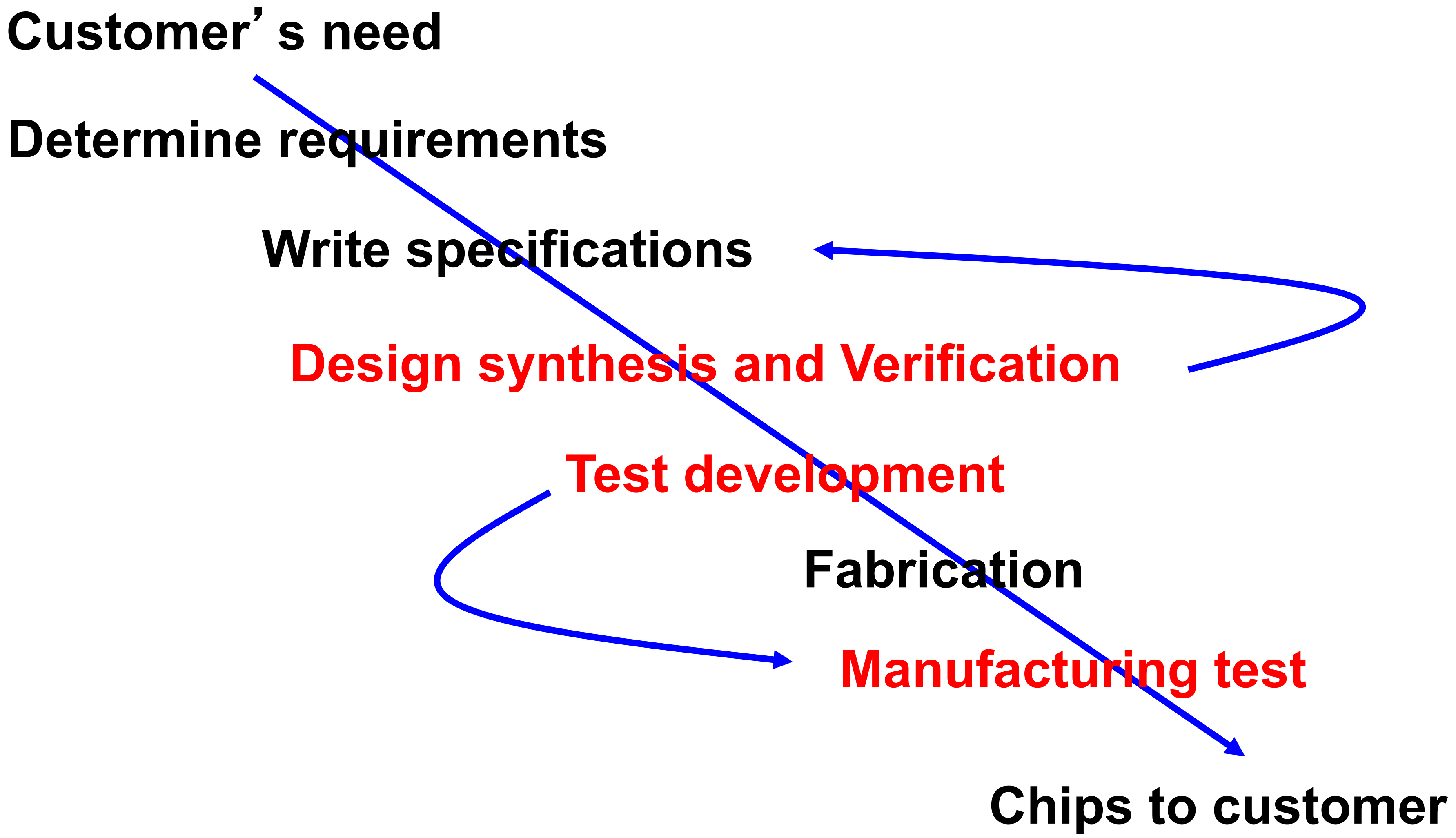
Introduction to Behavioural VHDL

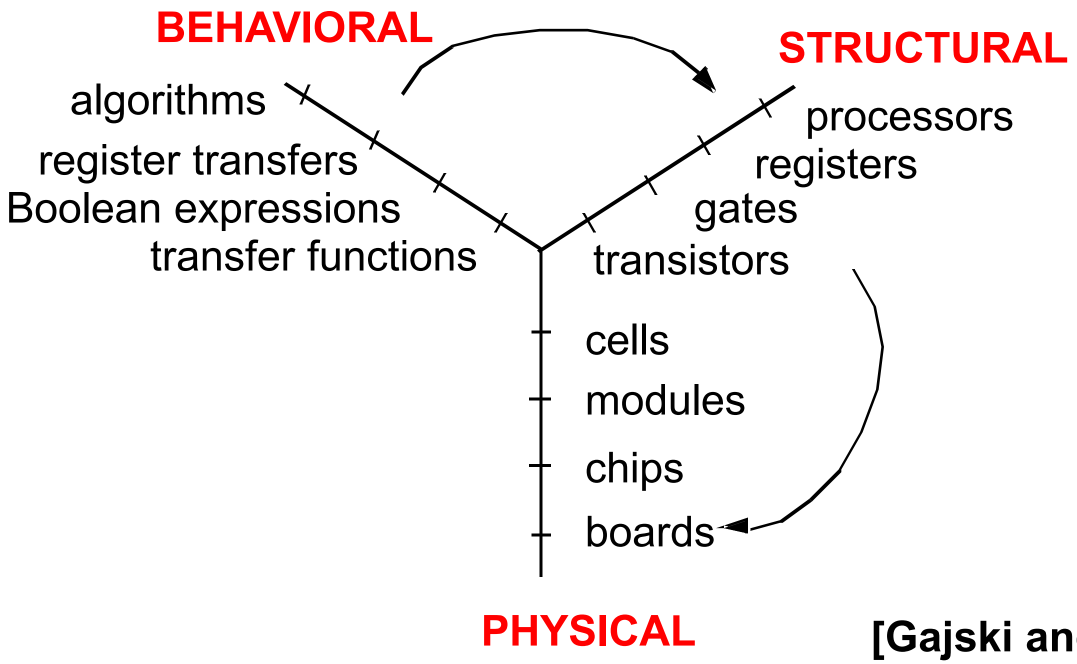
Virendra Singh
Professor
Dept. of Electrical Engineering, and
Dept. of Computer Science & Engg.
Indian Institute of Technology Bombay
http://www.ee.iitb.ac.in/~viren/
E-mail: viren@{ee,cse}.iitb.ac.in

7th June 2024

VLSI Realization Process



The Role of Hardware Description Languages



[Gajski and Kuhn]

- Design is structured around a hierarchy of representations
- HDLs can describe distinct aspects of a design at multiple levels of abstraction

Basic VHDL Concepts

- Interfaces
- Modeling (Behavior, Dataflow, Structure)
- Test Benches
- Analysis, elaboration, simulation
- Synthesis

Behavioral VHDL

Assignment Statement

- Signal assignment
- Variable assignment

```
Signal STAT_OUT: Std_Logic;
Stat_out <= not state_in;

Variable Preset, count:unsigned (0 to 3);
Count := preset +1;
```

Logical Operators

```
Library IEEE;
Use IEEE.Std_logic_1164.all;
Entity Full_Adder is
Port (A, B, CIN: in Std_logic;
        sum, cout: out Std_logic);
Architecture Dataflow of Full_adder is
Begin
 sum <= (A xor B) xor CIN;
Cout <= (A and B) or (B and CIN) or (A and CIN);
End;
```

Arithmetic Operators

```
Library IEEE;
Use IEEE.numeric_std.all;
Entity Unsigned_Adder is
Port (A, B: in unsigned (0 to 3);
        sum: out  unsigned (0 to 3));
Architecture Simple of Unsigned_adder is
Begin
 sum <= A + B;
End Simple;
```

Relational Operators

```
library IEEE;
use IEEE.numeric_std.all;
entity GT is
port (A, B: in unsigned (3 downto 0);
        Z: out  Boolean);

architecture DF of GT is
begin
 Z <= A (1 down to 0) > B (3 downto 2);
end Simple;
```

Vector and Slices

```
Library IEEE;
Use IEEE.std_logic_1164.all;
package ARRAYS is
Type BANK is array (0 to 1) of Std_logic_vector (3 downto 0);
End ARRAYS;
Library IEEE;
Use IEEE.std_logic_1164.all , work.arrays.all;
entity GT is
port (A, B, C: in std_logic_vector (3 downto 0);
            REG_FILE: inout BANK;
         Z: out  std_logic_vector (3 downto 0));
end GT;
```

Inferring Latches from If Statements

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity INCR is
port (A: in bit;
          Z: out  unsigned (0 to 1));
End INCR;
```

Inferring Latches from If Statements

```
architecture Example of INCR  is
Begin
    INCR_L: process (A)
        variable ONES: unsigned (0 to 1)
        begin
            if A = ‘1’ then
                ONES := ONES + 1;
            end if;
            Z <= ONES;
        end process INCR_L;
end Example;
```

Inferring Latches from If Statements

```
Package EXAM is
    type GRADE_TYPE is (FAIL, PASS, EXCELLENT);
end;
library IEEE;
use IEEE.std_logic_1164.all;
use work.exam.all;
Entity compute is
port (marks: in natural in range 0 to 10;
           grade: out  GRADE_TYPE);
End compute;
```

Inferring Latches from If Statements

```
architecture Example of compute  is
begin
    process (marks)
    begin
        if marks < 5 then
                grade <= FAIL;
        elsif marks >= 5 and marks < 7 then
                grade <= PASS;
        end if;
    end process ;
end Example;
```

Inferring Latches from If Statements

```
Package EXAM is
    type GRADE_TYPE is (FAIL, PASS, EXCELLENT);
end;
library IEEE;
use IEEE.std_logic_1164.all;
use work.exam.all;
Entity compute_mod is
port (marks: in natural in range 0 to 10;
            grade: out  GRADE_TYPE);
End compute_mod;
```

Inferring Latches from If Statements

```
architecture Example of compute_mod  is
begin
    process (marks)
    begin
          if marks < 5 then
                    grade <= FAIL;
          elsif marks >= 5 and marks < 7 then
                    grade <= PASS;
          else grade <= EXCELLENT;
          end if;
    end process ;
end Example;
```

Inferring Latches from If Statements: Exception for Variables

```
signal A, B, clk;
. . . .
P1: process (A, clk)
    variable p: std_logic;
begin
    if clk = '1' then
            B <= p;
            p := A;
    end if;
    end process ;
```

Inferring Latches from If Statements: Exception for Variables

```
P2: process (A, clk)
    variable q: std_logic;
begin
    if clk = '1' then
        q := A;
        B <= q;
    end if;
end process P2;
end
```

Case Statements

```
Package PACK_A is
    type OP_TYPE is (ADD, SUB, MUL, DIV);
end;
library IEEE;
use IEEE.std_logic_1164.all, IEEE.numeric_std.all;
use work.exam.all;
Entity ALU is
port (OP: in OP_TYPE;
          A, B: in  unsigned (0 to 1);
          Z: out unsigned (0 to 1));
End ALU;
```

Case Statements

```
architecture Example of ALU  is
begin
    process (OP, A, B)
            variable tmp: unsigned (3 downto 0);
    begin
            case OP is
                    when ADD =>
                            Z <= A + B;
                    when SUB =>
                            Z <= A - B;
```

Case Statements

```
                when MUL =>
                        tmp := A * B;
                        Z <= tmp (1 downto 0);
                when DIV =>
                        Z <= A / B;
        end case;
    end process ;
end Example;
```

Case Statements

```
package COLLECT is
    type STATES is (S0, S1, S2, S3);
end;
library IEEE;
use IEEE.std_logic_1164.all, work.pack_b.all;
entity state_update is
port (curr_state: in STATES;
          Z: out integer range 0 to 3);
end state_update;
```

Case Statements

```
architecture Example of state_update  is
begin
    process (curr_state)
            variable tmp: unsigned (3 downto 0);
    begin
            case curr_state is
                    when S0 | S3 =>
                            Z <= 0;
                    when S1 =>
                            Z <= 3;
                    when others =>
                            null;
            end case;
    end process;
end Example;
```

Case Statements

```
architecture Example of state_update  is
begin
    process (curr_state)
            variable tmp: unsigned (3 downto 0);
    begin
            Z <= 0;
            case curr_state is
                        when S0 | S3 =>
                                    Z <= 0;
                        when S1 =>
                                    Z <= 3;
                        when others =>
                                    null;
            end case;
    end process;
End Example;
```

FSM Implementation

```
architecture fsm_1 of fsm is
      signal cur_state: fsm_state;
      signal nxt_state: fsm_state;

State_change: process(clk, reset)
begin
  if (clk’event and clk=1)
     if reset = ‘1’ then
              cur_state <= S0;
     else
              cur_state <= nxt_state;
     end if;
  end if;
end process state_change;
```

FSM Implementation

```
Next_state_logic: process(cur_state, reset)
begin
  case cur_state is
            when S0 =>
              if ready = ‘1’ then
                nxt_state <= S1;
              else
              nxt_state <= S0;
              end if;
          when S1 =>
              nxt_state <= S2;
       . . . . . . . . . . .
   end case;
end process;
```

FSM Implementation

```
Output_logic: process(cur_state)
begin
  case cur_state is
              when S0 =>
                           A <= ‘0’;
                            B <= ‘0’;
                           C <= ’00”;
                           D <= “00”;
              when S1 =>
                           A <= ‘1’;
                            B <= ‘0’;
                           C <= ’11”;
                           D <= “01”;
. . . .
   end case
end process;
end;
```

Loop Statements

3 kinds of loop in VHDL

- While-loop
- For – loop
- Loop

For-loop is supported by synthesis

Loop Statements

```
library IEEE;
use IEEE.std_logic_1164.all, IEEE.numeric_std.all;
entity DEMUX is
port (A: in unsigned (1 downto 0);
          Z: out unsigned (3 downto 0));
end DEMUX;
```

Loop Statements

```
architecture For_Loop of DEMUX  is
begin
    process (A)
          variable tmp: integer range 0 to 3;
    begin
          tmp := to_integer (A);
          for J in Z`range loop
                    if tmp = J then
                              Z(J) <= ‘1’;
                    end if;
          end loop;
    end process ;
end Example;
```

Wait Statements

3 kinds of loop in VHDL

- Wait for *time*
- Wait until *condition*
- Wait on *signal-list*

Wait until is supported by synthesis

Imply synchronous logical behaviour

If used, it should be the first statement in the process

Condition must be one of the allowed clock expression

- Wait until *clock-name = clock_value;*
- Wait until *clock-name = clock_value and clock_name`EVENT*;

Loop Statements

```
library IEEE;
use IEEE.std_logic_1164.all, IEEE.numeric_std.all;
Entity INCR is
port (clk: in std_logic;
          counter: out unsigned (1 downto 0));
end DEMUX;
architecture FLOP of INCR is
Begin
   process
   begin
          wait until clk = ‘1’;
          counter <= counter + 1;
   end process;
End FLOP;
```

Loop Statements

```
architecture For_Loop of DEMUX  is
begin
    process (A)
            variable tmp: integer range 0 to 3;
    begin
            tmp := To_integer (A);
            for J in Z`range loop
                        if tmp = J then
                                    Z(J) <= ‘1’;
                        end if;
            end loop;
    end process ;
end Example;
```

Modeling Memories

- RAM can be modeled as registers
- Storage represented by array, bit vectors, or integers
- An address vector, converted to integer, is used to index the array
- Declaration of an array type and a signal of that type

```
type mem-array is array (0 to 2**depth -1) of
                std_ulogic_vector(width -1 downto 0);
signal RAM: mem_array;
```

Asynchronous RAM

- Level sensitive device
- Model like a latch
- Behaviour model

```
Asynch_RAM:   process (addr, din, we)
    begin
          if we = '1' then
                    RAM(to_integer (addr)) <= din;
          end if;
    end process ;
    dout <= RAM (to_integer (addr));
```

Synchronous RAM with Asynchronous Read

- Write operation is synchronous
- Have embedded registers that store address and data
- Read is asynchronous

```
Synch_RAM:    process (clk)
    begin
        if rising_edge (clk)  then
                if we = ‘1’ then
                        RAM(to_integer (addr)) <= din ;
                end if;
        end if;
    end process ;
    dout <= RAM (to_integer (addr));
```

Synchronous RAM with Synchronous Read

- Have embedded registers that store address and data
- Differ in data they read when write is also performed

```
Synch_RAM:    process (clk)
    begin
        if rising_edge (clk)  then
                dout <= RAM (to_integer (addr));
                if we = ‘1’ then
                        RAM(to_integer (addr)) <= din ;
                end if;
        end if;
    end process Synch_RAM;
```

Synchronous RAM with Synchronous Read

- Have embedded registers that store address and data
- Differ in data they read when write is also performed

```
Synch_RAM:    process (clk)
    begin
        if rising_edge (clk)  then
                if we = ‘1’ then
                        RAM(to_integer (addr)) <= din ;
                end if;
                dout <= RAM (to_integer (addr));
        end if;
    end process Synch_RAM;
```

Synchronous RAM with Synchronous Read

- Have embedded registers that store address and data
- Differ in data they read when write is also performed
- Provides new data

```
Synch_RAM:   process (clk)
   begin
      if rising_edge (clk)  then
         if we = ‘1’ then
            RAM(to_integer (addr)) <= din ;
         else
            dout <= RAM (to_integer (addr));
         end if;
      end if;
   end process Synch_RAM;
```

Synchronous RAM with Synchronous Read

- Add enable input

```
Synch_RAM:    process (clk)
    begin
        if rising_edge (clk)  then
                if en = ‘1’ then
                        dout <= RAM (to_integer (addr));
                        if we = ‘1’ then
                                RAM(to_integer (addr)) <= din ;
                        end if;
                end if;
        end if;
    end process Synch_RAM;
```

Thanks