Filename=”ch5.doc”

5. 0 VHDL OPERATORS

There are seven groups of predefined VHDL operators:

1. Binary logical operators: and or nand nor xor xnor
2. Relational operators: = /= < <= > >=
3. Shifts operators: sll srl sla sra rol ror
4. Adding operators: + - &(concatenation)
5. Unary sign operators: + -
6. Multiplying operators: * / mod rem
7. Miscellaneous operators: not abs **

The above classes are arranged in increasing priority when parentheses are not used.

Example1: Priority of operators. Let A=”110”, B=”111”, C=”011000”, and D=”111011”

(A & not B or C ror 2 and D) = “110010” ?

the operators are applied in the following order: not, &, ror, or, and, =

not B = ‘000”	--bit-by-bit complement
A & not B = “110000”	--concatenation
C ror 2 = “000110”	--rotate right 2 places
(A & not B) or (C ror 2) = “110110	--bit-by-bit or
(A & not B or C ror 2) and D = “110010”	--bit-by-bit and
[(A & not B or C ror 2 and D) = “110010”]=TRUE	--with parentheses the equality test is done last

Example 2: Shift operators. Let A = “10010101” --are in **IEEE.NUMERIC_BIT** or in **IEEE.NUMERIC_STD**

A sll 2 = “01010100”	--shift left logical, filled with ‘0’
A srl 3 = “00010010”	--shift right logical, filled with ‘0’
A sla 3 = “10101111”	--shift left arithmetic, filled with right bit
A sra 2 = “11100101”	--shift right arithmetic, filled with left bit
A rol 3 = “10101100”	--rotate left by 3
A ror 5 = “10101100”	--rotate right by 5

Example 3: arithmetic operators.

If the left and right signed operands are of different lengths, the shortest operand will be sign-extended before performing an arithmetic operation. For unsigned operands, the shortest operand will be extended by filling in 0s on the left.

signed:	“01101” + “1011” = “01101” + “11011” = “01000”
unsigned:	“01101” + “1011” = “01101” + “01011” = “11000”

TYPE SIGNED IS ARRAY(NATURAL RANGE <>) OF STD_LOGIC;
TYPE UNSIGNED IS ARRAY(NATURAL RANGE <>) OF STD_LOGIC;

When unsigned or signed addition is performed, the final carry is discarded, and overflow is ignored. If a carry is needed, an extra bit is appended to the leftmost bit.

Any overloaded binary operators perform binary operation with all argument of the same type. Vector arguments may be unequal in size, the smaller one is sign-extended to the same size as the larger argument before the operation is performed. For “+” operators,

```
FUNCTION “+” (arg1, arg2 : STD_LOGIC) RETURN STD_LOGIC;
FUNCTION “+” (arg1, arg2 : STD_ULOGIC_VECTOR) RETURN STD_ULOGIC_VECTOR;
FUNCTION “+” (arg1, arg2 : STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR;
FUNCTION “+” (arg1, arg2 : UNSIGNED) RETURN UNSIGNED;
FUNCTION “+” (arg1, arg2 : SIGNED) RETURN SIGNED;
```

```
CONSTANT A: unsigned(3 DOWNTO 0):= “1101”;
CONSTANT B: signed(3 DOWNTO 0):=”1011”;
VARIABLE  SUMU: unsigned(4 DOWNTO 0);
VARIABLE SUMS: signed(4 DOWNTO 0);
VARIABLE OVERFLOW: boolean;

SUMU:= ‘0’ & A + unsigned’(“0101”);        --result is “10010”  sum=2, carry=1
SUMS:=B(3) & B + signed(“1101”);           --result is “”11000” sum =8, carry=1
```

The algorithm for adding two numbers in sign-2’s-complement representation gives an incorrect result when an overflow occurs. This arises because an overflow of the number bits always changes the sign of the result and gives an erroneous n-bit answer. Consider the following example. Two signed binary numbers, 35 and 40, are stored in two 7-bit registers. The maximum capacity of the register is $(2^6–1)=63$ and the minimum capacity is $–2^6=-64$. Since the sum of the numbers is 75, it exceeds the capacity of the register. This is true if the numbers are both positive or both negative.

carries:	0 1	carries:	1 0
+35	0 100011	-35	1 011101
+40	0 101000	-40	1 011000
+75	1 001011	-75	0 110101

In either case, we see that the 7-bit result that should have been positive is negative, and vice versa. Obviously, the binary answer is incorrect and the algorithm for adding binary numbers represented in 2’s complement as stated previously fails to give correct results when an overflow occurs. Note that if the carry out of the sign-bit position is taken as the sign for the result, then the 8-bit answer so obtained will be correct.

An ***overflow*** *condition can be detected by observing the carry into the sign-bit position and the carry out of the sign-bit position. If these two carries are not equal, an overflow condition is produced. This is also detected if the sum in the sign-bit is different from the previous sum.*

5.1 Two’s Complement Integer Addition

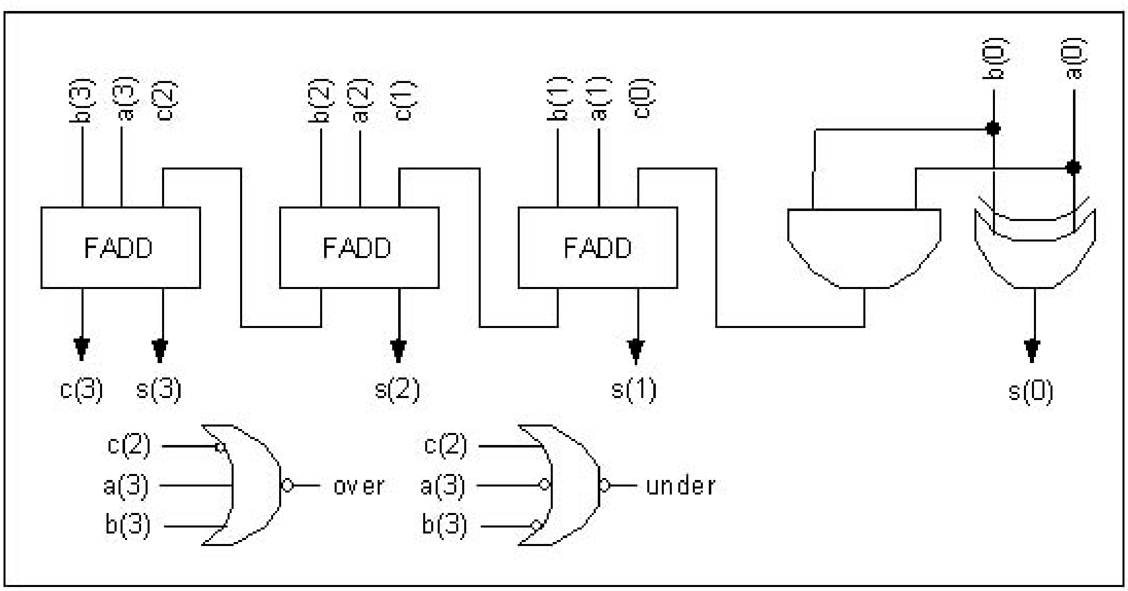
It is assumed that the input vectors are in 2’s complement format.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY ovrflo_undrflo IS
```

```
        PORT(a, b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
                sum : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
                under, over : OUT BIT);
END ovrflo_undrflo;

ARCHITECTURE arch_ovrflo_undrflo OF ovrflo_undrflo IS
BEGIN
        add:    PROCESS(a, b)
                        VARIABLE res : INTEGER;
                BEGIN
                        res := CONV_INTEGER(a) + CONV_INTEGER(b); --(1)
                        IF (res > 7) THEN
                                over <= '1';
                        ELSE
                                over <= '0';
                        END IF;
                        IF (res < -8) THEN
                                under <= '1';
                        ELSE
                                under <= '0';
                        END IF;
                        sum <= Conv_Std_Logic_Vector(res,4);            --(1)
                END PROCESS add;
END arch_ovrflo_undrflo;
```

NOTE 1: CONV_INTEGER is IEEE.std_logic_signed

The above vhdl code is implemented as shown below:



Underflow occurs when adding two negative numbers, the result is positive number. This happens when a(3)=b(3)=1, and no previous carry, c(2)=0. Overflow occurs when adding two positive number, the results is a negative number. This happens when a(3)=b(3)=0, and with previous carry , c(2)=1.

5.2 Two’s Complement Direct Integer Addition

This example uses an abstract integer ports. The integer addition can be done directly without integer-to-bit or bit-to-integer conversion. When using abstract port types, integer and user-defined enumerated ports are converted by Autologic VHDL to bit_vectors of the appropriate size. Only standard library is needed for this coding.

```
PACKAGE my_intgr IS
        SUBTYPE my_int IS INTEGER RANGE –8 TO 7;
END my_intgr;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE WORK.my_intgr.ALL;

ENTITY ovrflo_undrflo IS
        PORT(a, b : IN my_int;
                sum: OUT my_int;
                under, over : OUT BIT);
END ovrflo_undrflo;

ARCHITECTURE arch_ovrflo_undrflo OF ovrflo_undrflo IS
BEGIN
        add:    PROCESS(a, b)
                        VARIABLE res : INTEGER RANGE –16 TO 15 :=0;
                BEGIN
                        res := a + b;
                        IF(res >7) THEN
                                over <= ‘1’;
                        ELSE
                                over <= ‘0’;
                        END IF;
                        IF (res < -8) THEN
                                under <= ‘1’;
                        ELSE
                                under <= ‘0’;
                        END IF;
                        IF (over=’0’ AND under’0’)THEN sum <= res;  END IF;
                END PROCESS add;
END arch_ovrflo_undrflo;
```

The above implementation is identical to the one in 5.1

5.3 Addition Using Procedure Call

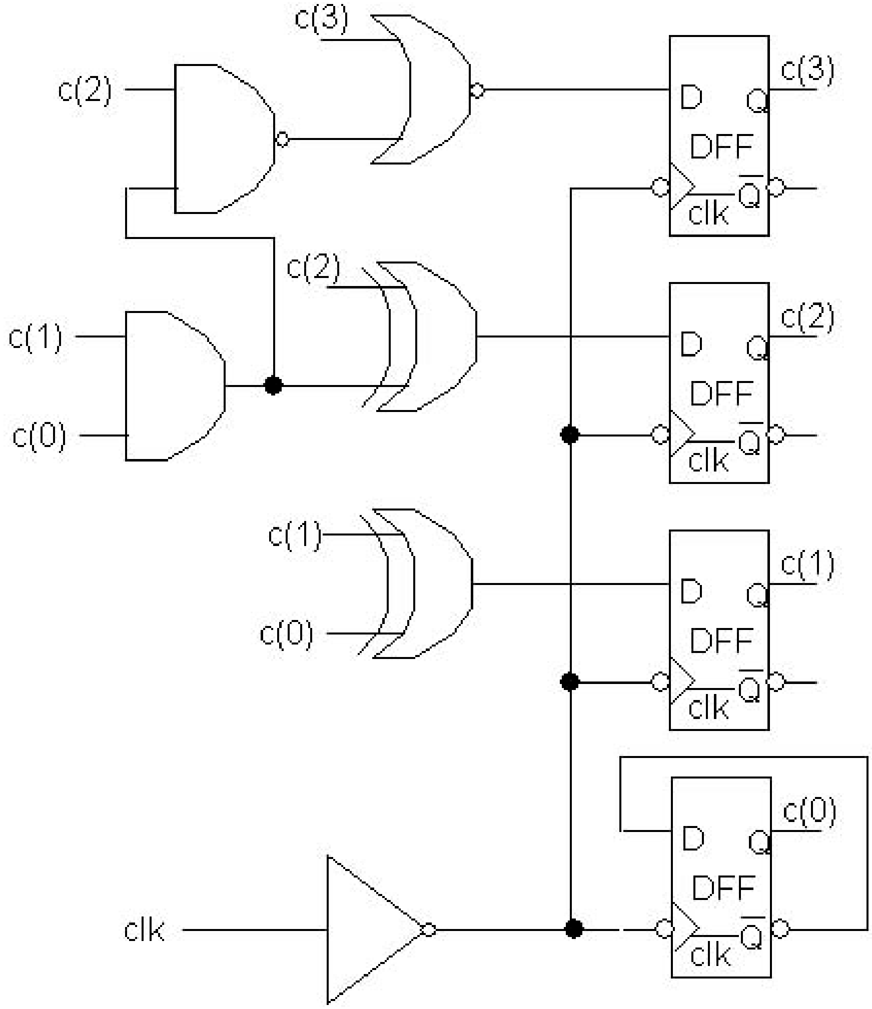
A procedure is a subprogram that can modify its parameters (signals and/or variables) and return new values for these parameters. A procedure is synthesized at each location it is called. This is analogous to a component instantiation in place.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY add IS
        PORT(a, b : IN STD_LOGIC_VECTOR(0 TO 3);
                enable: IN BIT;
                result: OUT STD_LOGIC_VECTOR(0 TO 3);
                carry: OUT STD_LOGIC);
END add;

ARCHITECTURE arch_add OF add IS
        PROCEDURE add_with_carry (SIGNAL g : IN BOOLEAN;
                        SIGNAL a1, a2 : IN STD_LOGIC_VECTOR(0 TO 3);
                        SIGNAL result: OUT STD_LOGIC_VECTOR(0 TO 3);
                        SIGNAL carry: OUT STD_LOGIC) IS
                VARIABLE temp: STD_LOGIC_VECTOR(0 TO 4);
        BEGIN
                IF (g) THEN
                        temp := (a1(0)&a1) + (a2(0)&a2);
                        carry <= temp(0);
                        result <= temp(1 TO 4);
                END IF;
        END add_with_carry;
 BEGIN
 blk1:  BLOCK( enable = ‘1’)
                BEGIN
                        add_with_carry(guard, a, b, result, carry);
                END BLOCK blk1;
END arch_add;
```

5.4 Binary Counter

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY binctr IS
        PORT(clk: IN STD_LOGIC; c : INOUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END binctr;

ARCHITECTURE arch_binctr OF binctr IS
BEGIN
        PROCESS(clk)
        BEGIN
                IF clk’EVENT AND clk=’1’ THEN
                        c <= c + “0001”;
                END IF;
```

```
	END PROCESS;
END arch_binctr;
```



5.4 Rotate 8-Bit Register by one

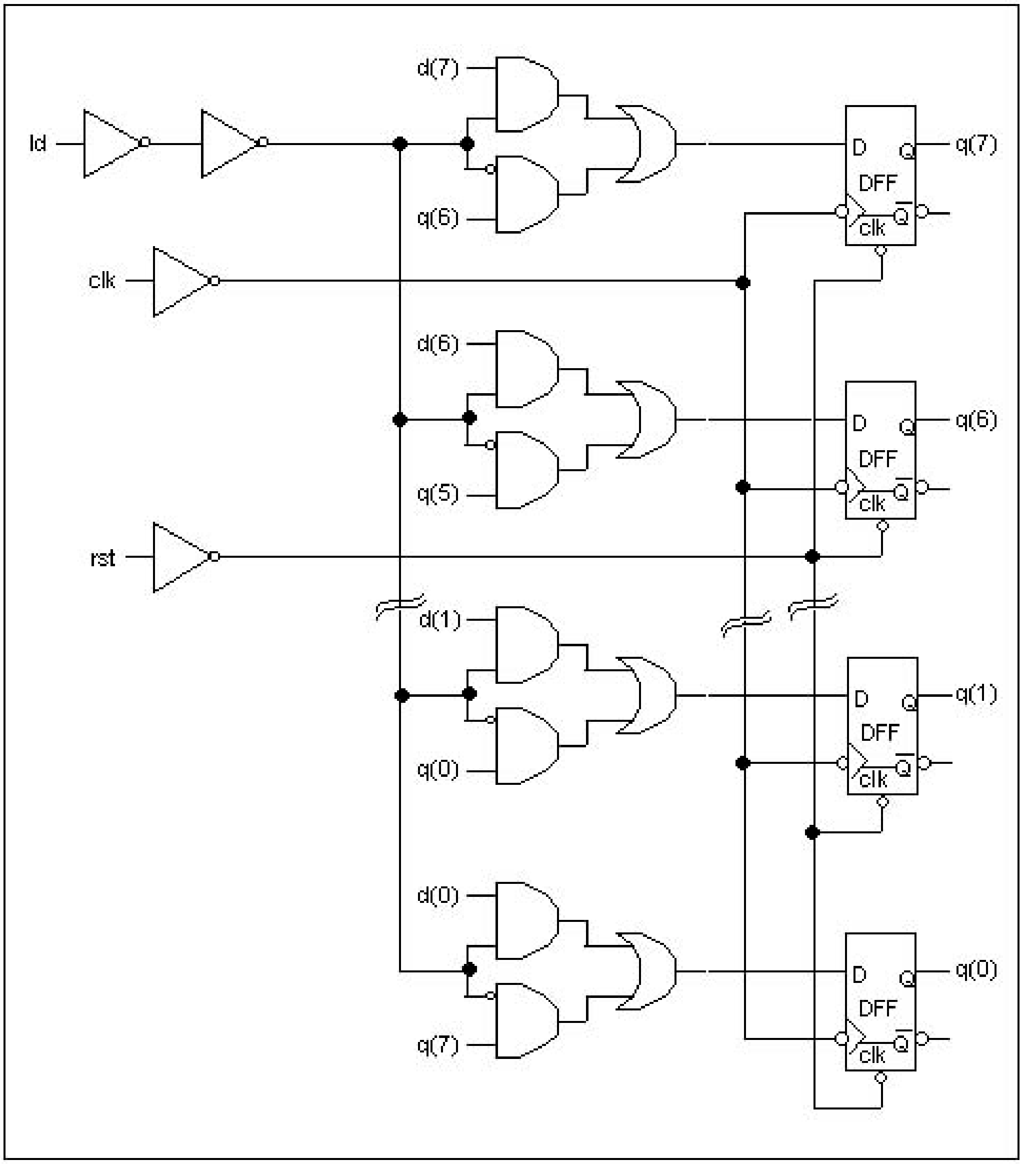
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY rotate IS
	PORT(clk, rst, ld : IN STD_LOGIC;
		d : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
		q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END rotate;

ARCHITECTURE arch_rotate1 OF rotate IS
	SIGNAL qtmp : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
	PROCESS(clk, rst)
	BEGIN
```

```
        IF rst = '1' THEN
            qtmp <= "00000000";
        ELSIF (clk = '1' AND clk'EVENT) THEN
            IF (ld = '1') THEN
                qtmp <= d;
            ELSE
                qtmp <=qtmp( 6 DOWNTO 0) & qtmp(7);
            END IF;
        END IF;
    END PROCESS;
    q <= qtmp;
END arch_rotate1;
```

Seperating combinatorial and sequential circuit portion using procedure call

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

PACKAGE mypackage IS
    PROCEDURE reg8(SIGNAL clk, rst : IN STD_LOGIC;
            SIGNAL d : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            SIGNAL q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END mypackage;

PACKAGE BODY mypackage IS

    PROCEDURE reg8(SIGNAL clk, rst : IN STD_LOGIC;
            SIGNAL d : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            SIGNAL q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)) IS
    BEGIN
        IF rst = '1' THEN
            q <= "00000000";
        ELSIF clk = '1' AND clk'EVENT THEN
            q <= d;
        END IF;
    END reg8;
END mypackage;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.mypackage.ALL;
ENTITY rotate IS
    PORT(clk, rst, ld : IN STD_LOGIC;
        d : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        q : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END rotate;

ARCHITECTURE arch_rotate2 OF rotate IS
    SIGNAL dtmp, qtmp : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
    dtmp <= d WHEN (ld = '1') ELSE  qtmp(6 DOWNTO 0) & qtmp(7);
    reg8(clk, rst, dtmp, qtmp);
    q <= qtmp;
END arch_rotate2;
```

Both vhdl codes have the same implementation shown below:


ld
clk
rst
d(7)
q(6)
d(6)
q(5)
d(1)
q(0)
d(0)
q(7)
D
Q
DFF
clk
q(7)
q(6)
q(1)
q(0)