High-Level Synthesis through Transforming VHDL Models

Anatoly Prihozhy
Belarusian State Polytechnic
aprihozhy@bspa.unibel.by

Abstract

In this paper a method of transforming a behavioral VHDL-model to a functionally equivalent model with one basic block is proposed. High-level synthesis techniques including scheduling, allocation, and binding are modified for the model. These reduce the number of control steps, FSM states, state transitions, functional and storage units in an RTL-structure.

1. Introduction

Usually, the designer starts by describing the required behavior and specifying the optimization goal and design constraints. Capturing this input, a high-level synthesis tool generates a structure at the register transfer level (RTL) [1-5]. If the designer is not satisfied with the synthesis result, she/he can modify the behavioral description, optimization criterion, or design constraints and try again. Nowadays the path-based high-level synthesis method is widely and successfully used in efficient synthesis tools [1-5,10,11]. It constitutes a theoretical basis for performing data-control flow graph (DCFG) analysis, scheduling, allocation, and binding for the whole behavior. At the same time, the method has the drawbacks as follows:

- The number of paths on the CFG can increase exponentially depending on the number of nodes (up to millions of paths are possible for real designs [10])
- Reordering of statements in a path is not allowed
- During scheduling, allocation and binding very complex combinatorial tasks have to execute for each path and the obtained results have to be combined.

In this paper we prove that any behavioral description can be equivalently transformed to a one basic block model (OBBM) allowing more powerful synthesis results than the traditional approaches allow. Section 2 presents the idea of transformational synthesis. In sections 3 and 4 transformation rules and techniques for inferring OBBM are described. Scheduling, allocation, and binding techniques for OBBM are presented in sections 5 and 6. Section 7 gives some results.

2. Transformation-based synthesis

Starting from a behavioral VHDL-description, we generate a number of equivalent descriptions [6-9] by applying transformations (Fig.1). The technique extends the set of design alternatives and supports an efficient design space exploration (Fig.2). In this paper we propose a method of transforming a behavioral VHDL-model to a functionally equivalent model constructed of one basic block. The existing scheduling, allocation and binding techniques can't be directly applied to the model. We modify the techniques by means of using several relations on the sets of variables and statements in order to improve parameters of the generated RTL-structure.

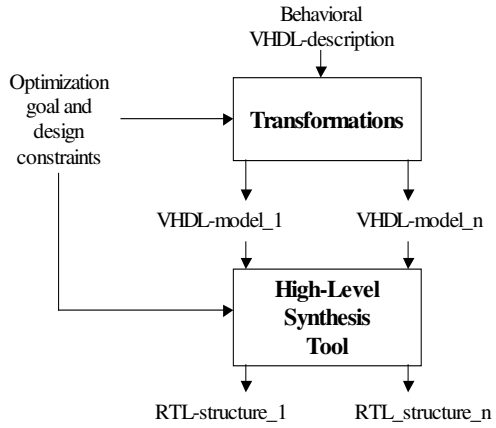

Figure 1. Synthesis through transformation

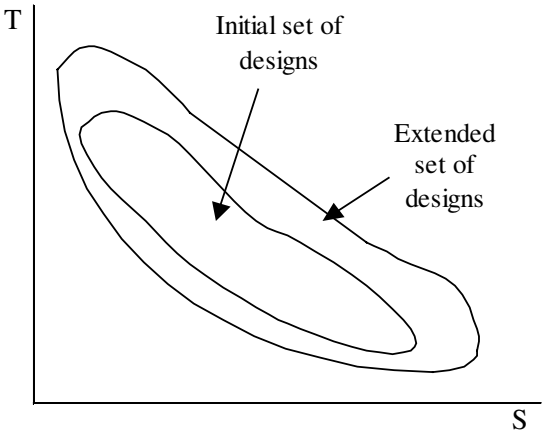

Figure 2. Design space exploration

3. Transformation of original model

We transform a behavioral sequential synchronous VHDL model with flexible or specified cycle behavior in order to obtain a VHDL-model that is more suitable for efficient high-level synthesis. Several types of useful transformations are used in the existing HLS tools [1-11]. We propose deeper transformations leading to significant reorganization of the source VHDL-model and its CFG: splitting statements, inserting statements, extracting computations, attaching statements, eliminating loops, eliminating exit- and next-statements, and others. The algebraic methods of transforming Boolean expressions will be intensively used as well. We will avoid multiple assignments to the same variable. Through the paper the following notations will be used: *V,V1*,…,*C,C1*,… are Boolean variables and expressions, *Q,Q1*,… are sequential statements, *S,S1*,… are sequences of statements, and *L,L1*,… are labels of loops.

3.1. Wait-statements

In the sequential synchronous behavioral model a sequence of statements may describe a specified cycle behavior or a flexible cycle behavior [1]. The VHDL-statement

wait until C **and** Clock'Event **and** Clock='1'; (3.1)

is used to control the specified cycle behavior, where *Clock* is the signal triggering the state transition. The fragment is equivalent to the loop-statement:

loop wait until Clock'Event **and** Clock='1'; (3.2)
exit when C; **end loop**;

3.2. Loop-statements

Our transformation rules are significantly based on using loop-statements without an iteration scheme: *L: loop S end loop L;*. The VHDL loop-statements *while* and *for* with ascending and descending ranges as follows:

while C **loop** S **end loop**;
for I **in** L **to** R **loop** S **end loop**; (3.3)
for I **in** L **downto** R **loop** S **end loop**

may be replaced with

loop V:=**not** C; **exit when** V; S **end loop**;
I:=L; **loop** V:=I>R; **exit when** V; S I:=I+1; **end loop**; (3.4)
I:=L; **loop** V:=I<R; **exit when** V; S I:=I-1; **end loop**;

Very often a loop-statement can be under an if-then-statement as in the following fragment:

if C **then loop** S **end loop**; **end if**; (3.5)

Inserting the if-statement into the loop-statement yields:

V:=true; **loop** V1:=V **and not** C; V:=false; (3.6)
exit when V1; S **end loop**;.

3.3. Exit-statements

An exit-statement works together with a loop: *L: loop S1 exit L when C; S2 end loop;*. If an exit-statement is under an if-statement:

if C1 **then exit** L **when** C2; **end if**; (3.7)

then the two statements may be merged as

V:=C1 **and** C2; **exit** L **when** V; (3.8)

If an exit-statement *exit L when C; S* is followed by a sequence *S* of statements that does not update the value of *C,* then the two statements may be reordered as:

if not C **then** S **end if; exit** L **when** C; (3.9)

Two exit-statements with the same loop-label:

exit L **when** C1; **exit** L **when** C2; (3.10)

may be merged as:

V:=C1 **or** C2; **exit** L **when** V; (3.11)

3.4. Next-statements

All the next-statements are eliminated from the VHDL-behavioral description. If a next-statement is under an if-statement:

if C1 **then next** L **when** C2; **end if**; (3.12)

then the two statements may be merged as

V:=C1 **and** C2; **next** L **when** V; (3.13)

The most general situation with a next-statement and two nested loops can be represented as:

L1: **loop** S1 L2: **loop** S2 (3.14)
next L1 **when** C; S3 **end loop** L2; S4 **end loop** L1;

Replacing the next-statement with exit- and if-statements yields:

V:=false; L1: **loop** S1 L2: **loop** S2 V:=C; (3.15)
exit L2 **when** V; S3 **end loop** L2;
if not V **then** S4 **end if**; **end loop** L1;

The same transformation can be performed on an arbitrary number of nested loops. It can imply introducing a label for an unlabeled loop.

3.5. Return-statements

The return-statements are used to exit from VHDL functions and procedures. We eliminate the statements from models and replace them with variable assignment, exit- and if-then-statements.

3.6. If-statements

An if-statement in VHDL selects no more than one sequence of statements and is represented in the following form:

if C1 **then** S1 **elsif** C2 **then** S2 • • • (3.16)
elsif Cn-1 **then** Sn-1 **else** Sn **end if**;

The if-statement may be split to the following sequence of variable assignment and if-then-statements:

V1:=C1; V2:=**not** C1 **and** C2; • • • (3.17)
Vn-1:=**not** C1 **and not** C2 **and** … **and** Cn-1;
Vn:=**not** C1 **and not** C2 **and** … **and not** Cn-1;
if V1 **then** S1 **end if**; **if** V2 **then** S2 **end if**; • • •
if Vn-1 **then** Sn-1 **end if**; **if** Vn **then** Sn **end if**;

The if-then-statement *if V then Q1;…Qk; end if;* containing a sequence of other statements may be split to a sequence of if-then-statements each of them with one statement inside:

if V **then** Q1; **end if**; … **if** V **then** Qk; **end if**; (3.18)

This transformation is eligible if *V* is not reassigned in *Q1, …,Qk-1*. If one if-statement is inside of another if-statement then the described splitting procedure can yield *if V1 then if V2 then Q; end if; end if;* that is reduced to

V:=V1 **and** V2; **if** V **then** Q; **end if**; (3.19)

In special case *if V1 then V2:=E; end if;* the if-then-statement may be replaced with a variable assignment statement:

V2:=(V1 **and** E) **or** (**not** V1 **and** V2); (3.20)

Very often two VHDL-fragments as follows will appear during transformations:

V2:=false; **if** V1 **then** V2:=E; **end if**; (3.21)
V2:=true; **if** V1 **then** V2:=E; **end if**;

These may be replaced with:

V2:=V1 **and** E; (3.22)
V2:=E **or not** V1;

If *E* implies *V1* in (3.20), that is $E \rightarrow V1$ holds for all the values of primary variables, and the previous value of *V2* is not determined, then (3.20) is reduced to *V2:=E;*. Here '$\rightarrow$' is the logical implication operation. Similarly, if the implication $V2 \rightarrow V1$ holds then (3.19) is reduced to:

if V2 **then** Q; **end if**;. (3.23)

3.7. Case-statements

In VHDL the case statement has the following form:

case E **is when** H11|…|H1k_1 => S1 • • • (3.24)
when Hr1|…|Hrk_r => Sr **when others** => Sr+1 **end case**;

where *E* is an expression of a discrete type and *Hij* is a choice defining a value or a range of values. The case-statement is equivalently transformed to the following if-statement:

V:=E; **if** R11 **or** … **or** R1k_1 **then** S1 • • • (3.25)
elsif Rr **or** … **or** Rrk_r **then** Sr **else** Sr+1 **end if**;

where *V* is a variable of the type associated with the expression *E* and *Rij* is a relational operator (expression) associated with the choice *Hij*. The if-statement is split as it was described in section 3.6.

3.8. Variable and signal assignments

A variable assignment statement with an aggregate in the left part is split into a sequence of assignment statements with a variable in the left part. If an assignment statement contains an expression *E* in the right part constructed of more than one operator, it is split implying the addition of variables of appropriate types. A signal assignment statement using operators in the right part is replaced with a signal assignment statement without operators by means of introducing an additional variable and variable assignment statement.

3.9. Procedure and function calls

All the transformation rules that are considered in the paper may be applied to the processes and subprograms. In particular, we can represent the subprogram body as one basic block. Two approaches to processing the subprograms during high-level synthesis are possible: performing high-level synthesis separately for each subprogram and merging the process and subprograms before synthesis. A drawback of the first approach is that it is not easy to optimize the design cost. A drawback of the second approach is that it is desirable to avoid multiple attaching the subprogram body to the process body because the VHDL-code can extend significantly when a lot of calls of the same subprogram exist. In order to unify the merging algorithm, all the functions are replaced with procedures and assignment statements *U:=F(P);* are replaced with procedure calls *F(P,U)* where *U* is a variable, *F* is a function name, and *P* is a list of actual parameters.

4. VHDL-model with one basic block

The number of paths on CFG significantly depends on the number of basic blocks. In this section we prove that any VHDL-process may be represented as one basic block in the statement part:

```
P: process Declars                                          (4.1)
begin Basic_Block end process P;
```

where *Declars* are local declarations and *Basic_Block* is a sequence of wait, variable and signal assignment statements either covered or not covered with if-then-statements. An if-then-statement describes a condition of operation execution and variable/signal assignment. The process parenthesis *"begin*" and *"end"* describe an infinite loop. The proposed transformation technique consists of the key steps as follows: inserting statements located after a loop into the loop, inserting statements located before a loop into the loop, merging neighbor nested loops, eliminating loop- and exit-statements, and eliminating subprogram calls.

4.1. Inserting statements into loop

We assume that all the loop-statements are labeled and all the exit-statements refer a loop-label. While inserting statements located after a loop, the number of exit-statements in the loop has to be taken into account as in the following VHDL-fragment:

```
L: loop S1 exit L when V1; S2                               (4.2)
   exit L when V2; S3 end loop L; S4
```

where *S4* is a sequence of statements that do not update the value of variables *V1* and *V2* (otherwise we can use additional variables). Inserting *S4* into the loop yields:

```
L: loop S1 if V1 then S4 end if; exit L when V1; S2         (4.3)
if V2 then S4 end if; exit L when V2; S3 end loop L;
```

The drawback is that two copies of *S4* appear in the loop-body. In order to have one copy we begin with reordering and merging the exit-statements:

```
L: loop S1 if not V1 then S2 end if; V3:=V1 or V2;          (4.4)
     exit L when V3; S3 end loop L; S4
```

and then the sequence *S4* is inserted into the loop:

```
L: loop S1 if not V1 then S2 end if; V3:=V1 or V2;          (4.5)
   if V3 then S4 end if; exit L when V3; S3 end loop L;
```

The technique is eligible for an arbitrary number of exit-statements in one loop. When statements located before a loop are inserted into the loop, the exit-statements are not used, so we consider the VHDL-fragment as:

```
S1 L: loop S2 end loop L;                                   (4.6)
```

that is functionally equivalent to:

```
V:=true; L: loop                                            (4.7)
if V then S1 end if; V:=false; S2 end loop L;
```

If the loop-statement is under an if-then-statement:

```
S1 if C then L: loop S2 end loop L; end if;                 (4.8)
```

then the inserting yields:

```
V:=true; L: loop if V then S1 end if;                       (4.9)
V1:=V and not C; V:=false; exit when V1; S2 end loop L;
```

Inserting of statements located after a loop is cheaper than inserting of statements located before the loop.

4.2. Generating nested loops

Usually a VHDL-process or subprogram contains a hierarchy of loops. In the hierarchy pairs of loops exist that either execute sequentially or one loop is in the body of other loop. For the both cases we prove that an equivalent system of nested loops may be constructed. First, we transform the VHDL-fragment:

```
L1: loop S1 if C1 then L2: loop S2 exit L2 when C2;        (4.10)
S3 end loop L2; end if; S4 end loop L1;
```

where the loop *L2* is in the body of the loop *L1* and is covered by an if-then-statement. After inserting the sequences *S1* and *S4* into the loop *L2* and attaching the exit-statements we have:

```
L1: loop V1:=true; L2: loop                                 (4.11)
     if V1 then S1 end if; V2:=V1 and not C1;
     V1:=false; V3:=not V2; if V3 then S2 end if;
     V4:=V3 and C2; if V4 then S4 end if;
     V5:=V2 or C2; exit L2 when V5; S3
end loop L2; end loop L1;
```

Second, we transform the sequence of two loops:

```
L1: loop S1 exit L1 when C1; S2 end loop L1;                (4.12)
S3 L2: loop S4 exit L2 when C2; S5 end loop L2;
```

After inserting the loop *L2* into the loop *L1* and inserting the statements located before *L2* into the loop we have:

```
L1: loop V1:=true; L2: loop                                 (4.13)
     if V1 then S1 end if; V2:=V1 and C1;
     if V2 then S3 end if; V3:=V1 and not C1;
     V1:=false; exit L2 when V3; S4 exit L2 when C2; S5
end loop L2; exit L1 when C1; S2 end loop L1;
```

Merging two exit-statements referring label *L2* and insert-ing the statements located after *L2* into the loop yields:

```
L1: loop V1:=true; L2: loop                                 (4.14)
     if V1 then S1 end if; V2:=V1 and C1;
     if V2 then S3 end if; V3:=V1 and not C1;
     V1:=false; V4:=not V3; if V4 then S4 end if;
     V5:=V3 or C2; V6:=V5 and C1; exit L1 when V6;
     if V5 then S2 end if; exit L2 when V5; S5
end loop L2; end loop L1;
```

Consecutively applying the described transformations to pairs of hierarchical and sequential loops, we can transform any VHDL-process with an arbitrary number and structure of loops to a process with nested loops.

4.3. Eliminating loop- and exit-statements

Two nested loops:

```
L1: loop  V1:=true;  L2: loop  S1  exit L2 when C1;      (4.15)
S2  end loop L2;  end loop L1;
```

may be replaced with one loop:

```
L1: loop  S1  if C1 then V1:=true; end if;               (4.16)
V2:=not C1; if V2 then  S2  end if;  end loop L1;
```

if the variable *V1* used during inserting statements takes initial value *true*. The exit-statement is replaced with a conditional *(C1)* assignment of value *true* to the control variable *V1*. The sequence *S2* also executes conditionally, but with the condition *not C1*. Applying the rule to nested loops (4.11) inferred from two hierarchical loops (4.10) and using rules (3.20) to (3.23) , we obtain one loop as follows:

```
L1:loop if V1 then  S1  end if; V2:=V1 and not C1;       (4.17)
     V3:=not V2; if V3 then  S2  end if;
     V4:=V3 and C2; if V4 then  S4  end if;
     V1:=V2 or C2; V5:=not V1; if V5 then  S3  end if;
end loop L1;
```

Similarly, nested loops (4.14) inferred from sequential loops (4.12) are transformed to one loop:

```
L1: loop  if V1 then  S1  end if; V2:=V1 and C1;         (4.18)
     if V2 then  S3  end if; V7:= not C1;
     V3:=V1 and V7; V4:=not V3; if V4 then  S4  end if;
     V1:=V3 or C2; V6:=V1 and C1;
     exit L1 when V6; if V1 then  S2  end if;
     V5:=not V1; if V5 then  S5  end if;
end loop L1;
```

Given a process with *k* nested loops the loops may be eliminated step by step. Finally, a process with one basic block is inferred. The process statement part is a loop that is suspended and resumed by events on the *Clock* signal.

4.4. Eliminating subprogram calls

We assume that a subprogram is called for more than once in a process (another subprogram) body. Let the bodies of subprogram and process be represented as one basic block. If each subprogram call is replaced with the subprogram body of significant size, the VHDL code can extend significantly. Our goal is to construct the process body in such a way to attach one copy of the subprogram body. The goal is achieved through using additional control Boolean variables and if-statements. We illustrate our approach with the following abstract VHDL-like fragment:

```
process  • • •                                           (4.19)
  procedure F(Pi, Po) is  Declars  begin  B  end F;
begin  S1  F(P1i,P1o);  S2  F(P2i,P2o);  S3  end process;
```

where *F* is a procedure name, *Pi* and *Po* are descriptions of input and output formal parameters, *Declars* is a list of declarative items, *B* is a procedure body represented as one basic block, and *P1i, P1o, P2i*, and *P2o* are input and output actual parameters of first and second procedure calls. After inserting the declarations *Pi, Po, and Declars* into the process declarative part, inserting the procedure body *B* into the process statement part, and reorganizing the statement part, the fragment is as follows:

```
process • • • Pi, Po, and Declars  modified              (4.20)
  variable V1,…,V7: Boolean;  begin
    V3:=V1 or V2; if V3 then  B  end if;
    V4:=not V3; if V4 then S1 end if;
    if V4 then Pi':=P1i; end if; V5:=V1 and not V2;
    if V5 then P1o:=Po'; end if; if V5 then S2 end if;
    if V5 then Pi':=P2i; end if; V6:=not V1 and V2;
    if V6 then P2o:=Po'; end if; if V6 then S3 end if;
    V7:=V1 xor V2;  V2:=V1;  V1:=not V7;
end process;
```

where *Pi'* and *Po'* are local variables representing the input and output formal parameters of procedure. In the process, all the variables that are used in *B* have to be unique. Increase in the number of procedure calls requires additional control variables, although the transformation method remains the same.

4.5. An example

Now we demonstrate the proposed transformation technique on a simple VHDL behavioral description. An original GCD algorithm is presented in Fig.3. First, we split all the control structures and remove the iteration scheme from the loop (Fig.4). Three additional Boolean variables *C1, C2,* and *C3* are introduced. Then, according to (4.5) and (4.7) we insert the statements that precede and succeed the loop into the loop (Fig.5). An additional variable *C0* is used. After that the loop is eliminated, the exit-statement is replaced with an if-statement, and the variable *C0* takes the initial value *true* (Fig.6). Finally, two statements that assign a value to variable *C0* are merged and the resulting variable assignment statement proves that *C0=C1*. The variable *C1* is removed and replaced with *C0* (Fig.7). The waveforms of input and output signals are the same for all the VHDL-models (Fig.8). Any behavioral VHDL-description can be transformed in the similar way.

```
entity GCD is
port(Clock, Reset: in Bit; XP, YP: in Bit_Vector(15 downto 0);
   Ready: out Bit; Res: out Bit_Vector(15 downto 0));
end GCD;
architecture Behavior1 of GCD is  begin
process  variable X,Y: Bit_Vector(15 downto 0); begin
   wait until Clock'Event and Clock='1'; Ready<='0'; X:=XP;
   Y:=YP; while (X/=Y) loop  wait until Clock'Event and
   Clock='1'; if (X<Y) then Y:=Y-X; else X:=X-Y; end if;
   end loop;  Ready<='1'; Res<=X;  end process;
end Behavior1;
```

Figure 3. Original VHDL-model of GCD

```
architecture Behavior2 of GCD is begin
 process variable X,Y: Bit_Vector(15 downto 0);
  variable C1,C2,C3: Boolean; begin
  wait until Clock'Event and Clock='1'; Ready<='0'; X:=XP;
  Y:=YP; loop C1:=X=Y; exit when C1;
  wait until Clock'Event and Clock='1';C2:=X<Y;C3:=not C2;
  if C2 then Y:=Y-X; end if; if C3 then X:=X-Y; end if;
  end loop; Ready<='1'; Res<=X; end process;
end Behavior2;
```

Figure 4. Transformed VHDL-model (step 1)

```
architecture Behavior3 of GCD is begin process
  variable X,Y: Bit_Vector(15 downto 0);
  variable C0,C1,C2,C3: Boolean; begin C0:=true; loop
    if C0 then wait until Clock'Event and Clock='1';end if;
    if C0 then Ready<='0'; end if; if C0 then X:=XP; end if;
    if C0 then Y:=YP; end if; C0:=false; C1:=X=Y;
    if C1 then Ready<='1'; end if; if C1 then Res<=X; end if;
    exit when C1; wait until Clock'Event and Clock='1';
    C2:=X<Y; C3:=not C2; if C2 then Y:=Y-X; end if;
    if C3 then X:=X-Y; end if; end loop; end process;
end Behavior3;
```

Figure 5. Transformed VHDL-model (step 2)

```
architecture Behavior4 of GCD is begin process
  variable X,Y: Bit_Vector(15 downto 0);
  variable C0,C1,C2,C3: Boolean:=true; begin
  if C0 then wait until Clock'Event and Clock='1'; end if;
  if C0 then Ready<='0'; end if; if C0 then X:=XP; end if;
  if C0 then Y:=YP; end if; C0:=false; C1:=X=Y;
  if C1 then Ready<='1'; end if; if C1 then Res<=X; end if;
  if C1 then C0:=true; end if; if not C1 then wait until
  Clock'Event and Clock='1'; end if; C2:=X<Y; C3:=X>Y;
  if C2 then Y:=Y-X; end if; if C3 then X:=X-Y; end if;
end process; end Behavior4;
```

Figure 6. Transformed VHDL-model (step 3)

```
architecture Behavior5 of GCD is begin process
  variable X,Y: Bit_Vector(15 downto 0);
  variable C0,C2,C3: Boolean:=true; begin
  if C0 then wait until Clock'Event and Clock='1'; end if;
  if C0 then Ready<='0'; end if; if C0 then X:=XP; end if;
  if C0 then Y:=YP; end if; C0:=X=Y;
  if C0 then Ready<='1'; end if; if C0 then Res<=X; end if;
  if not C0 then wait until Clock'Event and Clock='1'; end
  if; C2:=X<Y; C3:=X>Y; if C2 then Y:=Y-X; end if;
  if C3 then X:=X-Y; end if; end process;
end Behavior5;
```

Figure 7. Transformed VHDL-model (step 4)

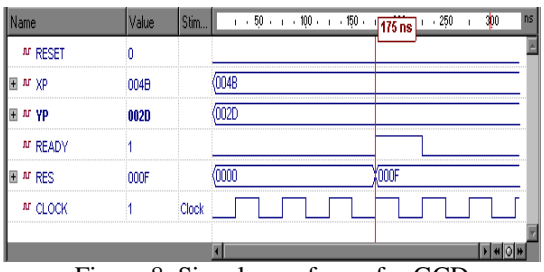

Figure 8. Signal waveforms for GCD

5. Scheduling for OBBM

Scheduling aims at introducing control steps and FSM states, distributing statements on the steps, and either minimizing the number of steps or minimizing the design cost. The scheduling results in generating a high-level state machine HLSM. The known scheduling techniques that process a traditional basic block are [1-5]: as soon as possible ASAP, as late as possible ALAP, list scheduling, integer linear programming formulation ILPF, freedom-based scheduling, force-directed scheduling, dynamic loop scheduling, scheduling for pipelines and other techniques. The path-based scheduling technique processes the whole CFG consisting of several basic blocks.

5.1. Scheduling methodology

In a traditional basic block all statements execute unconditionally. In the OBBM a statement may execute conditionally or unconditionally. Our scheduling methodology includes: analyzing data dependencies, generating orthogonal, implication, and equivalence relations on the set of control signals and variables, generating a set of pairs of orthogonal statements, analyzing operator compatibility and operator proximity, generating a statement precedence relation, performing a modified scheduling technique (ILPF, list scheduling, ASAP, and ALAP). The generated schedule is optimized due to orthogonal operations are mutually exclusive and may execute on the same functional unit concurrently without additional resources. As a result, the statements are reordered, grouped, and an optimized HLSM is generated.

5.2. Relations on the set of variables

Signals and variables (mostly variables) of Boolean type control the execution of if-then-statements. Relations on the set of control variables define in which manner the operators and assignments that are under an if-then-statement execute. Two control variables *V1* and *V2* are:

- Orthogonal (#) if the variables take never value *true* simultaneously
- Variable *V1* implies variable *V2* (→) if *V2* takes never value *false* when *V1* equals *true*
- Variable *V2* implies variable *V1* (←) if *V1* takes never value *false* when *V2* equals *true*
- Equivalent (↔) if the variables can't take different values simultaneously
- Independent (–) if neither of the previous cases takes place.

There are two main sources for inferring the relations. First, the VHDL relational operators are analyzed. Given two assignment statements *V1:=X R1 Y;* and *V2:=X R2 Y;* with the same operands where *R1,R2*∈ *{=, /=, <, <=, >, >=}* are relational operators, the relations #, →, ←, and ↔ between *V1* and *V2* are inferred through using Table 1. The similar table is used for a pair of statements *V1:=X R1 L1;* and *V2:=X R2 L2;* where *L1* and *L2* are

Inferring relations between *V1* and *V2* Table 1

N	Operator *R1*	Operator *R2*	Relation between *V1* and *V2*
1	=	=	↔
2	=	/=, <, >	#
3	=	<=, >=	→
4	/=	=	#
5	/=	/=	↔
6	/=	<, >	←
7	<	=, >, >=	#
8	<	/=, <=	→
9	<	<	↔
10	<=	=, <	←
11	<=	<=	↔
12	<=	>	#
13	>	=, <, <=	#
14	>	/=, >=	→
15	>	>	↔
16	>=	=, >	←
17	>=	<	#
18	>=	>=	↔

Inferring relations using logical operators Table 2

Statement	Existing relations	Inferred relations
A:=B **and** C;	–	A→B and A→C
	B→D or C→D	A→D
	D#B	D#A
	D#C	D#A
A:=B **or** C;	–	B→A and C→A
	D→B or D→C	D→A
	B→D and C→D	A→D
	D#B and D#C	D#A
	B→C	B→A
	C→B	C→A
A:=B **xor** C;	B#C	B→A and C→A
	B→D and C→D	A→D
	D→B and D→C	A#D
A:=B **nand** C;	D→B and D→C	A#D
	D#B or D#C	D→A
	B→C	A#B
	C→B	A#C
A:=B **nor** C;	–	A#B and A#C
	D→B	A#D
	D→C	A#D
	B#D and C#D	B→A
A:=**not** B;	–	A#B
	D→B	A#D
	B#D	D→A
–	A→B and B→C	A→C

literals for which one of the relations *=, /=, <, <=, >, >=* holds. Second, the VHDL logical operators are analyzed (Table 2) to infer additional relations between other pairs of control variables. In the table, *A, B, C,* and *D* are Boolean variables. There are similar rules for pairs of variable assignment statements and for pairs of relations.

5.3. Orthogonal statements

All the sequential statements in OBBM may be considered as an if-then-statement. If a statement *Q* is not conditional one, it may be replaced with *if true then Q end if;*. Two if-then-statements *if V1 then Q1 end if;* and *if V2 then Q2 end if;* are defined to be orthogonal if the variables *V1* and *V2* are orthogonal. One orthogonal statement cannot precede another orthogonal statement. The orthogonal statement bodies are mutually exclusive and may execute on the same functional unit concurrently in the same control step. VHDL-like fragment (4.20) uses seven Boolean variables *V1,…,V7*. The relations on the set of variables are described by the matrix:

R=

	V1	V2	V3	V4	V5	V6	V7
V1	↔	#	→	#	←	#	→
V2	#	↔	→	#	#	←	→
V3	←	←	↔	#	←	←	←
V4	#	#	#	↔	#	#	#
V5	→	#	→	#	↔	#	→
V6	#	→	→	#	#	↔	→
V7	←	←	→	#	←	←	↔

The orthogonal variables graph is presented in Fig.9. The following pairs of control variables are orthogonal: *(V3,V4), (V4,V5), (V4,V6), (V5,V6)*. As a result the pairs of statement sequences as follows are mutually exclusive: *(B,S1), (S1,S2), (S1,S3), (S2,S3)*.

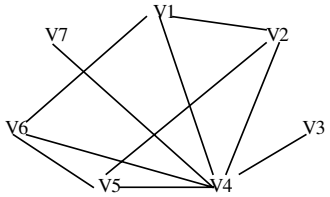

Figure 9. Orthogonal variables graph

5.4. Operators compatibility and proximity

There are two cases for operators to be compatible within one control step: the operators belong to orthogonal statements and may execute on the same type of functional unit and the operators are relational and have identical operands. The compatible operators may execute within one control step without additional resources. The operator proximity is used to select compatible operators to be merged. We estimate the proximity of two operators as the number of common inputs and outputs. Maximizing the proximity of operators leads to minimizing the cost of interconnect units in resulting RTL-structure.

5.5. Precedence of statements

The statement precedence relation $PRE=VL \cup US \cup WT$ is a union of three sub-relations:

- A statement *i* precedes a statement *j* $((i,j) \in VL)$ if *i* and *j* are not orthogonal and *i* has an output variable that is an input variable for *j*
- A statement *i* precedes a statement *j* $((i,j) \in US)$ if *i* and *j* are not orthogonal and *i* uses a value of an input variable to be assigned a new value by *j*

- A statement *i* precedes a statement *j* *((i,j)∈WT)* if a wait-statement *w* exists such that the pairs *(i,w)* and *(w,j)* of statements are not orthogonal and *i* precedes *w* and *j* succeeds *w* in the VHDL-text.

The relation *PRE* can be represented as a statement precedence graph. Statements *i* and *j* are sequential if a path exists between *i* and *j* on the graph, otherwise, the statements are concurrent. The statement precedence graph *PRE* and its sub-graphs *VL*, *US*, and *WT* for the GCD including 12 statements (Fig.7) are shown in Fig.10.

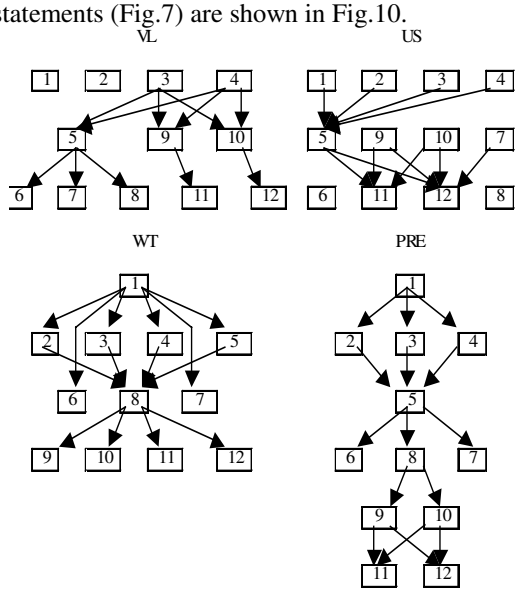

Figure 10. Statement precedence graphs for GCD

5.6. Extending scheduling techniques

All the traditional techniques [1-11] scheduling a basic block assume that the sequential statements may not be an if-then-statement and, therefore, may not be orthogonal. Our method allows the orthogonal statements to execute in the same control step in parallel without additional resources. This implies increase in the average number of statements in a step and decrease in the total number of steps. As a result, the number of values computed in one control step and used in another step is reduced as well as the number of registers. The extended ASAP and ALAP being feasible-constrained techniques use the statement precedence graph at input. The modified list scheduling being a resource-constrained technique uses a status of each statement:

- All the predecessors have been already scheduled
- There is a predecessor has not been scheduled
- The statement may be scheduled on an existing functional unit using the orthogonal relation
- Addition of a functional unit is required and

others. The modified time-constrained scheduling problem ILPF can be formulated as minimizing

$$\sum_{k=1}^{m} (s_k * M_k) \qquad (5.1)$$

subject to

$$\sum_{r=1}^{c} z_{r,j,k} \le M_k \qquad \text{for } 1 \le j \le s,\ 1 \le k \le m$$

$$z_{r,j,k} = min(1,\ \sum_{i \in FU_k} y_{r,i,j}) \qquad \text{for } 1 \le r \le c,\ 1 \le j \le s,\ 1 \le k \le m$$

$$y_{r,i,j} \le C_{i,r} * x_{i,j} \qquad \text{for } 1 \le r \le c,\ 1 \le i \le n,\ 1 \le j \le s$$

$$\sum_{r=1}^{c} y_{r,i,j} = 1 \qquad \text{for } 1 \le i \le n,\ 1 \le j \le s$$

$$\sum_{j=S_i}^{L_i} x_{i,j} = 1 \qquad \text{for } 1 \le i \le n$$

$$\sum_{j=S_i}^{L_i} (j * x_{i,j}) - \sum_{j=S_k}^{L_k} (j * x_{k,j}) = 1 \qquad \text{for all } i \rightarrow k$$

where n is the number of statements; s is the number of control steps; M_k and s_k are the number and cost of functional units of type k; m is the number of functional unit types; C is the set of cliques of the orthogonal statements graph; c is the number of cliques in C; $C_{i,r}$ equals *1* if statement i belongs to clique r; $z_{r,j,k}$ is an integer variable that equals *1* if at least one statement of clique r is associated with the type k of functional unit and executes in step j, otherwise, equals *0*; $y_{r,i,j}$ is a variable that equals *1* if statement i executes in step j and is associated with clique r, otherwise, equals *0*; $x_{i,j}$ is a variable that equals *1* if statement i executes in step j, otherwise, equals *0*; S_i and L_i are the earliest and latest possible time of statement i; $i \rightarrow k$ denotes i precedes k.

5.7. High-level state machine

The HLSM initially includes a sequence of states and a set of control variables (Fig.11). Each state has exactly one transition. To speed up the HLSM operation, additional direct transitions are added. Thus, the HLSM for GCD (Fig.12) initially included two sequential states *s0* and *s1*. All the statements in state *s0* where covered by an if-then-statement with the condition *C0*. There was a sense

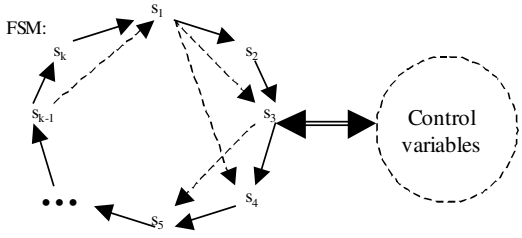

Figure 11. Control part of RTL-structure

```
architecture HLSM of GCD is type State_Type is (s0,s1);
  signal State: State_Type; begin process (Clock,Reset)
  variable X,Y: Bit_Vector(15 downto 0);
  variable C0,C2,C3: Boolean:=true; begin
if Reset='0' then State<=s0; elsif Clock'Event and Clock='1'
then case State is
  when s0 => Ready<='0'; X:=XP; Y:=YP; State <= s1;
  when s1 => C0:=X=Y; if C0 then Ready<='1'; end if;
    if C0 then Res<=X; end if; C2:=X<Y; C3:=X>Y;
    if C2 then Y:=Y-X; end if; if C3 then X:=X-Y; end if;
    if C0 then State <= s0; else State <= s1; end if;
end case; end if; end process; end HLSM;
```

Figure 12. High-level state machine for GCD

to come to state *s0* if and only if *C0* equals *true*. We have added a direct transition from *s1* to *s1* and replaced the statement *Next_State<=s0;* in the end of case-statement with the if-statement as shown in Fig.12.

6. Allocation and binding for OBBM

The RTL-structure being an output of high-level synthesis consists of two parts: a data path (DP) and a finite state machine (FSM). Allocation aims at minimizing the DP cost and defining the set of functional, storage, and interconnect units in the RTL-structure. Binding aims at mapping the elements of behavioral description to the structure components. A lot of allocation and binding techniques have been developed [1-7]: path-based, rule-based, branch and bound, clique partitioning, integer linear programming, simulated-annealing, graph coloring, and other algorithms. The OBBM being a purely data flow representation supports an efficient allocation and binding:

- Analysis of variable lifetime on one basic block
- Generating the variables compatibility relation
- Generating the operators compatibility and proximity relations
- Deriving and folding the DP from the DFG.

6.1. Variable lifetime analysis

Assuming that all the statements are distributed on the sequence of HLSM states, a variable v lifetime is represented as an interval $l_v=[s^b_v, s^e_v]$ where s^b_v, and s^e_v are the earliest and latest states in which the variable is alive. In order to compute the interval we use the function $Inc: V\times S\rightarrow\{\varnothing, \{in\}, \{out\}, \{in,out\}\}$ mapping the pairs variable/state to subsets of the set $\{in,out\}$. It is easy to derive from the function the first state s_v^{first} in which v is used as output and the last state s_v^{last} in which v is used as input. The values s^b_v, and s^e_v are computed from the values s_v^{first} and s_v^{last} taking into account the fact that the basic block is a body of an infinite loop. The function *Inc* for the GCD represented as the HLSM in Fig.12 is described in Table 3. The variable lifetimes are $l_X=l_Y=[s0,s1]$ and $l_{C0}=l_{C2}=l_{C3}=[s1,s1]$.

Function *Inc* for GCD lifetime analysis Table 3

State	Variable				
	X	Y	C0	C2	C3
S0	{out}	{out}	∅	∅	∅
S1	{in, out}	{in, out}	{in, out}	{in, out}	{in, out}

6.2. Variable and operator compatibility

A variable $v\in V_W$ which is alive within one HLSM state is implemented as a wire. A variable $v\in V_M$ which is alive in several HLSM states is mapped to a register, RAM, or ROM. Two variables v_1 and v_2 of V_M may be mapped to the same storage unit if they are compatible. The variables are compatible if either their lifetime intervals are not intersected or each statement using v_1 is orthogonal to each statement using v_2. We describe the variable compatibility as a binary relation C_V. The operator compatibility is represented by a binary relation C_O that is computed using the formula:

$$C_O=(U_O\cap\sim(S_O\backslash O_O))\cup R_O, \quad (6.1)$$

where '\' is a set subtraction operation; $\sim A$ is complementation of set A; U_O is the set of pairs of operators executed on the same type of functional unit; S_O is the set of pairs of operators executed in the same HLSM state; O_O is the orthogonal relation; R_O is the set of pairs of relational operators which execute in the same HLSM state and use the same operands. The operator proximity is estimated as it was described in section 5.4 and represented with a relation P_O. Five operators are used in the GCD HLSM shown in Fig.11: $\{'=','<','>','-','-'\}$. The relations C_O, U_O, S_O, O_O, and R_O on the set are presented in Fig.13.

U_o =

	=	<	>	-	-
=	0	1	1	0	0
<	1	0	1	0	0
>	1	1	0	0	0
-	0	0	0	0	1
-	0	0	0	1	0

S_o =

	=	<	>	-	-
=	0	1	1	1	1
<	1	0	1	1	1
>	1	1	0	1	1
-	1	1	1	0	1
-	1	1	1	1	0

O_o =

	=	<	>	-	-
=	0	0	0	0	0
<	0	0	0	0	0
>	0	0	0	0	0
-	0	0	0	0	1
-	0	0	0	1	0

R_o =

	=	<	>	-	-
=	0	1	1	0	0
<	1	0	1	0	0
>	1	1	0	0	0
-	0	0	0	0	0
-	0	0	0	0	0

C_o =

	=	<	>	-	-
=	0	1	1	0	0
<	1	0	1	0	0
>	1	1	0	0	0
-	0	0	0	0	1
-	0	0	0	1	0

Figure 13. Compatibility relations for GCD

6.3. Folding data flow graph (data path)

Folding the DFG and DP aims at minimizing the design cost. The folding scheme is shown in Fig.14. Two types of optimization algorithms have been developed:

- Global optimization that merges the variables,

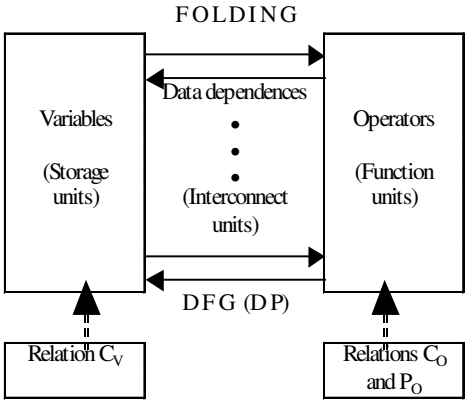

Figure 14: Folding DFG and DP

operators, and data dependencies in parallel

- Local optimization that separately merges the variables, operators, and data dependencies.

The algorithms of the first type find a cheaper design, while of the second type are faster.

7. Results

A software has been developed that implements the high-level synthesis techniques based on transformation of VHDL-models. Experimental results have been obtained on several benchmarks. Some advantages and drawbacks of the proposed method are listed in Table 4. Table 5 presents experimental results for three VHDL-models of the Bubble benchmark [5]. In model 1 all the expressions and conditional statements are split and the iteration scheme of loops is removed. In model 2 the statements located between and after loops are inserted into the loops. In model 3 the loop- and exit-statements are eliminated. The number of control steps is decreased twice due to the transformations. A non-significant increase in the register cost is due to scheduling for minimum register count has not been used. Table 6 proves that the proposed transformations lead to additional possibilities of operations execution in parallel. The use of 2 adders for the original VHDL-model of the Pid benchmark [5] has not decreased the number of control steps. The use of 2 adders for a transformed model has decreased the number of steps from 23 to 16.

Advantages and drawbacks of OBBM Table 4

Advantages	Drawbacks
Reduction in the number of: • Basic blocks in CFG • Control steps and FSM states • State transitions • Storage units • Functional units	The transformed behavioral VHDL-model differs from the initial VHDL-specification
More opportunities for pipelining, chaining, multi-cycling, and asynchronous high-level synthesis	Extending the set of control variables

Bubble benchmark Table 5

Parameter	Behavioral VHDL-model		
	Bub_1	Bub_2	Bub_3
Loops	9	9	0
Control steps	20	15	10
Registers	7	9	10
Register width (bits)	104	106	107
Multiplexers	4	4	6
Multiplexer width (bits)	68	68	70
Multiplexer inputs	13	14	18
RAM	1	1	1

Pid benchmark Table 6

Parameter	Model 1	Model 2	
	1-2 adds	1 adder	2 adds
Control steps	23	22	16
Registers	13	16	15
Register width (bits)	389	423	422
Multiplexers	8	8	10
Multiplexer width (bits)	227	227	291
Multiplexer inputs	33	34	40
Collectors	9	9	9
ROM	1	1	1

8. References

[1] R.A.Bergamaschi, *"High-Level Synthesis in a Production Environment"*, Fundamentals and Standards in Hardware Description Languages, J.P. Mermet, ed., Kluwer Academic Publishers, Norwell, Mass., 1993, pp.195-230.

[2] R.Camposano and W.Rosensteil, *"Synthesizing Circuits from Behavioral Descriptions"*, IEEE Trans. CAD, Vol.CAD-8, Feb. 1989, pp.171-180.

[3] D.D.Gajski et al., *"High-Level Synthesis: Introduction to Chip and System Design"*, Kluwer Academic Publishers, Norwell, Mass., 1992.

[4] T.Hwang, J.Lee, Y.Hsu, "A Formal Approach to the Scheduling Problem in High- Level Synthesis", *IEEE Trans.on CAD*, Vol.10, No.4, 1991.

[5] A.A.Jerraya, I.Park, and K.O'Brien, *"Amical: An Interactive High-Level Synthesis Environment"*, Proc. European Design Automation Conf.93, IEEE Computer Society Press, Los Alamitos, Calif., 1993.

[6] A.Prihozhy, *"Net Scheduling in High-Level Synthesis"*, IEEE Design & Test of Computers, Spring, 1996, pp.26-35.

[7] A.Prihozhy "Asynchronous Scheduling and Allocation", Proc. DATE 98, *IEEE CS Press*, CA, 1998.

[8] A.Prihozhy and F.Buijs "*Transformations of Behavioral VHDL-Descriptions*", National Academy of Sciences, Belarus, 1994.

[9] A.Prihozhy, *"Methods for Logical Algorithm Equivalent Transformation in VLSI CAD"*, Trans. Physics & Mathematics, National Academy Sciences, Belarus, 1992, N 2, pp.86-92. (in Russian).

[10] W.Rosensteil, *"Experiences with High-Level Synthesis from VHDL-Specifications"*, Proc. Workshop on Design Methodologies for Microelectronics and Signal Processing, Gliwice-Cracow, 1993, pp.405-412.

[11] E.Villar and P.Sanches, *"Synthesis Applications of VHDL"*, Fundamentals and Standards in Hardware Description Languages, J.P. Mermet, ed., Kluwer Academic Publishers, 1993, pp.231-262.