Semantics and Synthesis of Signals in Behavioral VHDL

Loganath Ramachandran
Frank Vahid
Sanjiv Narayan
Daniel D. Gajski

Technical Report #92-28
March 23, 1992

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

ramachan@ics.uci.edu
vahid@ics.uci.edu
narayan@ics.uci.edu

Abstract

Signals are a fundamental part of VHDL behavioral descriptions. There are many kinds of VHDL signals, each possesing complex and hence often misunderstood semantics. The result is that synthesis tools often inadequately address synthesis of signals. In this report, we first make clear the semantics of the various signal kinds shared by multiple processes through the use of conceptual hardware, rather than just text. Second, with the semantics firmly understood, we discuss techniques and issues in synthesizing actual hardware for shared signals. This information can be used to take a step towards synthesizing correct hardware from VHDL descriptions while greatly reducing current restrictions imposed by synthesis tools on allowable VHDL behavior.

Contents

List of Figures

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. A key aspect of an HDL behavioral description is the signal (and port), which is an extension of a variable in that a signal possesses a value at a specific time, and hence can be used when concurrency is modeled. 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 [2, 3, 4, 5] 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, most approaches do not distinguish between the three VHDL signal kinds (simple, bus, and register), even though their functionalities are different. Some approaches always map signals to wires, which is incorrect in situations when storage of the signal's value is necessary.
- Restrict the allowable use of signals in the VHDL description to a synthesizable subset of functionality. For example, a common restriction permits no more than one VHDL process to write to a given signal, which greatly simplifies synthesis but also severely undermines a signal's usefulness. Another common restriction forbids the use of signals that are two-dimensional (e.g. an array of bit-vectors).

Many synthesis tools would be unable to correctly handle even the simple description in Figure 1. The information in this report enables synthesis tools to eliminate many restrictions on signals and to create hardware that correctly implements intended functionality, thus providing for a large step toward the goal of synthesizing correct hardware for general VHDL behaviors. The formal semantics of VHDL signal attributes is presented in [6]. Our approach is to first *clearly define* the semantics of the various kinds of VHDL signals, as well as the related constructs of ports and resolution functions. This is achieved by describing much of their semantics using a *conceptual hardware*, rather than just a textual description as is normally used. The conceptual hardware uses well-understood components, such as a latch with a load line, to concisely indicate functionality. An analogous use of conceptual hardware would be defining the semantics of a multiplexor through the use of AND and OR gates. Anyone familiar with such gates will then understand the functionality of a multiplexor. However,

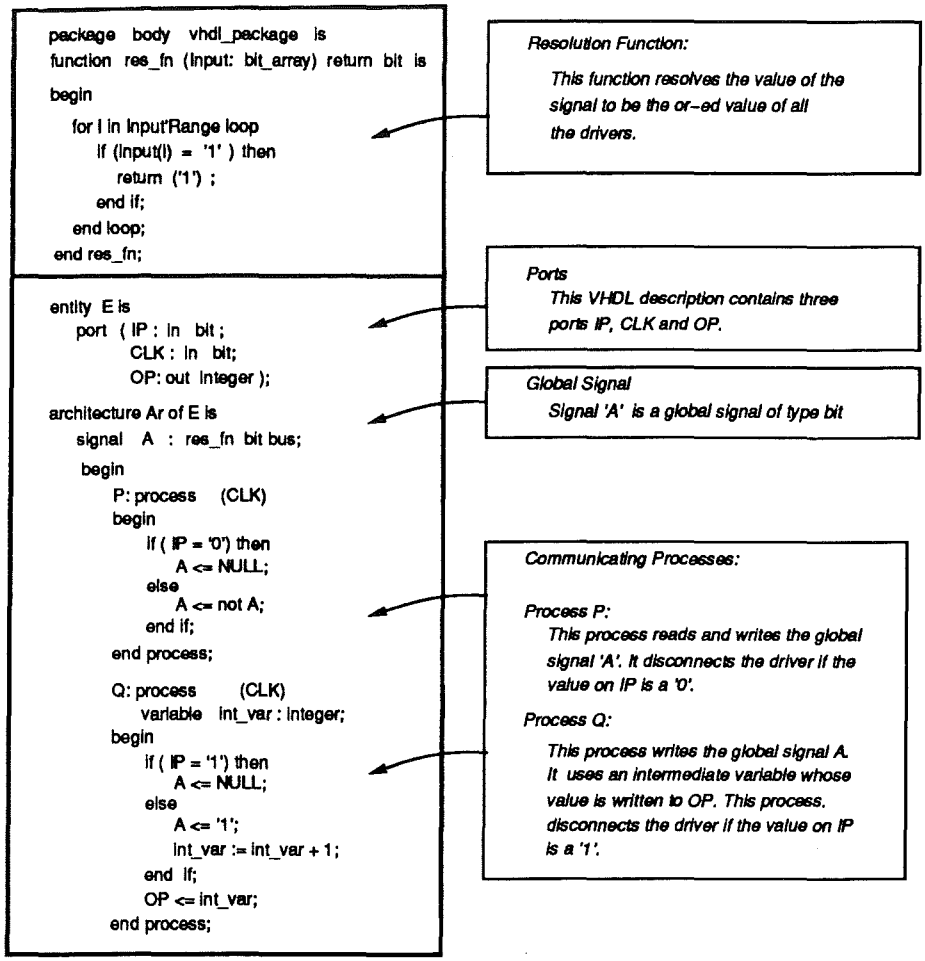


Figure 1: A simple VHDL example showing the use of ports and signals

the conceptual hardware bears no relation to any particular implementation of the multiplexor, i.e. an implementation may use NOR gates. Likewise, our conceptual hardware uses wires, latches, and buffers to make clear many commonly misunderstood issues related to signals, although synthesized hardware need not match this conceptual hardware.

Once the semantics are clearly defined, we discuss synthesis of actual hardware for signals, ports, and resolution functions. We show that our conceptual hardware can serve as real hardware for signals of certain data types, while in a few cases different hardware is necessary for other data types. We show that ports are synthesized almost identically to signals. We also discuss restrictions on allowable VHDL behavior which, although far less constraining than those in previous approaches, are still necessary to assure that synthesis can generate feasible hardware. 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. Synthesis issues such as scheduling which are internal to a process are not discussed.

2 Resolution Function

2.1 Semantics

In order to understand the semantics of the various signal-kinds in VHDL it is necessary to briefly examine resolution functions. 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(e.g. P and Q in Figure 1). Due to the independent nature of the processes it is possible that a global signal is written by more than one process at a given instant.

With multiple drivers co-existing on a single wire it becomes necessary to *determine a single value* for the signal. In VHDL, resolution functions are used for this purpose. Resolution functions determine a single value for a signal from the set of all values contributed by the different drivers of the signal. In Figure 1 we show a typical use of a resolution function. The conceptual hardware for the resolution function is shown as a bold dot in Figure 2.

There are certain semantic requirements that must be followed when using resolution functions in a VHDL model. Since the input to a resolution function is a set of values of the signal's type, each process must provide a value of that type. Hence, if a signal is composite, each process must assign a value to all of the composite signal's subelements. For example, if a signal A's type is an array of 1000 integers, a process must assign all 1000 integers of A, and these 1000 integers make up a value that is passed to the resolution function.

The VHDL language does not specify how to use the resolution function's capability. The modeler could decide to make the resolution function very complex. However, it should be remembered that in hardware, the value of a wire with multiple drivers is determined by the technology (e.g. ECL,TTL, or

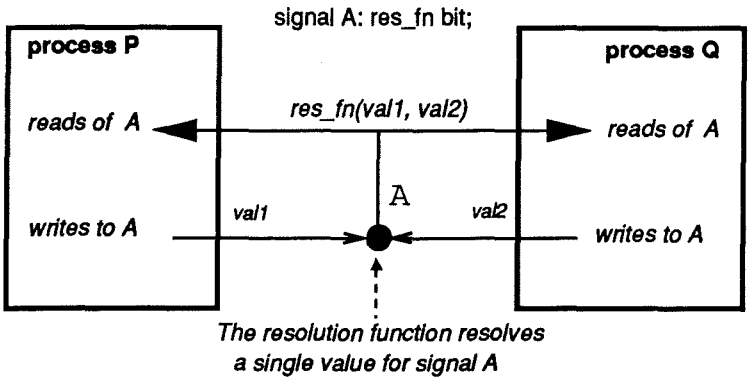


Figure 2: Conceptual hardware for a resolution function

CMOS). 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.* For example, in some technologies, multiple drivers on the same wire result in a wired-or value. This can be modeled by using a resolution function.

2.2 Synthesis

When synthesising hardware from a given behavioral description, it is necessary to consider the type of constraints that a particular resolution function imposes on the synthesis process. Since resolution functions are to be used only to represent the target technology's wire characteristic, the synthesis tool ignores the resolution function during synthesis. However, the implementation technology for the synthesized circuit should be constrained to the technology where the specified resolution function holds.

3 Signals in VHDL

Having discussed resolution functions, we shall now describe signal semantics using conceptual hardware. We would again like to emphasize that conceptual hardware is used only to understand signal semantics, and for now should not be confused with synthesized hardware.

Signals are generally used to model communication between processes. Each process which writes to a signal is called a *driver* for that signal. The most basic view of a signal is that it is a value which can be read and written by each process to which it is visible. A signal can model a wire, bus, or register. Signals can be of any data type, such as an integer, bit, pointer, array, or an arbitrarily complex user-defined type.

The syntax for a VHDL signal declaration is:

signal identifier : [resolution-function-name] type [signal-kind] [:= expression];

signal-kind ::= **register** | **bus**

We have simplified the actual VHDL signal syntax to focus on the relevant issues. Boldface items are keywords. Bracketed items are optional. The identifier is the name by which the signal is refered to in a process. [1, 7] provide english descriptions of the signal syntax. Signals can be of three kinds: simple (or no-kind), bus, or register. Their semantics are examined separately. The conceptual

hardware for the three signal kinds is shown in Figure 3. The semantics for the three signal kinds is summarized in Figure 4.

3.1 Semantics

3.1.1 Simple (No-Kind) Signals

First, consider the simple signal declaration:

signal S : [resolution-function-name] type [:= expression];

A simple signal can be written to by multiple drivers. Resolution functions are not needed for the signal if there is exactly one driver. The drivers of a simple signal 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.

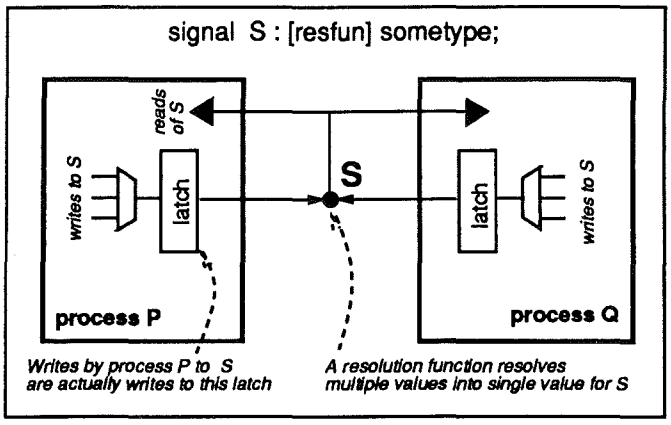
The declaration of a signal instantiates a *virtual wire* in conceptual hardware. The wire has a value of the type specified in the declaration. Possible types include scalars (e.g. integers, bits, reals, strings) and composites (e.g. arrays, records) which are composed of scalars. In conceptual hardware, the virtual wire for any signal is assumed to be one bit wide even though the signal may be several bits wide in reality. All reads of the signal by any process are of this virtual wire's value.

Figure 3(a) shows the conceptual hardware of a simple signal. The virtual storage is implemented by a virtual latch. The bitwidth of the latch is shown to be one in conceptual hardware. The value of the signal S is actually the value on the virtual wire emerging from the dark oval representing the resolution function associated with the signal.

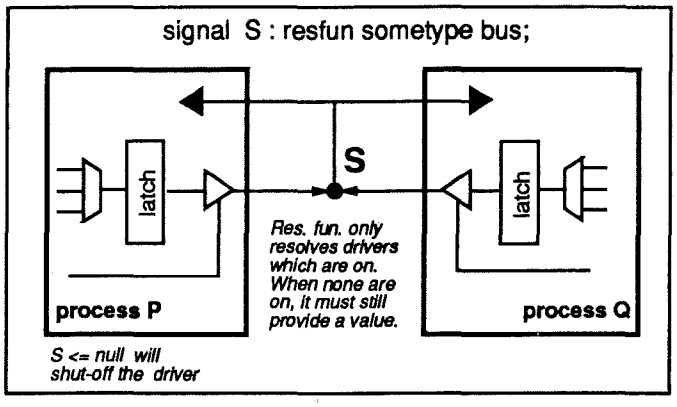
Writing a value to a signal in a process instantiates virtual storage in that process. The storage is of the type specified in the declaration. The storage output feeds the virtual wire of the signal. This storage is refered to as the process *driver* of the signal.

To see why virtual storage is needed within each process that writes to the signal in our conceptual hardware model, consider the following VHDL code segment:

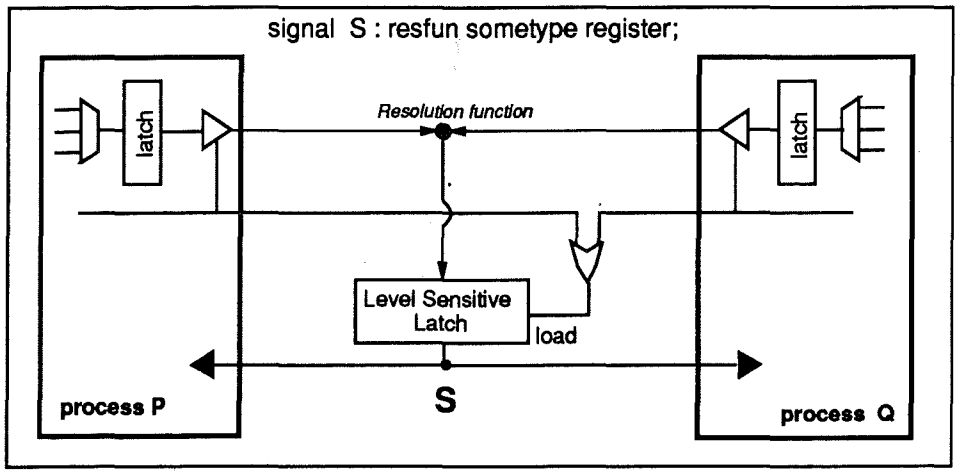
```
signal S : integer;
...
P: process
```



(a) Simple signal (no kind)



(b) Bus-kind signal



(c) Register-kind signal

Figure 3: Conceptual hardware for the three signal kinds

```
    variable v : integer;
begin
    v := 1;
    S <= v;
    wait for 50 ns;
    v := 2;
end process;
```

According to VHDL signal semantics, process P should continue to drive S with the value 1 even after 50 ns, at which time v is set to 2. Since the source of the value written to S may change, virtual storage of the value is implied. Situations in which this virtual storage is not required are discussed later.

3.1.2 Bus-Kind Signals

We now consider **bus** signals, the syntax of which is:

signal S : [resolution-function-name] type **bus** [:= expression];

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.* We indicate this in our conceptual hardware model by the addition of a virtual buffer, as in Figure 3(b).

A process shuts-off its driver for signal A by assigning a *null* value to it as in "A <= *null*;". In the case of a signal of kind bus, a resolution function is required not only for resolving multiple values into one, but also for indicating a value when when all drivers are shut-off.

One should not confuse the conceptual hardware's buffer with a tristate buffer. The former either contributes a value to the virtual wire or it does not and is independent of the signal type. A tristate buffer, on the other hand, outputs either a '1', '0', or high-impedance value 'Z'. Tristate logic is represented in VHDL as a type with the appropriate resolution function. Our conceptual hardware is independent of the signal type.

3.1.3 Register-Kind Signals

We now consider **register** signals, the syntax of which is:

signal S : [resolution-function-name] type **register** [:= expression];

The characteristics of the register-kind signal is given in Figure 4. 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.* The resolution function is *not* called in this case as it was with bus signals. This implies storage in our conceptual hardware, as shown in Figure 3(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.

	Signal Kind		
	No Kind	**Bus**	**Register**
Number of Drivers Allowed	Multiple	Multiple	Multiple
Resolution Function Needed ?	Only when Multiple drivers exist	Always	Always
Drivers can be turned off ?	No	Yes	Yes
Value when Multiple Drivers Active	Value determined by Resolution function	Value determined by Resolution function	Value determined by Resolution function
Value when Zero Drivers Active	Impossible	Value determined by Resolution function	Last resolved value of the signal

Figure 4: Characteristics of various signal kinds

3.1.4 Alternative resolution for composite signals

Until now, no mention was made of composite types, i.e. types composed of other types, such as arrays or records. The virtual latch in our conceptual hardware holds any type, such as a bit, an array of bits, or even an array of integers. In the latter case, the entire array of integers is sent over the virtual wire to be resolved with other values. However, VHDL offers an alternative resolution method for composite types, where each *element* is resolved individually. The declaration of such a signal and its conceptual hardware representation are shown in Figure 5.

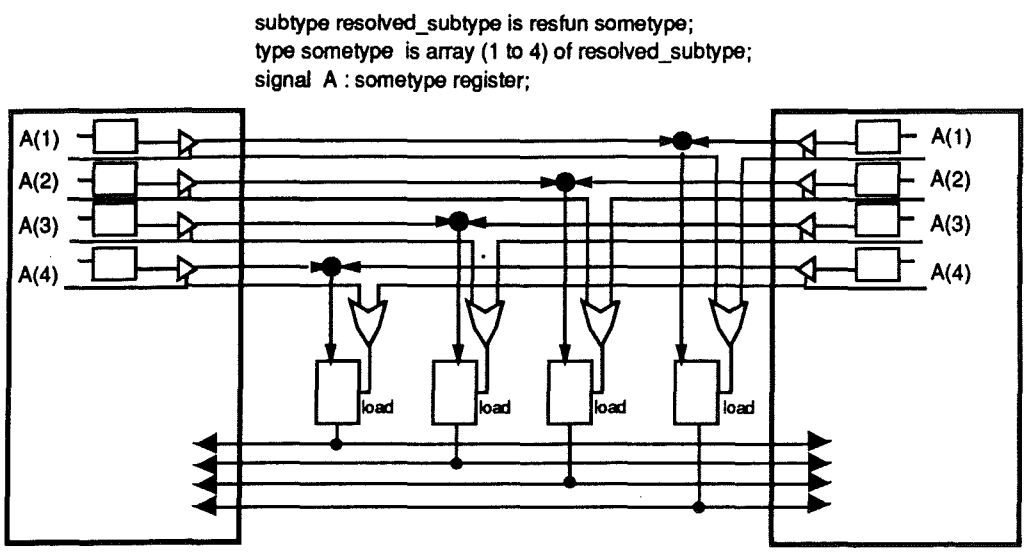


Figure 5: Resolving a composite's subelements

3.2 Synthesis

3.2.1 Synthesizing Hardware for Signals

In the previous section, we introduced the various signals kinds and explained their semantics by introducing each signal kind through the use of *conceptual hardware*, rather than through english. We now consider the issue of implementing signals using *real* hardware. Figure 6 summarizes the hardware implementations of the three signal kinds. However, synthesis of signals may still be difficult or unfeasible in other cases which are discussed below.

One problem is the use of virtual wires which 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 bits or sets of bits. Scalar signal-types such as booleans, integers, characters, strings, and enumerations are encoded into a one or more bits. For example, a signal of type "integer range 0 to 20" is encoded into five bits. A composite signal-type such as an array must have its elements encoded. For example, an array of integers becomes an array of bit-vectors. A composite signal-type such as a record can have each of its elements treated as 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.

Once all signals have been bit-encoded, we determine if the number of wires and the sizes of the latches in the conceptual hardware are feasible for implementation. For example, an array of 1000 16-bit bit-vectors would require a latch of 16000 bits within each process (or 1000 16-bit latches

	Signal Kind		
	No Kind	**Bus**	**Register**
Implementation	Wire	Wire	Level Sensistive Latch + Wire
Local Latches in each Driver required ?	Yes	Yes	Yes
Tristate Buffers required in each Driver ?	No	Only if driver is ever turned off (eg. S <= null;)	Only if driver is ever turned off (eg. S <= null;)

Figure 6: Hardware implementation of various signal kinds

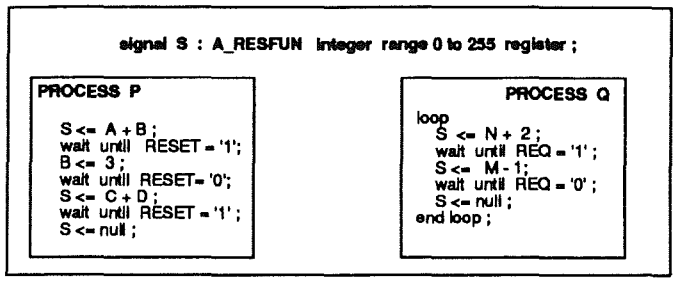
if resolution is that of Figure 5), which is clearly infeasible. Such a signal is better implemented using a memory, as will be discussed later. Certain heuristics are required to select signals which are implementable using the latch/wire conceptual hardware. Such heuristics may range from simply selecting all scalars (i.e. non arrays) to selecting signals such that area and routing constraints of the entire entity are satisfied. We shall refer to signals which are selected to be implemented as latches by the term *latchable signals.*

Latchable signals are implemented with hardware that is similar to that Figure 3. 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, such as choosing between level-triggered or edge-triggered latches, loading the appropriate value into the latch at the correct times, and setting the value of the buffer's tri-stating input. 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:

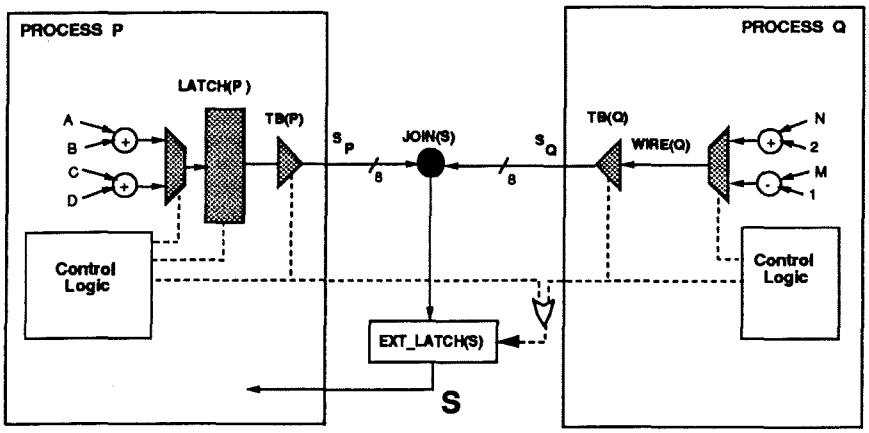
```
P: process(B,C)
begin
   A <= B + C;
end process;
```

```
Q: block
   begin
      A <= B + C;  -- concurrent signal assignment
   end block;
```

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. Likewise, if a process assigns only constants to a signal, then only a mux having the appropriate constants as inputs is needed. A latch is needed in a process that drives a signal if and only if a current source of the signal S (i.e. variable or another signal which occurs on the right-hand side of the previous



(a) Two VHDL processes writing to a register signal, S



(b) Hardware implementation for S

Figure 7: Synthesizing an integer signal of kind register.

assignment to S) is updated with a new value in a clock cycle, but the signal itself is not.

Figure 7 shows how we can synthesize an integer signal of kind register. In Figure 7(a), signal S has two drivers – written by two processes P and Q. In P, B is a source for signal S in the first assignment statement. Since S is not updated when B is assigned a new value (second assignments statement), we will require a latch for S in process P to store its previously assigned value. No such latch is needed in process Q. Since both processes have a null assignment to S, both have a tristate buffer to turn off the drivers for the signal in both processes. Both drivers for S (i.e., P, Q) may possibly be turned off simultaneously. Thus, we need an external latch $EXT\_LATCH(S)$ to store the most recent resolved value, as desired by VHDL signal semantics for register kind signals. The output of this latch is the value of S which can be used by process P when it referenced in the right-hand side of an assignment statement.

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

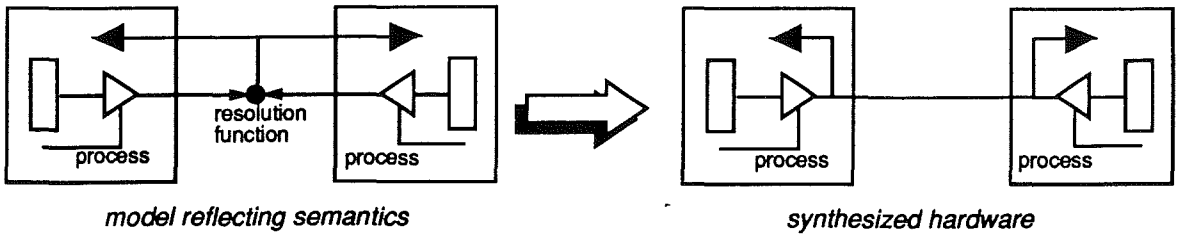


Figure 8: Simplifications made possible by restricted resolution function

and does not posses other functionality. Figure 8 illustrates this concept.

Figure 9 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.

In addition to signal assignments in processes, VHDL also has concurrent signal assignments. As stated in [1], concurrent signal assignment statements represent an equivalent process statement. Thus, concurrent signal assignments are handled in a manner similar to that of signal assignments in processes.

3.2.2 Memory Signals and Arbitration

When it is not feasible to implement a signal as a latch within each process or to have one wire for each bit that comprises the signal, the signal is implemented as a memory. We shall refer to these as *memory signals.* We shall assume that all memory signals are 2-dimensional (arrays of bit-vectors). Large 1-dimensional signals (bit-vectors) are easily converted to 2-dimensions. Signals with 3-dimensions or more must be converted to 2-dimensions or must be considered unimplementable.

The memory resides outside the processes, regardless of whether or not the signal was declared a register kind. Each access to the memory by a process is replaced by a channel consisting of address, data, and control. A process may have more than one such channel to allow for concurrent accesses, as described below. Latches may be needed within the process to hold the address and data values. Their necessity is based on the same criteria as was that for latchable signals discussed above.

During conversion to a control/dataflow graph (CDFG), a memory access is modeled as an operation with an address input and a data input or output for writes and reads respectively, as demonstrated in Figure 10. If scheduling of the CDFG is performed before allocation, then an N-port memory is used, where N is the maximum number of accesses of the memory during any single control-step. If

```
procedure GENERATE_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 whenever a current source of S is assigned a new value
          then
            Implement S as a latch within P_i, LATCH(P_i)
            LATCH(P_i) is enabled whenever S is written to in 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|register) and (S has a null signal assignment in P_i)
          then
            Add a tristate-buffer, TB(P_i), to WIRE(P_i) or the output of 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)
        endif

    endfor

    Connect all the outputs S_i from each process P_i which write to S
    Let this connection be called JOIN(S)

    if (S.kind = register) and (S has a null signal assignment in each process P_i which writes to it )
      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 if (S.kind = no_kind|bus)
        The value at JOIN(S) represents the signal S
    endif

end
```

Figure 9: Synthesizing hardware for latchable signals.

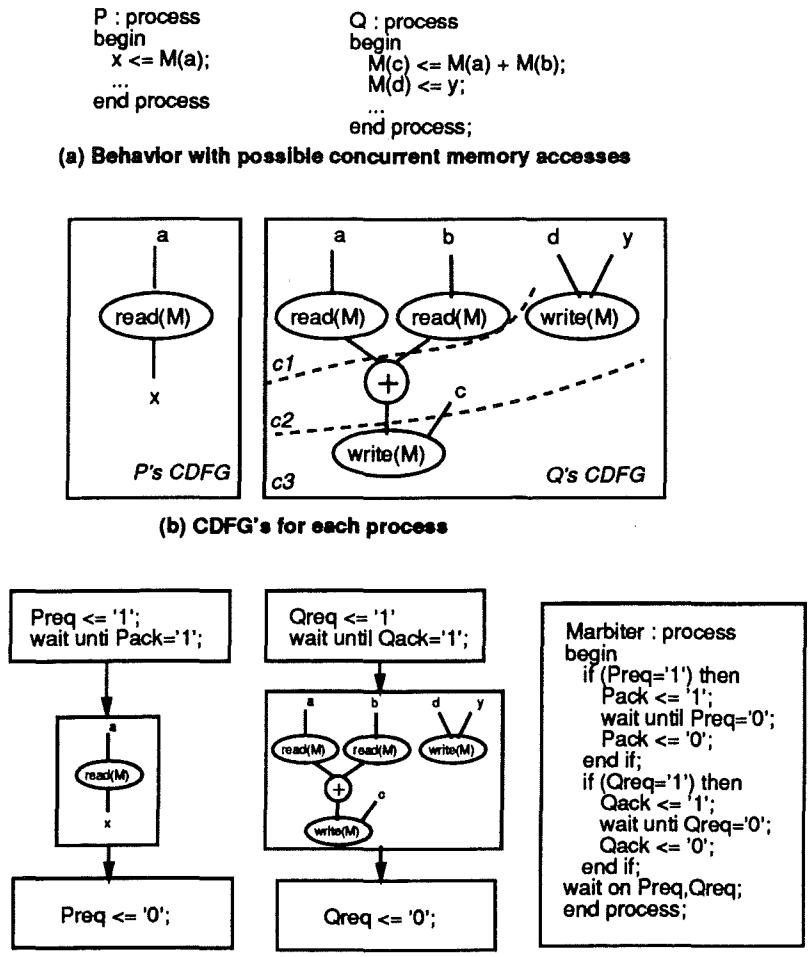


Figure 10: Memory accesses in a CDFG

allocation is performed first and an N-port memory is selected, then this constrains the schedule to no more than N accesses of the memory in any single control-step.

Within a single process, determining the number of accesses to a memory during any one clock cycle is trivial. However, since signals can be accessed by multiple processes, this number must be determined over all processes. In general, this is extremely complex since it is likely that the processes do not operate in sync, but instead can have varying cycle times. For example, in Figure 10(b) Q's CDFG has been scheduled into three control steps. Suppose processes P and Q are activated on events $e1$ and $e2$, respectively. If $e1, e2$ are external events, it cannot be determined if P's read will occur during $c1, c2$ or $c3$. Even if the events are internally generated, such determination is likely to still be impossible or very difficult. *The only way to ensure that memory accesses in two processes do not occur at the same time is to explicitly arbitrate between them.*

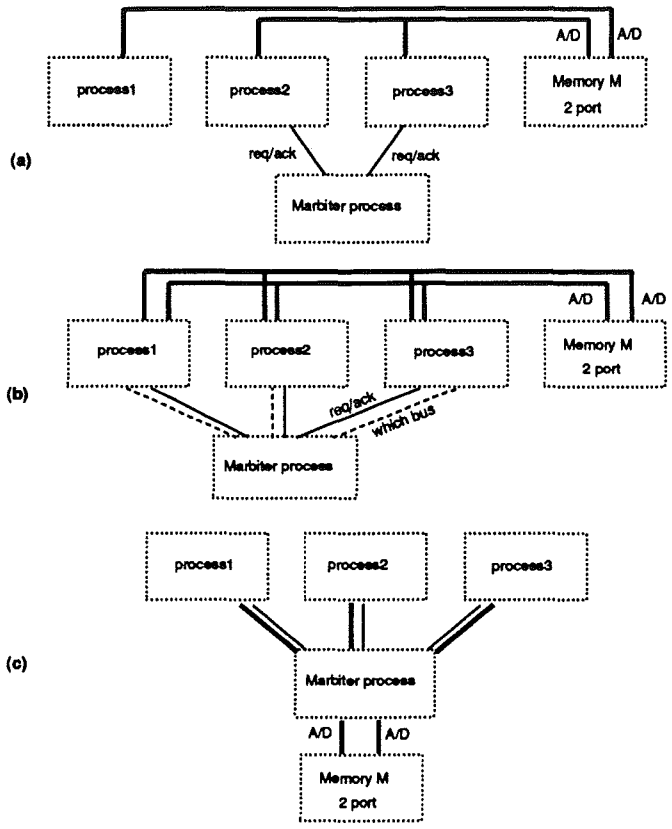


Figure 11: Various models of arbitration

Arbitration is accomplished by adding functionality to existing processes' behaviors. Each process must request permission to access the memory. After receiving permission and then performing the access, the process must relinquish its access rights. This handshake is implemented with two signals (request, acknowledge). A new process is created which monitors the request lines of each process and grants permissions based on some priority (e.g. fixed or rotating). Figure 10(c) shows an example.

Note that arbitration is never needed within a single process, since the number of concurrent accesses to the memory can always be controlled by scheduling.

Arbitration limits access to a given resource. Limited 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 or decongest routing. Depending on the arbitration model used, the resource may be the memory itself or a particular port of the memory. Several alternative arbitration models are shown in Figure 11. In Figure 11(a), each access is statically assigned to a specific port. Hence each port is a resource and the concurrent accesses over a port are arbitrated. In Figure 11(b) and (c), accesses are assigned to ports dynamically. Hence the two ports comprise a single resource. The arbiter must limit accesses to two processes at any time. In Figure 11(b), each process can access the ports directly,

```
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

                                        /* Introduce Req/Ack signals in the processes */
    for each process P_i which accesses MEM do
        Precede all accesses of MEM in P_i by the following
            MEM_req_i <= '1' ;
            wait until (MEM_ack_i = '1') ;
        Append the following after all accesses of MEM in P_i
            MEM_req_i <= '0'
            wait until (MEM_ack_i = '0') ;
    endfor

                                        /* Generate Arbiter Process */
    P_k = head(PLIST)
    PLIST = tail(PLIST)
    Add the following statements at the head of the arbiter process
        wait until MEM_req_1 or MEM_req_2 or .. or MEM_req_N ;
        if ( MEM_req_k = '1' ) then
            MEM_ack_k <= '1' ;
            wait until ( MEM_req_k = '0' ) ;
            MEM_ack_k <= '0' ;

    while PLIST ≠ φ do
        P_k = head(PLIST)
        PLIST = tail(PLIST)
        Append the following statements to the arbiter process
            elsif ( MEM_req_k = '1' ) then
                MEM_ack_k <= '1' ;
                wait until ( MEM_req_k = '0' ) ;
                MEM_ack_k <= '0' ;
    endfor

    Append the following statements to the arbiter process
        end if ;
end
```

Figure 12: Generating VHDL description for a Fixed Priority Memory Arbiter

so the arbiter merely informs a process which port it has permission to use. In Figure 11(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 has the simplest arbiter, but may result in poor performance since port assignment decisions are made statically. Hence a process may spend much time waiting for a port to become available even though a another port is unused. The second model alleviates this static decision problem, but requires a more complex arbiter as each process is connected to multiple ports, which may result in routing problems. The model also requires each process to be slightly more complex in order 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.

Figure 12 shows how a memory arbiter which implements a fixed priority scheme is generated for a memory signal, MEM. The arbiter process will grant access rights to MEM based on a priority ordering of the processes specified by the user. An alternative way of computing priorities is by calculating the average number of accesses to MEM made by each process. The process with the highest number of accesses will be assigned the highest priority. The arbiter generation algorithm requires that the priorities of processes be specified as a list, $PLIST$. Each access to MEM in the processes is modified to incorporate a handshake communication mechanism between the process and the arbiter. At any given time, all processes which need to access MEM will request the arbiter for permission. The arbiter process generated by the algorithm in Figure 12 simply scans access requests from the processes in order of their priority, and grants access to the highest priority request.

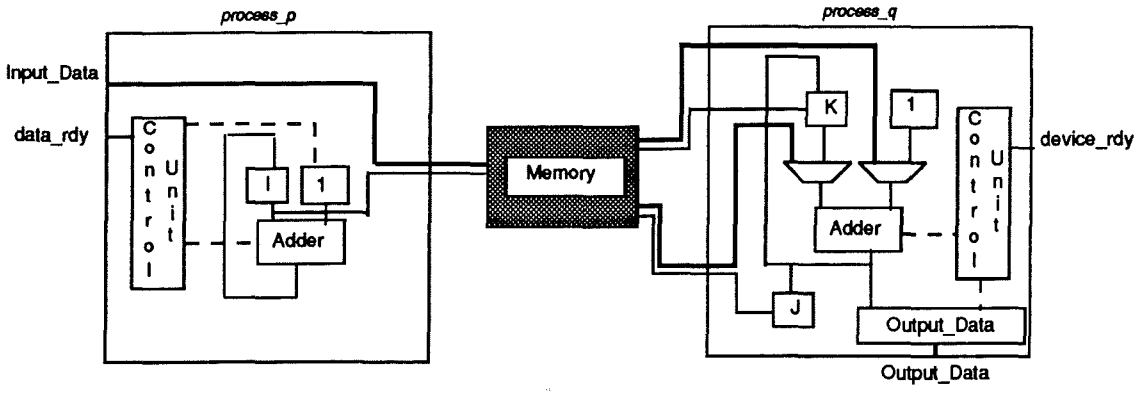
Figure 13 shows the results of synthesis on a multi-process VHDL description with multiple accesses to a global memory. As shown in Figure 13(a), three simultaneous accesses are possible on the global memory. The results of synthesis using a 3-port memory is shown in Figure 13(b). In this case, data access could be done independently on a separate port. However if a single-port memory were to be used, then an arbitration module is required to control the accesses to the memory. The results of such a synthesis is shown in Figure 13(c). In Figure 13(c) we do not arbitrate between the two memory accesses $M(K)$ and $M(J)$ because scheduling process Q' will ensure that they are scheduled in different control steps.

3.2.3 Variables

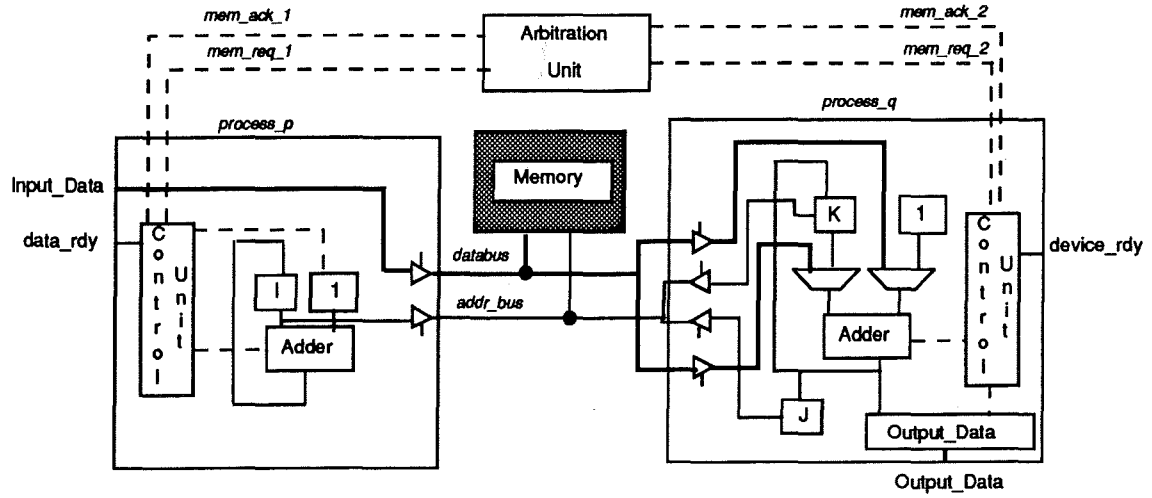
We have so far focussed our discussion on signals, which are external to VHDL processes. We now briefly discuss the synthesis of variables, which are local to a process. Variables are used as temporary value holders that can be used to store intermediate results of a computation. By removing the concept

```
port( data_rdy: In Bit;   devic_rdy : In Bit;   Input_data : in Integer;  Output_data : out Integer) ;
        type Memory is array (0 to 256) of Integer;
        signal M : Memory;

Process P (data_rdy)
variable I : Bit;
begin
    .....
    M(I) <= Input_data;
    I := I + 1;
end;

Process Q  (device_rdy)
variable K,J : Bit;
begin
    .....
    J := I;
    K := K + 1;
   Output_data <= M(K) + M(J);
end;
```

(a) Multi-process VHDL description with array accesses



(b) Synthesized Design with a 3-port Memory



(c) Synthesized Design with Arbitration for 1-port Memory

Figure 13: An example of synthesis with arbitration

of 'time', variables ensure sequentiality of variable assignment statements and lend themselves to simple descriptions.

To illustrate the use of variables, let us use the following descriptions which contains two processes P and Q. Both these processes are identical in functionality. However they differ in the use of a temporary variable *temp*. P uses a temporary variable, *temp* while computing A, but process Q does not.

```
P: process(B,C)
   variable  temp : integer ;
   variable A : integer ;
begin
   temp := B + C ;
   A    := temp + D ;
end process ;
```

```
Q: process(B, C)
   variable A : integer ;
begin
   A := B + C + D ;
end process ;
```

It is very difficult to predict how the variables would be synthesised without taking into account the available resources to implement the functionality. For example in the above descriptions if the user had given process P and a single adder was allocated, then the synthesis program would store the value for *temp* in a register. Hence *temp* would get synthesised into a register. However if two adders were available then both the additions could be performed in the same state. Thus *temp* does not require a register.

Similar arguments can be given for the description Q. In general, the variables may turn into a wire or a register depending on the available resources.

4 Ports

4.1 Semantics

An entity may have ports for interfacing with an external environment. In VHDL, a port is a signal declared in the interface list of an entity or component declaration. The semantics of ports are identical to that of signals except that in addition, ports have an associated mode which constrains the direction of the data flow allowed through the port. Let us consider the modes *in*, *out*, and *inout*.

In Figure 14(a) and Figure 14(b) we show straightforward conceptual hardware representations of

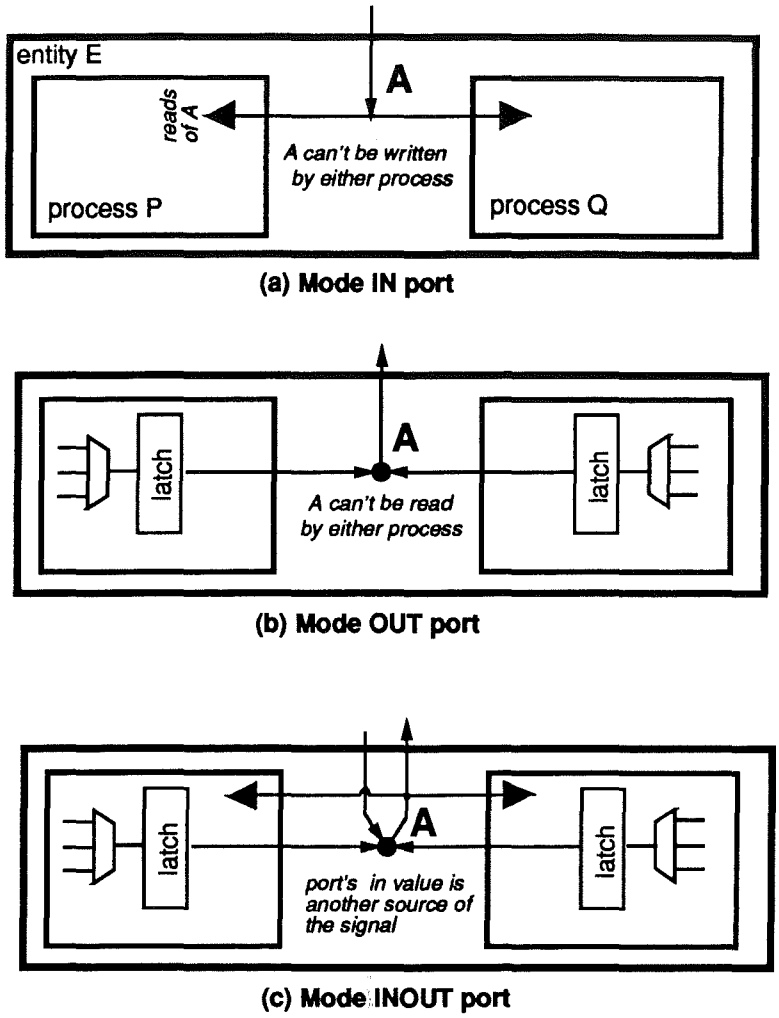


Figure 14: Conceptual hardware for three port modes

in and *out* ports. An *in* port cannot be written by any of the processes in the VHDL description. However, the value can be simultaneously read by all the processes. On the other hand an *out* port can only be written to by the processes. The value of the port is the resolved value from the resolution function. None of the processes can read a value from an out port.

In Figure 14(c), we show the conceptual hardware for an *inout* port. As shown, all process can read and write a value onto the port, thereby requiring the use of a resolution function. In addition, the external environment also contributes another value to the resolution function for the signal. This resolved value is what is read by any processes and is also the output value of the port. Note that 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.

VHDL allows an *out* or *inout* port to be declared as bus-kind. The semantics discussed for bus-kind signals is also applicable to ports. Hence, the extension to the conceptual hardware for bus kind ports is identical to that shown for signals in Figure 3(b). 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 14 would be the actual hardware. However simplifications are possible in the case of inout port. In Figure 15 the *in* and *out* values for an *inout* port are implemented as a single bus although they were conceptually shown as separate buses. Again, this simplification is possible only when the resolution function reflects the ports physical characteristics and does not include any other functionality.

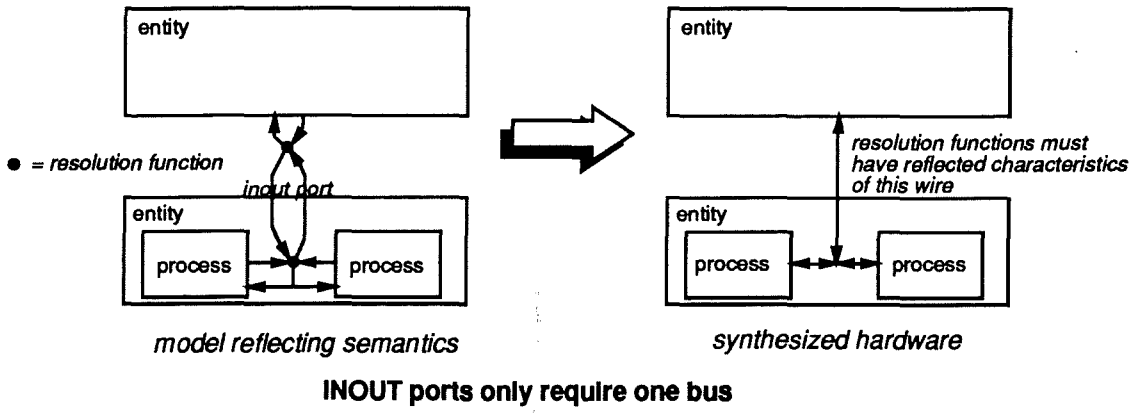


Figure 15: Synthesis Optimization for Inout Ports

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 in the context of 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 shown how hardware can be synthesized for such constructs. While almost all 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. The synthesis guidelines presented in this paper are being incorporated into the VHDL synthesis environment which we are currently developing.

6 Acknowledgements

This work was supported by the Semiconductor Research Corporation (grant #91-DJ-146). We are grateful for their support.

7 References

[1] *IEEE Standard VHDL Language Reference Manual*, 1988.

[2] L. Saunders, "The IBM VHDL Design System," in *Proceedings of the 24th Design Automation Conference*, 1987.

[3] R. Camposano, L.F. Saunders, and R. M. Tabet, "VHDL as Input for High Level Synthesis," *IEEE Design and Test of Computers*, March 1991.

[4] J. Lis and D. Gajski, "Synthesis from VHDL," in *Proc. of the ICCD*, 1988.

[5] J. Roy and R. Vemuri, "DSS: A Distributed Synthesis System for VHDL Specifications," in *VHDL Users' Group Spring 1991 Conference*, 1991.

[6] A. Salem and D. Borrione, "Formal semantics of VHDL timing constructs," in *Proceedings of Euro-VHDL*, 1991.

[7] R. Lipsett, C.F. Schaefer, and C. Ussery, *VHDL : Hardware Description and Design.* Kluwer Academic Publishers, 1989.