
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 step 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 Mechanism

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
the transmitter, signalling that the datum has been received correctly. One clock cycle later the transmitter sets port \texttt{ack\_in} to '0' which finishes the receive operation at the ports belonging to the transmission channel in.

The \texttt{send} operation starts immediately after the \texttt{receive} operation by putting the received data to port \texttt{data\_out} and setting port \texttt{data\_ok} to '1'. Then Port \texttt{ok\_out} is polled until it gets the value '1'. This finishes the \texttt{send} operation.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{transmitter.png}
\caption{Waveform of a Transmitter}
\end{figure}

\begin{verbatim}
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;
  ok\_in <= '1';
  ack\_in <= '1';
  ok\_out <= '1';
  data\_out <= buf;
  wait until clk = '1';
  loop -- receive
    exit when ack\_out = '1';
    wait until clk = '1';
  end loop:
  ok\_out <= '0';
end process;
\end{verbatim}

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 \textit{merging} the tail of the \texttt{receive} operation and the head of the \texttt{send} operation as shown in Listing 2.

\begin{verbatim}
transmitter: process
  variable buf : bit\_vector( 7 downto 0 );
begin
  loop
    exit when ok\_in = '1';
    wait until clk = '1';
  end loop:
  buf := data\_in;
  ack\_in <= '1';
  ok\_out <= '1';
  data\_out <= buf;
  wait until clk = '1';
  loop
    exit when ack\_out = '1';
    wait until clk = '1';
  end loop:
  ok\_out <= '0';
end process;
\end{verbatim}

Listing 2: The Merged Transmitter.

In this case, merging is performed by moving the tail statements of the receive protocol (here: \texttt{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, \texttt{sel='1'} represents any not constant condition, whatever.

\begin{verbatim}
transmitter: process
  variable buf : bit\_vector( 7 downto 0 );
  variable receive\_ready : boolean := TRUE;
begin
  if receive\_ready = FALSE then
    ok\_out <= '1';
    -- force to finish
    wait until clk = '1';
    -- receive
  \end{verbatim}
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 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 \(\text{wait until clk = '1'}\) or equivalent statements, which specify the timing offset for one clock cycle, are used in sequential or behavioral VHDL descriptions for this reason. These wait statements constitute the set \(V_W\). More complex wait statements like \(\text{wait on clk until clk = '1'}\) and \(\text{en = '0'}\) 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

---

1It 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.

```plaintext
procedure Merging( CFG: P ) is
    Natural: W := 0;
    List : V_{L,B}, V_{L,E}, V_{L,M}, V_{W} := null;
    List0CFG : M := null;
begin
    IdentifyProtocols( { P, V_{L,B}, V_{L,E}, V_{L,M}, V_{W}, W } );
    CutProtocolMerges( { V_{L,B}, V_{L,E}, V_{L,M}, M, V_{W}, W } );
    InsertProtocolEnds( M, V_{W} );
end Merging;
```

```plaintext
procedure IdentifyProtocols( { CFG,P,List:V_{L,B},V_{L,E},V_{L,M},V_{W}:Natural,W } )
    is
        Node : n;
begin
    for all n in P loop
        case kind(n) is
            when ProtocolBeginLabel => V_{L,B} = V_{L,B} \cup N;
            when ProtocolEndLabel   => V_{L,E} = V_{L,E} \cup N;
            when MergeBeginLabel    => V_{L,M} = V_{L,M} \cup N;
            when WaitStatement      =>
                V_{W} = V_{W} \cup N;
                if IsUnconditionalStatement( N ) then
                    W := W + 1;
                end if;
         end if;
    end case;
end loop;
for all n in P loop
    if kind( n ) = ConditionalStatement then
        IdentifyProtocols( { SubCFG(n), V_{L,B}, V_{L,E}, V_{L,M}, V_{W}, W } );
    end if;
end loop;
end IdentifyProtocols;
```

```plaintext
    is
        Node : n,b,e,m,n;
        CFG : m;
begin
    for all n,b,e,m,n in V_{L,B},V_{L,E},V_{L,M} loop
        m = CutSubCFG( m, n, b, e );
        if W = 0 then
            InsertSubCFG( m, n, b, e );
        else
            for all n in m loop
                if kind( n ) = WaitStatement then
                    V_{W} = V_{W} \setminus n;
                end if;
            end loop;
        end if;
    end loop;
end CutProtocolMerges;
```

```plaintext
procedure InsertProtocolEnds( List0CFG:M,List:V_{W} )
    is
        Node : n;
        CFG : m;
begin
    for all n in V_{W} loop
        for all m in M loop
            AppendSubCFG( m, n );
        end loop;
    end loop;
end InsertProtocolEnds;
```

Listing 4: The Merging Algorithm

The complexity $C$ of the merging algorithm can be formulated as

$$C = |S| + |P| + |W| \cdot |P| = O(|S|^2)$$

considering the three passes of the merging algorithm. Here, $|S|$ is the number of statements, $|P|$ is the number of statements in the protocols and $|W|$ 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 negligible 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 time-slots 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 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 acc <= '1' and acc <= '0'. Due to data dependency, scheduler, as referenced above, add an additional time constraint between acc <= '0' and acc <= '1', assuming that acc 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.

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 merge_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 automatically due to the fact, that detailed information about the protocol is required as shown in Section 3.

### Table 1: Analysis of the optimized VHDL-Code

<table>
<thead>
<tr>
<th>circuit</th>
<th>original</th>
<th>refined</th>
<th>merged</th>
<th>folded</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec_send</td>
<td>13</td>
<td>32</td>
<td>45</td>
<td>60</td>
</tr>
<tr>
<td>rec_send1</td>
<td>10</td>
<td>33</td>
<td>52</td>
<td>77</td>
</tr>
<tr>
<td>sem_rec</td>
<td>12</td>
<td>44</td>
<td>78</td>
<td>140</td>
</tr>
<tr>
<td>semtrans</td>
<td>22</td>
<td>79</td>
<td>503</td>
<td>375</td>
</tr>
<tr>
<td>semtrans1</td>
<td>25</td>
<td>79</td>
<td>503</td>
<td>384</td>
</tr>
<tr>
<td>split</td>
<td>23</td>
<td>58</td>
<td>58</td>
<td>84</td>
</tr>
<tr>
<td>tee</td>
<td>18</td>
<td>53</td>
<td>193</td>
<td>231</td>
</tr>
<tr>
<td>transmitter</td>
<td>29</td>
<td>56</td>
<td>91</td>
<td>114</td>
</tr>
<tr>
<td>xii</td>
<td>42</td>
<td>95</td>
<td>621</td>
<td>799</td>
</tr>
</tbody>
</table>

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.

```vhdl
procedure receive
  signal clk : in bit;
  signal data : in bit_vector;
  signal ok : in bit;
  signal ack : out bit;
  variable buff : out bit_vector)
begin
  proc_begin:
    -- mark begin of the protocol
  loop
    exit when ok = '1';
    wait until clk = '1';
  end loop;
  buff := 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.

```vhdl
transmitter: process
  variable buff: bit_vector(7 downto 0);
begin
  receive( data_in, ok_in, ack_in, buff, clk);
  send( data_out, ok_out, ack_out, buff, 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.
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 sub-routines, as soon as the VHDL frontend supports VHDL’92.

Future effort lies on the optimization of multi-cycle 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