Behavioral Synthesis Methodology for HDL-Based Specification and Validation

D. Knapp, T. Ly, D. MacMillen, R. Miller

Synopsys Inc.
700B E. Middlefield Rd
Mountain View, CA USA 94043

Abstract

This paper describes a HDL synthesis based design methodology that supports user adoption of behavioral-level synthesis into normal design practices. The use of these techniques increases understanding of the HDL descriptions before synthesis, and makes the comparison of pre- and post-synthesis design behavior through simulation much more direct. This increases user confidence that the specification does what the user wants, i.e. that the synthesized design matches the specification in the ways that are important to the user. At the same time, the methodology gives the user a powerful set of tools to specify complex interface timing, while preserving a user's ability to delegate decision-making authority to software in those cases where the user does not wish to restrict the options available to the synthesis algorithms.

1.0 Overview

This paper describes a synthesis methodology that uses high-level synthesis (HLS) of behavioral hardware-description language (HDL) descriptions. HLS has the distinguishing characteristic that operations are automatically *scheduled,* i.e. assigned to states, as opposed to lower-level synthesis, in which operations are assigned to states by the user [1, 2, 3]. For example, in an HDL description of a square root function, an operand *x* would be loaded, a series of operations would follow, and a single result *r* would be returned. The read *x* and the write *r* might be fixed to particular states or times by a communication protocol, but the internal operations that compute the square root would be automatically scheduled.

A prospective user of HLS will then ask a number of questions. These will likely include the following:

- How can I constrain I/O operations to fall into particular cycles, or range of cycles, to meet existing protocols?
- How can I constrain I/O operations to have particular timing relationships? For example, how can I constrain a data ready strobe to be synchronous with data on data ports?
- How can I be confident that my interface timing specification really works with the surrounding hardware?
- How can I give the scheduling software optimization opportunities when my timing specification is not rigid? For example, I might not care exactly when data was transferred, as long as a corresponding strobe remains synchronized with the data. Thus the strobe and data should be locked together, but the locked strobe/ data pair of operations could move.
- How can I be confident that the synthesized hardware will really do what I want it to:
    1. In the sense that it computes the right result,
    2. In the sense that scheduling of I/O operations does not 'break' its I/O protocols.

These questions can be reformulated as requirements on the HDL description methodology to be used in conjunction with HLS:

- The original HDL description should be simulatable.
- There should be a mode wherein the cycle by cycle I/O timing of the original HDL description is preserved exactly; i.e., no I/O timing difference will be allowed between the pre-and post-synthesis descriptions. This will allow direct comparison, on a cycle by cycle basis, of the pre- and post-synthesis designs; it will also allow the user to meet the most rigid cycle-based timing protocols.
- There should be a mode wherein timing relationships between I/O signals can be simply and easily preserved across synthesis, but where 'stretching' (cycle level delay insertion) is permitted, so that the user does not have to specify exactly how many cycles a computation will take. This mode should allow manual constraints. Such a mode allows comparison of pre- and post-synthesis I/O timing between "similar points" of the pre- and post-synthesis waveforms.
- There should be a mode in which the user explicitly specifies all timing constraints without reference to the simulation behavior of the HDL; the only timing constraints inferred from the HDL description are ordering constraints among I/O operations sharing a port. This mode gives the greatest flexibility, both for optimization and for specification of complex timing relationships; it is also the most difficult to use.

We call these three modes the *cycle-fixed* IO *scheduling mode,* the *superstate-fixed IO scheduling mode,* and the *free-floating IO scheduling mode* respectively. Each has consequences for the style of HDL description and validation methodology. These modes give the user a wide range of choices in specifying I/O timing, with a corresponding range of ways in which validation of the specification and comparison of the implementation with the specification can be performed.

1.1 Structure of this paper

The balance of this paper is structured as follows. In Section 1.2, related work in this field is discussed. Following that, in Section 2, some mode-independent considerations and assumptions are described. In Section 3, the cycle-fixed mode is described in detail. Then in Section 4, the superstate-fixed mode is described. In Section 5, the free-floating mode is described. In Section 6, experience with the current software is described; finally, in Section 7 the paper is summarized and conclusions are drawn.

1.2 Related Work

High-level synthesis has been well described in the literature; see, for example, Camposano[1], Gajski[2], Maerz[3]. These tutorial papers describe the basics of HLS systems. CALLAS [4] describes work in the area of maintaining simulated behavior that is exactly the same pre- and post-synthesis; this idea is reflected

32nd ACM/IEEE Design Automation Conference ®

in the cycle-fixed mode described here. The superstate-fixed mode is related the High Level State machine of of [5], and to the behavioral finite state machines (BFSM's) of [6]. Our approach of validation through simulation is typical of current industry practice; it complements, but cannot completely replace, more formal methods [7].

2.0 Basic assumptions

The circuit to be synthesized by HLS consists of a collection of always blocks (VHDL processes); each always block will be mapped to hardware consisting of a datapath and a control FSM. Each will be synthesized separately.
Control over timing makes use of clocking statements in the source HDL. In Verilog, this can be done by use of **@**(**posedge** clock) or **@**(**negedge** clock) statements[1]. These are used to separate I/O events that are to happen in different clock cycles. Event triggers using other signals are specifically disallowed, with the exception of asynchronous reset and a special gating methodology described in Section 2.2, used for synchronizing I/O.

2.1 Reset

In order to handle resets in an intuitively appealing way, we call attention to the **always** block (VHDL **process**) that will be scheduled. In our methodology this block contains a single all-encompassing, nonterminating loop, here called *reset_loop*.

```
always begin: b1
    begin: reset_loop
        // reset sequence behaviors
        forever begin
            // normal mode behaviors
        end
    end
end
```

Inside *reset_loop* is a *reset sequence*; this consists of all behaviors associated with reset. For example, in a microprocessor the reset sequence would clear the program counter, disable interrupts, and initialize the stack pointer. The reset sequence may contain many clock cycles, e.g. to initialize a RAM. Following the reset behavior is the 'normal mode' loop, which does not terminate either; this loop contains behaviors that are executed until the next reset occurs. In a microprocessor, for example, the normal mode loop would be the fetch / execute cycle.

In order to simulate the effect of synchronous resets correctly in the source HDL description, the user must insert a statement of the form[2]

```
if (reset == 1'b1) disable reset_loop;
```

after every **@posedge** statement. This **disable** has the effect of restarting the block (process) following a clock edge upon which reset is found to be true. Simulation of synchronous resets can be matched both pre- and post-synthesis.

Another capability can also be provided in which the user declares a reset pin to the synthesis software, which then synthesizes the reset; but because the reset behavior is not encoded in the HDL, resets cannot be simulated correctly before synthesis using this technique.

Scheduling cannot handle exits triggered by a reset in the same way as other exits, because there may be read-before-write accesses in the HDL. Consider the following: In this situation, the assignments

```
begin: reset_loop
    outport <= x; // x is read before write!
    begin: main_loop
        x = v1;
        @(posedge clock);
        if (reset == 1'b1) disable reset_loop;
        x = v2;
    end
end
```

of *x* cannot be rescheduled, because this would change the observable behavior of the circuit immediately following a reset pulse. If, for example, the second write to *x* was rescheduled before the clock edge, then the output immediately following a reset pulse would be *v2* in the scheduled design; but it would be *v1* in the original description. So if we are to allow read before write in the HDL, we must either relax the requirement that all behaviors must be identical, or we must forbid movement of such side effects across clock boundaries. Side effects on variables that are always written before they are read are not affected.

2.2 Registered outputs

VHDL signals and Verilog **reg** variables behave like register or latch outputs. That is, they hold their values once set. For implementation reasons, we chose to register all outputs of HLS synthesized designs; thus a nonblocking (signal) assignment becomes a register write. This has the consequence that responses to external events cannot happen until the cycle after the external event, as shown in Fig. 1.

Figure 1 shows the behavior of a synthesized circuit where the HDL input is of the general form

```
if (Ready == 1'b1) then Data <= foo;
@(posedge clock);
```

This timing corresponds to both input and output. Notice that this

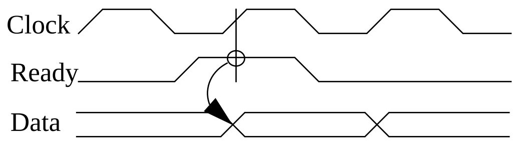

Fig. 1. Response to an external event.

timing diagram implies that the control FSM for the synthesized data path is a Mealy machine; and that the overall synthesized design is a Moore machine.

Here is an example combining an asynchronous reset and a compact busy wait on a data strobe.

```
while (strobe != 1) begin
    @(posedge clock or posedge reset);
    if (reset == 1'b1) disable reset_loop;
end
```

3.0 Cycle-fixed mode

High-level synthesis in *cycle-fixed* mode can be described by the following statement:

- Cycle-by-cycle I/O timing is identical between the pre-and post-synthesis designs.

This means that validation by simulation is straightforward: a user need merely simulate the pre- and post-synthesis designs side by side, and check for differences in the outputs. Alternatively, the synthesized design can be inserted into the original test bench without modifying the test bench. The only differences that are visible involve combinational delays in the form of setup and hold times; for example, a delta-delay setup time would become a real setup

1. In VHDL "wait until clock'event and clock = '1';" gives us a rising-edge clock.

2. In VHDL this would be "when reset = '1' exit reset_loop".

time, and a registered output pin will not transition exactly on the clock edge, as it would in the pre-synthesis simulation[1]. This is shown in Fig. 2.

Notice that this mode only constrains the I/O operations of the design. That is, the reads and nonblocking (signal) writes of the HDL are tied to particular cycles. But this still leaves optimization opportunities for the scheduling algorithm: other operations (e.g. additions, memory operations, and register reads and writes) can be shifted in time, as long as they consume data after it has been read in, and produce data in time to write it out. The I/O operations provide a series of 'stakes in the ground' that define time frames within which all other operations are free to move.

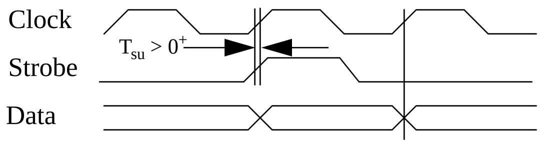

Fig. 2a. Simulation of specified design (pre-synthesis)

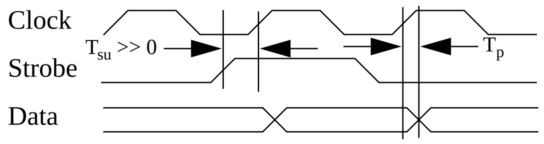

Fig. 2b. Simulation of synthesized design (post-synthesis)

Fig. 2. Comparison of simulation in cycle-fixed mode.

The main advantage of cycle-fixed mode is that the user can synthesize exactly the same timing diagram that the original HDL specification shows in simulation; thus, if the simulated HDL specification works in a particular context, then the synthesized design will also work, assuming only that setup, hold, and propagation delays, etc. as shown in Fig. 1b meet the clock cycle time.

A further advantage of cycle-fixed mode is that simulation of a zero-gate-delay model of the synthesized design will match the original specification exactly; hence a simple file difference program can be used to compare pre- and post-synthesis designs. This is expected to have a profound effect on user acceptance of HLS as a viable tool in the design cycle: users are able to simply and efficiently check the equivalence of designs before and after synthesis.

There are a number of methodological and implementation considerations that affect the way we can write and implement cycle-fixed mode. These will now be described.

3.1 Numbers of clock edges

One consequence of the commitment to maintain exact I/O equivalence in cycle-fixed mode is that numbers of clock edges cannot be varied inside the scope of loops and conditionals. To do so would distort the I/O timing of the design.

1. In zero-delay simulation one should ensure that data transitions occur slightly after clock transitions; failing to do this is the most common source of simulation mismatches. The problem comes about because of varying numbers of simulation-cycle delays in the clock and data wires of the circuit: the clock can arrive 'after' the data by an infinitesimal (zero-time) amount. This causes something analogous to a setup-time violation.

3.2 Loop boundaries

Every loop of an always block must contain at least one clock edge statement. The only exception to this is loops with constant iteration bounds, which can be unrolled during synthesis.
A loop can be thought of as a subgraph of a finite-state machine (FSM) which forms a cycle. The synthesized design will enter this cycle when the loop is executed, and leave it when the loop is exited. Such a loop is shown in Fig. 3.

```
o1 <= v1;
while (c) begin: loop
    o2 <= v2;
    @(posedge clock);
end
o3 <= v3;
@(posedge clock);
```

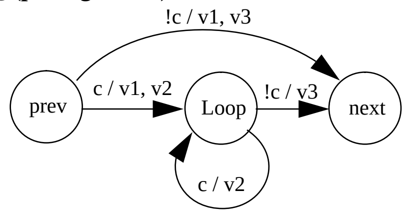

Fig. 3. Loop and corresponding state graph

The loop of Fig. 3 corresponds to the state labeled 'Loop'. During each pass of the loop, the value of *v2* will be written to the output port *o*.

The main consequence of matching this behavior is the splitting of the conditional test *c*. Notice that it was necessary, in order to capture the timing of the original, to have a state transition that bypassed the loop altogether if *c* was false when it was first tested. This means that the test must be performed in two places: once in state *prev*, and once in state *loop*. In general, it is necessary to unroll the first state of the first pass through a **while** loop in order to capture this behavior correctly.

If we wish to avoid unrolling the first pass, then it is necessary to rewrite the loop so that (1) there is a clock edge on all paths between the writes of *o1* and *o3*, and (2) there is a clock edge between the conditional test and any succeeding I/O, as shown in Fig. 4.

```
o1 <= v1;
while (c) begin: loop
    @(posedge clock);
    o2 <= v2;
end
@(posedge clock);
o3 <= v3;
```

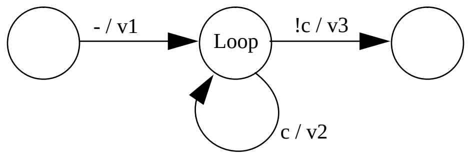

Fig. 4. Loop that does not need partial unrolling.

3.3 Conditional multicycle operations

A *multicycle* operation is one that has a longer combinational delay than the clock cycle. This imposes special constraints on synthesis in cycle-fixed mode, because it is necessary to stabilize all data and control inputs to the hardware block that implements the multicycle operation. This includes all the control inputs of all multiplexers that drive multicycle operations; clearly we cannot afford glitches on these paths.

But inserting these registers means that we need to know what to strobe into the registers one cycle before the multicycle operation is to begin. Thus we need to add extra time, under some circumstances, so that the stabilizing registers can be properly loaded. This is illustrated in Fig. 5; we assume

```
@(posedge clock);
if (input_signal == 1'b1) begin
        x = input_read_1;
        y = input_read_2;
        tmp = x + y; // 2 cycle addition
        @(posedge clock); // strobe stab regs
        @(posedge clock); // 1st cycle of add
        @(posedge clock); // 2nd cycle of add
        out <= tmp;
 end
 @(posedge clock);
```

Fig. 5. HDL description for a multicycle addition.

Notice that we needed three clock cycles to do this properly: one to get the condition and strobe the stabilizing registers, and two to perform the multicycle addition. Notice also that such delays can often be hidden, where the multicycle operations are not constrained by I/O; but that in this case there is no opportunity to hide the additional delay associated with stabilizing the inputs.

3.4 Loop pipelining in cycle-fixed mode

Loop pipelining is a technique whereby a loop can be made to act like a pipeline. Thus the loop has a relatively long latency, i.e. the time from a data input to the corresponding data output; and a shorter initiation interval, which is the rate at which data can be delivered to and read out from the loop. In cycle-fixed mode, and with some extra constraints in the other modes, a simple way to imply loop pipelining while maintaining timing equivalence is to use a delayed assignment (in VHDL, a transport delay) on the output statement. Suppose, for example, we have a loop whose latency is ten cycles, but whose initiation interval is two cycles; we can put an output write after the second clock edge statement, with a delay of eight cycles. This will simulate the same way both before and after synthesis.

```
while (condition) begin
    @(posedge clock); // 10 ns clock
    @(posedge clock);
    out <= #80 value; // delayed by 8 cycles
end
```

4.0 Superstate-fixed Mode

The *superstate-fixed* I/O mode is used where the I/O should inherit its general structure from the HDL, but where there is some freedom to shift I/O operations in time. Consider, for example, the two-wire handshaking protocol shown in Fig. 6.

The two-wire protocol is insensitive to the time between transitions; this makes it ideal for many applications. In a case like this, the only things we really need to assure in order to have correct timing are that (1) the signal transitions occur in the right order, and (2) that the transitions of *Strobe* and *Data* maintain a lockstep relationship. Beyond that, the user might not care very much how many clock cycles were inserted by scheduling; other design optimization criteria (such as the number of gates to compute the data value) might dictate more or fewer clock cycles for this transaction. The cycle-fixed mode is unsuitable for this kind of loosened specification of timing: the user could be forced to edit the code many times, with varying numbers of clock edge statements each time, looking for the best implementation.

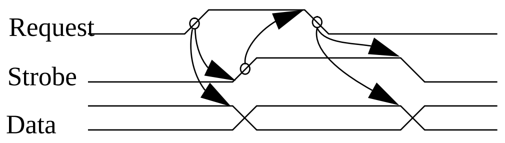

Fig. 6. Two-wire handshaking protocol.

The *superstate-fixed I/O scheduling mode* can be expressed by the following statements:

- Adjacent pairs of clock edge statements in the HDL form the boundaries of superstates.
- All I/O operations in a superstate remain in that superstate.
- A superstate may be expanded by the scheduler, which can add clock cycles to lengthen a superstate.
- All I/O writes in a superstate will always take place in the last clock cycle of the superstate.
- I/O reads may float within a superstate.

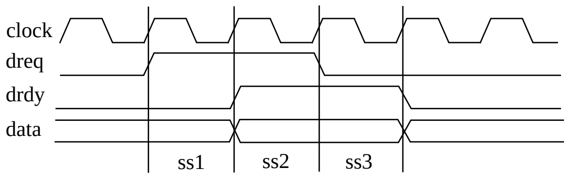

Fig. 7a. Simulation before superstate-fixed scheduling.

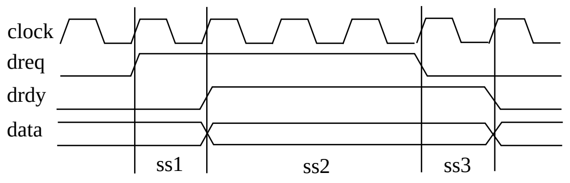

Fig. 7b. Simulation after superstate-fixed scheduling.

These rules, taken together, mean that an HDL scheduled in superstate mode will show the same signal transitions and ordering as the original HDL; but that the original timing may potentially be 'stretched' by the addition of new clock edges. This is illustrated in Fig. 7, where the original HDL simulation of an I/O transfer taking three cycles has become five cycles long by the addition of two extra cycles to the second superstate.

4.1 Protocols in superstate mode

One of the major advantages of superstate mode is that handshaking I/O protocols are not distorted by the addition of clock cycles to superstates. This has two beneficial conseqences: first, comparison of simulated pre- and post-synthesis designs is straightforward; and second, protocols that are insensitive to increased numbers of clock cycles will not be 'broken' by superstate scheduling. Hence if a design consists of many processes, each of which is to be scheduled, the use of handshaking communication in conjunction with superstate mode scheduling will ensure that the design will continue to work after synthesis. The same considerations apply to the simulation test bench as well: the test bench must communicate with the synthesized design(s) via handshaking protocols; otherwise it may have to be modified to communicate successfully with the synthesized design. This happens because the read and write operations occur at different times pre- and post-synthesis; the test bench must be able to tolerate this, or the user will have to retime the test bench.

Protocols that do not involve explicit requests and acknowledges can still be used; but care must be taken with data to be read in by the synthesized process. In particular, recall that read operations may move freely within their superstate. This means that data being presented to the synthesized circuit must be either valid during the entire superstate in which it is read, or else retimed after scheduling. This will ensure that the read operation always gets the correct data.

4.2 Constraints in superstate mode

The reason a designer would use superstate mode instead of cycle-fixed mode is that some part of the schedule does not have a fixed timing bound, and the user does not want to imply such a bound by using cycle-fixed I/O. However, the user may have a non-handshaking protocol, or a protocol that streams data once synchronization has been established by the protocol. In such cases the parts of the schedule that perform synchronization may need to be handled as if the scheduler was in cycle-fixed mode; while the other parts of the design can be allowed more freedom. For example, consider the fragment

```
while (ready == 1'b0) begin: handshaking_loop
    @(posedge clock);
end
@(posedge clock);
a1 = in_port; // label read_1
@(posedge clock);
a2 = in_port; // label read_2
@(posedge clock);
out_port <= long_involved_function(a1, a2);
out_ready <= 1'b1; // label done
@(posedge clock);
```

Here the external logic provides the data for *read_1* and *read_2* in the two cycles after the signal *ready* goes true; the synthesized system must pick it up then, or the protocol will be broken. Furthermore, insertion of extra cycles in the loop *handshaking_loop* will cause the interface to behave unpredictably. Thus cycle-fixed mode would seem to be indicated. However, suppose that there is no need for the output to show up until 20 cycles after the input has been delivered; the designer will thus want to allow the scheduler authority to add cycles to the last superstate, and rely on a test of the *out_ready* pin to synchronize the data on *out_port*. Thus stretching can be allowed in the last superstate, but not in the first three.

This can be done by means of explicit point-to-point scheduling constraints; that is, constraints that tie two labeled operations together in a particular timing relationship. A constraint set that would serve the purpose is

1. The time from the beginning of *handshaking_loop* to its end should be exactly one cycle.
2. The time from the end of *handshaking_loop* to the beginning of *read_2* should be exactly one cycle.
3. The time from the end of *handshaking_loop* to the data ready strobe *done* is no greater than 21 cycles.

Notice that these constraints are not part of the HDL; but they are a necessary part of the methodology. They can be implemented as pseudo-comments, as attributes, or as directives in a separate scheduler command file. Notice also that they can be applied to non-I/O operations as well, in all three modes, to give the user a little extra control over the scheduling process.

4.3 Superstate HDL methodology

Superstate mode defines superstates as containing the I/O operations that fall between adjacent pairs of clock edge statements. This definition has the consequence that sometimes an HDL prepared for superstate mode needs clock edge statements that are not needed in cycle-fixed mode. For example, the text of Fig. 3 is ambiguous when the HDL is considered as input for superstate mode. This comes about because two writes are separated by a conditional @posedge. If the loop condition is true, then the writes should be in different superstates; if it is false, then they should be in the same superstate. Clearly there is no unique static assignment of I/O operations to superstates in this situation.

Furthermore, there is an implicit ordering of operations conferred by the sequencing of the HDL text; this ordering cannot be allowed to come into conflict with the ordering conferred by the migration of reads into any cycle of their superstate and writes into the last cycle of their superstate.

The HDL methodology rules that prevent ambiguities and contradictions in superstate mode are:

1. A superstate that contains a loop continue is called a continuing superstate. Implicitly, the last superstate of a loop is also a continuing superstate. A continuing superstate and the first superstate of the loop are really the same superstate; there is no clock statement on the execution path going from one to the other. If a continuing superstate contains a write, then the first state of the loop cannot contain any I/O, because a write belonging to the continuing superstate would be migrated to the end of the first loop superstate: this would result in a violation of the HDL's ordering constraints.
2. A superstate that contains a loop beginning cannot include both an I/O write before the loop beginning and any I/O operation inside the loop. For example,

```
@(posedge clock);
out_port <= write1_data;
while (cond) begin
    read1_data = in_port; // Illegal!
    @(posedge clock);
    ...
end
```

   the write in this fragment conflicts with the read in the beginning of the loop; they are in the same superstate.
3. A write cannot precede a while loop that is succeeded by any I/O operation, unless there is a clock edge statement between either the write and the loop begin, or between the loop end and the second I/O operation.
4. A loop having a superstate in which both a loop exit[1] and an I/O write are located must have a clock edge statement between the loop end and the next I/O operation.
5. A conditional clock edge (e.g. an @edge on one branch of a conditional) cannot be used to separate a write from another I/O operation. This fragment is illegal for that reason.

```
out_port <= v1;
if (cond) @(posedge clock);
v2 = in_port;
```

5.0 Free-floating I/O mode

It will sometimes be the case that a user will need to convey more freedom to the scheduler than is allowed by the superstate I/O mode. For example, the user may wish to allow two unrelated writes to be permuted. Consider the fragment of Fig. 8.

In this situation, the user might not care whether the first or the second function happens first; indeed, they could be interleaved and the user might not care. But neither superstate nor cycle-fixed mode will permit permutation of I/O operations and waits; so a more powerful mode is needed.

1. Other than a reset exit. Reset exits can be ignored after a preprocessing step in which they are detected and global reset behavior is enacted, as explained in Section 2.

The *free-floating* mode is characterized by implicit constraints on single I/O ports and explicit user constraints.

```
a1 = in_port1;
a2 = in_port2;
@(posedge clock);
out_port_1 <= long_function_1 ( a1, a2 );
@(posedge clock);
b1 = in_port3;
b2 = in_port4;
@(posedge clock);
out_port_2 <= long_function_2 ( b1, b2 );
```

Fig. 8. Writes to out_port_1 and out_port_2 may be permuted.

Implicit I/O port constraints are derived directly from the HDL text and are imposed on the sets of reads and writes that occur on a single port. These are formed into partially ordered sets, one for each port, where the ordering is derived from a static execution trace analysis of the source HDL. The schedule constructed by synthesis can only transpose two members of one of these sets if there is no ordering relationship between them.

This, however, says nothing about ordering of reads and writes that occur on different ports, which must be explicitly constrained by the user, by means of the explicit two-point constraints described in Section 4.2.

For example, in our experience a common early mistake in free-floating mode is to expect a data strobe's timing to be fixed with respect to that of the data being strobed. This will not necessarily be the case if the user does not issue explicit constraints.

The downside of this mode is the number of explicit constraints that the user must construct. This can easily be comparable in numbers of lines to the HDL input itself. In addition, it is very easy to get such constraints wrong, or to forget a crucial constraint; hence the cycle-fixed and superstate modes are simpler and less error-prone to use.

6.0 Experience

Support for the methodologies discussed above has been built into a commercial product, the Synopsys Behavioral Compiler(TM). This product is currently in use at a number of sites. Of these, about half use Verilog as their input HDL; the rest use VHDL.

Experience to date indicates that the superstate mode is usually the most convenient from the standpoint of ease of specification of complex timing behaviors. The next most convenient is usually the cycle-fixed mode. The reason for this is that the power of the free-floating mode comes at the price of manually added constraints; while the cycle-fixed mode requires the user to add clock cycles to the source HDL when, e.g., the duration of a particular loop is to be changed.

From the standpoint of ease of validation of results, the cycle-fixed mode is usually a little more convenient than the superstate mode. This is because the handshaking protocols necessary to get the design talking to the test bench after superstate-mode scheduling must be designed and written in both the test bench and the specification; or alternatively the test bench timing must be modified to match the schedule of I/O of the post-synthesis design.

One area in which the free-floating mode seems to be more convenient than the others is in that of exploration. Here the user is more interested in getting a rough idea of the cost and speed of a design or algorithm, than in getting its interfaces exactly right. In this context, the ease of turning the design around and the high degree of freedom from methodological constraints makes it simpler to change the design and resynthesize to see what the overall results are. Then when the general outlines of the algorithms, representations, etc. are clear the user can begin to worry about the detailed I/O timing.

The overall effort of getting I/O interfaces right using these three modes is usually less than the effort spent in getting the best possible quality of results. Even with behavioral synthesis, HDL writing styles still can have a large impact on the quality of the synthesized circuit. Examples that can affect synthesis quality are: loop ordering, assignment of variables and arrays to memories, choice of loop pipeline initiation intervals and latencies, pipelined components, embedding combinational logic in reusable function blocks, the tradeoff between multicycle operations and fast clock rates, and the partitioning of the design into datapath/controller subunits (i.e. always blocks; in VHDL, processes). All are potentially of great importance to the quality of results, and all represent true engineering decisions that must be carefully considered if a really good design is to be achieved.

7.0 Conclusion

We have presented HDL methodologies for the synthesis of various kinds of I/O timing and protocols, and for simulation-based validation of the synthesized design against the original specification. Three modes of scheduling I/O operations have been presented:

1. Cycle-fixed, in which the design has exactly the same cycle-level I/O timing before and after synthesis;
2. Superstate-fixed, in which I/O operations are grouped by pairs of @posedge statements; post-synthesis timing behavior is a (potentially) stretched version of the pre-synthesis timing; and
3. Free-floating, in which the only constraints on I/O scheduling are either between operations sharing a port or supplied by the user.

Some of the implications of the scheduling modes were described. In the cycle-fixed and superstate modes, these involve the placement of clock edge statements, loop boundaries, conditionals, and I/O operations; while in the free-floating mode there are no rules of this kind.

Experience with production software which implements these methodologies has been described, and conclusions based on that experience have been drawn.

8.0 References

1. R. Camposano, W. Wolf. Trends in High-Level Synthesis. Kluwer, 1991.
2. D. Gajski, N. Dutt, A. Wu, S. Lin. High-Level Synthesis: Introduction to Chip and System Design. Kluwer, 1992.
3. S. Maerz, High Level Synthesis. In *The Synthesis Approach to Digital System Design*, P. Michel, U. Lauther, P. Duzy, eds., Chapter 6. Kluwer, 1992.
4. A. Stoll and P. Duzy, High-Level Synthesis from VHDL with Exact Timing Constraints, *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pp. 188-193, IEEE, 1992.
5. R. A. Bergamaschi, A Kuehlmann, S-M. Wu, V. Venkataraman, D. Reischauer, and D Neumann, A Methodology for Production Use of High Level Synthesis, Workshop Proceedings, Sixth International Workshop on High Level Synthesis, (1992).
6. W. Wolf, S. Takach, C.-Y. Huang, R. Manno, E. Wu. The Princeton University Behavioral Synthesis System. *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pp. 182-187, IEEE, 1992.
7. K. L. McMillan, Fitting Formal Methods into the Design Cycle, *Proceedings of the 31st ACM/IEEE Design Automation Conference*, pp. 314-319, IEEE, 1994.