CPE 487: Digital System Design

Spring 2018

Lecture 8

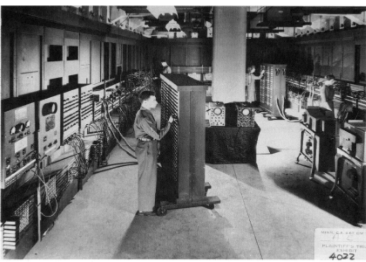
Structural Modeling

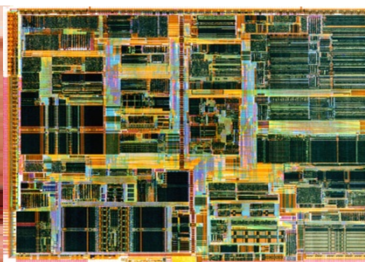
Bryan Ackland

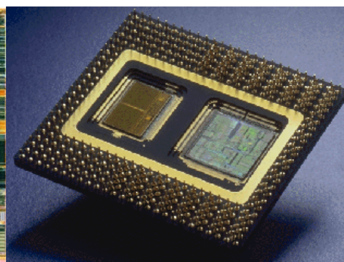
Department of Electrical and Computer Engineering

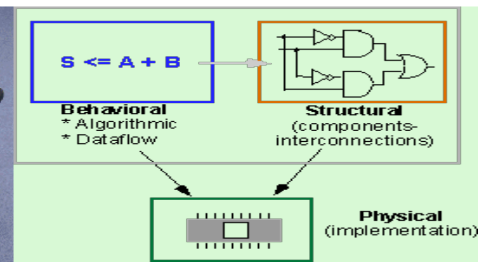
Stevens Institute of Technology

Hoboken, NJ 07030







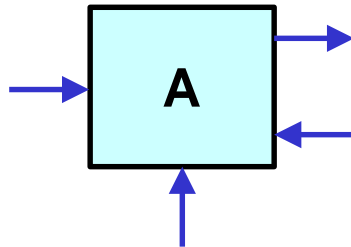


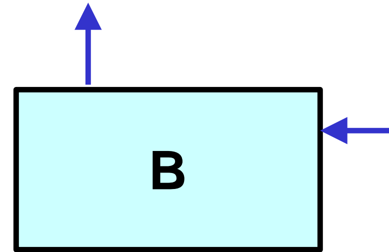
Abstraction & Hierarchy

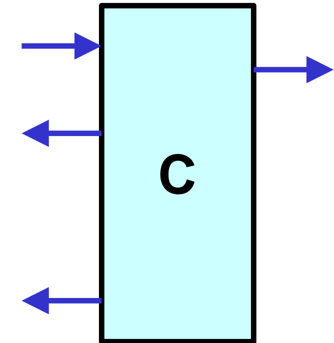
- In order to create detailed model of a complex system, we need to use abstraction & hierarchy
- Behavioral modeling provides abstraction
  - so far all our models have been described as one entity
- Structural modeling supports hierarchy & re-use
- Structural modeling describes physical connection between subsystems whose behavior and/or structure has already been defined
- Structural modeling supports designer directed partitioning of a system
  - important in synthesis
- Structural modeling facilitates sharing and re-use of designs

Building a Structural Hierarchy

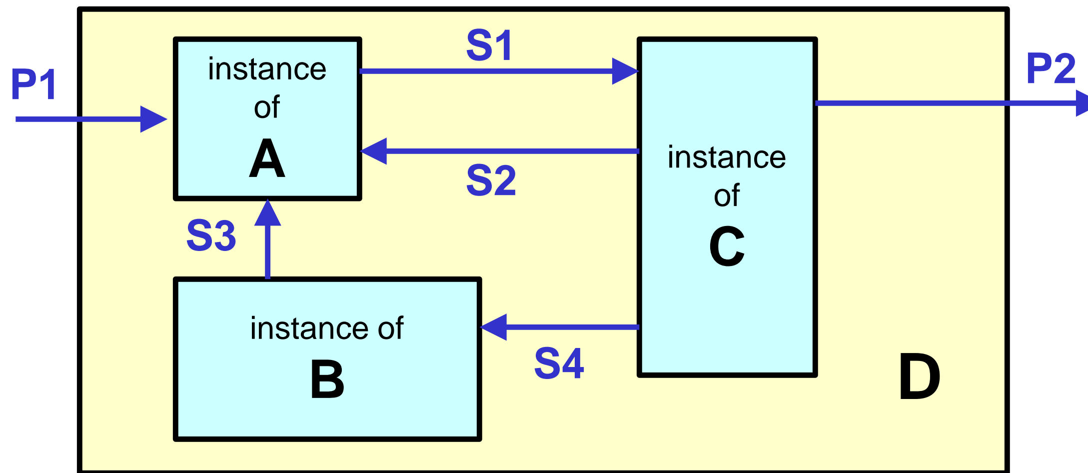
- Design a set of components

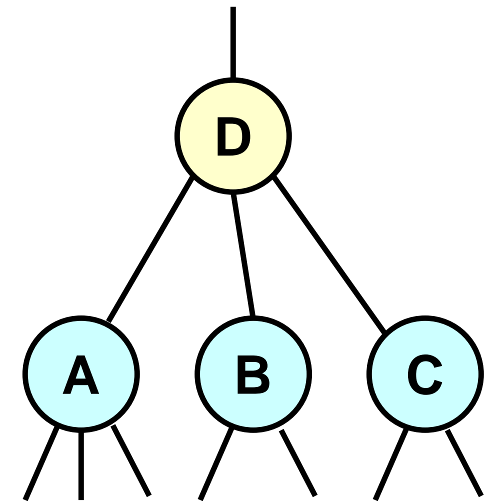






- Instantiate these components in a new (higher level) component

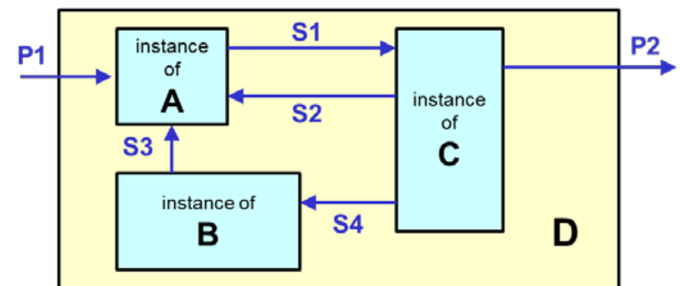




- Connect components together with signals

Modeling a Structural Hierarchy

```
entity D is
port(P1:in bit;
     P2:out bit);
end entity D;

architecture structural of D is
component A is
port(a1,a2,a3:in bit;
     a4:out bit);
end component A;

component B is
port(b1:in bit;
     b2:out bit);
end component B;

component C is
port(c1:in bit;
     c2,c3,c4:out bit);
end component C;

signal s1,s2,s3,s4: bit;

begin
Inst1: A port map (a1=>P1,
          a2=>S2, a3=>S3,
          a4=>S1);
Inst2: B port map (b1=>S4,
          b2=>S3);
Inst3: C port map (c1=>S1,
          c2=>S2, c3=>S4,
          c4=>P2);

end architecture structural;
```



Elements of a Structural Model

1. Ensure you have a behavioral or structural description of each component in the system
   - i.e., you have a correct entity-architecture description of each component defined elsewhere (in this or another VHDL file or a package)

2. In architecture of system:

```
architecture arch_name of entity_name is
-- declare various components
-- declare signals that will interconnect instantiated components
begin
    -- instantiate one or more instances of each component using
    -- port map to connect component ports to system ports & signals
end architecture arch_name;
```

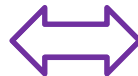
Component Declaration

- A component declaration declares the name and the interface of a component.
- It appears in the declarations part of an architecture part, or in a package declaration.

```
component component-name [is]
      [port (list-of-interface-ports);]
end component [component-name];
```

- component name and port names & types must match those in original entity description

```
component flipflop
      port(D : IN std_logic;
      clk : IN std_logic;
      Q ,Qb: OUT std_logic);
end component;
```



```
entity flipflop is
      port(D : IN std_logic;
      clk : IN std_logic;
      Q ,Qb: OUT std_logic);
end entity;
```

Component Instantiation

- Defines and labels a specific instance of a declared subcomponent.
- Associates the ports of the entity & the signals of the architecture with the ports of the subcomponent.

Component-label : component-name [**port map** (association-list)];

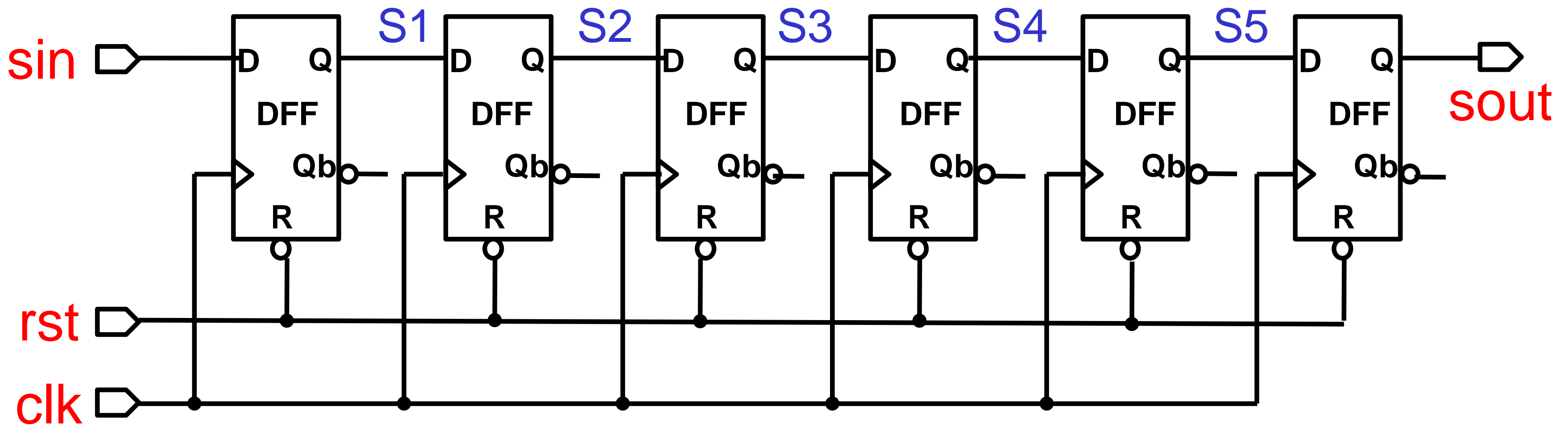
- Association-list associates signals in the entity, called actuals, with the ports of a component, called formals.

(*formal*1=>*actual1*, *formal2*=>*actual2*,…) --etc.

- An actual may be the keyword open to indicate a port that is not connected.

FF1: flipflop **port map** (clk=>ckin, D=>d3, Q=>dout, Qb=>**open**);

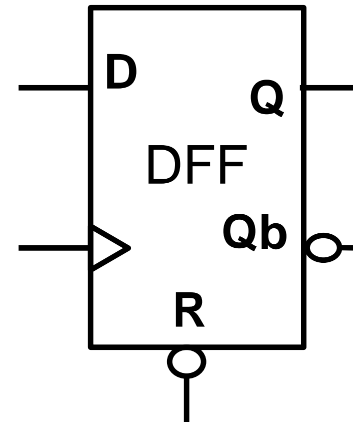
Example: 6-element shift register



SR6

6-bit SR: DFF component model

```
entity DFF is
  port (
    R, ck, D : in  std_logic;
    Q, Qb : out std_logic);
end DFF;

architecture behave of DFF is
begin
dfp: process (R, ck)
  begin
    if (R = '0') then
      Q <= '0';
      Qb<=‘1’;
    elsif (ck'event and ck = '1') then
      Q <= D;
      Qb<= not D;
    end if;
  end process;
end behave;
```

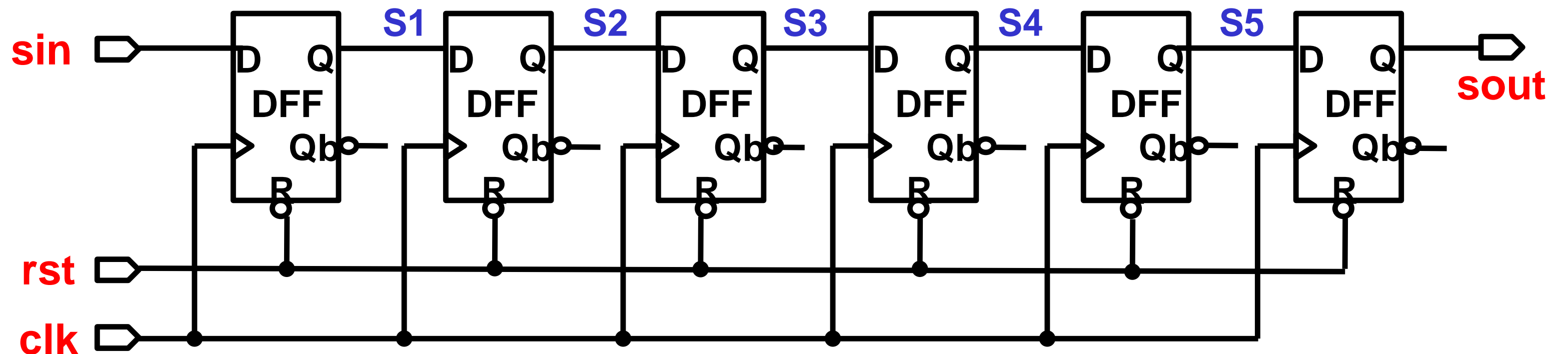


6-bit SR: SR6 declarations

```
entity SR6 is
   port (
     rst, clk, si: in  std_logic;
     so: out std_logic);
end SR6;

architecture RTL of SR6 is
   component DFF
   port (
      R, ck, D : in  std_logic;
      Q, Qb:out std_logic);
   end component;

signal s1,s2,s3,s4,s5:  std_logic;

begin
```

6-bit SR: SR6 declarations

```
bit0 : DFF port map (R => rst, ck => clk, D=>sin, Q=>S1, Qb=>open);
bit1 : DFF port map (R => rst, ck => clk, D=>S1, Q=>S2, Qb=>open);
bit2 : DFF port map (R => rst, ck => clk, D=>S2, Q=>S3, Qb=>open);
bit3 : DFF port map (R => rst, ck => clk, D=>S3, Q=>S4, Qb=>open);
bit4 : DFF port map (R => rst, ck => clk, D=>S4, Q=>S5, Qb=>open);
bit5 : DFF port map (R => rst, ck => clk, D=>S5, Q=>sout, Qb=>open);

end architecture RTL;
```



Named & Positional Association

- In previous example, we used named association
  - allows associations to be made in any order

(*formal*1=>*actual1*, *formal2*=>*actual2*,…) --etc.

- Positional association only names actuals in the same order as the formals were listed in the component declaration
  - like order based subroutine parameter passing in conventional programming languages

(*actual1*, *actual2*, *actual3*…) --etc.

  - less verbose, but more prone to error

Named & Positional Association

- Component declaration:

```
component DFF port (
R, ck, D : in  std_logic;
Q, Qb:out std_logic);
end component;
```

- Component instantiation – named association

```
bit0: DFF port map (ck=>clk, R=>rst, D=>sin, Q=>S1, Qb=>open);
```

- Component instantiation – positional association

```
bit0: DFF port map (rst, clk, sin, S1, open);
```

Signals used in Port Map

- Actual signals used in port map can be:
  - scalars
  - vectors
  - array elements
  - literals

```
INST6B : CHILD6 port map (
        INP (2 downto 0) => QB (4 downto 2),
        INP (3) => reset,
        INP (4) => QB(0),
        outp => SET,
        enable=>'1',
        A(0 to 5)=>  O"35";
```

Sidebar: Array Attributes

- In addition to signal attributes, there are array attributes that return index range information

array'**left**	returns the left bound of the array index
array'**right**	returns the right bound of the array index
array'**high**	returns the upper bound of the array index
array'**low**	returns the lower bound of the array index
array'**length**	returns the number of elements in the array
array'**range**	returns the index range of the array
array'**ascending**	returns TRUE if array uses ascending index
array'**descending**	returns TRUE if array uses descending index

Examples of Array Attributes

signal abc: std_logic_vector (7 **downto** 2);

abc'left	returns 7
abc'right	returns 2
abc'high	returns 7
abc'low	returns 2
abc'length	returns 6
abc'range	returns 7 downto 2
abc'ascending	returns FALSE
abc'descending	returns TRUE

Unconstrained Ports

- A component entity may be defined with an input or output port that is an array type with no specification of index constraints:

```
entity mbit_and is
   Port (din:in  std_logic_vector;
         z:out  std_logic);
end mbit_and;
```

```
architecture mba of mbit_and is
begin
p1:process(din)
      variable av: std_logic;
      begin
            av:='1';
            for i in din'low to din'high loop
                        if din(i)='0' then av:='0';
                        end if;
            end loop;
            z<=av;
      end process;
end mba;
```

Constraint Passing

- The index bounds are not determined (constrained) until the component is instantiated:

```
component  mbit_and
   port( din : in  std_logic_vector;
       z : out std_logic);
end component;
…
signal pbus : std_logic_vector(3 downto 0);
signal bb : std_logic;
 …
begin
and1: mbit_and port map (din => pbus, z => bb);
```

Example: 4x4 Unsigned Multiply

- You are provided with two basic components: a 1-bit full adder and a 2-input and gate. Build a structural model of a 4x4 unsigned multiplier

```
entity fadd is
   Port (a,b,cin: in std_logic;
            sum,cout: out std_logic);
end fadd;

architecture gate of fadd is
begin
   sum <= a xor b xor cin after 5 ns;
   cout <= (a and b) or (a and cin)
                or (b and cin) after 5 ns;
end gate;
```

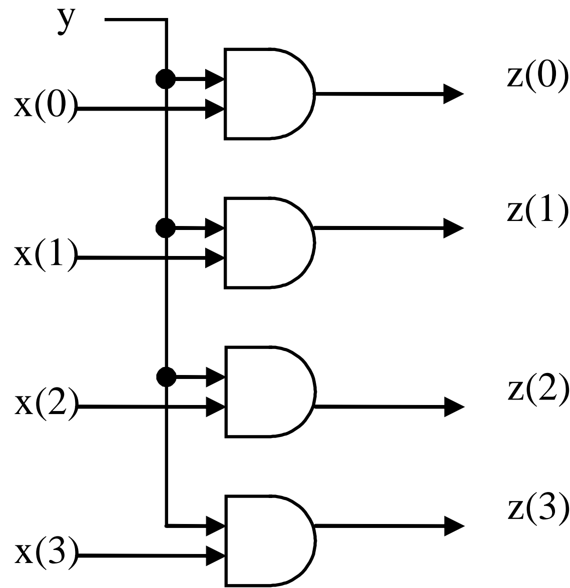
```
entity and2 is
   Port (a,b: in std_logic;
         c: out std_logic);
end and2;

architecture gate of and2 is
begin
   c<= a and b after 3 ns;
end gatel;
```

Unsigned Multiply Operations

Multiplicand	>					**X3**	**X2**	**X1**	**X0**
Multiplier	>				x	**Y3**	**Y2**	**Y1**	**Y0**
1st partial product	>					Y0X3	Y0X2	Y0X1	Y0X0
2nd partial product	>				Y1X3	Y1X2	Y1X1	Y1X0	
3rd partial product	>			Y2X3	Y2X2	Y2X1	Y2X0		
4th partial product	>	+	Y3X3	Y3X2	Y3X1	Y3X0			
Final product	>	**P7**	**P6**	**P5**	**P4**	**P3**	**P2**	**P1**	**P0**

- Components operations are:
  - 4x1-bit multiplies
  - 4-bit additions

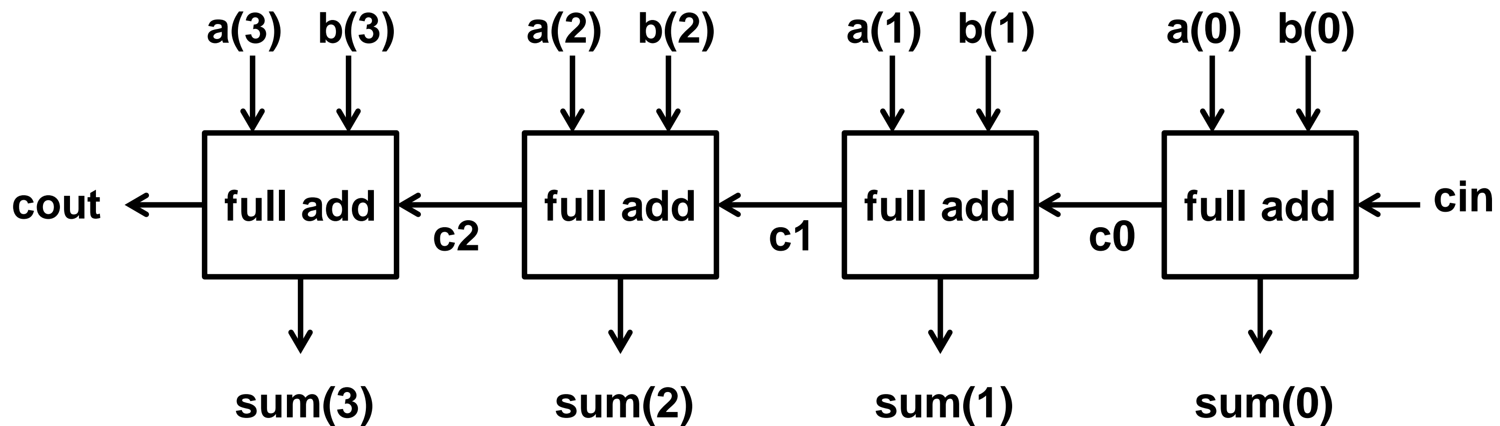
4x1 Unsigned Multiply




```
entity mpy4x1 is
   Port(x: in std_logic_vector(3 downto 0);
        y: in std_logic;
        z: out std_logic_vector(3 downto 0));
end mpy4x1;
```

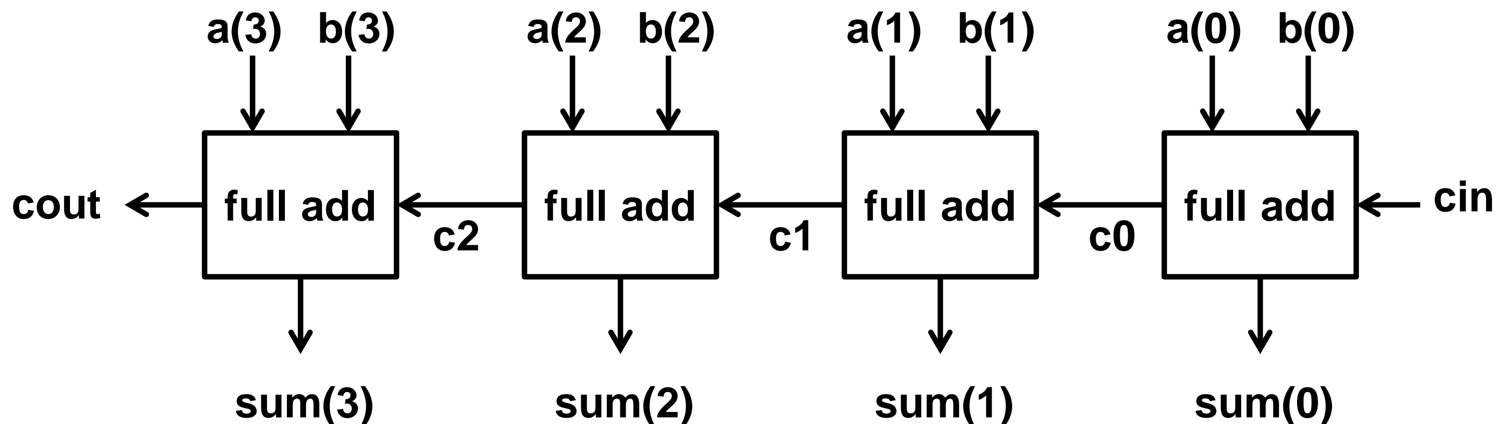
```
architecture struct of mpy4x1 is
    component and2
    port(a,b: in std_logic;
         c: out std_logic);
    end component;

begin
    bit3: and2 port map(a=>x(3),
        b=>y, c=>z(3));
    bit2: and2 port map(a=>x(2),
        b=>y, c=>z(2));
    bit1: and2 port map(a=>x(1),
        b=>y, c=>z(1));
    bit0: and2 port map(a=>x(0),
        b=>y, c=>z(0));
end struct;
```

4-bit adder




```
entity add4 is
   Port( a, b: in std_logic_vector(3 downto 0);
      cin: in std_logic;
      sum: out std_logic_vector(3 downto 0);
      cout: out std_logic);
end add4;
```

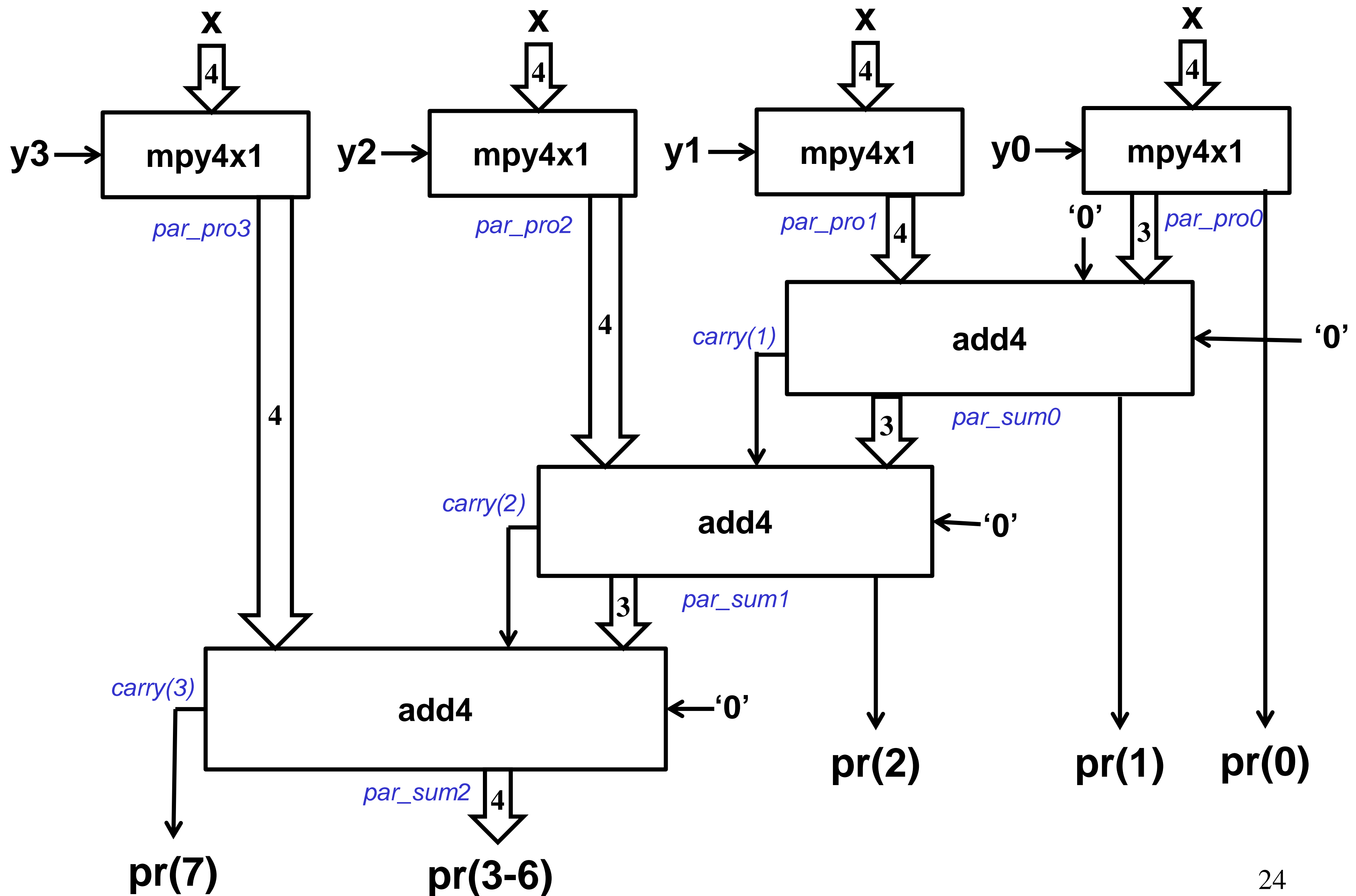
4-bit adder (cont.)




```
architecture struct of add4 is
   signal c: std_logic_vector(2 downto 0);
   component fadd
   port(  a, b, cin : in std_logic;
          sum, cout :out std_logic);
   end component;
```

```
begin
x0: fadd port map(a=>a(0), b=>b(0),
   cin=>cin, sum=>sum(0),cout=>c(0));
x1: fadd port map(a=>a(1), b=>b(1),
   cin=>c(0), sum=>sum(1),cout=>c(1));
x2: fadd port map(a=>a(2), b=>b(2),
   cin=>c(1), sum=>sum(2),cout=>c(2));
x3: fadd port map(a=>a(3), b=>b(3),
   cin=>c(2), sum=>sum(3),cout=>cout);
end struct;
```

4x4 multiplier – *putting it all together*



4x4 multiplier – entity

```
entity mult4x4 is
    Port ( x, y : in std_logic_vector(3 downto 0);
               pr : out std_logic_vector(7 downto 0));
end mult4x4;
```

4x4 multiplier – architecture declarations

```
architecture struct of mult4x4 is
signal par_pr0, par_pr1, par_pr2, par_pr3: std_logic_vector( 3 downto 0);
signal par_sum1, par_sum2, par_sum3: std_logic_vector( 3 downto 0);
signal carry: std_logic_vector(3 downto 1);

component mpy4x1 is
    port(x : in std_logic_vector(3 downto 0);
         y : in std_logic;
         z : out std_logic_vector(3 downto 0));
end component;
component add4 is
    port( a,b : in std_logic_vector(3 downto 0);
         cin : in std_logic;
         sum :out std_logic_vector(3 downto 0);
         cout : out std_logic);
end component;
```

4x4 multiplier – architecture instantiations

```
begin

    mpy0: mpy4x1 port map(x => x, y => y(0), z  => par_pr0);
    mpy1: mpy4x1 port map(x => x, y => y(1), z  => par_pr1);
    mpy2: mpy4x1 port map(x => x, y => y(2), z  => par_pr2);
    mpy3: mpy4x1 port map(x => x, y => y(3), z  => par_pr3);

    subadd1:  add4 port map(a => ‘0’ & par_pr0(3 downto 1), b => par_pr1,
        cin => '0',  sum => par_sum1, cout => carry(1) );
    subadd2:  add4 port map(a => carry(1) & par_sum1(3 downto 1), b => par_pr2,
        cin => '0',  sum => par_sum2, cout => carry(2) );
    subadd3:  add4 port map(a => carry(2) & par_sum2(3 downto 1), b => par_pr3,
        cin => '0',  sum => par_sum3, cout => carry(3) );

    z <= carry(3) & par_sum3 & par_sum2(0) & par_sum1(0) & par_pr0(0);

end struct;
```

Exploiting Regularity

- Models so far have captured hierarchy but not regularity
- Instantiating many copies of same component is tedious
  - Imagine a 64x64 bit multiplier!
- Take for example, 4-bit adder:

```
architecture struct of add4 is
  signal c: std_logic_vector(2 downto 0);
  component fadd
  port(  a, b, cin : in std_logic;
         sum, cout :out std_logic);
  end component;
begin
x0: fadd port map(a=>a(0), b=>b(0),
  cin=>cin, sum=>sum(0),cout=>c(0));
x1: fadd port map(a=>a(1), b=>b(1),
  cin=>c(0), sum=>sum(1),cout=>c(1));
x2: fadd port map(a=>a(2), b=>b(2),
  cin=>c(1), sum=>sum(2),cout=>c(2));
x3: fadd port map(a=>a(3), b=>b(3),
  cin=>c(2), sum=>sum(3),cout=>cout);
end struct;
```

Generate Statement

- Allows us to generate an “array” of instantiations:

```
architecture struct of add4 is
   -- signal c: std_logic_vector(2 downto 0);
   component fadd
   port(  a, b, cin : in std_logic;
          sum, cout :out std_logic);
   end component;
   signal carry: std_logic_vector(4 downto 0);

begin
   carry(0)<=cin;
   GADD: for k in 3 downto 0 generate
      FA: fadd port map(a=>a(k), b=>b(k), cin=>carry(k),
          sum=>sum(k),cout=>carry(k+1));
   end generate GADD;
   cout<=carry(4);
end struct;
```