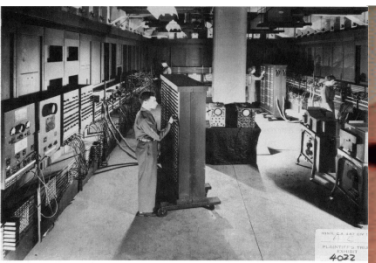
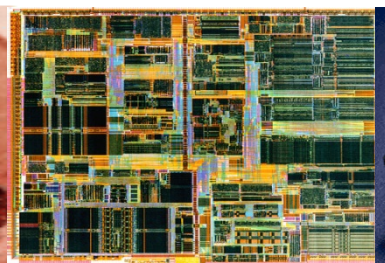
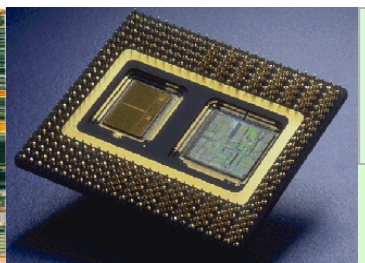
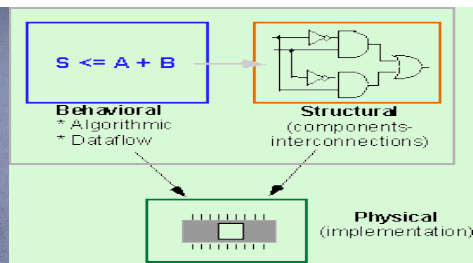
CpE 690: Introduction to VLSI Design

Lecture 3
FPGA Design and VHDL – Part II

Bryan Ackland
Department of Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ 07030



Behavioral Modeling

Higher Levels of Abstraction

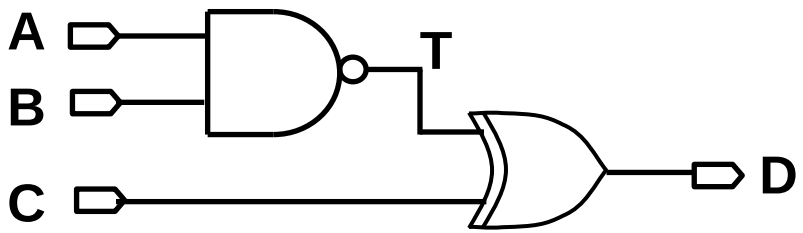
- Conditional CSA's allow us to move from the gate level to the register transfer level (registers, multiplexers, adders etc.)
- Impractical for higher levels of abstraction where we want to focus on high level function rather than structural implementation.
- For example: 32-bit microprocessor core
  - or even a 9-stage, 16 bit FIR filter ??

Process Construct

- Sequentially executed block of code
  - much like conventional programming languages
  - allows for complex computation of results
  - executes in zero time
- Supports variables as well as signals
- Powerful control flow constructs
- More control over when assignments are executed

Process Example

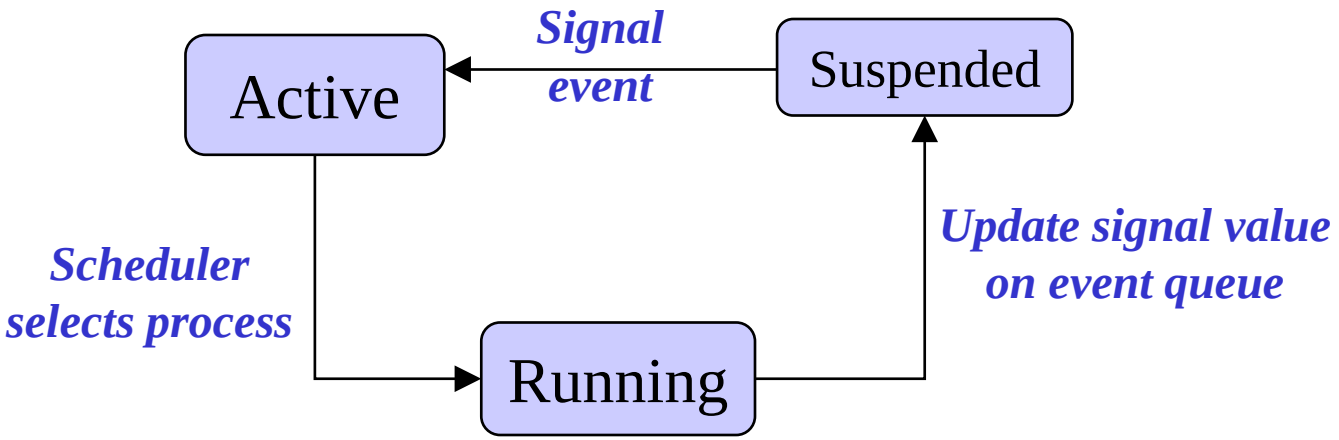
```
entity NANDXOR is
   port (
      A, B  : in  std_logic;
      C     : in  std_logic;
      D     : out std_logic);
end NANDXOR;
architecture RTL of NANDXOR is
   signal T : std_logic;
begin
   p0 : T <= A nand B after 2 ns;
   p1 : process (T, C)
   begin
      D <= T xor C after 3 ns;
   end process p1;
end RTL;
```



Sensitivity List

p1 : **process** (T, C)

- (T,C) is the process sensitivity list of process p1
- A process is executed whenever an event occurs on any signal in the sensitivity list
- Statements in the process are executed sequentially
- Process is then suspended until an event occurs on one of signals in process sensitivity list



Concurrent Signal Assignment or Process?

```
p1 : process (T, C)
  begin
    D <= T xor C after 3 ns;
  end process p1;
```

or

```
p1 : D <= T xor C after 3ns;
```

- These two representations are equivalent!
- CSA's are implemented as processes
- A CSA is a short-hand method of defining a process that schedules events on only one output signal
- Each process can be thought of as a concurrent assignment that can:
  - use complex sequential code to calculate a result
  - schedule events on more than one signal

Process Programming – If then Else

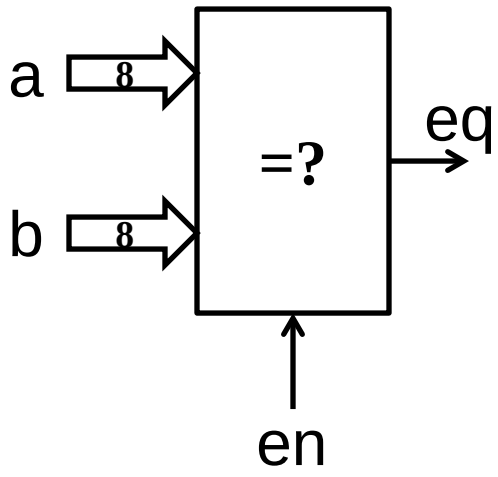
- An if statement selects a sequence of statements for execution based on the value of a condition (Boolean value).

if boolean-expression **then**
sequential-statements
{**elsif** boolean-expression **then**
sequential-statements}
[**else**
sequential-statement]
end if;

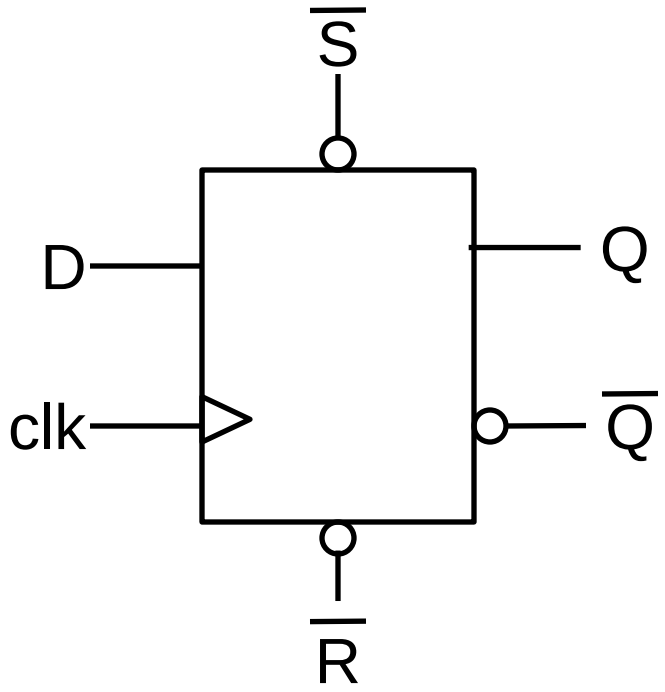
- First clause whose expression is true is executed
  - order of clauses matters
- Note that **elsif** is one word but **end if** is two words

Example 8-bit comparator

```
entity cmp_8 is
   port (
      a,b: in  std_logic_vector (7 downto 0);
      en: in std_logic;
      eq: out std_logic);
end cmp_8;
architecture behavior of cmp_8 is
begin
      cmp_proc : process (a,b,en)
      begin
            if en='0' then
               eq <= '0' after 4 ns;
            elsif a=b then
               eq <= '1' after 7 ns;
            else
               eq <= '0' after 7 ns;
            end if;
   end process cmp_proc;
end behavior;
```



D Flip-Flop



$\overline{S}$	$\overline{R}$	clk	D	Q	$\overline{Q}$
0	1	X	X	1	0
1	0	X	X	0	1
1	1	↑	1	1	0
1	1	↑	0	0	1
0	0	X	X	?	?

- Rising edge triggered sequential circuit
- D flip-flop captures value of D when clk goes from ‘0’ to ‘1’
- S and R are asynchronous over-riding set and reset
- In order to model, we need to know on which input an event has occurred

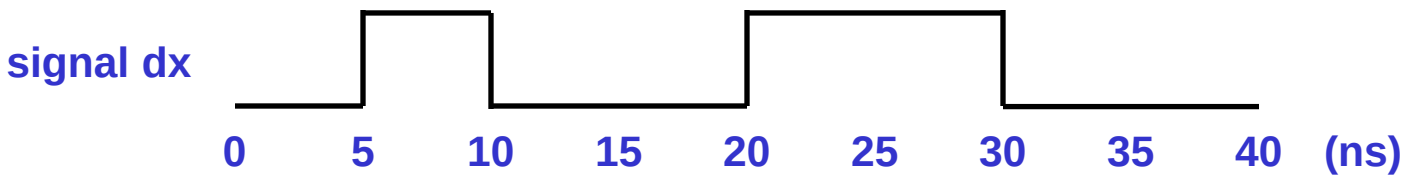
Sidebar: Attributes

- Attributes return information about a signal, e.g.:

Attribute	**Function**
signal_name**'event**	returns the Boolean value True if an event on the signal occurred at current time, otherwise returns False
signal_name**'active**	returns the Boolean value True there has been a transaction (assignment) on the signal at the current time, otherwise returns a False
signal_name**'transaction**	returns a signal of the type "bit" that toggles (0 to 1 or 1 to 0) every time there is a transaction on the signal.
signal_name**'last_event**	returns the time elapsed since the last event on the signal
signal_name**'last_active**	returns the time elapsed since the last transaction on the signal
signal_name**'last_value**	returns the value of the signal before the last event occurred on the signal
signal_name**'delayed(T)**	returns a signal that is the delayed version (by time T) of the original one. [T is optional, default Δ]
signal_name**'stable(T)**	returns a Boolean value, True, if no event has occurred on the signal during the interval T, otherwise returns a False. [T is optional, default Δ]
signal_name**'quiet(T)**	returns a Boolean value, True, if no transaction has occurred on the signal during the interval T, otherwise returns a False. [T is optional, default Δ]

Sidebar: Attribute Examples

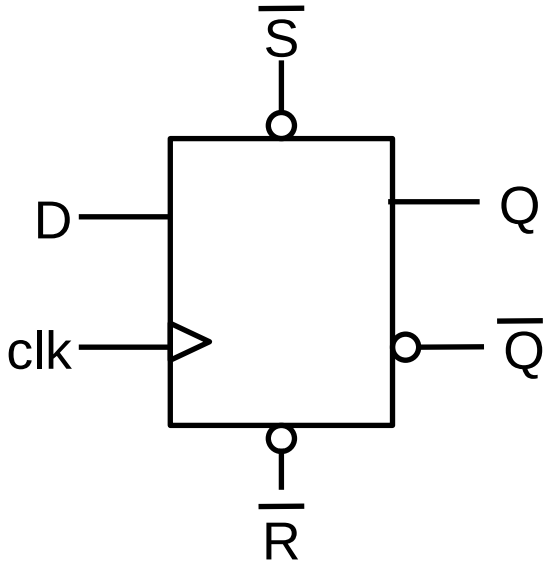
dx <= ‘0’ **after** 0ns, ‘1’ **after** 5ns, ‘0’ **after** 10ns, ‘0’ **after** 15ns, ‘1’ **after** 20ns, ‘0’ **after** 30ns;



dx’event	has the value TRUE at t=10ns
dx’event	has the value FALSE at t=15ns
dx’active	has the value TRUE at t=15ns
dx’last_event	has the value 5ns at t=15ns
dx’last_value	has the value ‘1’ at t=15ns
dx’delayed(8ns)	has the value ‘1’ at t=15ns
dx’stable(8ns)	has the value FALSE at t=15ns
dx’stable(2ns)	has the value TRU at t=15ns
dx’delayed(8ns)’event	has the value TRUE only at times 13, 18, 28, and 38ns.

Example: D Flip-Flop

```
entity Dff is
   port (
     clk,D,Rb,Sb: in  std_logic;
     Q,Qb: out std_logic);
end entity Dff;
```




```
architecture DA1 of Dff is
begin
     ff_proc: process (clk,Rb,Sb)
     begin
          if Rb='0' then
             Q<='0' after 5ns;
             Qb<='1' after 5ns;
          elsif Sb='0' then
             Q<='1' after 5ns;
             Qb<='0' after 5ns;
          elsif clk'event and clk='1' then
             Q<=D after 7 ns;
             Qb<= not D after 7 ns;
          end if;
       end process ff_proc;
end architecture DA1;
```

Process Programming: Case Statement

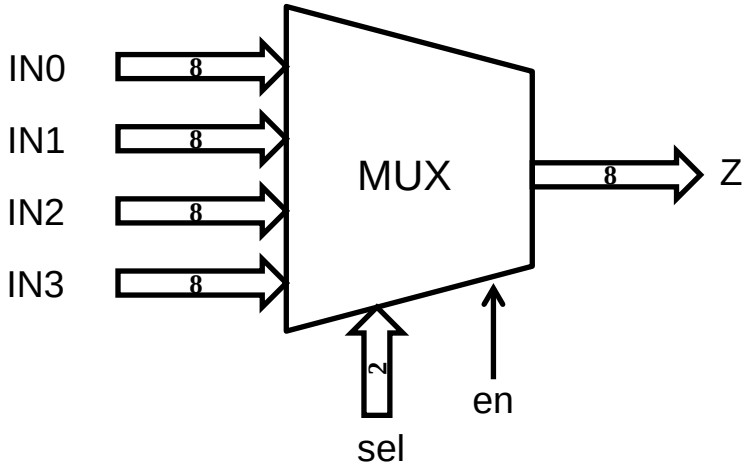
- A case statement selects one of several branches for execution based on the value of expression

case *expression* **is**
when *choices* => *sequential-statements*
when *choices* => *sequential-statements*
-- can have any number of branches
[**when others** => *sequential-statements*]
end case;

- The set of *choices* must be mutually exclusive and cover all possible values of the *expression*.
  - order of clauses does not matter

Example: 4-way 8-bit multiplexer

```
entity mux4 is
        port ( IN0, IN1, IN2, IN3: in std_logic_vector (7 downto 0);
        sel: in std_logic_vector (1 downto 0);
        en: in std_logic;
        Z: out std_logic_vector (7 downto 0));
end entity mux4;

architecture using_case of  mux4 is
begin
        P1: process (IN0,IN1,IN2,IN3,sel,en)
        begin
                if en='0' then
                        Z<= x"00" after 5ns;
                else
                        case sel is
                        when "00" =>    Z<= IN0 after 5ns;
                        when "01 "=>    Z<= IN1 after 5ns;
                        when "10" =>    Z<= IN2 after 5ns;
                        when "11" =>    Z<= IN3 after 5ns;
                        when others =>  Z<="XXXXXXXX" after 5ns;
                        end case;
                end if;
        end process;
end architecture using_case;
```



Sidebar: Bit String Literals

- Special forms of string literals that are used to represent binary, octal, or hexadecimal numeric data values. The numerical value is given in double quotes (") and the representation is specified by a character preceding the quoted value.
- The underscore character can be used for convenience and clarity - it does not change the represented value. For example:

  - Binary data: **B**"0110_1101_1111_0010”
  - Octal data: **O**“16_67_62".
  - Hexadecimal data: **X**“6DF2"
  - Binary data: “0010110111110010”

- Note that binary is assumed when no base specified, bit underscore cannot be used in this case.

Process Programming: Loop Statement

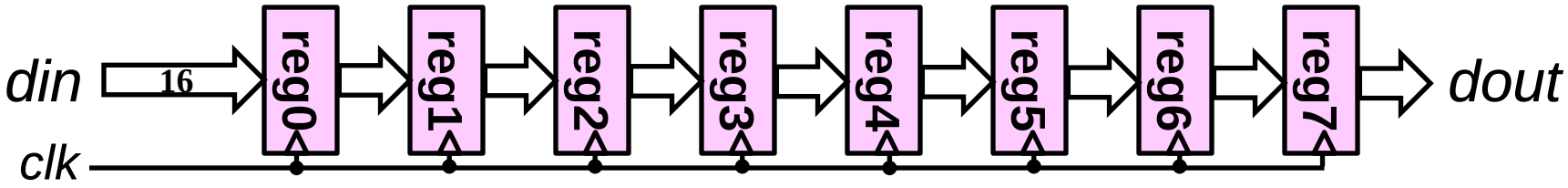
- The loop statement is used to iterate through a set of sequential statements.

[*loop-label* :] iteration-scheme **loop**
sequential-statements
end loop [*loop-label*];

Three types of iteration scheme:

1. **for** *identifier* **in** *range*
2. **while** *boolean-expression*
3. No iteration scheme specified

Example: 8-stage,16-bit register pipeline




```
entity pipe8 is
    port ( din:in std_logic_vector(15 downto 0);
    clk:in std_logic;
    dout:out std_logic_vector(15 downto 0));
end entity pipe8;

architecture pipe_be of  pipe8 is
type sig8x16 is array (0 to 7) of  std_logic_vector(15 downto 0);
signal regfile: sig8x16;
begin
    rproc: process (clk) is
    begin
        if clk='1' then
            regfile(0)<=din after 5ns;
            for i in 1 to 7 loop
                regfile(i)<=regfile(i-1) after  5ns;
            end loop;
        end if;
    end process;
    dout<=regfile(7);
end architecture pipe_be;
```

Loop Statement: For Iteration

[*loop-label* :] **for** *index* **in** *range* **loop**
sequential-statements
end loop [*loop-label*];

- *index* is implicitly declared in the loop statement
- *index* is local to the loop and read-only

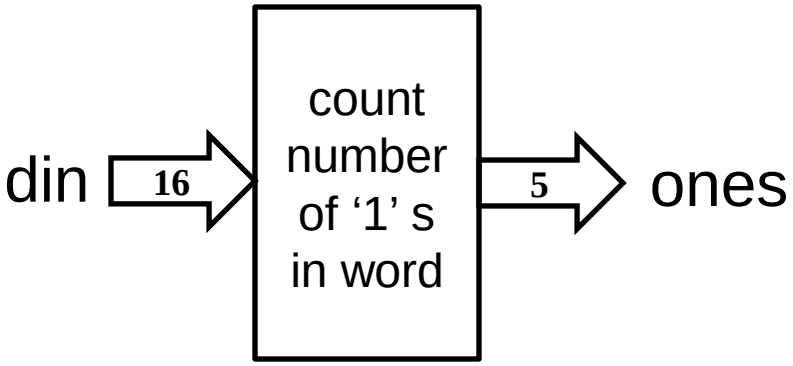
- Loop statement is most powerful when used with variables

Variables

- In addition to signals, VHDL supports variables
  - Variables in VHDL are similar to variables in conventional programming languages
- Like signals, each variable has a type
- Like signals, variables have a present value
- Unlike signals, variables have no concept of future time
  - simpler to implement in simulator
  - no events associated with variables
- Variables are defined within a process and are not visible outside of the process
- Signals represent physical interconnect in circuits
- Variables are local values used to simplify process of calculating a result

Example: Count number of “ones”

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

entity count1s is
   port (
     din: in  std_logic_vector (15 downto 0);
      ones: out std_logic_vector(4 downto 0));
end count1s;
```




```
architecture A1 of count1s is
begin
      p1: process (din)
      variable count: integer;
      begin
            count:=0;
            for i in 0 to 15 loop
               if din(i)=‘1’ then
                     count:= count+1;
               end if;
            end loop;
       ones<=conv_std_logic_vector(count,5)
                     after 5ns;
   end process;
end A1;
```

Variable Assignment Statement

Variable-object := *expression;*

- Expression may include both variables and signals. The present value of a signal is used in the computation
- Computation is performed in zero time (no delta delay)
- Can only occur within process

Review Architecture & Process

```
architecture RTL of OVERALL is
-- signals and constants can be declared here
-- variables CANNOT be declared here
begin
    -- concurrent signal assignment statements here
    -- NO variable assignment statements
    P1: process (SENSITIVITY_LIST)
    -- variables and constants can be declared here
    -- signals CANNOT be declared here
    begin
        -- sequential variable assignment statements here
        -- sequential signal assignment statements here
    end process P1;
end architecture RTL;
```

Wait Statement

- When process has sensitivity list, process is suspended until there is an event on one of the sensitive signals
- Alternatively, process can be suspended with use of wait statements:

wait for *time expression*;

wait for 25ns;

wait on *signal*;

wait on clk, reset;

wait until *condition*;

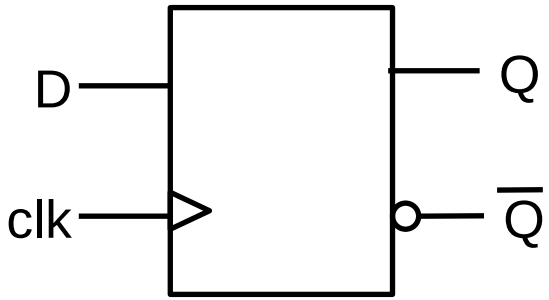
wait until index=0;

wait; -- means wait forever;

- When using wait statements, the process does not suspend at the last statement in the process code, but continues executing from the top of the process.

Example: D Flip-Flop

```
entity Dff 2 is
   port (
     clk,D: in  std_logic;
     Q,Qb: out std_logic);
end entity Dff2;
```




```
architecture  DB of Dff2 is
begin
      ff2_pr: process
      begin
            wait until clk'event and clk='1' ;
            Q<=D after 7 ns;
            Qb<= not D after 7 ns;
      end process ff2_pr;
end architecture DB;
```

Wait Statement

- Possible to use multiple conditions, e.g:

  wait on X,Y **until** Z=0 **for** 100ns;

  means: wait for a maximum of 100ns for an event on X or Y when Z=0

- tpr: **process** (a,b)
  begin
  …..
  end process;

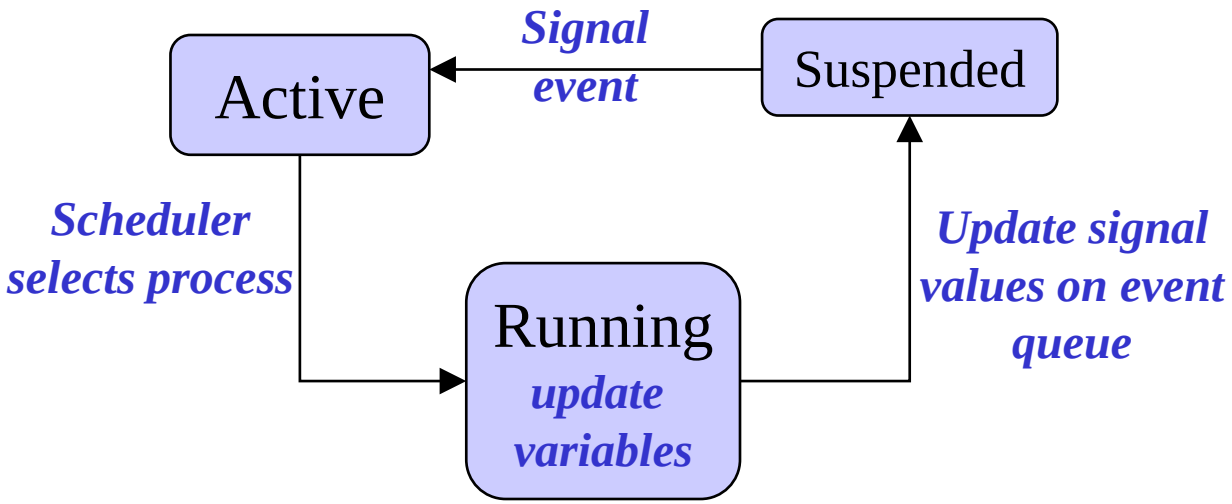
  is the same as

  tpr: **process**
  begin
  …..
  wait on a,b**;**
  end process;

- A process ***must have*** (a sensitivity list) ***or*** (one or more wait statements) ***but not*** both

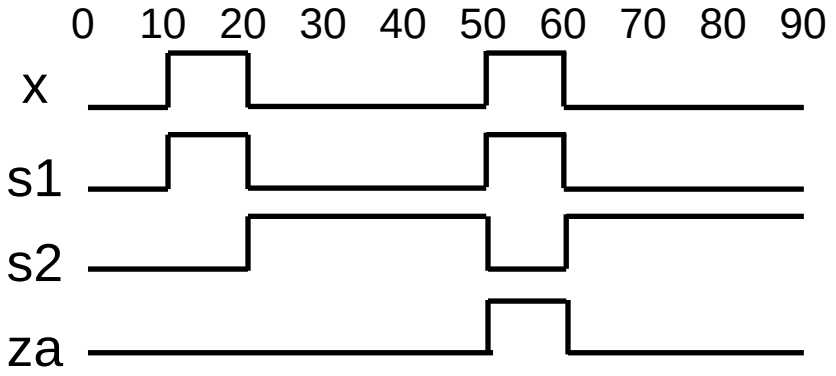
Timing of Variable and Signal Assignments

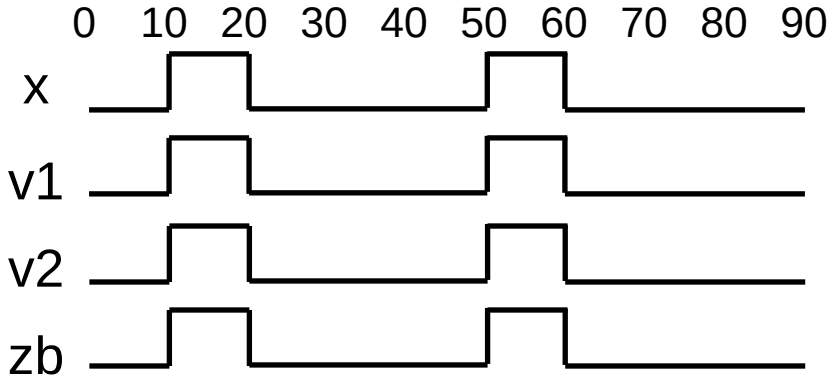
- Variables are assigned at the same time that the variable assignment is executed (zero-time)
  - Order of sequential variable assignment statements is important!
- Signals are assigned when the process is suspended
  - Can specify inertial, transport or zero delay
  - Zero delay signal assignment occurs at present time + Δ
  - How about order of sequential signal assignment statements?



Variable and Signal Timing Example

```
architecture A1 of sig_var is
signal s1, s2, x, za, zb: std_logic;
begin
    x<='0','1' after 10ns, '0' after 20ns, '1' after 50ns, '0' after 60 ns;
    pa: process (x)
    begin
        s1<=x;
        s2<=s1;
        za<=s2;
    end pa;
    pb: process(x)
    variable v1,v2:std_logic;
    begin
        v1:=x;
        v2:=v1;
        zb<=v2;
    end pb;
end A1;
```

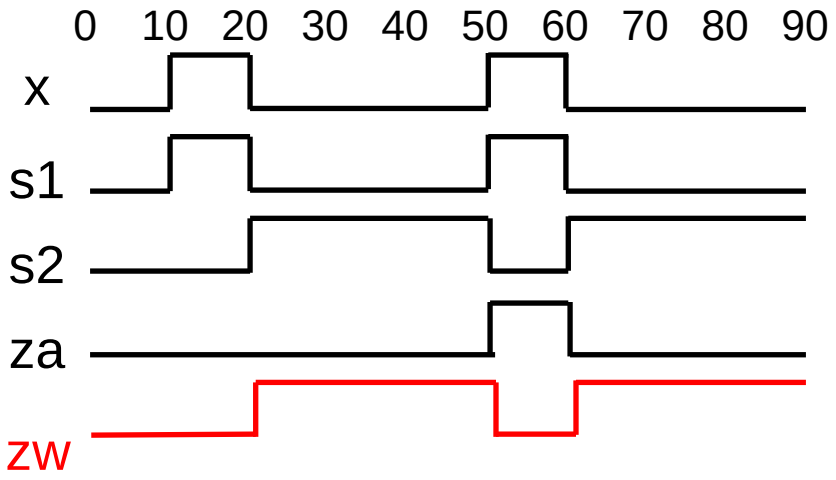




Wait for 0

- "Wait for 0" suspends process and then allows it to restart after a delay of only Δ
- Allows signal assignment to take effect before next statement is executed

```
architecture A1 of sig_var is
signal s1, s2, x, za, zb, zw: std_logic;
begin
    x<='0','1' after 10ns, '0' after 20ns, '1' after 50ns, '0' after 60 ns;
    pa: process
    begin
        wait on x;
        s1<=x;
        s2<=s1;
        za<=s2;
        wait for 0ns;
        zw<=s2;
    end pa;
end A1;
```



Loop Statement: While Iteration

[*loop-label* :] **while** *condition* **loop**
sequential-statements
end loop [*loop-label*];

- *condition* is expression using previously declared signals and/or variables
- These signals and variables can be modified within the loop

Other Useful Sequential Control Instructions

- **exit** [*loop label*] [**when** *condition*];
  - Exit from loop (like C-language *break*). Must be enclosed by a loop statement with the same loop label. If the loop label is not specified, the *exit* always applies to the innermost loop

- **next** [*loop label*] [**when** *condition*];
  - Skip remaining statements in current iteration of the loop (like C-language *continue*). If the loop label is not specified, the *next* always applies to the innermost loop

Loop Examples: Factorial Calculation

```
factorial := 1;
FLP:    for number in 2 to N loop
            factorial := factorial *number;
end loop;
```

```
j := 2;
factorial := 1;
WLP :   while j<=N loop
            factorial := factorial *j;
            j:= j +1;
end loop;
```

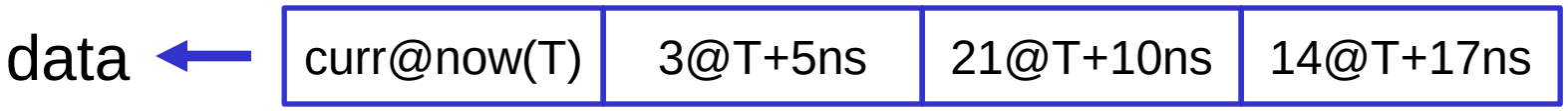
```
k := 1;
factorial :=1;
NLP :   loop
            factorial := factorial *k;
            k:= k +1;
            exit when k > N;
end loop;
```

These are all equivalent

Signal Drivers

- Outside of a process, multiple assignments to same signal are either illegal or invoke a resolution function
- Inside a process, multiple signal assignment behaves differently.
- A single driver is created for every signal that is assigned a value in a process
  - The driver holds its current value and all its future values
  - All transactions on a driver are ordered in increasing time

```
signal integer data;
p1: process
begin
    data<= 3 after 5ns, 21 after 10ns, 14 after 17ns;
```



Multiple Assignment: Transport Delays

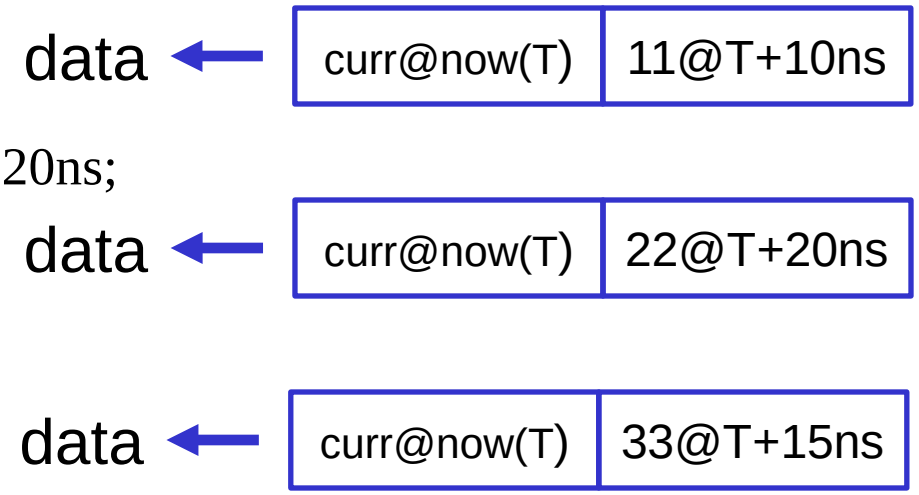
- Multiple Assignments update driver according to the order in which they are executed.
- Transport delay rules:
  1. All transactions that occur at or after the delay time of the first new transaction are deleted.
  2. All the new transactions are added at the end of the driver

```
data<= transport 11 after 10ns;
       …
       …
data<= transport 20 after 22ns;
       …
       …
data<= transport 35 after 18ns;
```

Multiple Assignment: Inertial Delays

- Inertial delay rules:
  1. All transactions that occur at or after the delay time of the first new transaction are deleted.
  2. Add all the new transactions to the driver
  3. Delete old transactions that occur within pulse rejection limit of first new transaction if value is different to value of first new transaction

```
data<= 11 after 10ns;
       …
       …
data<= reject 15ns inertial 22 after 20ns;
       …
       …
data<= 33 after 15ns;
```



Sidebar: Signed & Unsigned Vectors

- We frequently use multi-bit digital words to represent integer values on which we would like to perform arithmetic and relational operations
- The std_logic_vector type is simply an array of bits with no implied digital value
  - Only logical operators (nand, xor, not etc.) are defined in the *IEEE.std_logic_1164* library
  - No arithmetic (+, - etc.) or relational (>, <= etc.) because these would require understanding of meaning of vector
  - Does it represent signed, unsigned, signed-magnitude, floating etc. ?

Operations on Unsigned, Signed Numbers

- USE ieee.numeric_std.all
  and
  signals of the type UNSIGNED, SIGNED
  and conversion functions:
  std_logic_vector(), unsigned(), signed()

OR

- USE ieee.std_logic_unsigned.all
  and
  signals of the type STD_LOGIC_VECTOR
  - ***all*** STD_LOGIC_VECTOR objects will be treated as unsigned
  - approach used in Yalamanchili

- There is also an ieee.std_logic_signed.all
  - ***all*** STD_LOGIC_VECTOR objects will be treated as signed
  - do not use both!

Unsigned Arithmetic Example

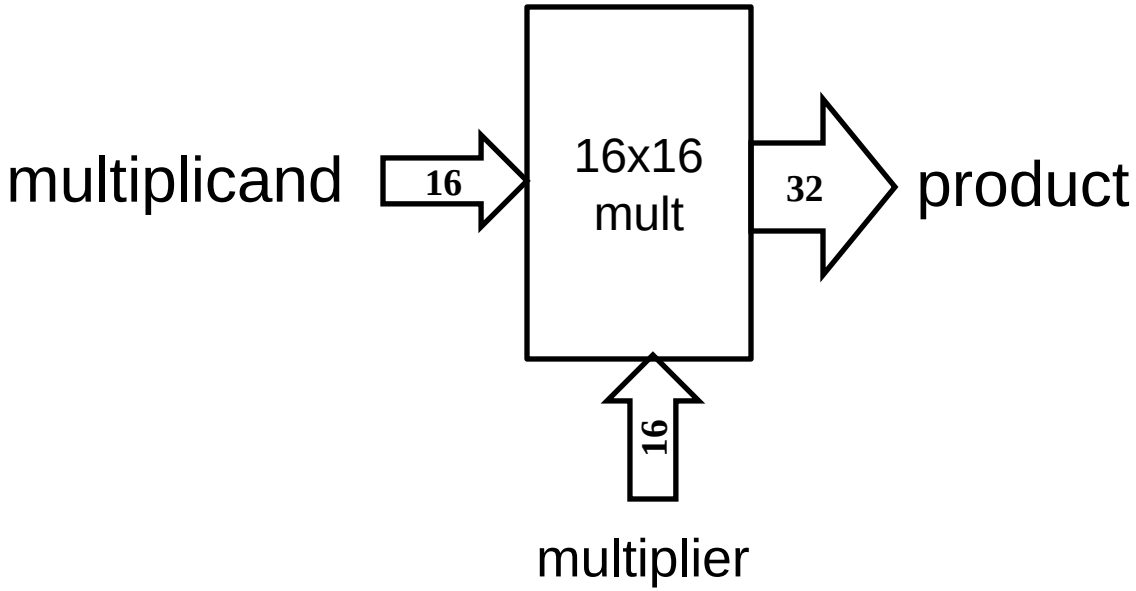
Suppose we want to add two 8-bit unsigned std_logic_vectors v1 and v2 to produce an 8-bit unsigned result v3 plus a carry-out

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

signal v1, v2, v3:
            std_logic_vector (7 downto 0);
signal  carry: std_logic;
signal u1, u2: unsigned (7 downto 0);
signal u3: unsigned (8 downto 0);

u1 <= unsigned (v1);
u2 <= unsigned (v2);
u3 <= ('0'&u1) + u2;
v3 <= std_logic_vector(u3(7 downto 0));
carry <= u3(8);
```

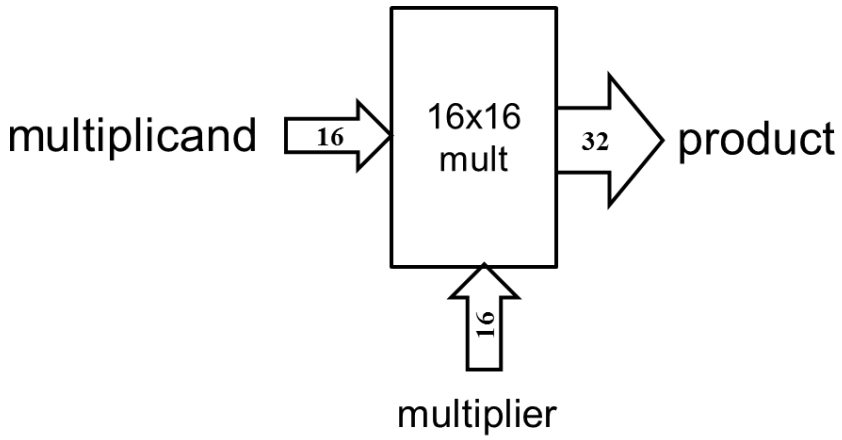
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

signal v1, v2, v3:
            std_logic_vector (7 downto 0);
signal vtemp:
            std_logic_vector (8 downto 0);
signal  carry: std_logic;

vtemp <= ('0'&v1) + v2;
v3 <= vtemp(7 downto 0));
carry <= vtemp(8);
```

Example: 16-bit unsigned multiplier

- Construct a “shift and add” behavioral model of a 16x16 bit unsigned multiplier using a process and variables



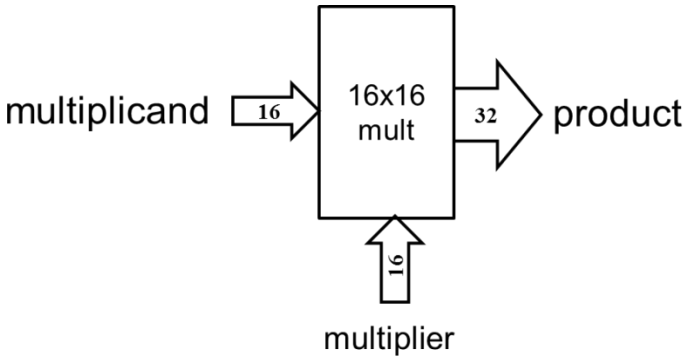
Example: 16-bit unsigned multiplier (cont.)




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

entity mult16 is
port( multiplicand, multiplier: in std_logic_vector(15 downto 0);
      product: out std_logic_vector (31 downto 0));
end entity mult16;
```

Example: 16-bit unsigned multiplier (cont.)

```
architecture behavioral of mult16 is
begin
mproc: process (multiplicand, multiplier)
    variable acc: std_logic_vector(32 downto 0);
    begin
        acc := '0' & x"00000000";
        for i in 0 to 15 loop
            if multiplier(i) = '1' then
                acc := acc + multiplicand & x"0000");
            end if;
            acc := '0' & acc(32 downto 1);
        end loop;
        product <= acc(31 downto 0) after 10ns;
    end process;
end architecture behavioral;
```



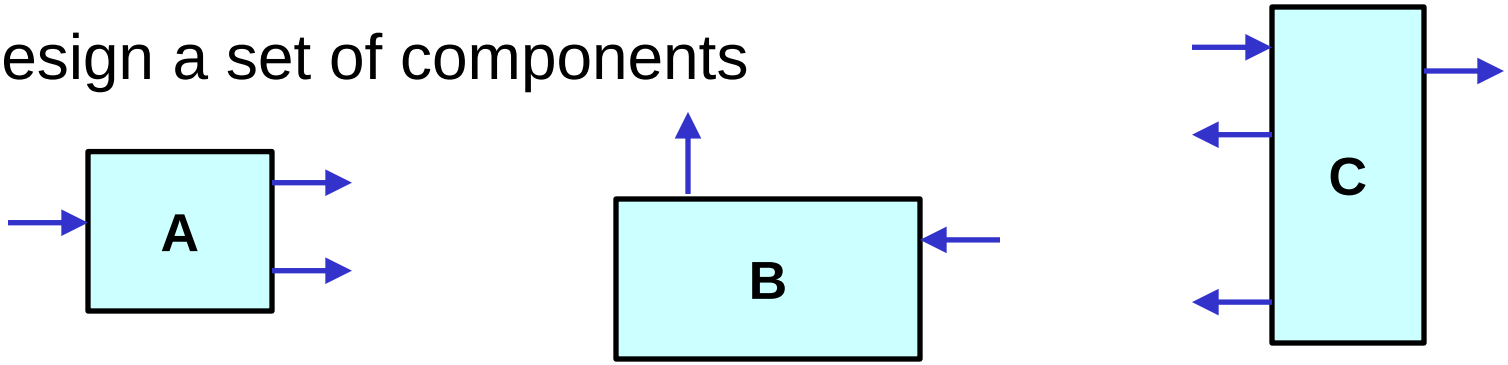
Structural Modeling

Abstraction & Hierarchy

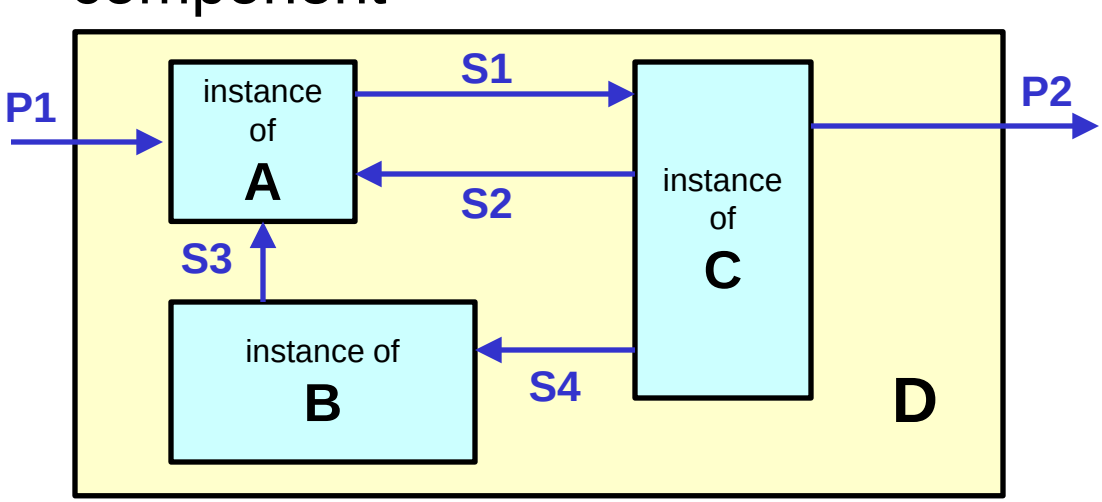
- In order to create detailed model of a complex system, we need to use abstraction & hierarchy
- Behavioral modeling provides abstraction
  - so far all our models have been described as one entity
- Structural modeling supports hierarchy
- Structural modeling describes physical connection between subsystems 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

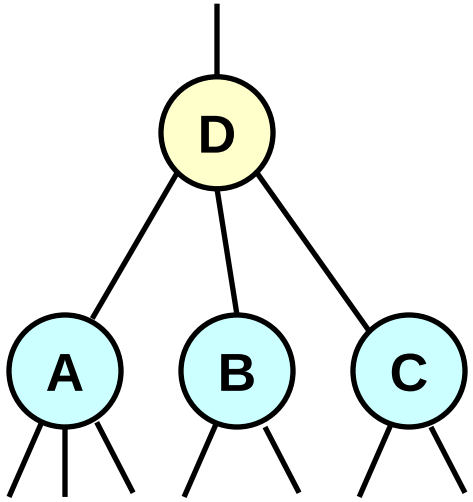
Building a Structural Hierarchy

- Design a set of components



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

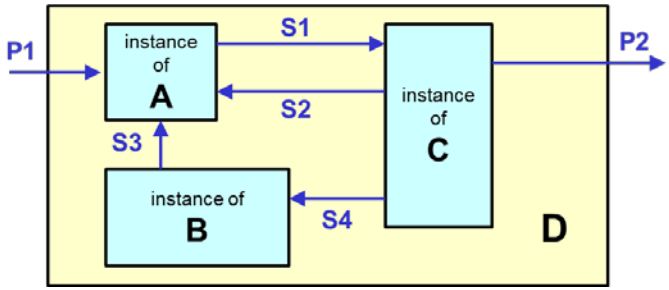




- Connect components together with signals

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
        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)];

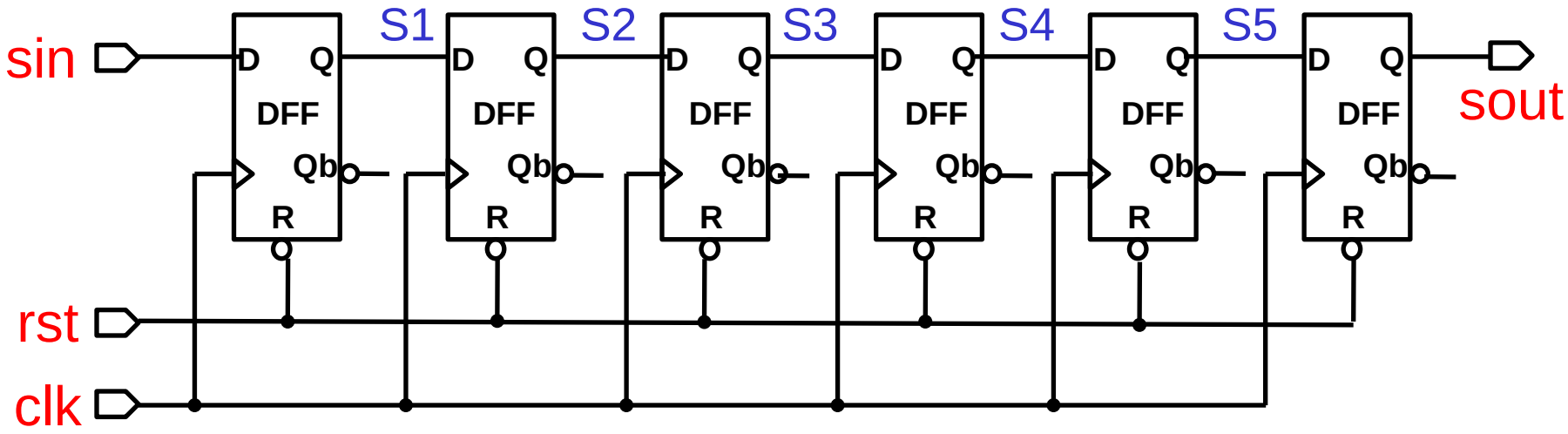
- Association-list associates signals in the entity, called actuals, with the ports of a component, called formals.

(*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**);

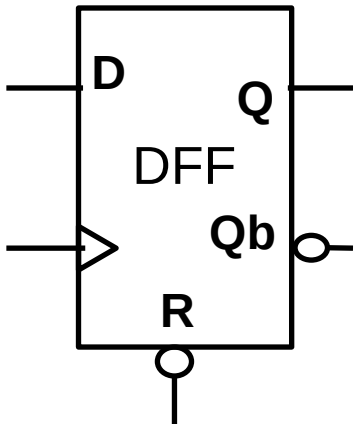
Example: 6-element shift register



SR6

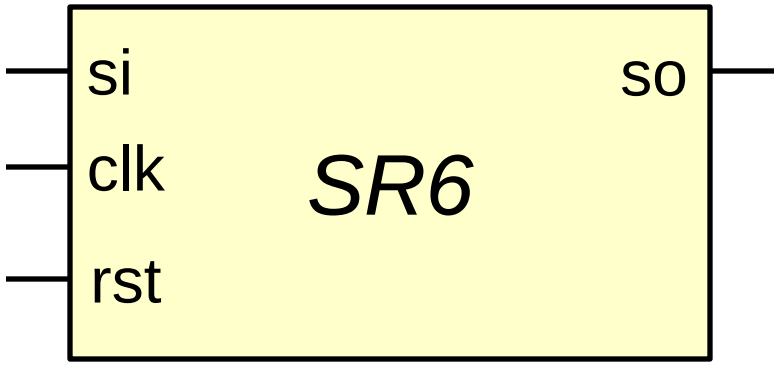
6-bit SR: DFF component model

```
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

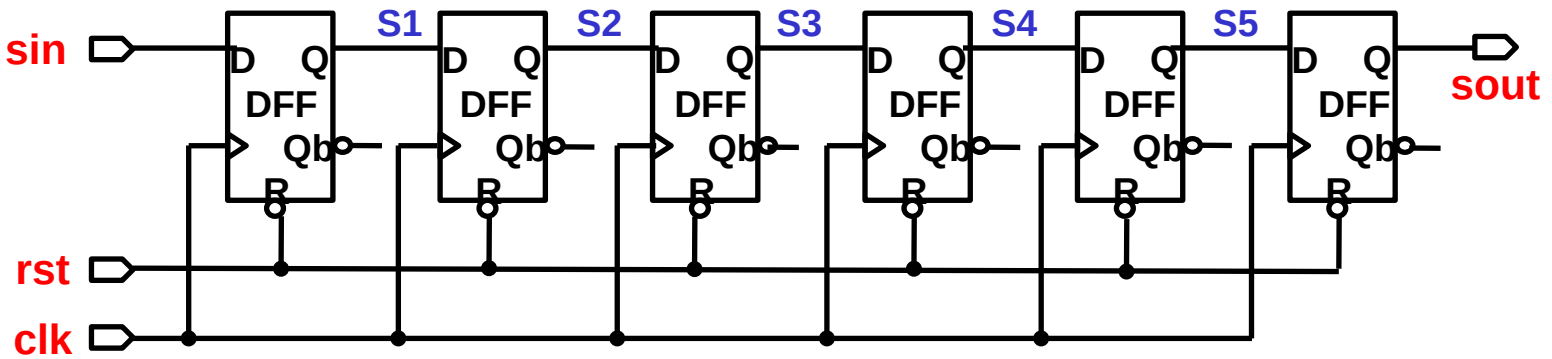
```
entity SR6 is
   port (
     rst, clk, si: in  std_logic;
     so: out std_logic);
end SR6;

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

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;
```

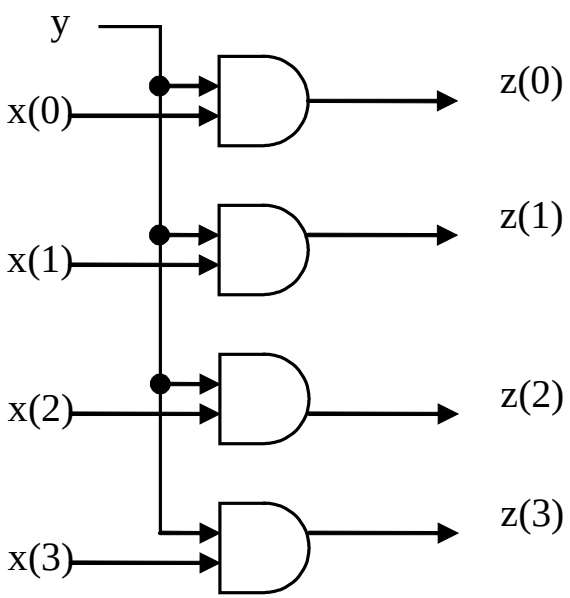
```
entity and2 is
   Port (a,b: in std_logic;
         c: out std_logic);
end and2;

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

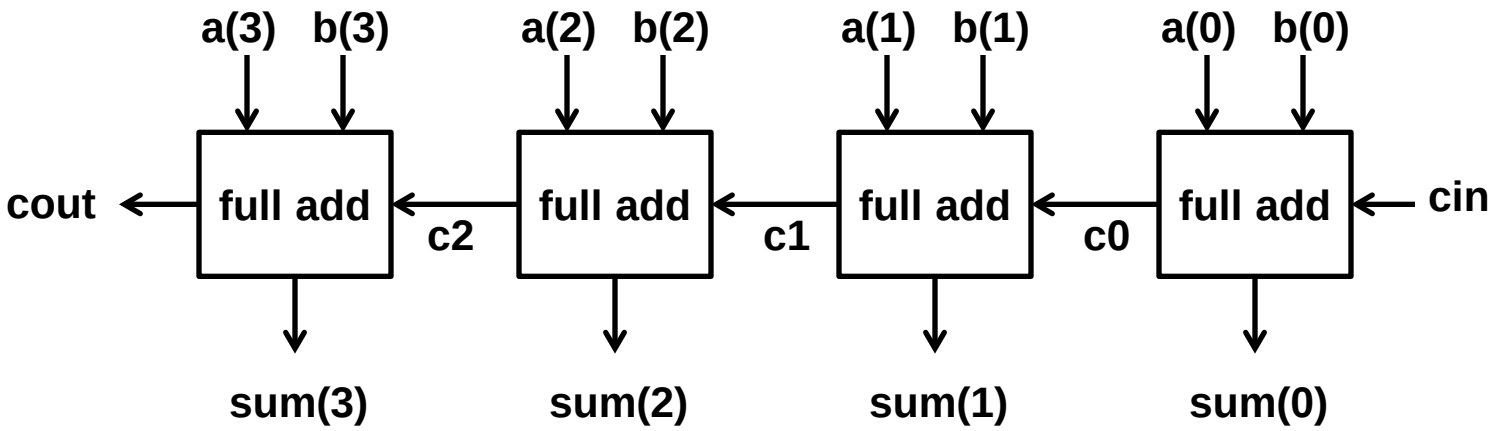
4x1 Unsigned Multiply




```
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;
```

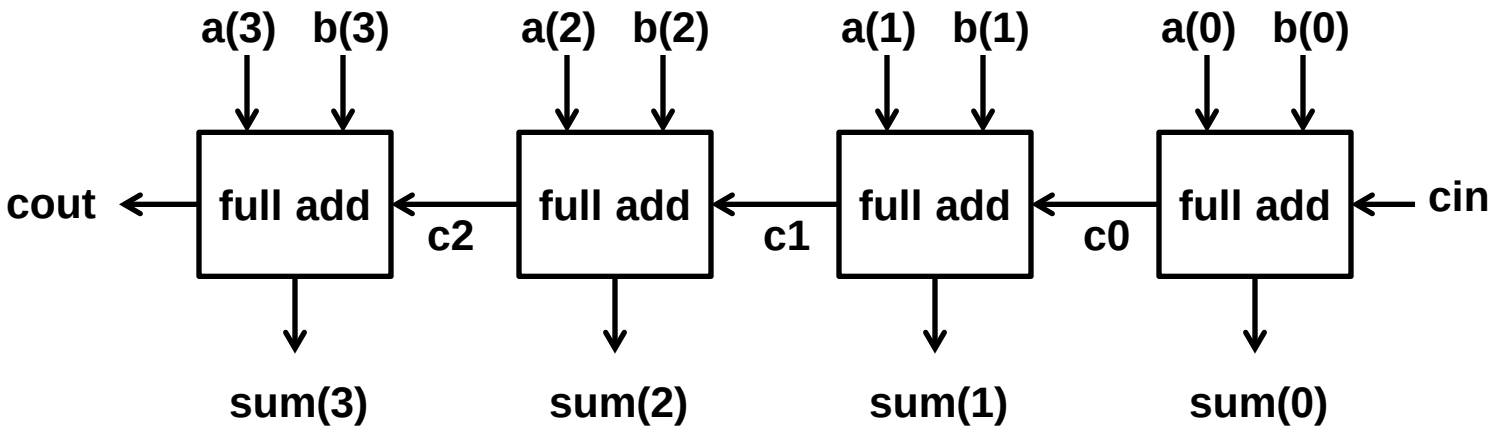
```
architecture struct of mpy4x1 is
    component and2
    port(a,b: in std_logic;
         c: out std_logic);
    end component;

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;
```

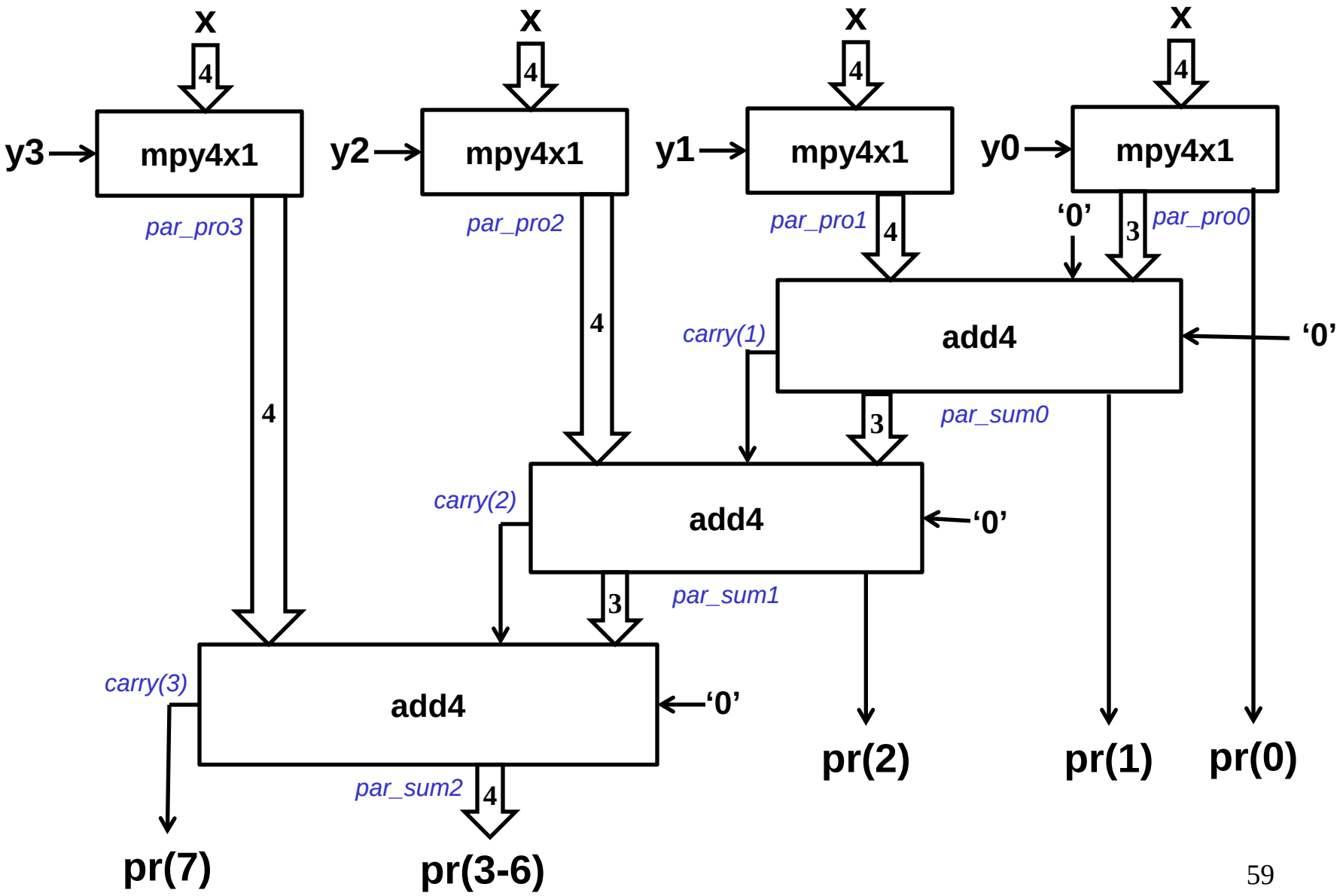
4-bit adder (cont.)




```
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;
```

Subprograms & Overloading

Subprograms

- As VHDL description of a system grows, we need mechanisms to help structure code and facilitate re-use
  - similar to procedures, subroutines, function calls in conventional programming languages
- A subprogram defines a sequential algorithm that performs a certain computation. There are two kinds of subprograms:
  - Function:
    - computes a single value.
    - executes in zero simulation time
  - Procedure:
    - can compute several values
    - may not execute in zero simulation time

Example of Function

- A function to return the maximum of two integers…

```
function max (variable  A, B: in integer) return  integer  is
-- declarations of constants & variables local to function here
-- no signal declarations allowed here
begin
--
-- body: sequential statements
--
return (expression)
end max;
```

- A function has a number of input parameters characterized by their class, mode and type
- A function has a single output (the returned value) characterized only by type

Function Input Parameters

- Class can be **signal, variable, constant** (or **file**)
  - Default class is **constant**
- Mode can only be **in**
  - Default mode is **in**
- Parameter names in function definition are called formal parameters
- When function is called e.g. `next := max (count, index)`
  - Actual parameters *count* and *index* take place of formal parameters *A* and *B*
  - Actuals may be associated with formals by name or position
- Actual parameter must match formal parameter in class, mode and type
  - Except formal parameter of class **constant** can match actual parameter of class signal, variable, constant or expression)

Using Function Max

```
architecture behavioral of xyz is
function max (variable A, B: in integer) return integer is
variable result : integer ;
begin
        result := A;
        if B > A then result := B;
        end if;
        return (result);
end max;

begin
p0: process
        variable v1, v2, v3: integer;
        begin
                …
                v3 := max (v1, v2);    -- or v3 := max (A=>v1, B=>v2);
                …
        end process p0;
end architecture behavioral;
```

Example: Function Rising Edge

```
architecture behavioral of dff is
function rising_edge (signal clock : std_logic)
      return boolean is
variable edge : boolean :=  FALSE;
begin
      edge := (clock = '1' and clock'event);
      return (edge);
end rising_edge;

begin
output: process
      begin
            wait until (rising_edge(Clk));
            Q <= D after 5 ns;
            Qbar <= not D after 5 ns;
      end process output;
end architecture behavioral;
```

Properties of Functions

- Functions cannot modify parameters
  - no side effects
- Functions only execute when called
  - Execute in zero time
  - Wait statements not permitted
  - Terminate when value is returned
- Variables are initialized on each call
- Compare to properties of process

Scope and Placement of Functions

Function code can be placed in:

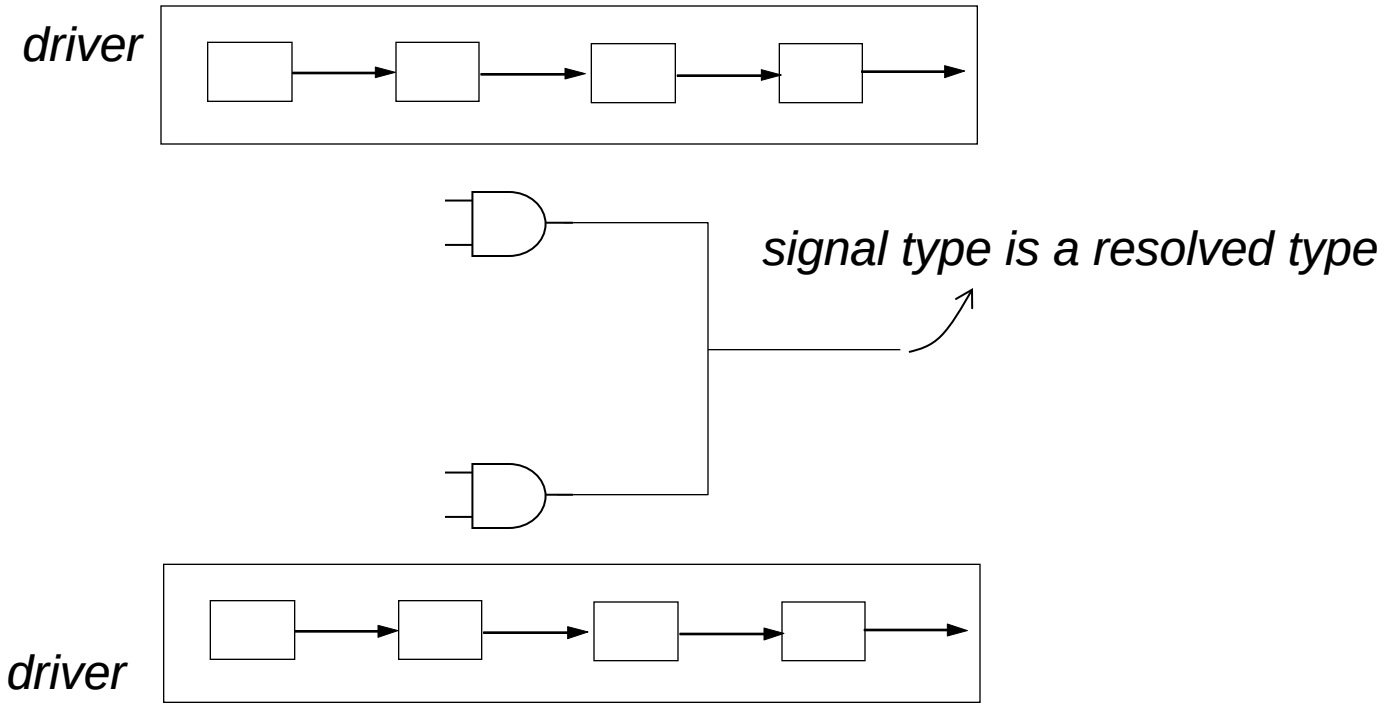
- Declarative section of a process
  - visible (can be called) only in that process
- Declarative section of an architecture
  - visible to CSA expressions and all processes in architecture
- In package declaration
  - visible to all code units that use that package

Example: Type Conversion Function

- Type conversion is common use of functions
  - for example: std_logic_vector to bit_vector

```
function to_bitvector (svalue : std_logic_vector) return bit_vector is
variable outvalue : bit_vector (svalue'length-1 downto 0);
begin
    for i in svalue'range loop -- scan all elements of the array
        case svalue (i) is
            when '0' => outvalue (i) := '0';
            when '1' => outvalue (i) := '1';
            when 'H' => outvalue (i) := '1';
            when others => outvalue (i) := '0';
        end case;
    end loop;
    return outvalue;
end to_bitvector
```

Resolution Functions



- Resolution function is invoked whenever an event occurs on this signal

Std_Logic Revisited

- Declaration of resolved type in IEEE std_logic_1164.vhd:

```
type std_ulogic is (
        'U',    -- Uninitialized
        'X',    -- Forcing Unknown
        '0',    -- Forcing 0
        '1',    -- Forcing 1
        'Z',    -- High Impedance
        'W',    -- Weak Unknown
        'L',    -- Weak 0
        'H',    -- Weak 1
        '-',    -- Don't care
        );

function resolved (s: std_ulogic_vector) return std_ulogic;

subtype std_logic is resolved std_ulogic;
```

declaration of function "resolved"

assigned as resolution function of type std_logic

Creating Resolved Type

- Four steps in creating a resolved signal type:

1. Start with unresolved type that can take on required range of values
   - e.g. **type** abc **is** (‘U’, ‘0’, ‘1’);
2. Create a new type that is a 1-D array of unresolved type
   - e.g. **type** abc_vector **is array** (**natural range** <>**) of** abc;
   - used by VHDL to capture multiple current assignments to a signal
3. Construct a resolution function that takes as input an array of unresolved signals and outputs a single resolved value
   - e.g. **function** res_abc (svec: abc_vector) **return** abc;
4. Declare new resolved type that is a sub-type of unresolved type with the associated resolution function
   - e.g. **subtype** abc_logic **is** res_abc abc;

Example: Resolved Logic

- *Create a resolved data type that can be 0, 1 or X (undefined)*

```
architecture behave of res_ex is

type mylogic is ('X', '0', '1');
type mylogic_vec is array (natural range <>) of mylogic;

function connect(mvec: mylogic_vec) return mylogic is
variable cml: mylogic;
begin
    cml:=mvec(mvec'left);
    for i in mvec'range loop
        if  (mvec(i)/= cml) then
            return('X');
        end if;
    end loop;
    return(cml);
end function connect;

subtype reslogic is connect mylogic;
```

	X	0	1
X	X	X	X
0	X	0	X
1	X	X	1

Procedures

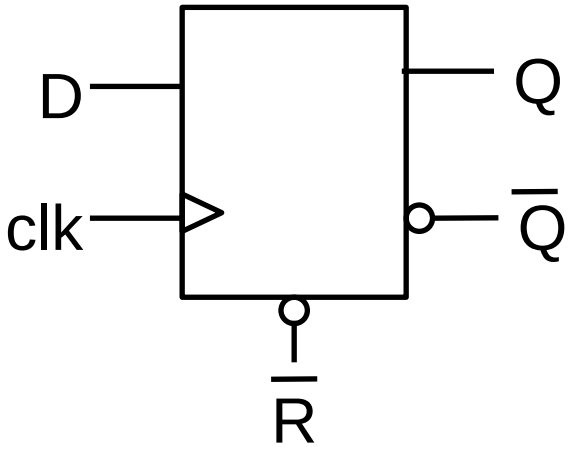
- Procedure is more powerful construct used to decompose large, complex behaviors into modular sections
- Unlike a function a procedure can modify parameters
  - parameter mode can be in, out or inout
  - default class of in parameters is constant
  - default class of out and inout parameters is variable
- No return statement
- Like functions:
  - Actual parameters must match formals in class, mode and type
  - Locally declared variables are initialized on each call

Procedures and Simulation Time

- Unlike functions, procedures do have a concept of time
  - do not necessarily execute in zero time
- Procedures can include signal assignment statements
  - to modify signals in parameter list
  - can also modify other signals (e.g. ports) – not recommended
- Procedures can be suspended with wait statements
  - unless called from a process that has sensitivity list

Example: D Flip-flop as Procedure

```
procedure DFF (signal D, clk, Rbar : in std_logic;
signal Q, Qbar : out std_logic) is
begin
    if (Rbar = '0') then
        Q <= '0' after 5 ns;
        Qbar <= '1' after 5 ns;
    elsif (rising_edge(clk)) then
        Q <= D after 5 ns;
        Qbar <= (not D) after 5 ns;
    end if;
end DFF;
```



Scope and Placement of Procedures

Procedure code can be placed in:

- Declarative section of a process
  - visible (can be called) only in that process
- Declarative section of an architecture
  - visible to CSA expressions and all processes in architecture
- In package declaration
  - visible to all code units that use that package

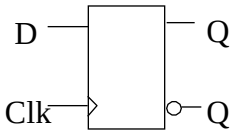
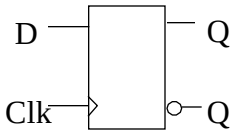
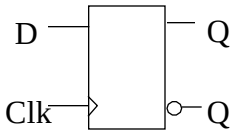
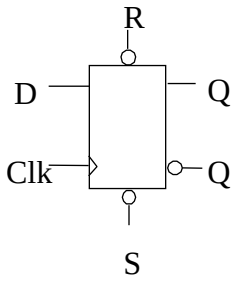
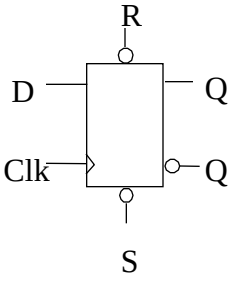
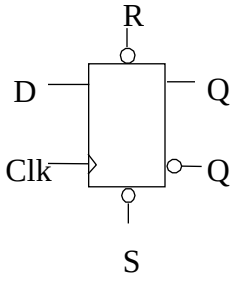
Subprogram Overloading

- One of the more powerful aspects of VHDL subprograms (functions & procedures) is ability to overload the sub-program name
- Overloading is giving two or more sub-programs the same name e.g.:

```
function negate(arg: integer) return integer;

function negate(arg: bit) return bit;
```

- When a call to negate is made, it is possible to identify the exact function to which the call is made from the number and type of actuals passed
- e.g., negate(20) vs. negate('1')

Example: How many D flip-flops do we need?

type / **style**	bit	std_logic	std_logic_vector (register)
simple D			
D with async R/S			

- How many flip-flop procedures do we need to create?

  dff_bit (clk, d, q, qbar)

  asynch_dff_bit (clk, d,q,qbar,reset,clear)

  dff_std (clk,d,q,qbar)

  asynch_dff_std (clk, d,q,qbar,reset,clear)

  etc.

D Flip-flops with overloaded names

- Solution: give all D flip-flop procedures same name
- Allow compiler to work out which procedure is appropriate
- If there is ambiguity, compiler will generate an error.

```
-- call a simple D flip-flop operating on bit signals
signal clk, d, q, qbar: bit
dff (clk, d, q, qbar);

-- call an RS 8-bit register operating on  8-bit std_logic_vector signals
signal clk, reset, clear: std_logic;
signal d, q, qbar: std_logic_vector (7 downto 0);
dff (clk, d, q, qbar, reset, clear);
```

Operator Overloading

- When a standard operator symbol is made to behave differently based on the type of its operands, the operator is said to be overloaded.
- For example in the standard package, **and** operation is only defined for arguments of type BIT and BOOLEAN, and for one-dimensional arrays of BIT and BOOLEAN.
- What if the arguments were of type MVL (where MVL is a user defined enumeration type with values ‘U’, ‘0’, ‘1’ and ‘Z’?)
- It is possible to augment the **and** operation as a function that operates on arguments of type MVL – the **and** operator is then said to be overloaded.

Operator Overloading: MVL Data Type

- In package:

  type MVL **is** (‘U’, ‘0’, ‘1’, ‘Z’);

  function “**and**” (L, R : MVL) **return** MVL;

  function “**or**” (L, R : MVL) **return** MVL;

  function “**not**” (R : MVL) **return** MVL;

  -- note: since *and*, *or* and *not* operators are predefined operator symbols, they have to be enclosed within double quotes when used as overloaded operator function names.

- In architecture:

  signal A, B, C : MVL;

  signal X, Y, Z : BIT;

  A <= C **or** ‘1’; --- refer to the overloaded operator

  B <= “**or**” (C, ‘1’); --- function call notion

  X <= **not** Y; -- refer to predefined operator

  Z <= X **and** Y; -- refer to predefined operator

  C <= (A **or** B) **and** (**not** C); -- refer to the overloaded operator

  Z <= (X **and** Y) **or** A; -- this is error: