There is considerable overlap between high-level synthesis and register-transfer level synthesis. This is due to the fact that many systems do not provide all the synthesis steps, but start at the register-transfer level.

2.3.4. Logic-Level Synthesis

As a result of the register-transfer level synthesis, the system to be designed is broken down into blocks of combinational logic and storage elements. The behavior of combinational logic is described by Boolean functions. Their optimization and mapping to a gate level hardware structure is the task of *logic-level synthesis*.

The translation from behavioral representation to structural representation is straightforward at the logic level. Thus, the main problem in logic synthesis is optimization, specifically *logic* minimization in order to obtain a minimal area.

2.3.5. Technology Mapping

The result of logic synthesis is a network of abstract gates. To implement the abstract network in hardware, groups of abstract gates are mapped to physical library cells of a given target technology. This step is called *library mapping* or *technology mapping*.

Technology mapping is not restricted to the logic level. Some systems perform technology mapping after register-transfer level optimizations, mapping functional units to macro-blocks which are described both in the structural and the physical domain.

Further steps of the synthesis process deal with the transition from the structural to the physical domain at lower levels of abstraction, i.e., layout generation, which is not presented here.

2.4. VHDL Hardware Description Language

2.4.1. Hardware Description Languages

Traditional design methodologies for hardware systems employ design specification at the conceptual level in a natural language, design description using schematics, and then simulation of the description. The initial specification in a natural language is extended with block diagrams, state diagrams and timing diagrams. In this process, the initial specification is refined by the designer, which adds new information until a complete design at the register transfer level is obtained.

As today's systems become more complex, a new approach is required at the conceptual level. Designers need to describe the system's specifications in an *executable specification* language or hardware description language (HDL). Such an approach has several advantages.

- First, simulating an executable specification allows the designer to verify the correctness of the system's intended functionality. In the traditional approach, which starts with a natural-language specification, such verification would not be possible until a simulatable system description had been obtained (usually, gate-level schematics).
- The second advantage is that the specification can serve as an input to synthesis tools, which, in turn, can be used to obtain an implementation of the system, reducing the design time by a significant amount.
- Third, such a specification can serve as comprehensive documentation, providing an unambiguous description of the system's intended functionality.
- Finally, as the designs become more complex, description languages allow to increase the level of abstraction needed in order to deal with this complexity.

Many HDLs exist, for example, AHPL, CDL, CONLAN, DDL, ISP, PMS, HardwareC, Verilog, VHDL. As most of them as derived from general programming languages, e.g., PL/1, Pascal, ALGOL, C, ADA, a close relationship exists between the use of HDLs in hardware design and of general programming languages in software engineering.

2.4.2. Introduction to VHDL

VHDL (VHSIC *Hardware Description Language*) has its origin in the programming language ADA. The development of the language started in 1980 and was initiated by the US *Department of Defense* (DoD), with the intention to create a standard HDL for its VHSIC (*Very High Speed Integrated Circuit*) project. The aim of this project was to develop the next generation of integrated circuits. In the process of developing complex integrated circuits, the designers found that the available design tools were not appropriate, because they were based on the gate level design. The designers needed a new description method, and proposed a new description language, VHDL.

The original version of the language was released by the DoD in 1985. The language was improved for several years, and became an IEEE standard in 1987 (1076-1987). Later, the DoD required all military design submissions to be in VHDL. Shortly thereafter, VHDL became an industry standard. A new version of the language was standardized in 1993 (IEEE standard 1076-1993).

The language concepts of ADA, which was also initiated by the DoD, were extended by the following features to handle hardware design:

- Description of hardware types (e.g., signals, bit strings);
- Description of timing behavior;
- Data driven control structures, in addition to algorithmic control structures.

VHDL provides a formal notation for communication between vendors and users of hardware devices, between designers, and between design tools. It is a power-

ful language which supports descriptions of hardware at many design levels, and is used to specify full multiprocessor systems from the system level to the actual implementation in hardware. VHDL also supports verification, synthesis and testing of the design.

The main characteristics of the VHDL language are the following:

- It supports various styles of description, such as behavioral, data flow, and structural. *Behavioral* description represents an algorithmic description of the function of the hardware system, without any structural information, enhanced optionally with some timing information. *Data flow* description represents the flow of data within a hardware system; it represents behavior while implying structure. A typical example is a register-transfer description, where data is exchanged between registers and similar complex objects. *Structural* description represents components (blocks) and their interconnections.
- It supports multiple design methodologies, such as top-down (the divide-and-conquer approach), bottom-up, and mixed designs. In top-down design methodology, a system is hierarchically divided into a set of subsystems that is easier to design. In bottom-up design methodology, the designer starts with basic components (or subsystems) and proceeds to design a complete system.
- It supports various digital modeling techniques. These include algorithmic descriptions, Boolean equations, and finite-state machines.
- It supports various approaches to timing. For example, it supports a synchronous approach in which signal operations take an integer number of clock cycles. It also supports an asynchronous approach that allows cycles of various lengths.
- It supports various hardware technologies, including new primitive logic types, new primitive components, and technology-dependent attributes.
- It supports concurrent statements. VHDL is a concurrent language that can model simultaneous events that occur in hardware by executing several statements at the same time.
- It allows the user to define new data types, which allows for a high level of abstraction when describing and simulating new design techniques.

Despite all these capabilities, some designers do not think VHDL works well as a design tool. They emphasize that VHDL is too verbose. The amount of VHDL code needed to describe a design is usually twice that needed by other HDLs. In addition, the inherent parallelism of the language make the debugging process a difficult task. Finally, VHDL does not support visual representation (however, some CAD systems allow visual representation of designs). Some designers believe that diagrams provide better readability than languages.

2.4.3. VHDL Styles of Description

The main abstraction of VHDL is the design *entity*, which identifies and represents a single part of a design, executes a specific function, and has well-defined inputs and outputs. That is, an entity represents a black box; one or more input and output lines enter and leave the black box, and a delay is identified with each line. A VHDL model consists of at least one design entity.

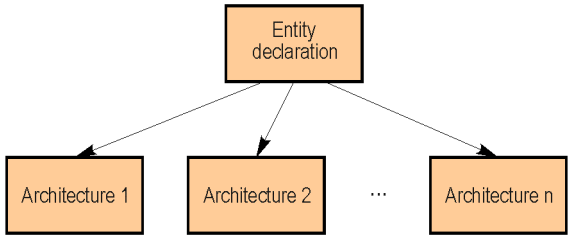

Figure 2.3. Basic structure of a VHDL entity.

There are two main elements that describe an entity: the *entity declaration* and the *architecture body*. The entity declaration contains the *interface* of the model to the environment, *attribute* descriptions, specifications and further descriptions (i.e., *assertions*) common to all architectures. An architecture represents one possible implementation of the model. Different implementations can result from different design variants or from different levels of abstraction in the design process. Several architectures can be associated with the same entity declaration (Figure 2.3).

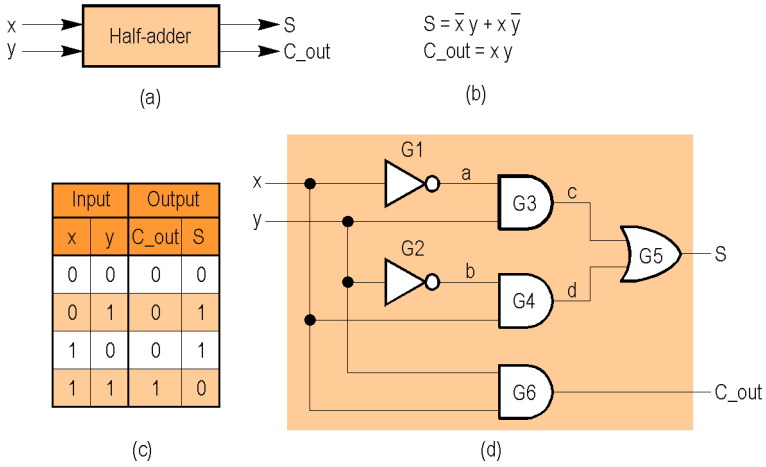

Figure 2.4. Different representations of a half adder: (*a*) block symbol; (*b*) Boolean equations; (*c*) truth table; (*d*) logic diagram.

VHDL offers three styles of description or views for a given entity: the *behavioral* style, the *data flow* style, and the *structural* style. A combination of these styles is possible within the same model. We show the differences between behavioral, data flow, and structural methodologies by describing the design of a half adder in each style. A half adder adds two 1-bit inputs *x* and *y* to produce a 2-bit result consisting of a sum bit *S* and a carry bit *C_out*.

Figure 2.4 shows different representations of a half adder in digital logic, using a block diagram, Boolean equations, truth table, and logic diagram.

The *behavioral* style presents an algorithmic description similar to most high-level programming languages. It uses control structures such as `if` statements and loops, as well as Boolean and arithmetic expressions. Figure 2.5 presents a behavioral architecture based on the truth table of a half adder.

```
entity half_adder is
   port (x, y: in bit;           -- input ports
         S, C_out: out bit);     -- output ports
end half_adder;

architecture behaviorial of half_adder is
begin
   process -- a process contains several statements
           -- executed in a sequential manner
      variable i: integer; -- index variable
      constant C_vector: bit_vector(0 to 2) := "001";
      constant S_vector: bit_vector(0 to 2) := "010";
      -- C_vector and S_vector are derived from
      -- the truth table of the half adder
   begin
      i := 0;
      if x = '1' then i := i+1; end if;
      if y = '1' then i := i+1; end if;
      C_out <= C_vector(i) after 20 ns;
      S <= S_vector(i) after 20 ns;
      wait on x, y;
      -- process is sensitive to signals x, y
   end process;
end behavioral;
```

Figure 2.5. Behavioral VHDL description of a half adder from truth table.

The keyword `entity` starts the description of the interface between the half adder and its environment, followed by the name of the entity. The optional `port` declaration defines the input signals *x* and *y* of type `bit`, and output signals *S* (sum) and *C_out* (carry) of the same type. `Bit` is a predefined enumeration type ('0', '1'). The keyword `end` followed by the entity name indicates the end of the entity declaration. Each statement in VHDL is concluded by a semicolon. Comments start with "`--`" and extend to the end of a line.

The keyword `architecture` begins the description of the architecture body. The name `behavioral` is a name given to this architecture. In general, the

architecture definition consists of a declarative part and an instruction part. In this case, the declarative part is empty. The instruction part contains a `process` statement, which is the language construct provided for the description of the behavior of a hardware component or even a hardware system. The `process` statement contains a declarative part and a statement part. The variable `i` stands for the number and position of the 1's in the inputs. The constants `C_vector` and `S_vector` are initialized to "001" and "010", respectively. The outputs *C_out* and *S* are set to the value of the i^{th} position in the array `C_vector` and `S_vector`, respectively. For example, the truth table in Figure 2.4 shows that if one of the inputs is 1 and the other is 0, the carry becomes 0 and the sum 1. This fact is represented by the bit in position 1 of vectors `C_vector` and `S_vector`, which are referred to by variable `i`. Because one of the inputs is 1 and the other is 0, `i` becomes 1. Thus,

```
C_out = C_vector(1) = 0
S = S_vector(1) = 1
```

The symbol `<=` is called *signal assignment*, as opposed to the symbol `:=`, which indicates a variable assignment. A variable assignment is performed as soon as the statement is executed and the expression on the right of the symbol `<=` is evaluated. A signal assignment may contain a specified delay. For instance, to indicate that it takes some time for the carry signal to change in response to a change of the input signals *x* and *y*, the `after` keyword is used.

Two types of delays are possible in a signal assignment: *inertial delay* or *transport delay*. In an inertial delay, exemplified in the preceding VHDL description, the output signal changes if and only if the input signal stays at a certain level at least for the specified delay time in the statement. In other words, the output signal does not change for an input transient of shorter duration than that of the delay time. In a transport delay, indicated by the `transport` keyword, a change of the input signal is always propagated to the output. If a statement does not contain a delay time of greater than zero, an elementary delay, called *delta delay*, will be assumed.

The `wait` statement within a `process` statement causes the process to be suspended until the value of the input signals *x* or *y* changes. Once a change appears in any of the inputs, the process starts all over again and changes the output values as necessary.

The *data fl*ow style describes a network of signals in which the flow of signal values is supervised by a set of control statements. For example, at the register transfer level of description, the data flow style presents the description of a circuit in terms of a set of concurrent register and signal assignments. The transfer between the registers are monitored by a set of control elements.

The data flow description does not have to define the components it uses. It often uses the Boolean operators and arithmetic expressions for describing a circuit. Another feature of this description is that it can show the amount of time elapsed in each element of the circuit. Figure 2.6 shows a data flow architecture based on the Boolean equations of the half adder.

The architecture body contains a *block* statement, delimited by the `begin` and `end` keywords. All statements within a block statement are initiated in parallel. The block consists of two signal assignment statements. A signal assignment is executed in two steps. First, the expression on the right hand side of the equation sign is evaluated after each change of any variable term involved. Then the resulting value is assigned to the signal after the specified delay. Any changes of the expression value in this time interval have (in general) no effect on a previous assignment. This is an example of data-driven control structure, when the execution of certain statements in a VHDL model depends on a specific change of variable or signal values.

```
entity half_adder is
   port (x, y: in bit;            -- input ports
         S, C_out: out bit);      -- output ports
end half_adder;

architecture data_flow of half_adder is
begin
   S <= (not x and y) or (x and not y) after 30 ns;
   C_out <= x and y after 10 ns;
end data_flow;
```

Figure 2.6. Data flow VHDL description of a half adder from Boolean equations.

The first signal assignment specifies that the signal *S* will get the exclusive-or value of the signals *x* and *y* after 30 ns. The 30 ns can be interpreted as the delay of an XOR gate. The second signal assignment specifies that the signal *C_out* will get the AND value of the input signals after 10 ns.

The *structural* style corresponds closely to the hardware structure of the circuit. A structural architecture can be described by declaring a set of components and connecting them with a set of signals. If a structural architecture is compared to a C program, the component declarations are similar to function declarations, and the input and output ports or signals are similar to the parameters for the functions. Figure 2.7 represents a structural architecture based on the logic diagram of the half adder.

The architecture body is divided into two parts: the declaration part, which appears before the keyword `begin`, and the interconnection part, which appears after `begin`. The declaration part consists of three `component` statements and a `signal` statement. The first `component` statement defines a NOT gate. The signal *i* is the input, and the signal *o* is the output. The second and third `component` statements define an AND gate and an OR gate, respectively. The `signal` statement defines a series of signals that are used for interconnecting the components. For example, the signal *a* is used to connect the *G*1 NOT gate to the *G*3 AND gate.

The design part includes a set of *component instantiation* statements. Each statement creates an instance (or copy) of a component. A statement starts with a label followed by the component name and a `port map` clause, which specifies the connections between the component instances in the form of a network specification or *netlist*. Each entry of the `port map` clause refers to one of the component's ports or a

locally declared signal. A port of a component is connected to a port of another component if they have a common element in the `port map` list. For example, at label *G*1, signal *x* is fed into a NOT gate, with signal *a* as the output. At label *G*3, signals *a* and *y* are fed into an AND gate, with signal *c* as the output.

```
entity half_adder is
   port (x, y: in bit;            -- input ports
         S, C_out: out bit);      -- output ports
end half_adder;

architecture structural of half_adder is
component not_gate
   port (i: in bit; o: out bit);
end component;
component and_gate
   port (i1, i2: in bit; o: out bit);
end component;
component or_gate
   port (i1, i2: in bit; o: out bit);
end component;
signal a, b, c, d: bit;
begin
   G1: not_gate port map (x, a);
   G2: not_gate port map (y, b);
   G3: and_gate port map (a, y, c);
   G4: and_gate port map (b, x, d);
   G5: or_gate  port map (c, d, S);
   G6: and_gate port map (x, y, C_out);
end structural;
```

Figure 2.7. Structural VHDL description of a half adder from logic diagram.

2.4.4. The Time Model in VHDL

The three dimensions of time in VHDL are illustrated in Figure 2.8.

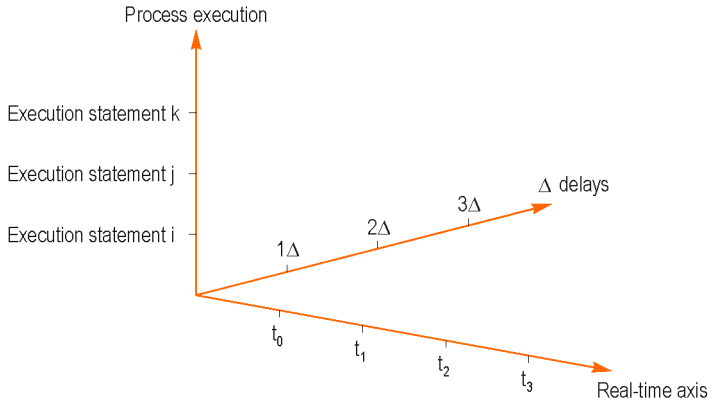

Figure 2.8. Three dimensions of time in VHDL.

The real-time axis reflects the advance of time in terms of discrete events. Besides the real-time axis, a delta-time delay allows the deterministic handling of concurrency and zero-delay in case of signal assignment. Any signal assignment is split into an *initiation activity*, the first step of a signal assignment execution, and an *execution activity*, the second step. The execution takes place at least one delta-cycle after the initiation. Uncertainties caused by multiple assignments to a signal have to be solved by the definition of a *resolution function*, provided by the user.

```
begin       -- block statement; all statements inside
            -- the block are initiated concurrently
   a <= b; -- signal assignment with zero delay
   b <= a; -- signal assignment with zero delay
end;
```

Figure 2.9. Concurrent signal assignments.

Figure 2.9 is an example of block statement with concurrent signal assignments. If the block statement is initiated at time t_0, both signal assignments are executed. Both expressions at the right hand side are evaluated, and the assignments of the values will take place within simulation cycle $t_0 + \Delta$. This means that signals *a* and *b* will change their values. This mechanism makes sure that the execution of the VHDL description is independent of a particular implementation by a simulator.

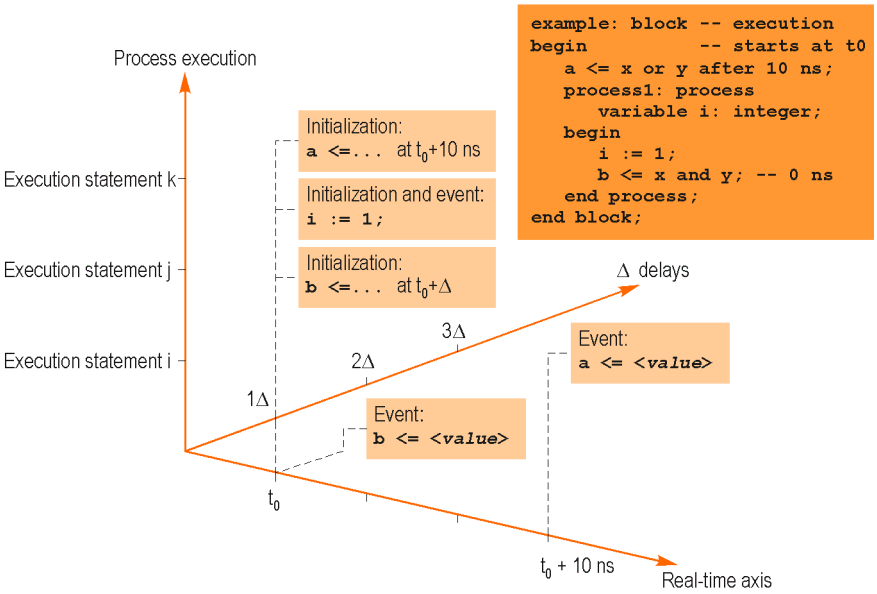

Figure 2.10. VHDL time model.

Additionally, VHDL distinguishes between an *inertial* and a *transport* delay. The inertial delay takes into account the inertia of the circuits, and is the default delay. A change of an output demands a minimum hold time of the corresponding input signals, otherwise there will be no effect. The transport delay is used to describe changes without considering inertia.

The third axis of the time model (Figure 2.8 and Figure 2.10) reflects the execution of statements within processes. For example, a process describes an algorithm without any real delay in terms of time. Variable assignments within a process, for example, consume no time. Therefore, no global variables exist in a VHDL model. Otherwise, what happens in case of multiple assignments to a variable at the same time has to be defined to avoid a simulator-dependent execution of a VHDL model.

2.4.5. Simulation of a Model

At the beginning of every simulation process, storage is allocated for all data objects declared in a VHDL design (such as variables, constants and signals). Also, initial values are assigned to these objects, and simulation time is reset to 0 ns. Simulation, then, starts adding time to proceed to the next event. Signals that need values at this time are assigned. All design units (processes) whose input signals changed are then executed. This sequence of adding time, signal change, and process execution continues until simulation suspends. Simulation stops when a time limit is reached that the user has specified, or the maximum time allowed by the simulator is reached.

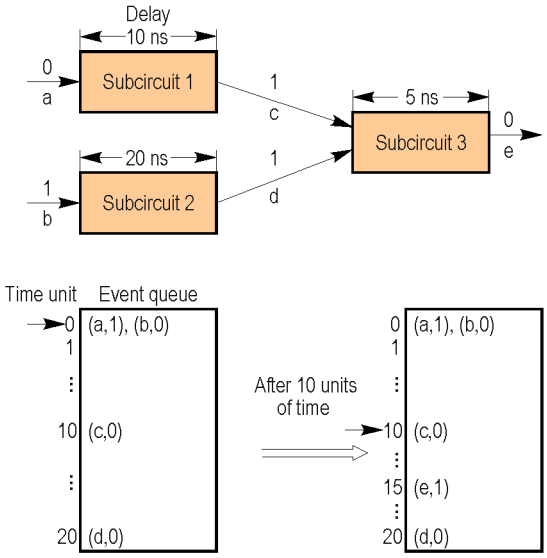

Figure 2.11. Simulation process.

As an example, consider the block diagram of the circuit in Figure 2.11. Assume that the signals *a*, *b*, *c*, *d*, and *e* are initialized to 0, 1, 1, 1, and 0, respectively. Also assume that a buffer, called the *event queue*, is allocated to schedule the changes to these signals. The events at time 0 in the event queue specify that *a* should change to 1 at time 0 and also *b* should change to 0 at the same time. When these changes have been done, the subcircuits that are affected by these signals are determined (subcircuit 1 and subcircuit 2). The changes on the outputs of these subcircuits will be determined and added to the event queues. Assume that subcircuit 1 produces 0 for the input 1. Since subcircuit 1 has a propagation delay of 10 ns, the event (*c*, 0) is scheduled for time 10 and is added to the event queue. Also, assume that subcircuit 2 produces 0 for the input 0. Since subcircuit 2 has a propagation delay of 20 ns, the event (*d*, 0) is scheduled for time 20 and is added to the event queue.

After the events at time 0 are processed, the simulation time is incremented by 1 and the corresponding events are processed. This process continues until it reaches time 10. At that time, event (*c*, 0) will be processed, which causes event (*e*, 1) to be scheduled for time 15. The simulation process continues until the event queue becomes empty or reaches some simulation time limit.