Computer-Aided Design

Instantiation & Testbench

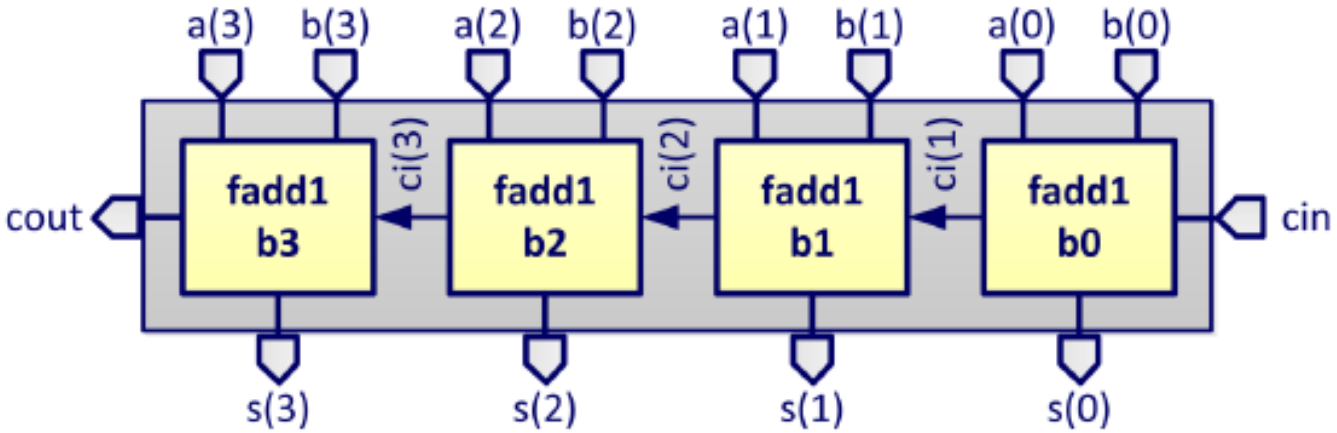
Mahdi Aminian

The University Of Guilan
Rasht - Iran

Some slides courtesy of:

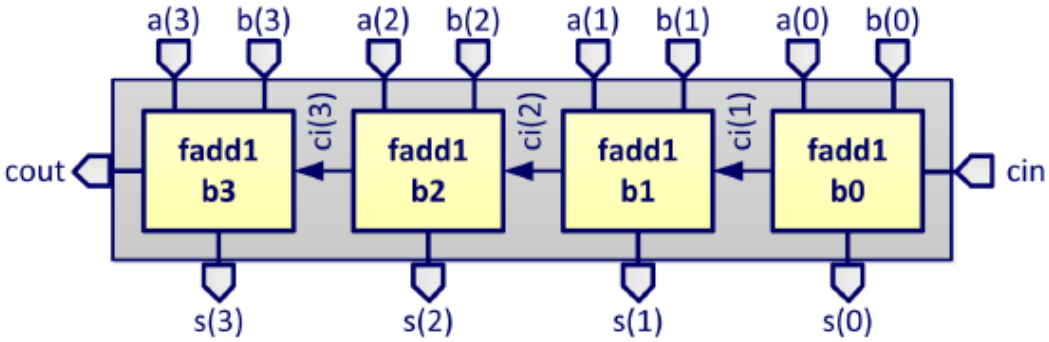
- “Hardware Systems Modeling”, A. Vachoux, EPFL
- CAD slides from Dr. Saheb Zamani

4-bit Ripple-Carry Adder




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

entity adder4 is
   port (
      signal a, b : in  std_logic_vector(3 downto 0);  -- operands
      signal cin  : in  std_logic;                     -- input carry
      signal sum  : out std_logic_vector(3 downto 0);  -- sum
      signal cout : out std_logic);                    -- output carry
end entity adder4;
```

4-bit Ripple-Carry Adder: Dataflow Architecture (1)

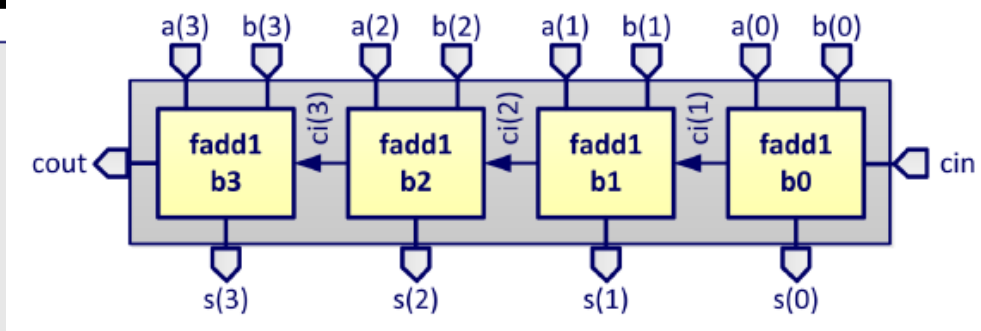



```
architecture dfl of adder4 is
   signal ci : std_logic_vector(1 to 3);
begin
   -- bit 0
   B0_S_0  : s(0)  <= a(0) xor b(0) xor cin;
   B0_CO_0 : ci(1) <= (a(0) and b(0)) or (a(0) and cin) or (b(0) and cin);
   -- bit 1
   B0_S_1  : s(1)  <= a(1) xor b(1) xor ci(1);
   B0_CO_1 : ci(2) <= (a(1) and b(1)) or (a(1) and ci(1)) or (b(1) and ci(1));
   -- bit 2
   B0_S_2  : s(2)  <= a(2) xor b(2) xor ci(2);
   B0_CO_2 : ci(3) <= (a(2) and b(2)) or (a(2) and ci(2)) or (b(2) and ci(2));
   -- bit 3
   B0_S_3  : s(3)  <= a(3) xor b(3) xor ci(3);
   B0_CO_3 : cout  <= (a(3) and b(3)) or (a(3) and ci(3)) or (b(3) and ci(3));
end architecture dfl;
```

- **Not easily scalable**
- **Repetitive behavior (or structure) should be better factorized**

4-bit Ripple-Carry Adder: Dataflow Architecture (2)

```
architecture dfl2 of adder4 is
   signal ci : std_logic_vector(0 to 4);
begin
   ci(0) <= cin;
   STAGES : for k in 0 to 3 generate
      BS  : s(k)    <= a(k) xor b(k) xor ci(k);
      BCO : ci(k+1) <= (a(k) and b(k)) or (a(k) and ci(k)) or (b(k) and ci(k));
   end generate STAGES;
   cout <= ci(4);
end architecture dfl2;
```



- **Use of a concurrent loop generate statement**
  - Sort of a pre-processing macro
  - May include any legal concurrent statement
  - Loop variable k is implicitly declared and only visible in the generate statement
  - Label (here, STAGES) is mandatory
- **No structure or hierarchy implied**

Component Declaration / Component Instantiation

- **Hierarchical/structural models**
  - Maintained through synthesis

```
component component-name is
   generic (generic-parameters);
   port (signal-ports);
end component component-name;
```

- **Component declaration**
  - In an architecture declarative part or in a package declaration
  - Similar to an entity declaration, **but**:

    A **component declaration** defines what is **needed**

    (prototype of a design entity)

    An **entity declaration** defines what is **available**

    A **configuration** may be required to bind both

- **Component instantiation**
  - In an architecture statement part

```
label : component component-name
   generic map (generic-parameter-bindings)
   port map (signal-port-bindings);
```

- **Direct instantiation does not need any component declaration**

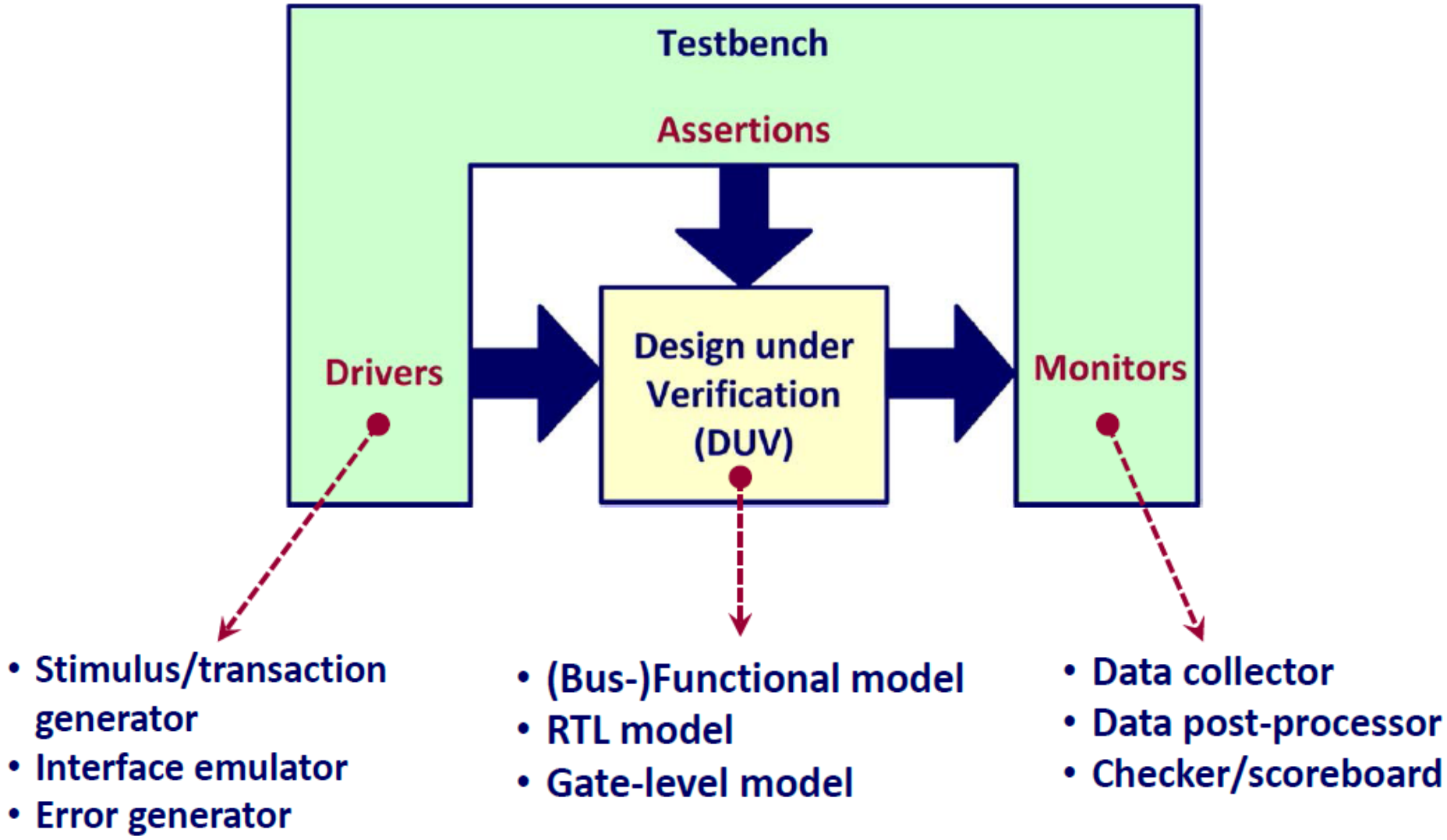
```
label : entity library-name.entity-name(architecture-name)
   port map (signal-port-bindings);
```

1-Bit Full Adder: Structural Architecture

```
Architecture Structural of adder4 is
  signal Ci : std_logic_vector (3 downto 1);
begin
  Add0: entity work.fadd1(dfl)
        port map (opa=>a(0), opb=>b(0),
                  cin=>cin, sum=> sum(0),cout=>Ci(1) );

  Add1: entity work.fadd1(dfl)
        port map (a(1), b(1), Ci(1), sum(1),Ci(2) );

  Add2: entity work.fadd1(dfl)
        port map (opb=>b(2), opa=>a(2),
                  cin=>Ci(2), cout=>Ci(3), sum=> sum(2) );

   Add3: entity work.fadd1(dfl)
        port map (a(3), b(3), Ci(3), sum(3),Cout );

end architecture structural;
```

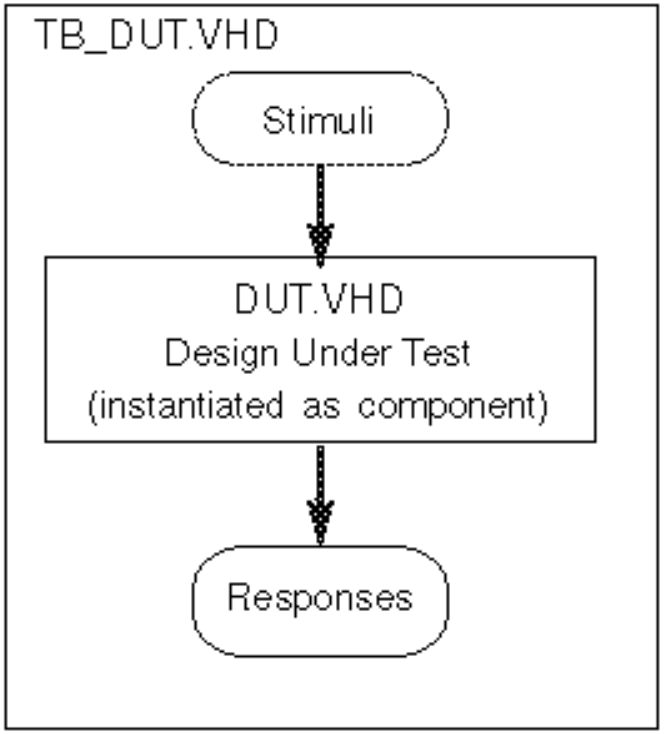
- **Component instances interconnected through signals**
  - port map
- **Direct instantiations of existing design entities from design library**
- **Each component instance is similar to a process**

Testbench Model Architecture



Testbench

- **VHDL code for top level**
- **No interface signals**
- **Instantiation of design**
- **Statements for stimuli generation**
- **Simple testbenches: response analysis by waveform inspection**
- **Sophisticated testbenches may need >50% of complete project resources**



Basic Structure of a VHDL Testbench Model

```vhdl
library ieee;
use ieee.std_logic_1164.all;
-- + other required packages...

entity testbench is
end entity testbench;

architecture bench of testbench is
   -- constant, signal, subprogram declarations...
begin

   -- instantiation of design entity under verif.
   DUV : entity work.uut_ent(uut_arch)
      generic map (...)   -- if needed
      port map (...);
   ...
   ...
   -- stimulus generation
   STIM : process
   begin
      ...
      wait; -- forever
   end process STIM;

   -- monitoring and checks
   CHK: process
   begin
      ...
   end process CHK;

end architecture bench;
```

- **Testbench interacts with DUV as if it was another hardware component**
  - Concurrency and timing issues
- **Processes STIM and CHK may also be defined as components**
- **Testbench is not intended to be synthesized**

Simple Testbench Example

```
entity ADDER IS
   port (A,B :   in   bit;
         CARRY,SUM : out bit);
end ADDER;
architecture RTL of ADDER is
begin
   ADD: process (A,B)
   begin
      SUM <= A xor B;
     CARRY <= A and B;
   end process ADD;
end RTL;
```

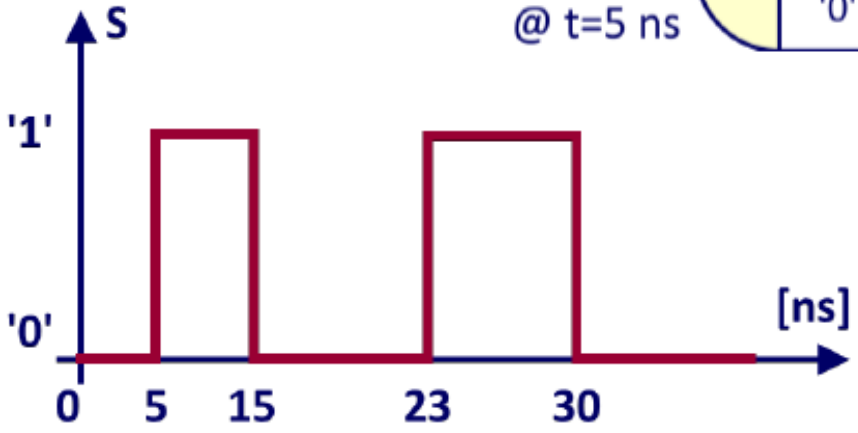
```
entity TB_ADDER IS
end TB_ADDER;
architecture TEST of TB_ADDER is
  component ADDER
      port (A, B:              in   bit;
          CARRY, SUM: out bit);
  end component;
  signal A_I, B_I, CARRY_I, SUM_I : bit;
begin
  DUT: ADDER port map (A_I, B_I, CARRY_I, SUM_I);
 STIMULUS: process
 begin
    A_I <= ´1´; B_I <= ´0´;
    wait for 10 ns;
    A_I <= ´1´; B_I <= ´1´;
    wait for 10 ns;
    -- and so on ...
 end process STIMULUS;
end TEST;
configuration CFG_TB_ADDER of TB_ADDER is
for TEST
   for DUT:ADDER use entity work.ADDER(RTL);
end for;
end CFG_TB_ADDER;
```

Examples of a Stimulus Generation Process

```
signal s : std_logic := '0';
```

```
STIM_GEN1 : process
begin
   wait for 5 ns;
   s <= '1', '0' after 10 ns, '1' after 18 ns, '0' after 25 ns;
   wait; -- forever
end process STIM_GEN1;
```

```
STIM_GEN2 : process
begin
   wait for 5 ns;
   s <= '1';
   wait for 10 ns;
   s <= '0';
   wait for 8 ns;
   s <= '1';
   wait for 7 ns;
   s <= '0';
   wait; -- forever
end process STIM_GEN2;
```



Current value @ t=5 ns

	15 ns	23 ns	30 ns
'1'	'0'	'1'	'0'

- **Two functionally equivalent processes**

Stimulus: Deterministic Waveform

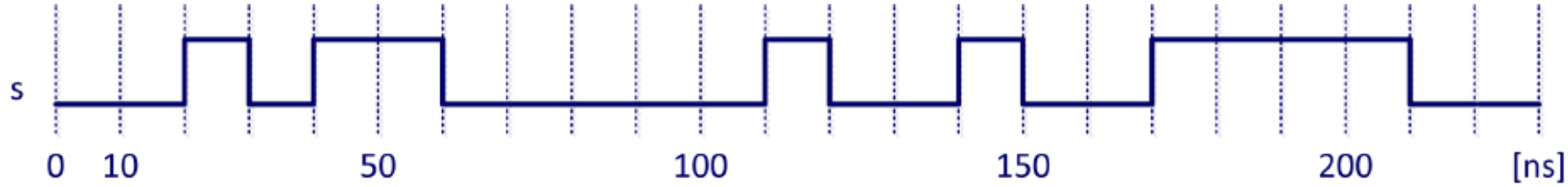
```
signal s : std_logic;
```

- **Concurrent or sequential signal assignment**

```
STIM : s <= '0',
            '1' after 20 ns,
            '0' after 30 ns,
            '1' after 40 ns,
            '0' after 60 ns,
            '1' after 110 ns,
            '0' after 120 ns,
            '1' after 140 ns,
            '0' after 150 ns,
            '1' after 170 ns,
            '0' after 210 ns;
```

- **Sequential signal assignment**

```
STIM : process
begin
   s <= '0'; wait for 20 ns;
   s <= '1'; wait for 10 ns;
   s <= '0'; wait for 10 ns;
   s <= '1'; wait for 20 ns;
   s <= '0'; wait for 50 ns;
   s <= '1'; wait for 10 ns;
   s <= '0'; wait for 20 ns;
   s <= '1'; wait for 10 ns;
   s <= '0'; wait for 20 ns;
   s <= '1'; wait for 40 ns;
   s <= '0'; wait for 20 ns;
   wait;
end process STIM;
```



Verification of a Combinational Design Entity

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

entity fadd1_tb is
end entity fadd1_tb;

architecture bench1 of fadd1_tb is
   constant DELAY : delay_length := 10 ns;
   signal opa, opb, cin : std_logic := '0';
   signal sum, cout : std_logic;
   ...
```

```
   ...
begin
   -- instantiation of design entity under verif.
   DUV : entity work.fadd1(dfl)
      port map (
         opa => opa, opb => opb, cin => cin,
         sum => sum, cout => cout);

   -- stimulus generation
   STIM1 : opa <= not opa after DELAY;
   STIM2 : opb <= not opb after 2*DELAY;
   STIM3 : cin <= not cin after 4*DELAY;

end architecture bench1;
```

- **DUV: 1-bit full adder**
- **Verification by inspection**