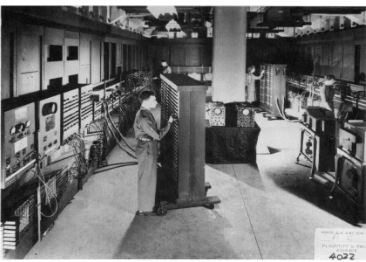
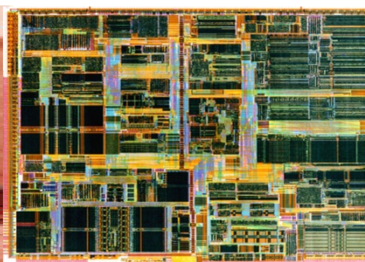
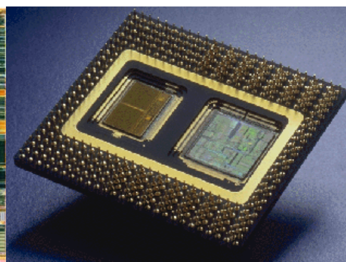
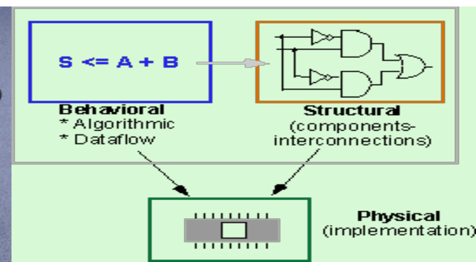
Lecture 10
Subprograms & Overloading

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



Subprograms

- As behavioral 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 (signal  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, 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 (signal 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;
signal s1, s2: integer;
begin
p0: process
    variable vr: integer;
    begin
        …
        vr := max (s1, s2);    -- or vr := max (A=>s1, B=>s2);
        …
    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

Pure vs. Impure Functions

[**pure** | **impure**] **function** *function-name* (*parameter-list*)
  return *return-type*

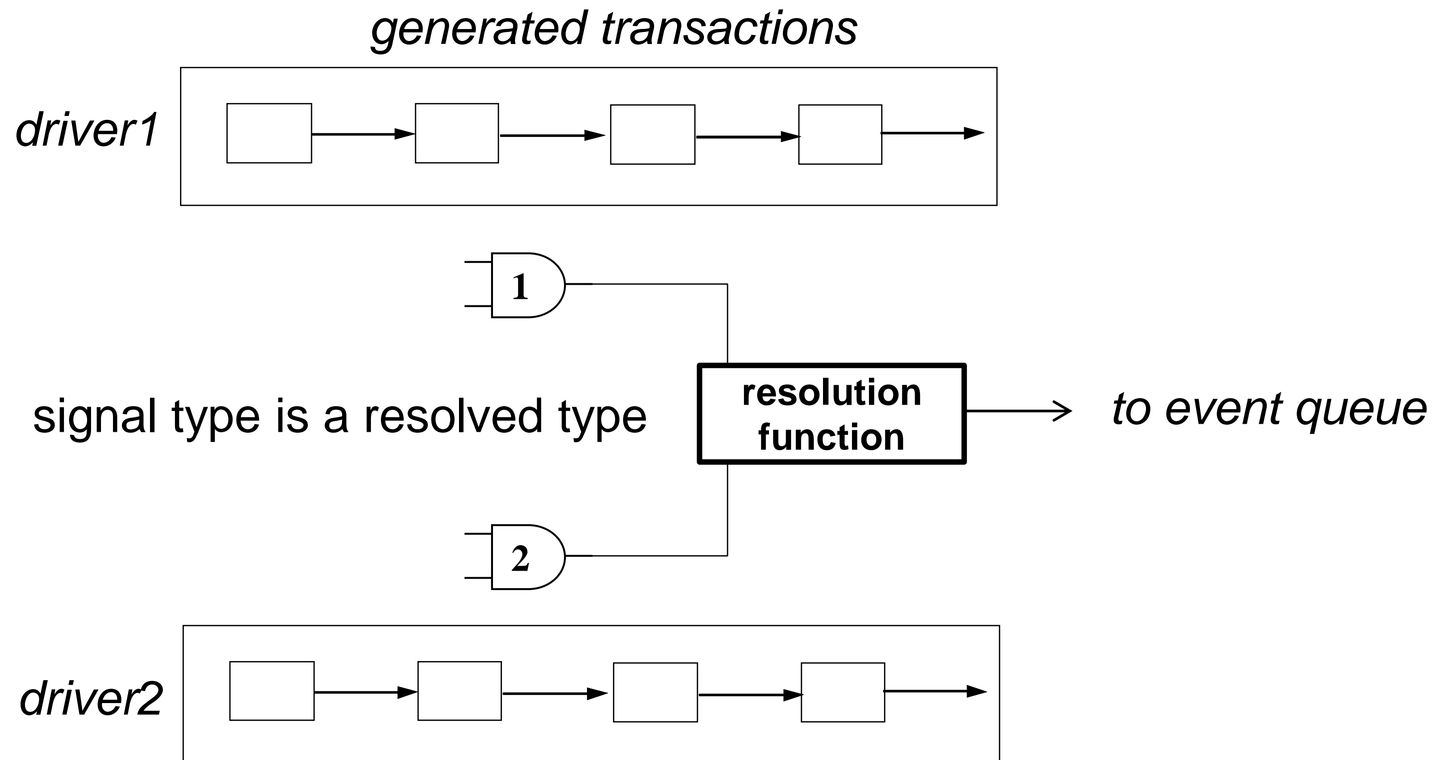
- By default, functions are pure
  - can only read parameters explicitly passed to the function
  - always return same value when called with same actuals
- Impure functions can access other signals & variables visible to parent (calling process or architecture)
  - can return different values when called with same actuals
  - examples are functions to return random number, simulation time, next word from text file etc.

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
        );

type std_ulogic_vector is array (natural range <>) of std_ulogic;

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’, ‘Z’, ‘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

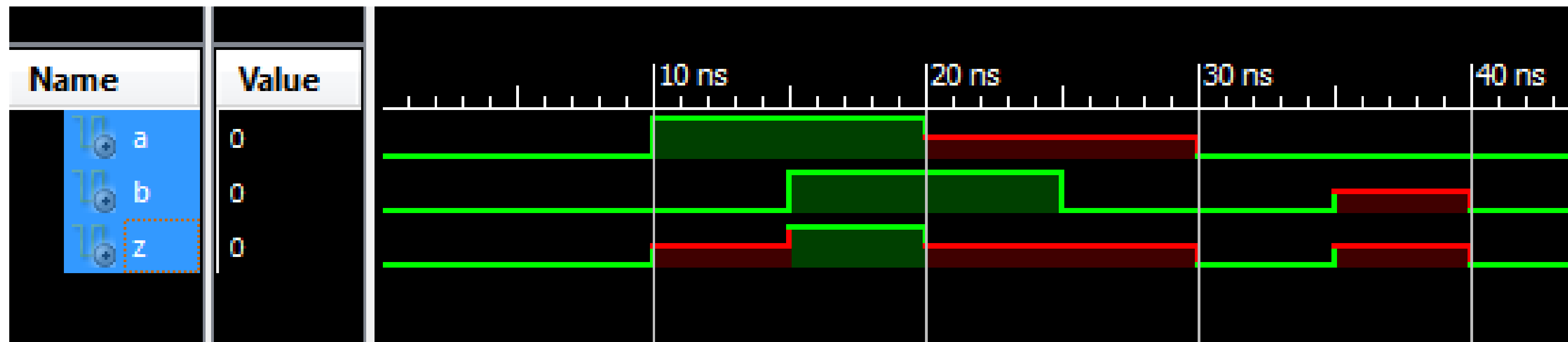
- *Create a resolved data type that can be 0, 1 or X (undefined)*

```
architecture behave of res_ex is

type mylogic is ('X', '0', '1');          -- step 1
type mylogic_vec is array (natural range <>) of mylogic;          -- step 2

function connect(mvec: mylogic_vec) return mylogic is          -- step 3
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;          -- step 4
```

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

Simulation of *reslogic* example

```
signal a,b,z: reslogic;

begin
tpr: process
    begin
        a<= '0', '1' after 10 ns, 'X' after 20ns, '0' after 30ns;
        b<= '0', '1' after 15 ns, '0' after 25ns, 'X' after 35ns;
        wait for 40 ns;
    end process;
    z<=a;
    z<=b;
end behave;
```



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 (with time delay)
  - can also modify other signals visible at the place procedure was called (e.g. ports) – not recommended
- Procedures can be suspended with wait statements
  - effectively suspends caller
  - cannot be called from a process that has sensitivity list

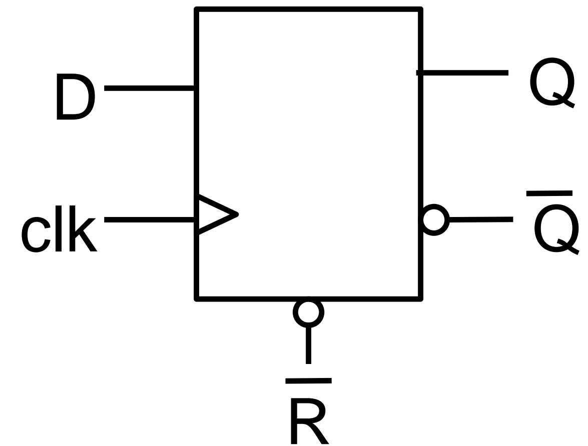
Procedure Declaration

```
procedure procedure_name (formal_parameter_declarations) is
-- declarations of constants & variables local to procedure here
-- no signal declarations allowed here
begin
--
-- body: sequential statements
--
end procedure_name;
```

- formal_parameter_declarations includes inputs and outputs
- Output parameters (signals & variables) may be modified by procedure

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;
```

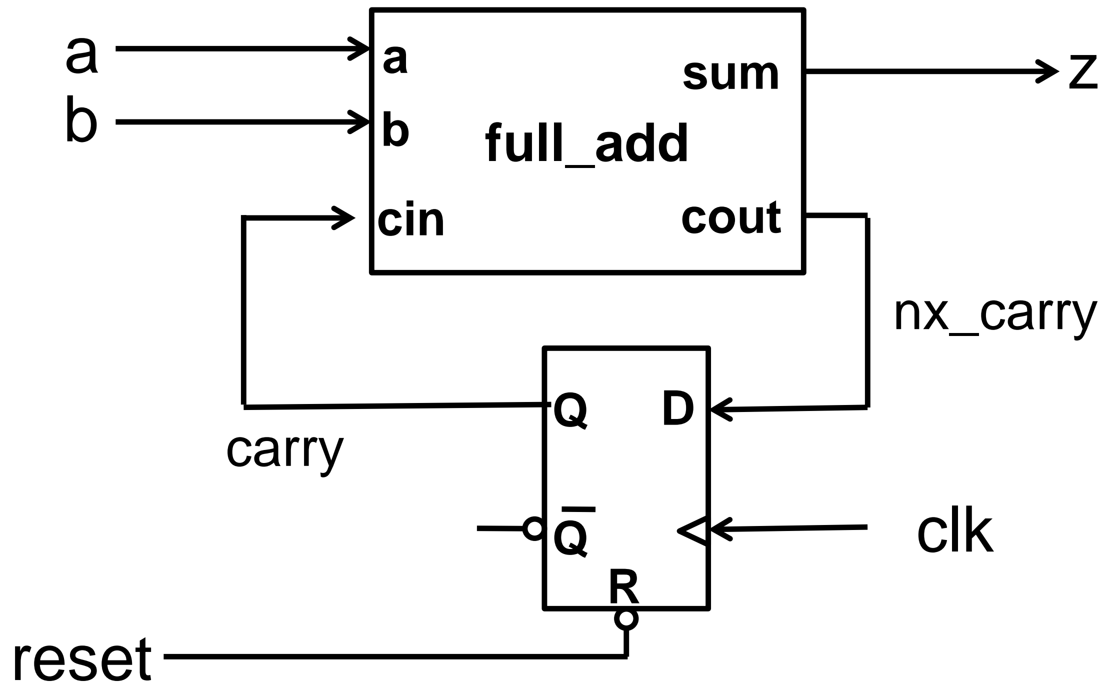


Scope and Placement of Procedures

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

Example: Bit-Serial Adder



reset: 1 1 1 1 1 1 1 1 0

cin: 0 0 1 1 1 1 1 0 X

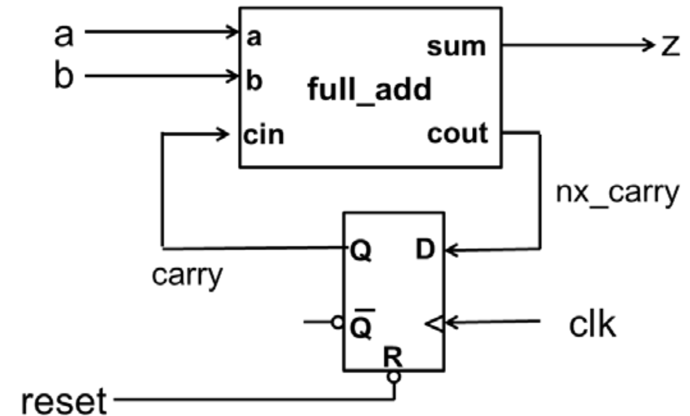
a: 0 0 0 1 1 0 1 1 0

b: 0 0 0 0 0 1 1 1 0

Z: 0 0 1 0 0 0 1 0 0

← *time*

Bit-Serial Adder with Concurrent Procedure Call



architecture structural **of** serial_adder **is**

component full_add
port (a, b, cin : **in** std_logic;
sum, cout : **out** std_logic);
end component;

procedure DFF(**signal** D, clk, Rbar: **in** std_logic, **signal** Q, Qbar: **out** std_logic) **is**
begin
-- *procedure body as described in earlier slide*
end procedure DFF**;**

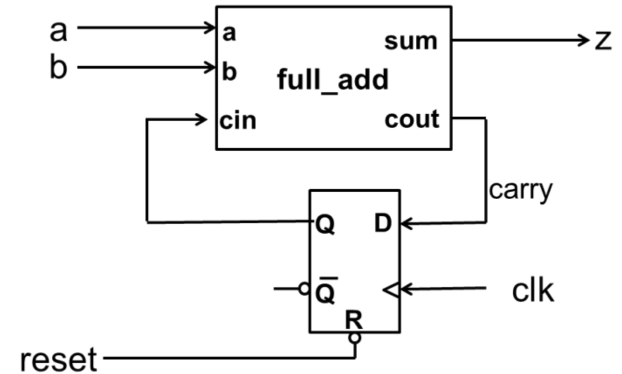
signal carry, nx_carry: std_logic;

begin
C1: full_add **port map** (a => a, b => b, cin => carry, sum =>z, cout => nx_carry);

DFF(clk => clk, Rbar =>reset, D=> nx_carry, Q=>carry, Qbar =>**open**);

end architectural structural;

Bit-Serial Adder with Sequential Procedure Call




```
architecture structural of serial_adder is

component full_add
port (a, b, cin : in std_logic;
    sum, cout : out std_logic);
end component;

procedure DFF(signal D, clk, Rbar: in std_logic, signal Q, Qbar: out std_logic) is
begin
    -- procedure body as described in earlier slide
end procedure DFF;

signal s1, s2 : std_logic;
begin
C1: full_add port map (a => a, b => b, cin => s1, sum =>z, cout => s2);
dpr: process
    begin
        DFF(clk => clk, Rbar =>reset, D=> s2, Q=>s1, Qbar =>open);
        wait on clk, reset;
    end process;
end architectural structural;
```

Subprogram Overloading

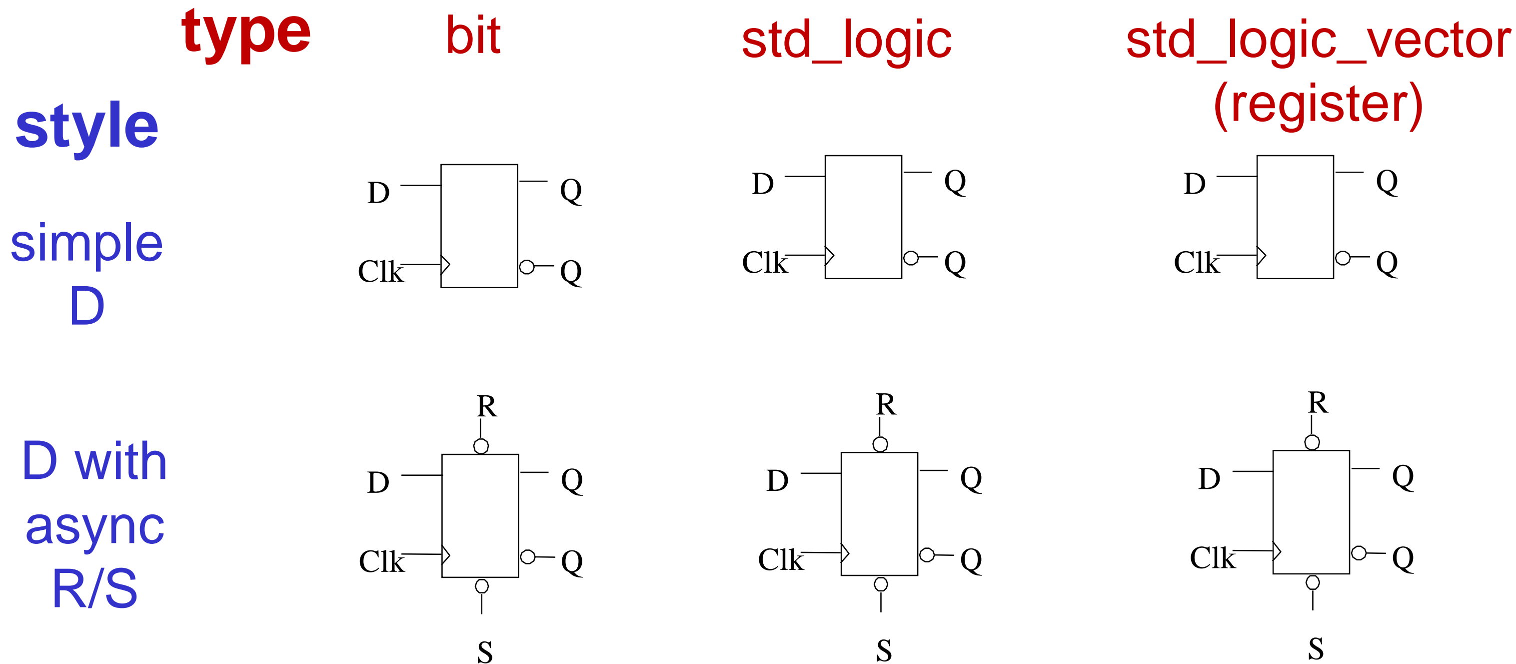
- One of the more powerful aspects of VHDL subprograms (functions & procedures) is ability to overload the sub-program name
- e.g., negate(20) vs. negate('1')
- Overloading is giving two or more sub-programs the same name e.g.:

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

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



- 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);
```

Hiding Existing Subprograms

- If there is a conflict in subprogram names, priority goes to subprogram with most local scope e.g.
- A call to function xyz within a process will use xyz declared in that process over xyz declared in the architecture over xyz declared in some external package
- Allows us to over-ride previously defined subprograms

Example using Packages

```
package P1 is
    function ADD (X, Y : BIT_VECTOR) return BIT_VECTOR;
end P1;

use WORK.P1.all;

architecture overloaded  of abc is
function ADD (X, Y : BIT_VECTOR) return BIT_VECTOR;
function ADD (A, B : BIT_VECTOR) return BIT_VECTOR;
signal IN1, IN2: BIT_VECTOR(3 downto 0);

begin
    SUM_CORRECT <= ADD (X => IN1, Y => IN2);
    SUM_ERROR <= ADD (IN1, IN2);     -- compiler error: ambiguous
end overloaded;
```

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 *my_logic* (where *my_logic* is a user defined enumeration type with values ‘0’, ‘1’ and ‘X’?)
- It is possible to augment the **and** operation as a function that operates on arguments of type *my_logic* – the **and** operator is then said to be overloaded.

Operator Overloading: *my_logic* Data Type

- In package:

```
type my_logic is ('X', '0', '1');
function "and" (L, R : my_logic) return my_logic is
begin
    if L='1' and R='1' return '1';
    elsif  L='X' or R='X' return 'X';
    else return '0';
    end if;
end function "and";
```

-- 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.

Operator Overloading: *my_logic* Data Type

- In package:

```
type my_logic is ('X', '0', '1');
function "and" (L, R : my_logic) return my_logic;
function "or" (L, R : my_logic) return my_logic;
function "not" (R : my_logic) return my_logic;
```

- In architecture:

```
signal A, B, C : my_logic;
signal X, Y, Z : BIT;
A <= C OR '1';          --- refer to the overloaded operator
B <= "or" ('0', A);     --- overloaded operator - function call notation
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(s)
Z <= (X and Y) or A;          -- this is error:
```