodes $\in V_L$ are also used to specify the beginning of the mergable part of the protocol in the CFG. Thus, V_L can be composed as follows: $V_L = V_{LB} \cup V_{LE} \cup V_{LM}$, where V_{LB} and V_{LE} are the sets of labeling nodes specifying the beginning and the end of a protocol respectively and V_{LM} is the set of labeling nodes specifying the beginning of the mergable part of the protocol.

V_P and V_C are nodes representing sub-CFGs. All nodes $\in V_P$ represent procedure calls and all nodes $\in V_C$ represent conditional statements. Loop statements are treated as conditional statements. Nodes $\in V_S$ represent the reminder of the sequential statements like signal assignment statements, variable assignment statements or null statements.

Time specifications in synchronous designs are clock related only. The simple wait statement `wait until clk = '1'` or equivalent statements, which specify the timing offset for one clock cycle, are used in sequential or behavioral VHDL descriptions for this reason [1]. These wait statements constitute the set V_W. More complex wait statements like `wait on clk until clk = '1' and en = '0'` must be built from loop statements and simple wait statements to enter the protocol merging step.

Edges $\in E$ represent the control flow. The subset $E_C \subset E$ specifies the execution order based on the statement sequence according to the text. E_V models control flow dependencies underlying conditional statements or the infinite process loop.

2.3 Algorithm Kernel

The protocol merging algorithm consists of three parts (see Listing 4): The first part identifies the mergable protocols in the description, collects all wait statements and counts all unconditionally executed wait statements. In this way, a frame for the schedule is generated. The second part cuts all mergable parts

[1] It is important to note, that the wait-statement specifies the schedule of the operations for RT-synthesis, too.

from the description, selects them in a list and introduces statements which force the execution of the protocol tail if necessary. The third part schedules parts of the protocols by inserting all mergable parts of a protocol behind a detected wait statement.

The node sets defined in 2.2 are represented as ordered sets or lists for performance reasons. The order is defined according to the sequential notation of the statements.

```
procedure Merging( CFG: P ) is
    Natural: W := 0;
    List : V_LB, V_LE, V_LM, V_W := ∅;
    ListOfCFG : M := ∅;
begin
    IdentifyProtocols ( P, V_LB, V_LE, V_LM, V_W, W );
    CutProtocolMergs ( V_LB, V_LE, V_LM, M, V_W, W );
    InsertProtocolEnds( M, V_W );
end Merging;

procedure IdentifyProtocols
    (CFG:P;List:V_LB,V_LE,V_LM,V_W;Natural:W)
is
    Node : n;
begin
    for all n ∈ P loop
        case kind( n ) is
        when ProtocolBeginLabel     => V_LB = V_LB ∪ N;
        when ProtocolEndLabel       => V_LE = V_LE ∪ N;
        when MergeBeginLabel        => V_LM = V_LM ∪ N;
        when WaitStatement          =>
            V_W = V_W ∪ n;
            if IsUnconditionalStatement( N ) then
                W := W + 1;
            end if;
        end case;
    end loop;
    for all n ∈ P loop
        if kind( n ) = ConditionalStatement then
            IdentifyProtocols ( SubCFG(n),
                V_LB, V_LE, V_LM, V_W, W );
        end if;
    end loop;
end IdentifyProtocols;

procedure CutProtocolMergs
    ( List:V_LB,V_LE,V_LM,V_W;ListOfCFG:M,Natural:W)
is
    Node : n_B, n_E, n_M;
    Node : n;
    CFG : m;
begin
    for all n_B ∈ V_LB, n_E ∈ V_LE, n_M ∈ V_LM loop
        m = CutSubCFG( n_M, n_E );
        if W = 0 then
            InsertSubCFG( m , n_B);
        else
            for all n ∈ m loop
                if kind( n ) = WaitStatement then
                    V_W = V_W \ n;
                end if;
            end loop;
        end if;
        DeleteWaitStatement( m );
        M := M ∪ m;
    end loop;
end CutProtocolMergs;

procedure InsertProtocolEnds
    ( ListOfCFG:M; List:V_W )
is
    Node : n;
    CFG : m;
begin
    for all n ∈ V_W loop
        for all m ∈ M loop
            AppendSubCFG( m , n );
        end loop;
    end loop;
end InsertProtocolEnds;
```

Listing 4: The Merging Algorithm

The complexity $\mathcal{C}$ of the merging algorithm can be formulated as

$$\mathcal{C} = |\mathcal{S}| + |\mathcal{P}| + |\mathcal{W}| * |\mathcal{P}| = O(|\mathcal{S}|^2)$$

considering the three passes of the merging algorithm. Here, $|\mathcal{S}|$ is the number of statements, $|\mathcal{P}|$ is the number of statements in the protocols and $|\mathcal{W}|$ is the number of wait statements. The formula shows, that the complexity of the merging algorithm is in worst case quadratic in number of processed VHDL statements.

The merging algorithm was implemented in C++ based on the procedural interface to the intermediate format (see [6]). It consists of about 500 lines of code. The CPU-time is neglectable in comparison to the subsequent high-level or RT-level synthesis steps.

3 Comparison with other Approaches

Methods as implemented in high-level synthesis tools (see eg. [1, 2, 3, 4]) typically do not attack the freedom of possibly overlapping protocols. Either the number of time slots (= number of required clock cycles) is minimized, or area and propagation delay are minimized subject to pre-scheduled IO-operations. In the first case the optimization of concrete protocols is inhibited, in the second case adjustments to the timeslots of I/O operations can not be made.

Different implementation alternatives for subroutines have been presented in [8]. None of the presented alternatives, allow for overlapping the subroutines. The same authors proposed in [9] a method for reduction of wire overhead by partial serialization but not by overlapping of protocols. Another approach for minimizing wires was presented in [10]. This approach, however, focusses on detection and removal of unnecessary synchronization lines and optimization of the according control logic.

A special approach in high-level synthesis, the dataflow oriented scheduling under consideration of relative time constraints, like presented eg. in [11] or [12], may generate some overlap of protocols. However, this approach optimizes level triggered protocols

equal to edge triggered protocols due to dataflow oriented schedule and wastes in this way one clock cycle per protocol.

Listing 5 and Figure 2 illustrates this fact. The wait-statement in listing 5 specifies a time constraint of one clock cycle between `ack <= '1'` and `ack <= '0'`. Due to data dependency, scheduler, as referenced above, add an additional time constraint between `ack <= '0'` and `ack <= '1'`, assuming that `ack` must hold the value `'0'` at least one clock cycle. This allows for an acknowledgment each two clock cycles only. Thus, level-sensitive protocols can not be fully optimized by this kind of scheduler, since these protocols support an acknowledgment each clock cycle (see Figure 2). The presented merging algorithm however is able to optimize both protocols using predefined labels and controlflow-oriented schedule instead of dataflow oriented schedule.

```
ack <= '1';
wait until clk = '1';
ack <= '0';
```

Listing 5: Specification of an Acknowledge-Signal

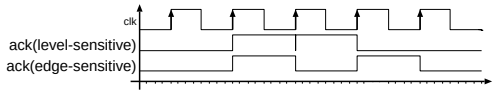

Figure 2: Acknowledge of Level- and Edge-Sensitive Protocols

Moreover, the presented merging algorithm allows for scheduling of not loop invariant operations into branches and loops with unknown number of loop iterations, which can not be performed by high-level synthesis algorithms.

4 Algorithm Application

4.1 Protocol Specification

The label `merg_this`, which shows the begin of the mergable part of a protocol, has to be set in the protocol specification to allow for automatic merging. Additional marks are necessary to mark the head (`proc_begin`) and the tail (`proc_end`) of a protocol. Since VHDL'87 does not generally allow for labeling sequential statements, dummy procedures are introduced for labeling. The recognition of the mergable part can not be performed automaticly due to the fact, that detailed information about the protocol is required as shown in Section 3.

circuit	original	inlined	merged	folded
rec_send	13	32	43	60
rec_send1	10	33	52	77
sem_rec	12	44	78	140
semtrans	22	79	503	375
semtrans1	25	79	503	384
split	23	58	58	84
tee	18	53	193	231
transmitter	29	56	91	114
xii	42	95	621	799

Table 1: Analysis of the optimized VHDL-Code

Listing 5 shows the specification of a receive operation based on a hand-shake protocol with parallel data transmission. Here, the dummy procedures `proc_begin`, `proc_end` and `merge_this` are used to mark the begin, the end and the mergable part of the protocol.

```
procedure receive
   ( signal   clk  : in bit;
     signal   data : in bit_vector;
     signal   ok   : in bit;
     signal   ack  : out bit;
     variable buf  : out bit_vector ) is
begin
   proc_begin;                --< mark begin of the protocol
   loop
      exit when ok = '1';
      wait until clk = '1' ;
   end loop;
   buf := data;
   ack <= '1';
   merge_this;                --< mark mergable part
   wait until clk = '1';
   ack <= '0';
   proc_end;                  --< mark end of the protocol
end receive;
```

Listing 5: Receive Operation

4.2 Protocol Application

Protocols specified as subroutines (see 4.1) allow to reuse protocol specifications and to hide merging labels from the user. In this way, the description of the transmitter example can be reduced to two subroutine calls. This is shown in Listing 6.

```
transmitter: process
   variable buf : bit_vector( 7 downto 0 );
begin
   receive( data_in , ok_in, ack_in, buf, clk );
   send( data_out , ok_out, ack_out, buf, clk );
end process;
```

Listing 6: Subroutine-Based Transmitter

5 Results

Table 1 shows the lines of VHDL-Code after each transformation step. In general, the final description after optimization has 4 up to 20 times more lines

circuit	not merged			merged		
	delay	area	cycles	delay	area	cycles
rec_send	5.78	135	4	5.60	126	3
rec_send1	5.49	136	7	6.02	138	6
sem_rec	5.48	139	2	5.37	139	1
semtrans	7.96	195	4	10.79	320	3
semtrans1	7.39	189	4	11.21	314	3
split	5.81	198	4	5.80	198	3
tee	5.67	181	6	6.83	209	5
transmitter	17.58	319	18	12.02	312	9
xii	17.62	607	36	28.17	3241	20

Table 2: Analysis of Synthesis Results

of code than the input description. This illustrates that it is nearly impossible, to perform the optimization manually. Two unexpected effects can also be observed: The number of lines may not increase after merging and the number of lines of code may decrease after wait-folding. The first effect occurs, if a lot of wait-statements have to be executed unconditionally and the second effect can be observed, if a lot of false paths are part of the description after merging.

Table 2 compares properties of circuits optimized with or without merging. All circuits are generated with a commercial RT-level synthesis tool. In all cases the number of clock cycles was reduced by applying protocol merging. A rough analysis did not allow to identify a correlation of the merging optimization with the propagation delay and area. A more detailed analysis results in the following assumptions:

A smaller and faster circuit can be achieved, if the circuit, which has to be optimized, possesses unconditionally executed wait statements. The reason is, that merging minimizes in this case the number of wait statements in the description and thus the number of states of the implicitly inferred finite state machine. Hence, the number of registers required for the hardware implementation can be smaller.

The size of the circuits is 1.5% to 40% larger, if no or only some unconditionally executed wait statements are part of the description before the merging optimization step. The area increase results mainly from an increasing number of states of the implicitly inferred finite state machine. New states, however, are forced by new conditionally executed wait statements, introduced by protocol merging.

The merging of more than one clock cycle generates circuits, which are about five times as big as the non-optimized circuits. The reason is the introduction of counters, which are necessary to count the number of executed respectively not executed cycles of a protocol. This overhead, however, is no weakness of the algorithm. It results from the optimization problem itself.

The difference in propagation delay in the merged and un-merged circuits relates to the differences in area. We observed, however, that the increased propagation delay satisfies in most cases the clock requirements.

6 Conclusion

A new method for clock cycle minimizing and protocol preserving scheduling of IO-operations with quadratic complexity was presented. The algorithm allows for optimizing edge triggered as well as level triggered protocols, which can not be handled by scheduling algorithms up to now. Finally, the scheduling algorithm is able to schedule operations in branches and loops with unknown number of iterations. It has to be pointed out however, that the algorithm was developed for protocol optimization only and not for resource minimization of complex operations.

Currently, a mechanism is included in the algorithm to allow for finishing protocols at specified places and thus to allow for data considering dependencies between different protocols and statements. It is planned to support labels on sequential statements instead of labeling dummy subroutines, as soon as the VHDL frontend supports VHDL'92.

Future effort lies on the optimization of multicycle merging and the improvement of the false path-analysis.

Acknowledgments

We would like to thank Michael Gasteiger, Wolfgang Glunz, Sabine März and Norbert Wehn for their support and for helpful discussions.

References

[1] P. Michel, U. Lauther, and P. Duzy. The Synthesis Approach to Digital System Design. Kluwer Academic Publishers, 1992.

[2] D.D. Gajski, A. Wu, N. Dutt, and S. Lin. High-Level Synthesis. Kluwer Academic Publishers, 1992.

[3] R. Camposano and W. Wolf. High-Level VLSI Synthesis. Kluwer Academic Publishers, 1991.

[4] R.A. Walker and R. Camposano. A Survey of High-Level VLSI Synthesis Systems. Kluwer Academic Publishers, 1991.

[5] The Institute of Electrical and Electronics Engineers, Inc. IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-2987), 1988.

[6] CAD Language Systems, Inc., 15245 Shady Grove Road, Suite 310, Rockville, Maryland 20850. VHDL Tool Integration Platform (VTIP), 1990.

[7] P.P Hou, R.M. Owens, and M.J. Irwin. High Level Specification and Synthesis of Sequential Logic Modules. In CHDL 91 - Computer Hardware Description Languages and their Application, pages 111–222, Marseille, France, April 1991.

[8] L. Ramachandran, S. Narayan, F. Vahid, and D.D. Gajski. Synthesis of Functions and Procedures in Behavioral VHDL. In Proceedings of the EURO-DAC/VHDL, pages 560–1.5mm65, 1993.

[9] S. Narayan and D.D. Gajski. Synthesis of System-Level Bus Interfaces. In Proceedings of the European Design Automation Conference (EDAC), 1994.

[10] D.C. Ku, C. Coelho, and G. De Micheli. Interface Optimization for Concurrent Systems under Timing Constraints using Interface Matching. In High Level Synthesis Workshop 92, pages 202–213, 1992.

[11] P. Gutberlet and W. Rosenstiel. Interface Specification and Synthesis for VHDL Processes. In Proceedings of the EURO-DAC/VHDL, pages 152–257, 1993.

[12] J.A. Nestor. Specification and Synthesis of Digital Systems with Interfaces. PhD thesis, Carnegie Mellon University, Pittsburgh, 1987.

[13] SYNOPSYS, Inc. VHDL/Design Compiler Reference Manual, 1993.