Semantics and Synthesis of Signals in Behavioral VHDL

Loganath Ramachandran, Frank Vahid, Sanjiv Narayan and Daniel D. Gajski
Department of Information and Computer Science
University of California, Irvine, CA, 92717

Abstract

Signals are a fundamental part of VHDL behavioral descriptions. There are many kinds of VHDL signals, each possessing complex and hence often misunderstood semantics. Synthesis tools often inadequately address synthesis of global signals. In this paper, we introduce conceptual hardware to understand the semantics of various signals. We then discuss techniques and issues in synthesizing actual hardware for shared signals, thus reducing current restrictions imposed by synthesis tools on allowable VHDL behavior.

1 Introduction

Since the standardization of VHDL [1], several efforts have been made to develop behavioral synthesis tools which synthesize structure from VHDL behavioral descriptions. The use of Signals (and ports) are a key aspect of a typical description. Signals are different from variables, in that they have a notion of time associated with them. This allows modeling of concurrency in the system and allows sharing of signals between concurrent processes. The formal semantics of VHDL signal attributes is presented in [2]. Figure 1 shows the use of signals and ports in a simple VHDL description.

There are many kinds of VHDL signals, and their semantics are complex. As a result of this fact, many previous synthesis approaches [3, 4, 5, 6] do one or more of the following:

• Discuss synthesis of signals incompletely or at a general level. For example, previous efforts have researched signals in the context of a single process only, ignoring the effects of several processes driving the same signal.

• Synthesize hardware for signals which is not functionally correct. For example, some approaches do not distinguish between the three VHDL signal kinds. Some approaches always map signals to wires, leading to incorrect results.

• Restrict the allowable use of signals in the VHDL description to aid the synthesis process. For example, a common restriction permits no more than one VHDL process to write to a given signal.

Such restrictions simplify the synthesis process tremendously, but they also undermine a signal's usefulness in a VHDL description. Many synthesis tools would find it difficult to handle even the simple description in Figure 1.

In this paper, we present a synthesis scheme that would enable synthesis tools to eliminate many restrictions on the use of signals and to create hardware that correctly matches VHDL signal semantics. Our approach is to first clearly define the semantics of the various kinds of VHDL signals and related constructs like ports and resolution functions. Even though descriptions of signal semantics are abundant [1, 7], confusion is still quite common [8]. We therefore introduce the notion of conceptual hardware, to simplify the understanding of signal semantics [9]. This conceptual hardware uses well-understood components like wires, latches, and buffers to make clear many commonly misunderstood issues related to signals. However, the conceptual hardware bears no relation to any particular implementation strategy.

Once the semantics are clearly defined, we discuss synthesis of actual hardware for signals, ports, and resolution functions. We show the relationship between our conceptual hardware and the synthesized hardware for different kinds of signals. We show that ports are synthesized almost identically to signals. We would like to emphasize at this point that this paper focusses only on signal semantics and synthesis external to the processes in the VHDL description. [10, 11] presents many research efforts that deal with issues like scheduling and register allocation which concern synthesis issues within a process. For the same reason, local variables are not discussed in this paper.

2 Resolution Function

2.1 Semantics

Any discussion of VHDL signal semantics would be incomplete if resolution functions were not included. In general, a VHDL description consists of a set of processes communicating using global signals. Thus a global signal represents a virtual wire that connects two or more independent processes. Due to the independent nature of the
In VHDL, signals are generally used to model communication between processes. A signal can model a wire, bus, or register and could represent any data type (like integers, bits, arrays or user-defined records).

In VHDL, signals can be of three kinds: simple (or no-kind), bus, or register. [1,7] provide details of the syntax for a signal declaration. The conceptual hardware (shown in Figure 2) clearly shows the signal semantics by using well-understood components. The final implementation of the signal need not resemble this conceptual hardware. The table in Figure 3 summarizes the signal semantics.

Figure 1: A simple VHDL example with ports and signals.

 processes it is possible that a global signal is written by more than one process at a given instant. Hence, it is necessary to determine a single value for the signal from the set of values written by different processes. In VHDL, resolution functions are used for this purpose. Figure 2(a) shows the conceptual hardware for a resolved signal $S$.

The input to a resolution function is a set of values of the signal's type. Hence each process must provide a value of that type. If a signal is composite, each process must assign a value to all of its subelements.

The VHDL language does not specify how to use resolution functions in a model. The resolution function could be made very complex. However, it should be remembered that in hardware, the value of a wire with multiple drivers is determined by the technology. Since a VHDL model is meant for hardware, resolution functions must be used to model only these technology-specific characteristics and should not contain any other functionality.

2.2 Synthesis

Since resolution functions are to be used only to represent the target technology's wire characteristic, the synthesis tool need not synthesize the resolution function. However, the implementation technology for the synthesized circuit should be one in which the specified resolution function holds.

3 Signals in VHDL

Signals are generally used to model communication between processes. A signal can model a wire, bus, or register and could represent any data type (like integers, bits, arrays or user-defined records).

In VHDL, signals can be of three kinds: simple (or no-kind), bus, or register. [1,7] provide details of the syntax for a signal declaration. The conceptual hardware (shown in Figure 2) clearly shows the signal semantics by using well-understood components. The final implementation of the signal need not resemble this conceptual hardware. The table in Figure 3 summarizes the signal semantics.

3.1 Semantics

3.1.1 Simple (No-Kind) Signals

The most important characteristic of the simple (or the no-kind) signal type is that these signal drivers cannot be turned off (i.e., a null assignment "$S <= null;" is not
permitted). Consequently, a simple signal has all of its drivers active at all times, and the value of the signal in the presence of multiple drivers is determined by the resolution function associated with the signal.

As shown in Figure 2(a) each signal instantiates a virtual wire in conceptual hardware. All virtual wires are shown as one bit wide signals, even though the actual signal may be several bits wide in reality. The virtual wire holds a value of the type specified in the signal declaration. All reads of the signal by any process are of this virtual wire's value.

When a value to a signal in a process instantiates virtual storage in that process. The storage is of the same type and width as the signal. This storage is referred to as the process driver of the signal. To see why virtual storage is needed within each process that writes to the signal in our conceptual hardware model, let us consider the following VHDL code segment:

\[
v := 1;
S <= v;
\]

wait for 50 ns;

\[
v := 2;
\]

According to VHDL signal semantics, the above process should continue to drive \( S \) with the value 1 even after the value of \( v \) is set to 2. Thus a virtual storage of \( S \) which holds the value 1 is implied. This virtual storage (shown as a virtual latch in conceptual hardware) could get eliminated during synthesis.

When multiple processes write to a global signal, a virtual latch is automatically implied in each of these processes (as shown in Figure 2(a)). The resolution function (represented by a dark oval) determines the resultant value for the global signal, which is read by different processes.

### 3.1.2 Bus-Kind Signals

The key difference between a bus signal and the simple signal described above is that a process can shut-off its signal driver to a bus signal. This is done by assigning a null value to the signal as in \( S <= \text{null;} \). In this case the resolution function has to also determine the resultant value when all signal drivers are turned off.

In our conceptual hardware model, we show this additional capability by introducing a virtual buffer (Figure 2(b)). This virtual buffer could either contribute a value or contribute nothing to the virtual wire, based on whether the driver is turned on or off.

### 3.1.3 Register-Kind Signals

The characteristics of the register-kind signal is given in Figure 3. Register-kind signals are identical to bus-kind signals with one exception. When all drivers are shut-off, a register-kind signal retains its last (resolved) value. This implies additional storage in our conceptual hardware, as shown in Figure 2(c). The virtual wire's resolved value is latched when at least one driver is on. All reads are from the output of this latch; hence when all drivers are shut-off, reads will be of the last latched value.

### 3.2 Synthesis

#### 3.2.1 Synthesizing Hardware for Signals

Having introduced a conceptual hardware model for the different signal kinds, we now consider the issue of implementing signals using real hardware. Figure 4 shows the hardware implementation strategy for the three signal kinds. The synthesis of real hardware from the conceptual model is not trivial in many cases.

<table>
<thead>
<tr>
<th>Number of Drivers Allowed</th>
<th>Signal Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple</td>
<td>All</td>
</tr>
<tr>
<td>Multiple</td>
<td>All</td>
</tr>
<tr>
<td>Multiple</td>
<td>All</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resolution Function Needed?</th>
<th>Signal Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only when multiple</td>
<td>All</td>
</tr>
<tr>
<td>drivers exist</td>
<td>All</td>
</tr>
<tr>
<td>Drivers can be turned off?</td>
<td>All</td>
</tr>
<tr>
<td>Value when Multiple</td>
<td>All</td>
</tr>
<tr>
<td>Drivers Active</td>
<td>All</td>
</tr>
<tr>
<td>Value determined by Res.</td>
<td>All</td>
</tr>
<tr>
<td>function</td>
<td>All</td>
</tr>
<tr>
<td>Value when Zero</td>
<td>All</td>
</tr>
<tr>
<td>Drivers Active</td>
<td>All</td>
</tr>
<tr>
<td>Value determined by Res.</td>
<td>All</td>
</tr>
<tr>
<td>function</td>
<td>All</td>
</tr>
<tr>
<td>Last resolved value of the signal</td>
<td>All</td>
</tr>
</tbody>
</table>

**Figure 3:** Characteristics of various signal kinds.

While virtual wires can carry values of any data type, real wires carry bits (i.e. voltages). Hence the first task of synthesis is bit-encoding: the conversion of all data types to sets of bits. Scalar signal-types such as booleans, integers, characters, strings, and enumerations are encoded into one or more bits. For example, a signal of type "integer range 0 to 20" is encoded into five bits, or an array of integers becomes an array of bit-vectors. A composite signal-type such as a record is treated as if each of its elements is a unique signal. Other types such as access types (pointers) and file types are simply too detached from hardware to be allowable. Algorithms for bit-encoding are beyond the scope of this paper.

After bit-encoding all signals to take care of complex data types, signals that can be implemented exactly as shown in the conceptual hardware model are selected using certain heuristics. We shall refer to such selected signals
as latchable signals. The other signals are not selected because they would require a large number of latches. For example, an array of 1000 16-bit bit-vectors would require a latch of 16000 bits within each process, which is infeasible. Such signals are implemented in a memory and are discussed later in this paper.

```
procedure GENERATE_SIGNAL_HARDWARE (S: signal) begin
for each process P(i) which writes to S do
  Let S(i) be the value of the signal S driven by process P(i)
  If S is NOT updated when a current source of S is updated then
    Implement S as a latch within P(i), LATCH(P(i))
    LATCH(P(i)) is enabled whenever S is written to P(i)
    The value S(i) is the output of LATCH(P(i))
  else
    Implement S as a wire driven by process P(i), WIRE(P(i))
    The value S(i) is the value on WIRE(P(i))
  endif
if (S.kind = bus I register) and (S has a null assignment in P(i)) then
  Add a tristate buffer, TB(P(i)) to WIRE(P(i)) or to LATCH(P(i))
  TB(P(i)) is disabled for the time duration between a null assignment and the next assignment to S in P(i)
  The value S(i) is then the output of the buffer TB(P(i))
else
  endif
endfor
Connect all the outputs S from each process P which write to S
Let this connection be called JOIN(S)
if(S.kind = register) and (S has a null assignment in all processes) then
  Add an external level sensitive latch, EXT_LATCH(S).
  EXT_LATCH(S) gets its input from JOIN(S).
  EXT_LATCH(S) is enabled whenever any process writes to S.
  The output of EXT_LATCH(S) represents the signal S
else
  endif
end
```

Figure 5: Procedure for synthesis of latchable signals.

Latchable signals are implemented with hardware that is similar to that Figure 2. Since the latch holds bits, a tristate buffer is used for the virtual buffer. The high-level synthesis algorithm applied to each process individually must determine the details related to the latch and the buffer within the process. The synthesis algorithm may also eliminate the latch and/or buffer when they are not necessary. For example, consider the VHDL code segments shown below:

```
P: process(B,C)
begin
  A <= B + C;
end process P;
```

The wire for A can be connected directly from the output of an adder with inputs B and C, because the sensitivity list of P indicates that A changes whenever there is a change in B or C. The general optimization rule for latch elimination is: A latch is needed in a process that drives a signal if and only if a current source of the signal S is updated with a new value in a clock cycle, but the signal itself is not. A source is any symbol appearing on the right side of an assignment to S.

Figure 6: Simplifications made possible by restricted resolution function.

Figure 5 outlines the synthesis algorithm to obtain a hardware implementation for a given signal. The synthesis algorithm determines how many processes write to the signal, whether a latch and tristate-driver is needed for the signal within each process, and finally how the values written by the several processes are combined.

Conceptually we represented a process' driving value of a signal and the read value as separate buses, to illustrate the job of the resolution function. In implementation, only one bus is needed. This simplification can be made only when the resolution function reflects a physical wire's characteristics and does not possess other functionality. Figure 6 illustrates this concept.

3.2.2 Memory Signals and Arbitration

When it is not feasible to implement a signal as a latch it is implemented as a memory. We shall refer to these as memory signals. We assume that all the memory signals are implemented as a two dimensional array of bit-vectors. (Other memory types are converted to two dimensions).

The memory resides outside the processes. Each access to the memory by a process is replaced by a channel consisting of address, data, and control. Latches may be needed within the process to hold the address and data values, since they are essentially global signals.
This external memory can be accessed by all the processes in the system. It is very difficult to ensure that all the processes access the memory at different times. Thus conflicts are possible. The only way to avoid these conflicts is to explicitly arbitrate between them.

Arbitration is accomplished by adding functionality to existing processes' behaviors. Each process must request permission to access the memory. After performing the access, the process must relinquish its access rights. This handshake is implemented with two signals (request, acknowledge). Also, a new arbiter process is created.

```
procedure GENERATE_ARBITER (MEM: 2D-array, PLIST: priority-list)
begin
    Let PLIST represent a list of the processes which access MEM, ordered in descending order of priority
    for each process P which accesses MEM do
        Proceed all accesses of MEM in P by the following:
        MEM_req_i <= '1';
        wait until (MEM_ack_i = '1');
        Append the following after all accesses of MEM in P:
        MEM_req_i <= '0';
        wait until (MEM_ack_i = '0');
    endfor
    while PLIST is not NULL do
        Generate Arbiter Process
        P_k = head(PLIST); PLIST = tail(PLIST);
        Append the following statements to the arbiter process:
        if MEM_req_k = '1' then
            MEM_ack_k <= '1';
            wait until (MEM_req_k = '0');
            MEM_ack_k <= '0';
            end if;
        endwhile
end;
```

Figure 8: Procedure for generation of VHDL description of a Fixed Priority Memory Arbiter.

Arbitration limits access to a given resource. Access constraints can arise when a memory is bound to a library component with fewer ports than needed or when buses are merged to satisfy pin constraints. Several alternative arbitration models are shown in Figure 7. In Figure 7(a), each access is statically assigned to a specific port. Here, concurrent accesses over a port are arbitrated. In Figure 7(b) and (c), accesses are assigned to ports dynamically. The arbiter must limit accesses to two processes at any time. In Figure 7(b), each process can access the ports directly, so the arbiter merely informs a process which port it has permission to use. In Figure 7(c), all communication occurs through the arbiter, so that it must physically route each process address/data to the available port.

The first arbitration model is simple, but may result in poor performance. A process may wait for a port to become available even though another port is unused. The second model alleviates this static decision problem, but requires a more complex arbiter. Also, each process has to be able to route its accesses to multiple buses. The third model eliminates this process complexity, but again results in a complex arbiter which may also have routing congestion problems.
quired to control the accesses to the memory. The results of such synthesis are shown in Figure 9(c).

4 Ports

4.1 Semantics

An entity may have ports for interfacing with an external environment. The semantics of ports are identical to that of signals except that in addition, ports have an associated mode (in, out, inout) which constrains the direction of the data flow allowed through the port.

In Figure 10(a) and Figure 10(b) we show straightforward conceptual hardware representations of in and out ports. The value of an out port is the resolved value from the resolution function. In Figure 10(c), we show the conceptual hardware for an inout port. Note that the external environment could also write to the port. The resolution function resolves all these values, to determine the actual value of the port. In conceptual hardware, the inout port is separated into an input and an output virtual wire. This is because the input to the port from the external environment may be different from the resolved value. With a restricted resolution function, the two wires can be merged into one.

VHDL also allows an out or inout port to be declared as bus-kind. The semantics and the conceptual hardware discussed for bus-kind signals is also applicable to ports. However, a port may not be declared as register-kind.

4.2 Synthesis

Since ports resemble signals in many aspects the synthesis also proceeds very similar to synthesis of hardware for signals. The conceptual hardware for ports shown in Figure 10 would be the actual hardware. However simplifications are possible in the case of inout port. Values for an inout port can be implemented as a single bus although they were conceptually shown as separate buses. This simplification is possible only when the resolution function reflects the ports physical characteristics and does not include any other functionality.

5 Conclusions

The research presented in this paper eases the restrictions placed by existing synthesis systems on the VHDL that can be used to specify designs. In order to obtain functionally equivalent hardware from VHDL descriptions, it is essential to understand the semantics of VHDL constructs, especially for signals driven by several processes. We have introduced a conceptual hardware representation to explain the semantics of signals, ports, and resolution functions. We have also given procedures to synthesize hardware for such constructs. While many restrictions have been eliminated for latchable signals, composite signals which are mapped to memories still possess some use restrictions, and may require arbitration which then adds new functionality. We see no alternative to these restrictions, unless there is a change in VHDL which adds a global memory construct that more closely matches synthesizable hardware.

References


621