VHDL Introduction

VHSIC **H**ardware **D**escription **L**anguage

A language for describing the structural, physical and behavioral characteristics of digital systems.

Execution of a VHDL program results in a simulation of the digital system.

VHDL supports both *structural* and *behavioral* descriptions of a system at multiple levels of abstraction.

Structure and behavior are complementary ways of describing systems.

- A description of the *behavior* of a system says nothing about the *structure* or the components that make up the system.
- There are many ways in which you can build a system to provide the same behavior.

Reference: "VHDL Starter's Guide", Sudhakar Yalamanchili, Prentice Hall

VHDL Introduction

VHDL programs are unlike programs written using *conventional* programming languages.

Digital systems are fundamentally about signals, specifically binary signals that may take values 0 or 1.

They are composed of components such as memories, registers, gates, caches, ALU's etc.

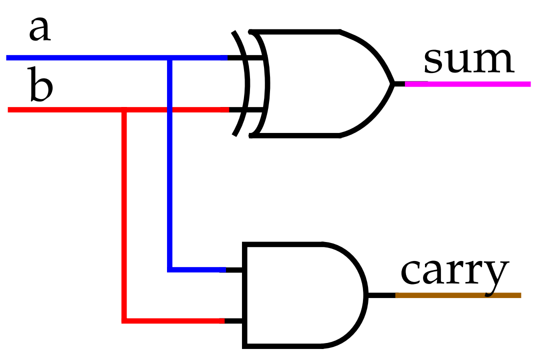
VHDL allows you to specify:

- The *components* of a circuit.
- Their interconnection.
- The behavior of the components in terms of their input and output signals.

For example :-

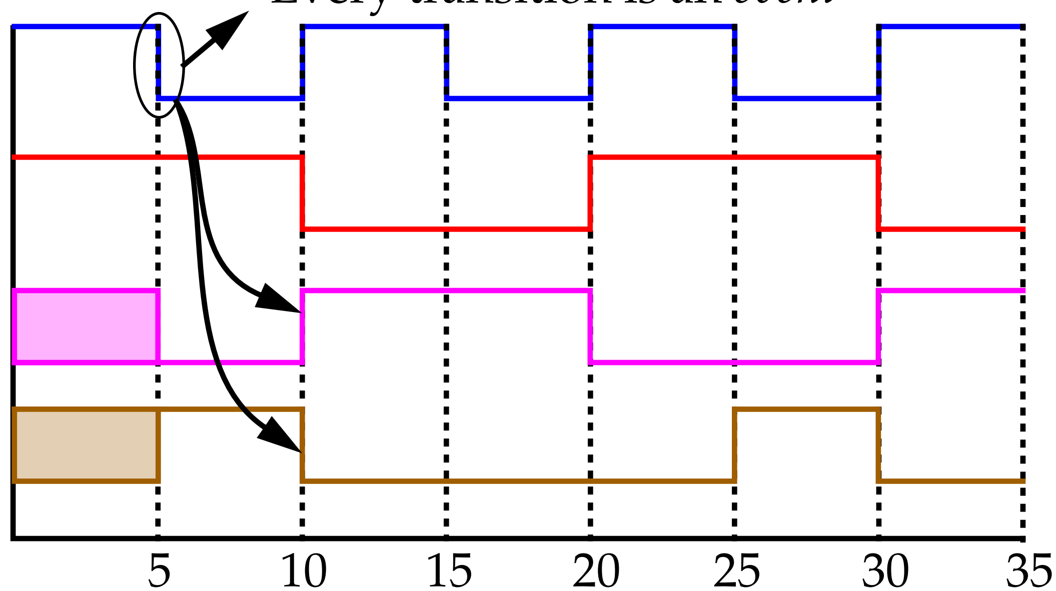
What are the behavioral properties of a half-adder circuit ?

Events, Propagation Delay and Concurrency



Half Adder

Every transition is an *event*



- *Event* on **a** from 1 to 0. (i.e. a time stamped transition)
- The output changes after a 5ns *propagation delay*.
  Both gates (and wires) have inertia or a natural resistance to change.
- A third property of this circuit is *concurrency*.

Both the xor and and gate compute new output values concurrently when an input changes state.

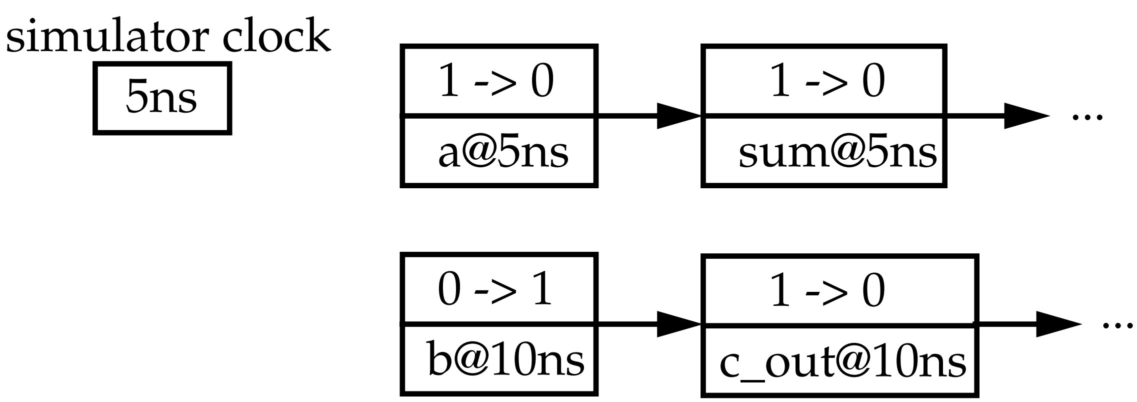
Data driven system: Events on signals lead to computations that may generate events on other signals.

Discrete Event Simulation

We can view VHDL as a programming language for describing the generation of events in digital systems supported by a *discrete event simulator*.

A *discrete event simulator* executes VHDL code, modeling the passage of time and the occurrence of events at various points in time.

It maintains an event list data structure to keep track of the order of all future events in the circuit.



An **event**:
A change in the value of a signal

Timestamp:
Time at which an event is to occur.

Advance simulation clock to time of next event, update signals receiving values.

Evaluate all components affected by signal updates and schedule new events.

Basic Language Concepts

Signals: Like variables in a programming language such as C, signals can be assigned values, e.g., 0, 1, Z, U.

However, signals also have an associated *time value*.

A signal receives a value at a specific point in time and retains that value until it receives a new value at a future point in time.

The sequence of values assigned to a signal over time is the *waveform* of the signal.

A variable always has one current value.

At any instant in time, a signal may be associated with several *time-value* pairs.

Just as variables can be of type integer, real etc. signals can be defined having specific types.

e.g. std_logic and std_logic_vector(X_n downto X_0)

Entity-Architecture

Design entity: A component of a system whose behavior is to be described and simulated.

Two components to the description:

The interface to the design: **entity** declaration.

The internal behavior of the design: **architecture** construct.

Entity example for half adder:

```
entity  half_adder  is port        -- Note: VHDL is CaSe insensitive.
(    a, b: in bit;
     sum, carry: out bit);
end half_adder;
```

half_adder is the name given to the design entity.

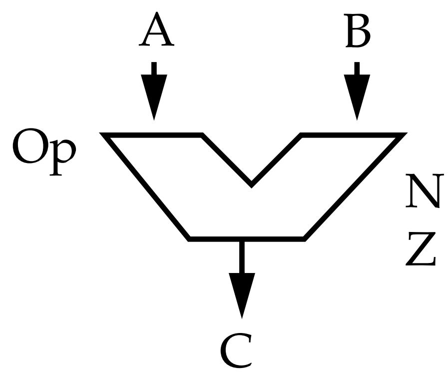
The input and outputs signals; **a**, **b**, **sum** and **carry**, are referred to as *ports*.

Entity-Architecture

Each port has a type, *bit* and *bit_vector* can assume values of 0 and 1.

Each port has a mode; *in*, *out* or *inout* (bidirectional signals).

Bit vectors are specified as:



entity ALU32 **is port**(
A, B: **in bit_vector**(31 **downto** 0);
C: **out bit_vector**(31 **downto** 0);
Op: **in bit_vector**(5 **downto** 0);
N, Z: **out** bit);
end half_adder;

A and **B** are 32 bits long with the most significant bit as 31.

A more general definition of *bit* and *bit_vector* are *std_logic* and *std_logic_vector*, which can assume more than just 0 and 1.

Entity-Architecture

Architecture construct:

```
architecture arch_name  of entity_name  is
    -- place declarations here
    begin
    --  place description of behavior here
    end half_adder_arch;
```

Concurrent statements:

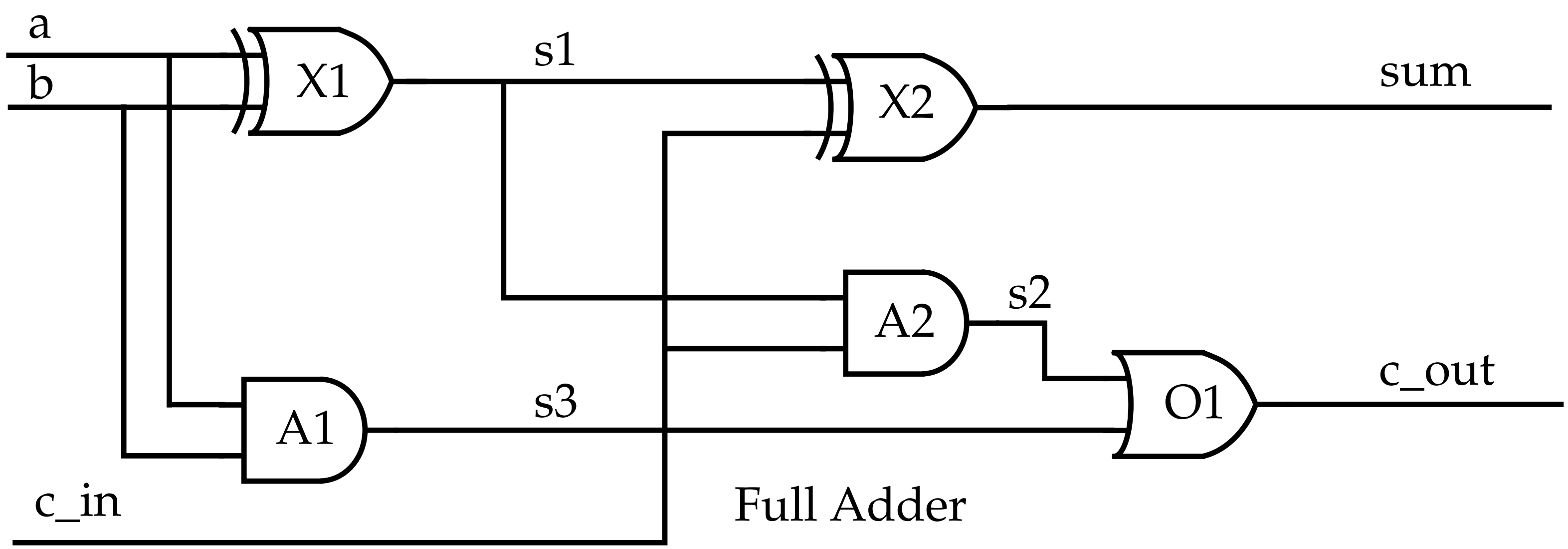
Signal assignment statements specify the new value and the time at which the signal is to acquire this value.

The textual order of the concurrent signal assignment statements (CSAs) do NOT effect the results.

```
architecture half_adder_arch of half_adder is
    begin
        sum <= (a xor b) after 5 ns;
        carry <= (a and b) after 5 ns;
    end half_adder_arch;
```

Entity-Architecture

We can also use (local) signals internal to the architecture, e.g., **s1**, **s2** and **s3** in the full adder circuit.

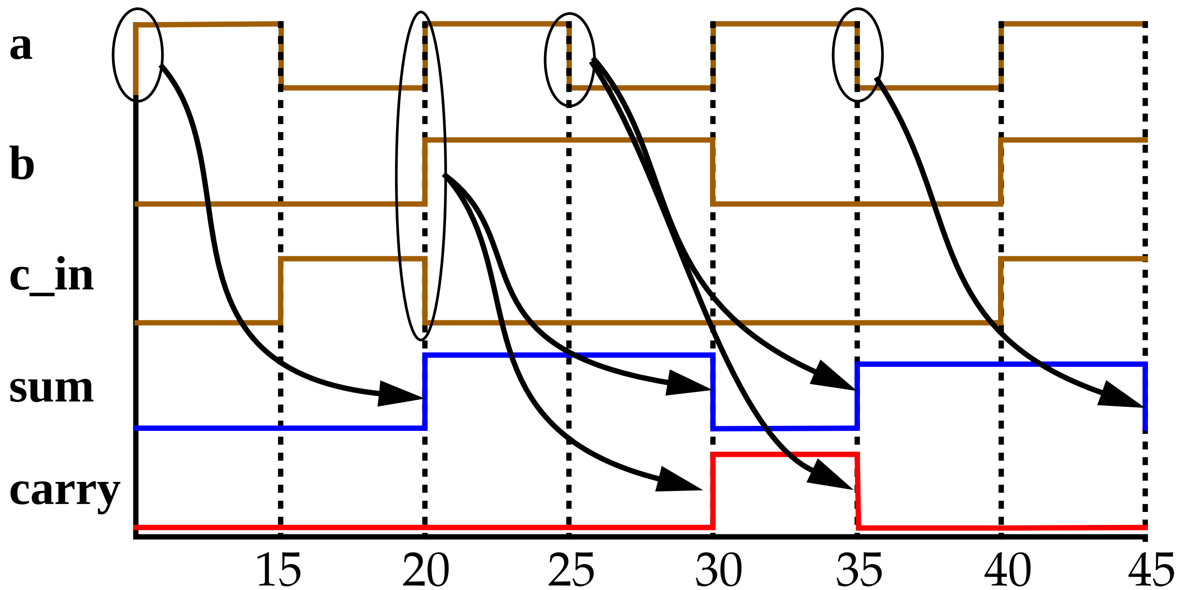


Full Adder

entity full_adder **is port**(
  a, b, c_in: **in bit**;
  sum, carry: **out bit**);
end half_adder;

Entity-Architecture

architecture full_adder_arch **of** full_adder **is**
signal s1, s2, s3: **bit**;
constant gate_delay: **Time**:= 5 **ns**;
begin
L1: s1 <= (a **xor** b) **after** gate_delay;
L2: s2 <= (c_in **and** s1) **after** gate_delay;
L3: s3 <= (a **and** b) **after** gate_delay;
L4: sum <= (s1 **xor** c_in) **after** gate_delay;
L5: carry <= (s2 **or** s3) **after** gate_delay;
end full_adder_arch;



Entity-Architecture

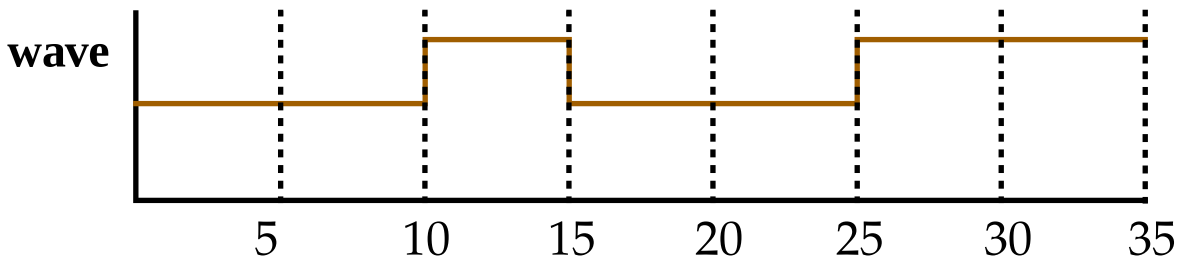
The following statements are also legal:

s1 <= (a **xor** b) **after** 5 **ns**, (a **or** b) **after** 10 **ns**, (**not** a) **after** 15 **ns**;

wave <= ’0’, ’1’ **after** 10 **ns**, ’0’ **after** 15 **ns**, ’1’ **after** 25 **ns**;

A *driver list* that specifies a waveform.

This statement generates a set of *transactions* (time-value pairs) to be carried out at distinct times in the future.



Other VHDL constructs

Conditional Signal Assignment Statement:

```
entity mux4 is port(
    in0, in1, in2, in3: in bit;
    S0, S1: in bit;
    Z: out bit_vector( 7 downto 0);
end mux4;

architecture behavioral of mux4 is
    begin
    Z <= in0 after 5 ns when S0 = '0' and S1 = '0' else
         in1 after 5 ns when S0 = '0' and S1 = '1' else
         in2 after 5 ns when S0 = '1' and S1 = '0' else
         in3 after 5 ns when S0 = '1' and S1 = '1' else
         "00000000" after 5 ns;
    end behavioral;
```

The first conditional found to be true determines the value transferred to the output.

The **Selected Signal Assignment Statement** behaves similarly.

```
with addr1 select
    reg_out <= reg0 after 5 ns when "000", ...
```

Modeling Behavior, Processes

Processes are used:

- For describing component behavior when they cannot be simply modeled as delay elements.
- To model systems at high levels of abstraction.

Process incorporate conventional programming language constructs.

A process is a **sequentially** executed block of code, which contains.

- arrays and queues.
- Variable assignments, e.g., **x** := **y**, which, unlike signals, take effect immediately.
- *if-then-else* and loop statements to control flow.
- Signal assignments to external signals.

Processes contain *sensitivity lists* in which signals are listed, which determine when the process executes.

In reality, CSAs are also processes without the *process*, *begin* and *end* keywords.

Modeling Behavior, Processes

```
entity half_adder is port(
    a, b: in bit;
    sum, carry: out bit;
end mux4;
architecture behavior of half_adder is
    begin
      sum_proc: process ( a, b )  begin
          if ( a = b)  then
              sum <= '0' after 5 ns;
          else
              sum <= ( a or b ) after 5 ns;
          end if;
          end process;
      carry_proc: process ( a, b )  begin
          case a is
              when '0' => carry <= a after 5 ns;
              when '1' => carry <= b after 5 ns;
              when others => carry <= 'X' after 5 ns;
          end case;
          end process;
    end behavior;
```

Modeling Behavior, Looping

Looping constructs include the *for* and *while* statements, e.g.,

```
for index in 1 to 32 loop
    if product_register(0) = '1' then
      product_register(63 downto32) := produce_register(63 downto 32) +
          multiplicand_register(31 downto 0);
     end if;
     product_register(63 downto 0) := '0' product_register(63 downto 1);
 end loop;
```

The loop *index* is implicitly declared, local to the loop and cannot be changed.

Alternatively;

```
while (j < 32) loop
     ...
     j := j + 1;
 end loop;
```

UMBC

Modeling Behavior, The Wait Statement

Processes are executed once upon initialization.

Thereafter, they are executed in a data-driven manner by either:

- an event on one or more signals in the *sensitivity list*.
- waiting for the occurrence of specific event using a *wait* statement.

The *wait* statement specifies the conditions under which a process may resume execution after being suspended, e.g.,

wait for *time expression*; -- wait for a specified time interval.
wait on *signal*; -- wait on a signal(s).
wait until *condition*; -- wait until *condition* becomes true;

The first and third form allow processes to model components that are not necessarily data driven.

wait statements also allow processes to suspend at multiple points, and not just at the beginning.

Modeling Behavior, The Wait Statement

For example, a positive edge-triggered flip-flop:

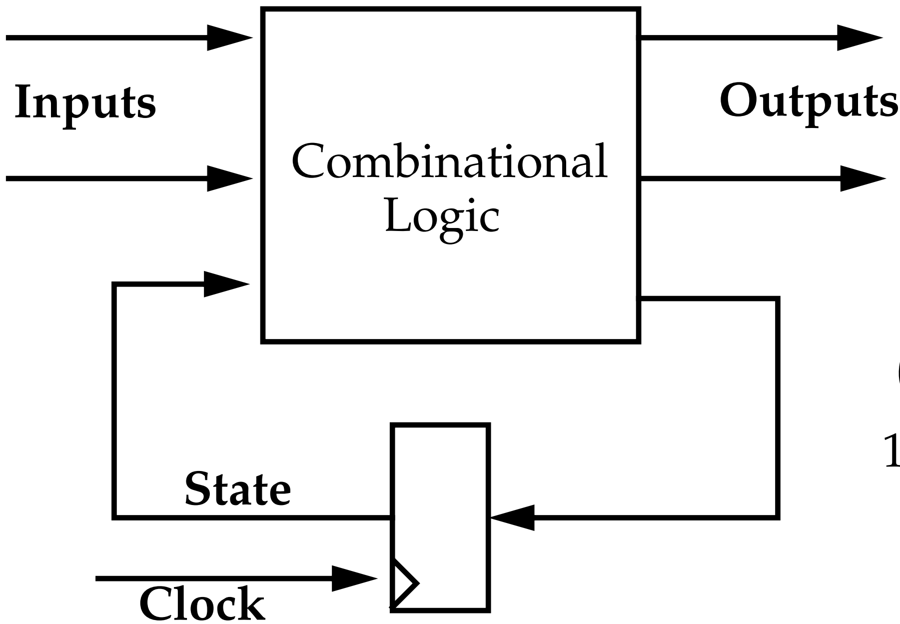
```
entity dff is port(
    D, Clk: in bit;
    Q, Qbar: out bit;
end dff;
architecture dff_arch of dff is
    begin
        output: process begin
            wait until ( Clk'event and Clk = '1' );
            Q <= D after 5 ns;
            Qbar <= not D after 5 ns;
        end process output;
    end dff_arch;
```

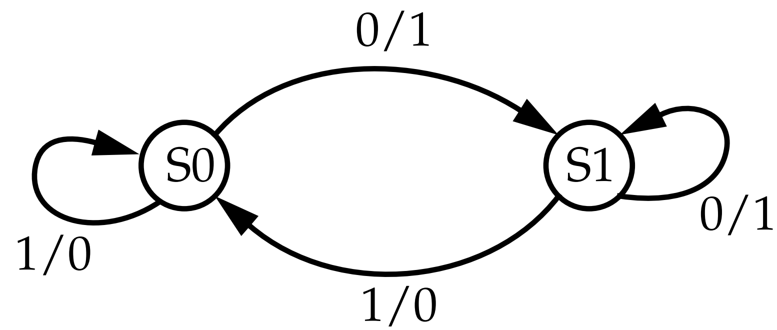
Note attribute *Clk*'**event** which is true when an event (rising or falling edge) occurs on signal *Clk*.

A D flip-flop with asynchronous reset (R) and set (S) inputs given in reference.

Modeling State Machines

A state machine (Mealy machine):





Combinational part implemented in one process, sensitive to events on the input signals or state variables.

Sequential part implemented in a second process, sensitive to the rising edge of the clock.

Modeling State Machines

```
entity state_machine is port(
    reset, clk, x: in bit;
    z: out bit;
end state_machine;

architecture behavioral of state_machine is
    type statetype is ( state0, state1);
    signal state, next_state: statetype := state0;
    begin
        comb_process: process (state, x) begin
            case state is
                when state0 => if x = ’0’  then next_state <= state1; z <= ’1’;
                                else next_state <= state0; z <= ’0’; end if;
                when state1 => if x = ’1’ then next_state <= state0; z<= ’0’;
                                else next_state <= state1; z <= ’1’; end if;
            end case; end process comb_process;
        clk_process: process begin
            wait until ( clk’event and clk = ’1’ )
            if reset = ’1’ then state <= statetype’left;
            else state <= next_state; end if;
            end process clk_process;
    end behavioral;
```

Modeling Structure

Structural model: A description of a system in terms of the interconnection of its components, rather than a description of what each component does.

A structural model does NOT describe how output events are computed in response to input events.

How do we simulate the circuit ?

Behavioral models of each component are assumed to be provided.

A VHDL structural description must possess:

- The ability to define the list of components.
- The definition of a set of signals to be used to interconnect them.
- The ability to uniquely label (distinguish between) multiple copies of the same component.

Modeling Structure

A structural description of a full adder:

```
entity full_adder is port(
    in1, in2, c_in: in bit;
    sum, c_out: out bit;
end full_adder;

architecture structural of full_adder is
    component half_adder
        port ( a, b: in bit; sum, carry: out bit );
    end component;
    component or_2
        port ( a, b: in bit; c: out bit );
    end component;

    signal s1, s2, s3: bit;

    begin
      H1: half_adder port map( a => in1, b => in2, sum =>s1, carry => s3 );
      H2: half_adder port map( a=> s1, b => c_in, sum => sum, carry => s2 );
      O1: or_2 port map( a => s2, b => s3, c => c_out );
    end structural;
```

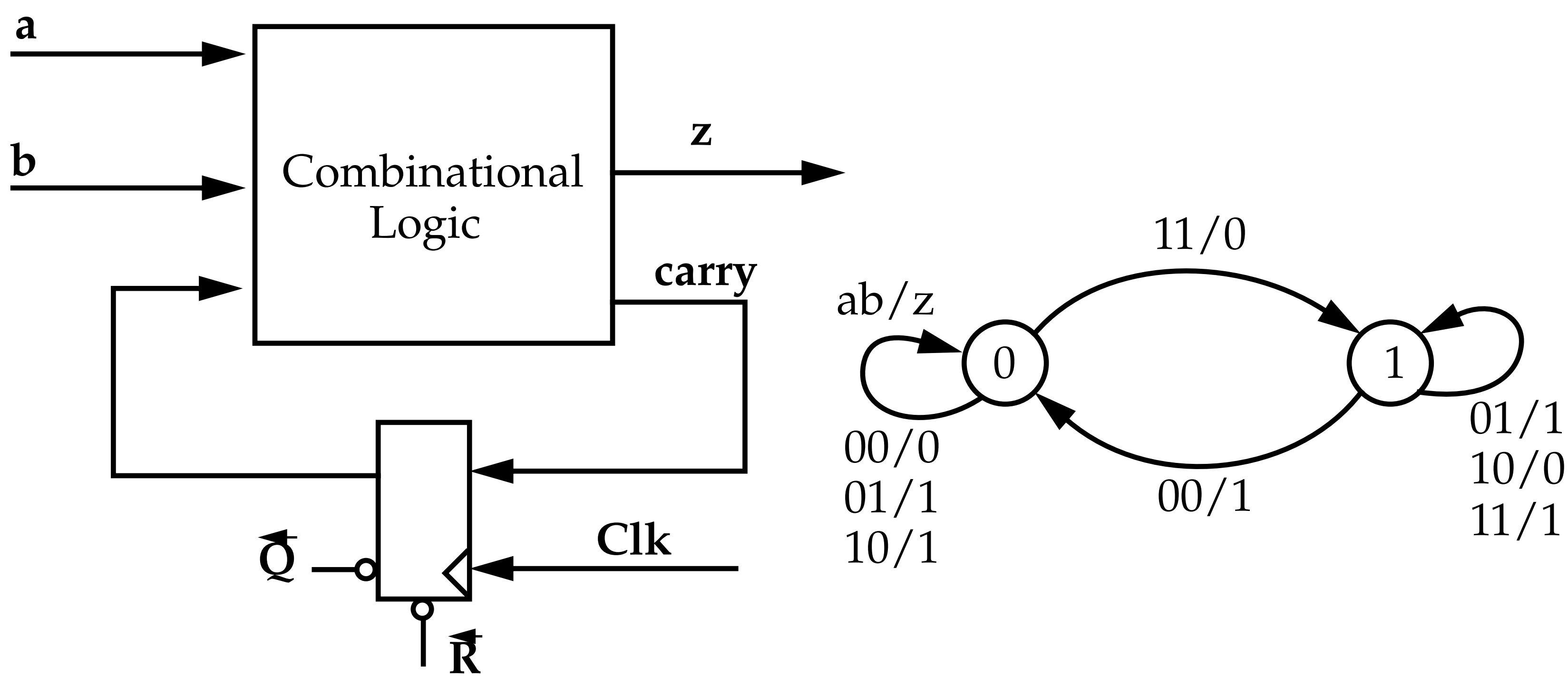
Component declarations

Signal declarations

Component Interconnection

Modeling Structure

A state machine of a bit-serial adder:



Modeling Structure

A structural description of a bit-serial adder:

```
entity serial_adder is port(
    a, b, clk, reset: in bit;
    z: out bit;
end serial_adder;

architecture structural of serial_adder is
    component comb
        port ( a, b, c_in: in bit; z, carry: out bit );
    end component;
    component dff
        port ( clk, reset, d: in bit; q, qbar: out bit );
    end component;

    signal s1, s2: bit;

    begin
      C1: comb port map( a => a, b => b, c_in => s1, z => z, carry => s2 );
      D1: dff port map( clk => clk, reset => reset, d => s2, q => s1,
        qbar => open );
    end structural;
```