Introduction

- A drawing of a circuit, or **schematic**, contains graphical information about a design
  - Inverter is above the OR gate, AND gate is to the right, etc.

- Such graphical information may not be useful for large designs

- Can use textual language instead
Textual Language – English

- Can describe circuit using English text rather than using a drawing
  - Of course, English isn't a good language for a computer to read
  - Need a more precise, computer-oriented language

(a) We'll now describe a circuit whose name is DoorOpener. The external inputs are c, h, and p, which are bits. The external output is f, which is a bit.

(b) We assume you know the behavior of these components:
    - An inverter, which has a bit input x, and bit output F.
    - A 2-input OR gate, which has inputs x and y, and bit output F.
    - A 2-input AND gate, which has bit inputs x and y, and bit output F.

The circuit has internal wires n1 and n2, both bits. The DoorOpener circuit internally consists of:

- An inverter named Inv_1, whose input x connects to external input c, and whose output connects to n1.
- A 2-input OR gate named OR2_1, whose inputs connect to external inputs h and p, and whose output connects to n2.
- A 2-input AND gate named AND2_1, whose inputs connect to n1 and n2, and whose output connects to external output f.

That's all.

Computer-Readable Textual Language for Describing Hardware Circuits: HDLs

- Hardware description language (HDL)
  - Intended to describe circuits textually, for a computer to read
  - Evolved starting in the 1970s and 1980s

- Popular languages today include:
  - VHDL – Defined in 1980s by U.S. military; Ada-like language
  - Verilog – Defined in 1980s by a company; C-like language
  - SystemC – Defined in 2000s by several companies; consists of libraries in C++
Combinational Logic Description using Hardware Description Languages

- **Structure**
  - Another word for "circuit"
  - An interconnection of components
  - Key use of HDLs is to describe structure

Note: The term "instantiate" will be used to indicate adding a new copy of a component to a circuit

The OR component

Three instances of the OR component

Describing Structure in VHDL

- **Entity** — Defines new item's name & ports (inputs/outputs)
  - std_logic means bit type, defined in ieee library
- **Architecture** — Describes internals, which we named "Circuit"
  - Declares 3 previously-defined components
  - Declares internal signals
    - Note "--" comment
  - Instantiates and connects those components

We'll now describe a circuit whose name is DoorOpener. The external inputs are c, h, and p, which are bits. The external output f, which is a bit.

We assume you know the behavior of these components:

- An inverter, which has a bit input x, and bit output F.
- A 2-input OR gate, which has inputs x and y, and bit output F.
- A 2-input AND gate, which has bit inputs x and y, and bit output F.

The circuit has internal wires n1 and n2, both bits.

The DoorOpener circuit internally consists of:

- An inverter named Inv_1, whose input c connects to external input c, and whose output connects to n1.
- A 2-input OR gate named OR2_1, whose inputs connect to external inputs h and p, and whose output connects to n2.
- A 2-input AND gate named AND2_1, whose inputs connect to n1 and n2, and whose output connects to external output f.

That's all.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity DoorOpener is
port ( c, h, p: in std_logic;
      f: out std_logic);
end DoorOpener;
architecture Circuit of DoorOpener is
component Inv
port (x: in std_logic;
     F: out std_logic);
end component;
component OR2
port (x,y: in std_logic;
     F: out std_logic);
end component;
component AND2
port (x,y: in std_logic;
     F: out std_logic);
end component;
signal n1,n2: std_logic; --internal wires
begin
Inv_1: Inv port map (x=>c, F=>n1);
OR2_1: OR2 port map (x=>h,y=>p,F=>n2);
AND2_1: AND2 port map (x=>n1,y=>n2,F=>f);
end Circuit;
```
Describing Structure in Verilog

- Modules defined for Inv, OR2, and AND2 (details omitted)
  - Note ‘//’ comment
- Module defined for DoorOpener
  - Lists inputs and outputs
  - Declares internal wires
  - Instantiates and connects three components

We'll now describe a circuit whose name is DoorOpener. The external inputs are c, h, and p, which are bits. The external output is f, which is a bit.

We assume you know the behavior of these components:
An inverter, which has a bit input x, and bit output F.
A 2-input OR gate, which has inputs x and y, and bit output F.
A 2-input AND gate, which has bit inputs x and y, and bit output F.

The circuit has internal wires n1 and n2, both bits.

The DoorOpener circuit internally consists of:
- An inverter named Inv_1, whose input connects to external input c, and whose output connects to n1.
- A 2-input OR gate named OR2_1, whose inputs connect to external inputs h and p, and whose output connects to n2.
- A 2-input AND gate named AND2_1, whose inputs connect to n1 and n2, and whose output connects to external output f.

That's all.

```
module Inv(x, F);
  input x;
  output F;
  // details not shown
endmodule
module OR2(x, y, F);
  input x, y;
  output F;
  // details not shown
endmodule
module AND2(x, y, F);
  input x, y;
  output F;
  // details not shown
endmodule
module DoorOpener(c, h, p, f);
  input c, h, p;
  output f;
  wire n1, n2;
  Inv Inv_1(c, n1);
  OR2 OR2_1(h, p, n2);
  AND2 AND2_1(n1, n2, f);
endmodule
```

Describing Structure in SystemC

- Module defined
  - Declares inputs and outputs
  - Declares internal wires
    - Note ‘/*’ comment
  - Declares three previously-defined components
  - Constructor function “CTOR”
    - Instantiates components
    - Connects components

We'll now describe a circuit whose name is DoorOpener. The external inputs are c, h, and p, which are bits. The external output is f, which is a bit.

We assume you know the behavior of these components:
An inverter, which has a bit input x, and bit output F.
A 2-input OR gate, which has inputs x and y, and bit output F.
A 2-input AND gate, which has bit inputs x and y, and bit output F.

The circuit has internal wires n1 and n2, both bits.

The DoorOpener circuit internally consists of:
- An inverter named Inv_1, whose input connects to external input c, and whose output connects to n1.
- A 2-input OR gate named OR2_1, whose inputs connect to external inputs h and p, and whose output connects to n2.
- A 2-input AND gate named AND2_1, whose inputs connect to n1 and n2, and whose output connects to external output f.

That's all.

```
#include "systemc.h"
#include "inv.h"
#include "or2.h"
#include "and2.h"

SC_MODULE (DoorOpener)
{
  sc_in<sc_logic> c, h, p;
  sc_out<sc_logic> f;
  // internal wires
  sc_signal<sc_logic> n1, n2;
  // component declarations
  Inv Inv1;
  OR2 OR2_1;
  AND AND2_1;
  // component instantiations
  SC_CTOR(DoorOpener):Inv_1("Inv_1"),
    OR2_1("OR2_1"),
    AND2_1("AND2_1")
  {
    Inv_1.x(c);
    Inv_1.F(n1);
    OR2_1.x(h);
    OR2_1.y(p);
    OR2_1.F(n2);
    AND2_1.x(n1);
    AND2_1.y(n2);
    AND2_1.F(f);
  }
};
```
Combinational Behavior

- **Combinational behavior**
  - Description of desired behavior of combinational circuit without creating circuit itself
  - e.g., \( F = c' \cdot (h + p) \) can be described as equation rather than circuit
  - HDLs support description of combinational behavior

Describing Combinational Behavior in VHDL

- Describing an OR gate's behavior
  - Entity defines input/output ports
  - Architecture
    - Process - Describes behavior
      - Process "sensitive" to \( x \) and \( y \)
        » Means behavior only executes when \( x \) changes or \( y \) changes
      - Behavior assigns a new value to output port \( F \), computed using built-in operator "or"

```
library ieee;
use ieee.std_logic_1164.all;

entity OR2 is
  port (x, y: in std_logic;
        F: out std_logic);
end OR2;

architecture behavior of OR2 is begin
process (x, y)
begin
  F <= x or y;
end process;
end behavior;
```
Describing Combinational Behavior in VHDL

• Describing a custom function's behavior
  – Desired function: f = c*(h+p)
  – Entity defines input/output ports (not shown)
  – Architecture
    • Process
      – Sensitive to c, h, and p
      – Assigns a new value to output port f, computed using built-in operators "not", "and", and "or"

```vhdl
architecture beh of DoorOpener is
begin
  process(c, h, p)
  begin
    f <= not(c) and (h or p);
  end process;
end beh;
```

Describing Combinational Behavior in Verilog

• Describing an OR gate's behavior
  – Module declares input/output ports
    • Also indicates that F is "reg"
    • Means F stores value
      – By default, ports are wires, having no storage
  – "always" procedure executes statement block when change occurs on x or on y
    • "Sensitive" to x and y
    • Assigns value to F, computed using built-in OR operator "|

```verilog
module OR2(x,y,F);
  input x, y;
  output F;
  reg F;
  always @(x or y)
  begin
    F <= x | y;
  end
endmodule
```
Describing Combinational Behavior in Verilog

- Describing a custom function's behavior
  - Desired function: \( f = c'(h+p) \)
  - Module defines input/output ports
    - Output \( f \) defined as "reg"
  - "always" procedure sensitive to inputs
    - Assigns value to \( f \), computed using built-in operators for NOT (~), AND (&), and OR (|)

```verilog
module DoorOpener(c, h, p, f);
    input c, h, p;
    output f;
    reg f;
    always @(c or h or p)
    begin
        f <= (~c) & (h | p);
    end
endmodule
```

Describing Combinational Behavior in SystemC

- Describing an OR gate's behavior
  - Module declares input/output ports
  - Constructor (CTOR)
    - Indicates module described by a method (procedure) "comblogic"
    - Sensitive to \( x \) and \( y \)
  - Method "comblogic" assigns \( F \) a new value using built-in OR operator "|
    - Reading input port done using .read() function defined for input port type; likewise, writing done using .write() function

```c
#include "systemc.h"

SC_MODULE(OR2)
{
    sc_in<sc_logic> x, y;
    sc_out<sc_logic> F;

    SC_CTOR(OR2)
    {
        SC_METHOD(comblogic);
        sensitive << x << y;
    }

    void blogic()
    {
        F.write(x.read() | y.read());
    }
};
```
Describing Combinational Behavior in SystemC

- Describing a custom function’s behavior
  - Desired function: \( f = c'*(h+p) \)
  - Module defines input/output ports
  - Constructor
    - Indicates module described by a method (procedure) "comblogic"
    - Sensitive to \( c \), \( h \), and \( p \)
  - "comblogic" method
    - Assigns value to \( f \), computed using built-in operators for NOT (~), AND (\&), and OR (|)

```cpp
#include "systemc.h"

SC_MODULE(DoorOpener)
{
    sc_in<sc_logic> c, h, p;
    sc_out<sc_logic> f;
    SC_CTOR(DoorOpener)
    {
        SC_METHOD(comblogic);
        sensitive << c << h << p;
    }

    void blogic()
    {
        f.write((~c.read()) & (h.read() | p.read()));
    }
};
```

Testbenches

- **Testbench**
  - Assigns values to a system's inputs, check that system outputs correct values
  - A key use of HDLs is to simulate system to ensure design is correct
Testbench in VHDL

- **Entity**
  - No inputs or outputs
- **Architecture**
  - Declares component to test, declares signals
  - Instantiates component, connects to signals
  - Process writes input signals, checks output signal
    - Waits a small amount of time after writing input signals
    - Checks for correct output value using "assert" statement

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity Testbench is
end Testbench;

architecture behavior of Testbench is
component DoorOpener
  port ( c, h, p: in std_logic;
         f: out std_logic);
end component;

signal c, h, p, f: std_logic;
begin
DoorOpener1: DoorOpener port map (c,h,p,f);

process begin
  -- case 0
  c <= '0'; h <= '0'; p <= '0';
  wait for 1 ns;
  assert (f='0') report "Case 0 failed";
  -- case 1
  c <= '0'; h <= '0'; p <= '1';
  wait for 1 ns;
  assert (f='1') report "Case 1 failed";
  -- (cases 2-6 omitted from figure)
  -- case 7
  c <= '1'; h <= '1'; p <= '1';
  wait for 1 ns;
  assert (f='0') report "Case 7 failed";
  wait; -- process does not wake up again
end process;
end behavior;
```

Testbench in Verilog

- **Module**
  - Three register signals for inputs (values must be written and stored)
  - One wire signal for output (value is only read, need not be stored)
  - Instantiates component, connects to signals
  - "initial" procedure executed only once at beginning
    - Sets input values
    - Displays output value

```verilog
module Testbench;
reg c, h, p;
wire f;

DoorOpener DoorOpener1(c, h, p, f);

initial begin
  // case 0
  c <= 0; h <= 0; p <= 0;
  #1 $display("f = %b", f);
  // case 1
  c <= 0; h <= 0; p <= 1;
  #1 $display("f = %b", f);
  // (cases 2-6 omitted from figure)
  // case 7
  c <= 1; h <= 1; p <= 1;
  #1 $display("f = %b", f);
end
```

```verilog
endmodule
```
Testbench in SystemC

- Module
  - Testbench is its own module
  - Three outputs, one for each input of system to test
  - One input, for the one output of the system to test
  - Constructor defined as THREAD
    - Like MODULE, but allows use of "wait" to control timing
  - testbench procedure writes outputs, waits for small amount of time, checks for correct output
    - assert function prints error if condition is not true

```
#include "systemc.h"

SC_MODULE(Testbench)
{
    sc_out<sc_logic> c_t, h_t, p_t;
    sc_in<sc_logic> f_t;

    SC_CTOR(Testbench)
    {
        SC_THREAD(testbench_proc);
    }

    void testbench_proc()
    {
        // case 1
        c_t.write(SC_LOGIC_0);
        h_t.write(SC_LOGIC_0);
        p_t.write(SC_LOGIC_0);
        wait(1, SC_NS);
        assert( f_t.read() == SC_LOGIC_0 );

        // case 1
        c_t.write(SC_LOGIC_0);
        h_t.write(SC_LOGIC_0);
        p_t.write(SC_LOGIC_1);
        wait(1, SC_NS);
        assert( f_t.read() == SC_LOGIC_1 );

        // (cases 2-6 omitted from figure)
        // case 7
        c_t.write(SC_LOGIC_1);
        h_t.write(SC_LOGIC_1);
        p_t.write(SC_LOGIC_1);
        wait(1, SC_NS);
        assert( f_t.read() == SC_LOGIC_0 );

        sc_stop();
    }

    return;
}
```

Sequential Logic Description using Hardware Description Languages

- Will consider description of three sequential components
  - Registers
  - Oscillators
  - Controllers
Describing a 4-bit Register in VHDL

- **Entity**
  - 4 data inputs, 4 data outputs, and a clock input
  - Use std_logic_vector for 4-bit data
    - \( I \): in std_logic_vector(3 downto 0)
    - \( I <= \text{"1000"} \) would assign \( I(3)=1, I(2)=0, I(1)=0, I(0)=0 \)

- **Architecture**
  - Process sensitive to clock input
    - First statement detects if change on clock was a rising edge
    - If clock change was rising edge, sets output \( Q \) to input \( I \)
    - Ports are signals, and signals store values – thus, output retains new value until set to another value

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity Reg4 is
  port ( I: in std_logic_vector(3 downto 0);
        Q: out std_logic_vector(3 downto 0);
        clk: in std_logic );
end Reg4;

architecture behavior of Reg4 is
begin
  process(clk)
  begin
    if (clk='1' and clk'event) then
      Q <= I;
    end if;
  end process;
end behavior;
```

Describing a 4-bit Register in Verilog

- **Module**
  - 4 data inputs, 4 data outputs, and a clock input
  - Define data inputs/outputs as vectors
    - input [3:0] \( I \)
    - \( I<=4'b1000 \) assigns \( I[3]=1, I[2]=0, I[1]=0, I[0]=0 \)
  - Output defined as register to store value
  - "always" procedure sensitive to positive (rising) edge of clock
    - Sets output \( Q \) to input \( I \)

```verilog
module Reg4(I, Q, clk);
  input [3:0] I;
  input clk;
  output [3:0] Q;
  reg [3:0] Q;
  always @posedge clk)
  begin
    Q <= I;
  end
endmodule
```
Describing a 4-bit Register in SystemC

- **Module**
  - 4 data inputs, 4 data outputs, and a clock input
  - Define data inputs/outputs as vectors
    - `sc_in<sc_lv<4> > I;`
  - Constructor calls seq_logic method, sensitive to positive (rising) edge of clock
    - seq_logic writes output $Q$ with input $I$
    - Output port is signal, and signal has storage, thus output retains value

```c
#include "systemc.h"
SC_MODULE(Reg4) {
  sc_in<sc_lv<4> > I;
  sc_out<sc_lv<4> > Q;
  sc_in<sc_logic> clk;
  SC_CTOR(Reg4) {
    SC_METHOD(seq_logic);
    sensitive += clk;
  }
  void seq_logic()
  {
    Q.write(I.read());
  }
};
```

Describing an Oscillator in VHDL

- **Entity**
  - Defines clock output
- **Architecture**
  - Process
    - Has no sensitivity list, so executes non-stop as infinite loop
    - Sets clock to 0, waits 10 ns, sets clock to 1, waits 10 ns, repeats

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity Osc is
  port ( clk: out std_logic );
end Osc;
architecture behavior of Osc is
begin
  process
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
  end process;
end behavior;
```
Describing an Oscillator in Verilog

- **Module**
  - Has one output, clk
  - Declare as "reg" to hold value
  - "always" procedure
    - Has no sensitivity list, so executes non-stop as infinite loop
    - Sets clock to 0, waits for 10 ns, sets clock to 1, waits for 10 ns, repeats

```verilog
module Osc(clk);
output clk;
reg clk;
always begin
  clk <= 0;
  #10;
  clk <= 1;
  #10;
end
endmodule
```

Describing an Oscillator in SystemC

- **Module**
  - Has one output, clk
  - Constructor creates single thread
    - Thread consists of infinite loop
      - while (true) {
    - Sets clock to 0, waits for 10 ns, sets clock to 1, waits for 10 ns, repeats

```systemc
#include "systemc.h"
SC_MODULE(Osc) {
  sc_out<sc_logic> clk;
  SC_CTOR(Osc) {
    SC_THREAD(seq_logic);
  }
  void seq_logic() {
    while (true) {
      clk.write(SC_LOGIC_0);
      wait(10, SC_NS);
      clk.write(SC_LOGIC_1);
      wait(10, SC_NS);
    }
  }
};
```
Describing a Controller in VHDL

• FSM behavior captured using architecture with 2 processes
  • First process models state register
    • Asynchronous reset sets state to "S_Off"
    • Rising clock edge sets currentstate to nextstate
  • Second process models combinational logic
    • Sensitive to currentstate and FSM inputs
    • Sets FSM outputs based on currentstate
    • Sets nextstate based on currentstate and present FSM input values
  • Note declaration of new type, statetype

Describing a Controller in Verilog

• FSM behavior captured using 2 "always" procedures
  • First procedure models state register
    • Asynchronous reset sets state to "S_Off"
    • Rising clock edge sets currentstate to nextstate
  • Second process models combinational logic
    • Sensitive to currentstate and FSM inputs
    • Sets FSM outputs based on currentstate
    • Sets nextstate based on currentstate and present FSM input values
  • Note state register size must be explicit – 2 bits, reg [1:0] currentstate
Describing a Controller in SystemC

- FSM behavior captured using 2 methods
  - First method models state register
    - Asynchronous reset sets state to "S_Off"
    - Rising clock edge sets currentstate to nextstate
  - Second process models combinational logic
    - Sensitive to currentstate and FSM inputs
    - Sets FSM outputs based on currentstate
    - Sets nextstate based on currentstate and present FSM input values
  - Note use of new type, statetype

```
#include "systemc.h"
enum statetype { S_Off, S_On1, S_On2, S_On3 };
SC_MODULE(LaserTimer) {
  sc_in<sc_logic> b, clk, rst;
  sc_out<sc_logic> x;
  sc_in<statetype> currentstate, nextstate;
  SC_CTOR(LaserTimer) {
    currentstate = S_Off; // Initial state
    nextstate = nextstate;
  }
  void statereg() {
    if ( rst.read() == SC_LOGIC_1)
      currentstate = S_Off; // initial state
    else
      currentstate = nextstate;
  }
  void combilogic() {
    switch (currentstate) {
    case S_Off:
      x.write(SC_LOGIC_0); // laser off
      if ( b.read() == SC_LOGIC_0 )
        nextstate = S_Off;
      else
        nextstate = S_On1;
      break;
    case S_On1:
      x.write(SC_LOGIC_1); // laser on
      nextstate = S_On2;
      break;
    case S_On2:
      x.write(SC_LOGIC_1); // laser still on
      nextstate = S_On3;
      break;
    case S_On3:
      x.write(SC_LOGIC_1); // laser still on
      nextstate = S_Off;
      break;
    }
  }
};
```

Datapath Component Description using Hardware Description Languages

- Will consider description of three datapath components
  - Full-adders
  - Carry-ripple adders
  - Up-counter
Describing a Full-Adder in VHDL

- **Entity**
  - Declares inputs/outputs

- **Architecture**
  - Described behaviorally (could have been described structurally)
  - Process sensitive to inputs
  - Computes expressions, sets outputs

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity FullAdder is
  port ( a, b, ci: in std_logic;
         s, co: out std_logic );
end FullAdder;

architecture behavior of FullAdder is
begin
  s <= a xor b xor ci;
  co <= (b and ci) or (a and ci) or (a and b);
end process;
end behavior;
```

\[ s = a \oplus b \oplus c_i \]
\[ c_o = b \cdot c_i + a \cdot c_i + a \cdot b \]

---

Describing a Full-Adder in Verilog

- **Module**
  - Declares inputs/outputs
  - Described behaviorally (could have been described structurally)
  - "always" procedure
    - Sensitive to inputs
  - Computes expressions, sets outputs

```verilog
module FullAdder(a, b, ci, s, co);
  input a, b, ci;
  output s, co;
  reg s, co;
  always @ (a or b or ci)
  begin
    s <= a ^ b ^ ci;
    co <= (b & ci) | (a & ci) | (a & b);
  end
endmodule
```

\[ s = a \oplus b \oplus c_i \]
\[ c_o = b \cdot c_i + a \cdot c_i + a \cdot b \]
Describing a Full-Adder in SystemC

- Module
  - Declares inputs/outputs
  - Described behaviorally (could have been described structurally)
  - *comlogic* method
    - Computes expressions, sets outputs

```
#include "systemc.h"

SC_MODULE(FullAdder)
{
  sc_in<sc_logic> a, b, ci;
  sc_out<sc_logic> s, co;

  SCCTOR(FullAdder)
  { SC_METHOD(comblogic);
    sensitive << a << b << ci;
  }

  void blogic()
  {
    s.write(a.read() ^ b.read() ^ ci.read());
    co.write((b.read() & ci.read()) |
              (a.read() & ci.read()) |
              (a.read() & b.read()));
  }
};
```

Describing a Carry-Ripple Adder in VHDL

- Entity
  - Declares inputs/outputs
  - Uses std_logic_vector for 4-bit inputs/outputs

- Architecture
  - Described structurally by composing four full-adders (could have been described behaviorally instead)
  - Declares full-adder component, instantiates four full-adders, connects
    - Note use of three internal signals for connecting carry-out of one stage to carry-in of next stage

```
library ieee;
use ieee.std_logic_1164.all;

entity CarryRippleAdder4 is
  port ( a, b, ci: in std_logic_vector(3 downto 0);
         s: out std_logic_vector(3 downto 0);
         co: out std_logic);
end CarryRippleAdder4;

architecture structure of CarryRippleAdder4 is
  component FullAdder
    port ( a, b, ci: in std_logic;
             s, co: out std_logic);
  end component;

  signal co1, co2, co3: std_logic;

  begin
    FullAdder1: FullAdder
      port map (a(0), b(0), ci, s(0), co1);
    FullAdder2: FullAdder
      port map (a(1), b(1), co1, s(1), co2);
    FullAdder3: FullAdder
      port map (a(2), b(2), co2, s(2), co3);
    FullAdder4: FullAdder
      port map (a(3), b(3), co3, s(3), co);
  end structure;
```
Describing a Carry-Ripple Adder in Verilog

• Module
  – Declares inputs/outputs
  – Uses vectors for 4-bit inputs/outputs
  – Described structurally by composing four full-adders (could have been described behaviorally instead)
  – Instantiates four full-adders, connects
    • Note use of three internal wires for connecting carry-out of one stage to carry-in of next stage

```verilog
module CarryRippleAdder4(a, b, ci, s, co);
  input [3:0] a;
  input [3:0] b;
  input ci;
  output [3:0] s;
  output co;
  wire co1, co2, co3;
  FullAdder FullAdder1(a[0], b[0], ci, s[0], co1);
  FullAdder FullAdder2(a[1], b[1], co1, s[1], co2);
  FullAdder FullAdder3(a[2], b[2], co2, s[2], co3);
  FullAdder FullAdder4(a[3], b[3], co3, s[3], co);
endmodule
```

Describing a Carry-Ripple Adder in SystemC

```c++
#include "systemc.h"
#include "fulladder.h"

SC_MODULE(CarryRippleAdder4)
{
  sc_in<sc_logic> a[4];
  sc_in<sc_logic> b[4];
  sc_in<sc_logic> ci;
  sc_out<sc_logic> s[4];
  sc_out<sc_logic> co;
  sc_signal<sc_logic> co1, co2, co3;

  FullAdder FullAdder_1;
  FullAdder FullAdder_2;
  FullAdder FullAdder_3;
  FullAdder FullAdder_4;

  SC_CTOR(CarryRipple4):
    FullAdder_1("FullAdder_1"),
    FullAdder_2("FullAdder_2"),
    FullAdder_3("FullAdder_3"),
    FullAdder_4("FullAdder_4")
    {
      FullAdder_1.a(a[0]); FullAdder_1.b(b[0]);
      FullAdder_1.ci(ci); FullAdder_1.s(s[0]);
      FullAdder_1.co(co1);
      FullAdder_2.a(a[1]); FullAdder_2.b(b[1]);
      FullAdder_2.ci(co1); FullAdder_2.s(s[1]);
      FullAdder_2.co(co2);
      FullAdder_3.a(a[2]); FullAdder_3.b(b[2]);
      FullAdder_3.ci(co2); FullAdder_3.s(s[2]);
      FullAdder_3.co(co3);
      FullAdder_4.a(a[3]); FullAdder_4.b(b[3]);
      FullAdder_4.ci(co3); FullAdder_4.s(s[3]);
      FullAdder_4.co(co);
    }
};
```
Describing an Up-Counter in VHDL

- Described structurally (could have been described behaviorally)
- Includes process that updates output port C whenever internal signal tempC changes
  - Need tempC signal because can’t read C due to C being an output port

Describing an Up-Counter in Verilog

- Described structurally (could have been described behaviorally)
- Includes always procedure that updates output C whenever internal wire tempC changes
  - Need tempC wire because can’t use C in the connection statements
Describing an Up-Counter in SystemC

- Described structurally (could have been described behaviorally)
- Includes method that updates output C whenever internal signal tempC changes
  - Need tempC signal because can't use C in the connection statements
- Can't use logic vector bits individually for connections, so needed tempC_b array too

```cpp
#include "systemc.h"
#include "reg4.h"
#include "inc4.h"
#include "and4.h"

SC_MODULE(UpCounter)
{
  sc_in<sc_logic> clk, cnt;
  sc_out<sc_lv<4>> > C;
  sc_out<sc_logic> tc;

  sc_signal<sc_lv<4>> tempC, incC;
  sc_signal<sc_logic> tempC_b[4];

  Reg4 Reg4_1;
  Inc4 Inc4_1;
  AND4 AND4_1;

  SC_CTOR(UpCounter) : Reg4_1("Reg4_1"), Inc4_1("Inc4_1"), AND4_1("AND4_1")
  {
    Reg4_1.I(incC); Reg4_1.Q(tempC);
    Reg4_1.clk(clk); Reg4_1.ld(cnt);
    Inc4_1.a(tempC); Inc4_1.s(incC);
    AND4_1.w(tempC_b[0]); AND4_1.x(tempC_b[1]);
    AND4_1.y(tempC_b[2]); AND4_1.z(tempC_b[3]);
    AND4_1.F(tc);
  }

  SC_METHOD(comblogic), sensitive << tempC;
  {
    void comblogic()
    {
      tempC_b[0] = tempC.read()[0];
      tempC_b[1] = tempC.read()[1];
      tempC_b[2] = tempC.read()[2];
      tempC_b[3] = tempC.read()[3];
      C.write(tempC);
    }
  }
};
```

RTL Design using Hardware Description Languages

- Will consider two forms of RTL descriptions
  - High-level state machine
  - Controller and datapath
High-Level State Machine of the Laser-Based Distance Measurer in VHDL

- Module similar to FSM, but single “always” procedure
- Asynch reset forces to state S0
- For each rising clock
  - Prepare to go to next state based on state and inputs

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity LaserDistMeasurer is
    port (clk, rst : in std_logic;
          B, S : in std_logic;
          L, D : out std_logic;
          Dctr : out unsigned(15 downto 0));
end;

architecture behavior of LaserDistMeasurer is

begin

  state_machine: process (clk, rst)
  begin
    if (rst='1') then
      L <= '0';  -- laser off
      D <= '0';  -- clear D
      state <= S1;  -- initial state
    elsif (clk='1' and clk'event) then
      case state is
        when S0 =>
          L <= '0';  -- laser off
          D <= '0';  -- clear D
          state <= S1;
        when S1 =>
          D <= U_ZERO;  -- reset count
          Dctr <= 0;
          if (B='1') then
            state <= S2;
          else
            state <= S1;
          end if;
        when S2 =>
          D <= U_ZERO;  -- laser on
          state <= S3;
          if (B='1') then
            state <= S4;
          end if;
        when S3 =>  
          D <= Dctr + 1;  -- calculate D
          state <= S4;
        when S4 =>
          D <= Dctr >> 1;  -- calculate D
          state <= S1;
      end case
    end if;
  end process
end;

end;
```

High-Level State Machine of the Laser-Based Distance Measurer in Verilog

- Module similar to FSM, but single “always” procedure
- Asynch reset forces to state S0
- For each rising clock
  - Prepare to go to next state based on state and inputs

```verilog
module LaserDistMeasurer(clk, rst, B, S, L, D);
input clk, rst, B, S;
output L;
output [15:0] D;

param S0 = 3'b000,
          S1 = 3'b001,
          S2 = 3'b010,
          S3 = 3'b011,
          S4 = 3'b100;

reg [2:0] state;
reg [16:0] Dctr;

always @ (posedge rst or posedge clk)
begin
  if (rst==1)
  begin
    L <= 0;
    D <= 0;
    state <= S0;  // initial state
  end
  else begin
    case state
      S0:
      begin
        L <= 0;  // laser off
        state <= S1;
      end
      S1:
      begin
        D <= U_ZERO;  // clear D
        if (B==1)
        begin
          state <= S2;
        end
        else
        begin
          state <= S1;
        end
      end
      S2:
      begin
        D <= Dctr + 1;  // calculate D
        state <= S4;
      end
      S3:
      begin
        D <= U_ZERO;  // laser off
        state <= S2;
      end
      S4:
      begin
        D <= Dctr >> 1;  // calculate D
        state <= S1;
      end
    end case
  end
endmodule
```
High-Level State Machine of the Laser-Based Distance Measurer in SystemC

- Module similar to FSM, but single method
- Asynch reset forces to state S0
- For each rising clock
  - Perform state's computation
  - Prepare to go to next state based on state and inputs

(continued from Figure 9.38)

Controller and Datapath of the Laser-Based Distance Measurer in VHDL

- At highest level, just connection of controller and datapath components
**Datapath of the Laser-Based Distance Measurer in VHDL**

- **Inputs:** B, S
- **Outputs:** L, Dreg_clr, Dreg_ld, Dctr_clr, Dctr_cnt

- **Datapath just another connection of components**
  - Assume up-counter, register, and shift-right components are already designed (similar to earlier-designed items)

- ** FSM similar to high-level state machine**
  - But high-level operations replaced by low-level datapath signals
  - Use two-process FSM description approach

---

**Controller of the Laser-Based Distance Measurer in VHDL**

```
library ieee;
use ieee.std_logic_1164.all;

entity LDM_Controller is
  port (clk, rst: in std_logic;
        B, S: in std_logic;
        D: out std_logic_vector(15 downto 0));
end LDM_Controller;

architecture behavior of LDM_Controller is
begin
  LDM_Controller: process (clk, rst)
  begin
    if (rst='1') then
      currentstate <= S0 -- initial state
    elsif (clk'event) then
      currentstate <= nextstate;
    end if;
  end process;

  comlogic: process (currentstate, B, S)
  begin
    case currentstate is
      when S0 =>
        when B='1' =>
          L <= '1'; -- laser on
          D <= '0';
          nextstate <= S1;
          Dreg_clr <= '1'; -- clear Dreg
          Dreg_ld <= '1'; -- load Dreg
          Dctr_cnt <= '0';
          Dctr_clr <= '1';
        else
          nextstate <= S3;
      when S1 =>
        when Dreg <= '1' =>
          D <= '1';
          nextstate <= S2;
                  nextstate <= S4;
                  if (S='1') then
                    D <= '0';
                    nextstate <= S4;
                  else
                    nextstate <= S3;
                  end if;
    end case;
  end process;
end behavior;
```

---

**Datapath**

- **Dreg:** 16-bit register
- **Dctr:** 16-bit register
- **Dreg clr:** clear Dreg
- **Dreg ld:** load Dreg
- **Dctr clr:** clear count
- **Dctr cnt:** stop counting
Controller and Datapath of the Laser-Based Distance Measurer in Verilog

- At highest level, just connection of controller and datapath components

```
module LaserDistMeasurer(clk, rst, B, S, L, D);
  input clk, rst, B, S;
  output L;
  output [15:0] D;
  wire Dreg_clr, Dreg_ld;
  wire Dctr_clr, Dctr_cnt;

LDM_Controller
  LDM_Controller_1(clk, rst, B, S, L, Dreg_clr, Dreg_ld, Dctr_clr, Dctr_cnt);
LDM_Datapath
  LDM_Datapath_1(clk, Dreg_clr, Dreg_ld, Dctr_clr, Dctr_cnt, D);
endmodule
```

Datapath of the Laser-Based Distance Measurer in Verilog

- Datapath just another connection of components
  - Assume up-counter, register, and shift-right components are already designed (similar to earlier-designed items)

```
module UpCounter16(clk, clr, cnt, C);
  input clk, clr, cnt;
  output [15:0] C;
  // details not shown
endmodule

module Reg16(I, Q, clk, clr, ld);
  input [15:0] I;
  input clk, clr, ld;
  output [15:0] Q;
  // details not shown
endmodule

module ShiftRightOne16(I, S);
  input [15:0] I;
  output [15:0] S;
  // details not shown
endmodule

module LDM_Datapath(clk, Dreg_clr, Dreg_ld, Dctr_clr, Dctr_cnt, D);
  input clk;
  input Dreg_clr, Dreg_ld;
  input Dctr_clr, Dctr_cnt;
  output [15:0] D;
  wire [15:0] tempC, shiftC;
  UpCounter16 Dctr(clk, Dctr_clr, Dctr_cnt, tempC);
  ShiftRightOne16 ShiftRight(tempC, shiftC);
  Reg16 Dreg(shiftC, D, clk, Dreg_clr, Dreg_ld);
endmodule
```
Controller of the Laser-Based Distance Measurer in Verilog

- FSM similar to high-level state machine
  - But high-level operations replaced by low-level datapath signals
  - Use two-procedure FSM description approach

```
module LDM_Controller(clk, rst, B, S, L, Dreg_clk, Dreg_ld, Dctr_CLR, Dctr_cnt);
  input clk, rst, B, S;
  output L;
  output Dreg_clk, Dreg_ld, Dctr_CLR, Dctr_cnt;
  reg
    L;
    Dreg_clk, Dreg_ld;
    Dctr_CLR, Dctr_cnt;
  parameter
    S0 = 3'b000, // initial state
    S1 = 3'b001,
    S2 = 3'b010,
    S3 = 3'b011,
    S4 = 3'b100;
  reg
    [2:0] currentstate;
    reg [2:0] nextstate;
  always @ (posedge rst or posedge clk)
    begin
      if (rst==1) currentstate <= S0;
      else currentstate <= nextstate;
    end
  always @ (currentstate or B or S)
    begin
      L <= 0; // laser off
      Dreg_CLR <= 1; // clear Dreg
      nextstate <= S1;
    end
    S1 begin
      L <= 1; // laser on
      Dctr_CLR <= 1; // clear count
      if (B==1) nextstate <= S2;
      else nextstate <= S1;
    end
    S2 begin
      Dreg_clr <= 0; // load Dreg
      Dreg_clk <= 1; // load Dreg
      nextstate <= S3;
    end
    S3 begin
      L <= 0; // laser off
      Dctr_cnt <= 1; // count up
      if (S==1) nextstate <= S4;
      else nextstate <= S3;
    end
    S4 begin
      Dreg_ld <= 1; // load Dreg
      Dctr_cnt <= 0; // stop counting
      nextstate <= S1;
    end
  endcase
endmodule
```

Controller and Datapath of the Laser-Based Distance Measurer in SystemC

- At highest level, just connection of controller and datapath components

```
#include "systemc.h"
#include "LDM_Controller.h"
#include "LDM_Datapath.h"

SC_MODULE(LaserDistMeasurer)
{
  sc_in<sc_logic> clk, rst;
  sc_in<sc_logic> B, S;
  sc_out<sc_logic> L;
  sc_out<sc_lv<16>> D;
  LDM_Controller LDM_Controller_1;
  LDM_Datapath LDM_Datapath_1;
  SCCTOR(LaserDistMeasurer) :
    LDM_Controller_1("LDM_Controller_1"),
    LDM_Datapath_1("LDM_Datapath_1")
  { 
    LDM_Controller_1.clk(clk);
    LDM_Controller_1.rst(rst);
    LDM_Controller_1.B(B);
    LDM_Controller_1.S(S);
    LDM_Controller_1.Dreg_clr(Dreg_clr);
    LDM_Controller_1.Dreg_ld(Dreg_ld);
    LDM_Controller_1.Dctr_CLR(Dctr_CLR);
    LDM_Controller_1.Dctr_cnt(Dctr_cnt);
    LDM_Datapath_1.clk(clk);
    LDM_Datapath_1.Dreg_clr(Dreg_clr);
    LDM_Datapath_1.Dreg_ld(Dreg_ld);
    LDM_Datapath_1.Dctr_CLR(Dctr_CLR);
    LDM_Datapath_1.Dctr_cnt(Dctr_cnt);
    LDM_Datapath_1.D(D);
  }
```

(continued from Figure 9.46)

```
#begin
LDM_Controller #begin
LDM_Datapath #begin
```
Datapath of the Laser-Based Distance Measurer in SystemC

- FSM similar to high-level state machine
  - But high-level operations replaced by low-level datapath signals
- Use two-procedure FSM description approach

Controller of the Laser-Based Distance Measurer in SystemC

(continued from Figure 9.50)

```c
$include "systemc.h"
$include "systemc.h"
$include "upcounter16.h"
$include "shiftrightone16.h"

SC_MODULE(LDM_Datapath) {
  Sc_logic clk, rst, B, S;
  Dreg<16> Dreg;
  Dctr<16> Dctr;
  Upcounter16 Dctr;
  Shiftrightone16 Shiftright;

  Sc_logic Dreg_clr, Dreg_ld, Dctr_clr, Dctr_cnt;

  void comblogic() {
    Sc_logic tempC;
    Sc_logic tempM;

    currentstate = nextstate;
    nextstate = S0;
    switch (currentstate) {
      case S0:
        write(Dreg_clr, SC_LOGIC_0); // laser off
        write(Dctr_cnt, 0);
        break;
      case S1:
        write(Dreg_clr, SC_LOGIC_1); // clear Dreg
        nextstate = S2;
        break;
      case S2:
        write(Dreg_clr, SC_LOGIC_0); // laser off
        Dreg.ld(Dreg_ld);
        nextstate = S3;
        break;
      case S3:
        write(Dreg_clr, SC_LOGIC_1); // clear count
        Dctr_clr = 1;
        nextstate = S4;
        break;
      case S4:
        write(Dreg_clr, SC_LOGIC_0); // load Dreg with Dctr/2
        L = 1;
        nextstate = S0;
        break;
    }

    L = (L & B) == 1; // laser on
  }
}
```

51

Digital Design
Copyright © 2006
Frank Vahid

52

Digital Design
Copyright © 2006
Frank Vahid
Chapter Summary

- Hardware Description Languages (HDLs) are widely used in modern digital design
  - Textual rather than graphical language sufficient for many purposes
  - HDLs are computer-readable
  - Great for simulation
- VHDL, Verilog, and SystemC are popular
- Introduced languages mainly through examples
- Numerous HDL books exist to teach each language in more detail