

# **Module 12 : Behavioral VHDL**

## **Tutorial and Exercises**

### **For the VeriBest Simulator**

#### **Copyright 1995-1999 SCRA**

**All rights reserved. This information is copyrighted by the SCRA, through its Advanced Technology Institute (ATI), and may only be used for non-commercial educational purposes. Any other use of this information without the express written permission of the ATI is prohibited. Certain parts of this work belong to other copyright holders and are used with their permission. All information contained, may be duplicated for non-commercial educational use only provided this copyright notice and the copyright acknowledgements herein are included. No warranty of any kind is provided or implied, nor is any liability accepted regardless of use.**

**The United States Government holds “Unlimited Rights” in all data contained herein under Contract F33615-94-C-1457. Such data may be liberally reproduced and disseminated by the Government, in whole or in part, without restriction except as follows: Certain parts of this work to other copyright holders and are used with their permission; This information contained herein may be duplicated only for non-commercial educational use. Any vehicle, in which part or all of this data is incorporated into, shall carry this notice .**

**See the [RASSP Disclaimer file](#) for additional RASSP Disclaimer, Warranty and Limitation of Liability Information concerning the material, VHDL code and software developed under the RASSP programs or incorporated in RASSP material.**

## 1. Getting Started

- 1.1. The Module 12 lab will provide practice with behavioral VHDL models. The VHDL created and compiled for the Module 10 and Module 11 labs will be reused through the use of library mappings.
- 1.2. The first VHDL files that you will need are **mult\_controller\_behav.vhdl** and **full\_mult\_str.vhdl**.

## 2. Examine and compile the state machine code for this lab

- 2.1. Open the file **mult\_controller\_behav.vhdl** using a text editor or a VHDL editing environment.

```
-----
-- Eight Bit Multiplier Controller Behavioral Example  --
--   RASSP E&F Module # 12 Behavioral VHDL             --
--   Robert Klenke UVa 28 May 1996                     --
-----

LIBRARY gate_lib;
USE gate_lib.resources.all;

ENTITY mult_controller_behav IS
    PORT(reset      : IN level;  --global reset signal
          start     : IN level;  --input to indicate start of process
          q0        : IN level;  --q0 ,input from data path
          clk       : IN level;  --clock signal
          a_enable  : OUT level;  --clock enable for A register
          a_reset   : OUT level;  --Reset control for A register
          a_mode    : OUT level;  --Shift or load mode for A
          c_enable  : OUT level;  --clock enable for c register
          m_enable  : OUT level;  --clock enable for M register
          q_enable  : OUT level;  --clock enable for Q register
          q_mode    : OUT level); --Shift or load mode for Q
END mult_controller_behav;

ARCHITECTURE state_machine OF mult_controller_behav IS

    SUBTYPE count_integer IS integer RANGE 0 to 8;
    TYPE states IS (idle, initialize, test, shift, add);
    SIGNAL present_state : states := idle;
    SIGNAL next_state    : states := idle;
    SIGNAL present_count  : count_integer := 0;
    SIGNAL next_count     : count_integer := 0;
```

## Module 12 – Behavioral VHDL Lab Tutorial

```
BEGIN

CLKD : PROCESS(clk,reset)
BEGIN
    IF(reset = '1') THEN
        present_state <= idle;
        present_count <= 0;
    ELSIF(clk'EVENT AND clk = '1' AND clk'LAST_VALUE = '0') THEN
        present_state <= next_state;
        present_count <= next_count;
    END IF;
END PROCESS CLKD;

STATE_TRANS : PROCESS(present_state,present_count,start,q0)
BEGIN
    next_state <= present_state;           -- default case
    next_count <= present_count;           -- default case
    CASE present_state IS
        WHEN idle =>
            IF(start = '1') THEN
                next_state <= initialize;
            ELSE
                next_state <= idle;
            END IF;
            next_count <= present_count;
        WHEN initialize =>
            next_state <= test;
            next_count <= present_count;
        WHEN test =>
            IF(present_count < 8) THEN
                IF(q0 = '0') THEN
                    next_state <= shift;
                ELSE
                    next_state <= add;
                END IF;
            ELSE
                next_state <= idle;
            END IF;
            next_count <= present_count;
        WHEN add =>
            next_state <= shift;
            next_count <= present_count;
        WHEN shift =>
            next_state <= test;
            next_count <= present_count + 1;
        WHEN OTHERS =>
            next_state <= idle;
```

## Module 12 – Behavioral VHDL Lab Tutorial

```
        next_count <= present_count;
    END CASE;
END PROCESS STATE_TRANS;

OUTPUT : PROCESS(present_state)
BEGIN
    CASE present_state IS
        WHEN idle =>
            a_enable <= '0';
            a_reset  <= '1';
            a_mode   <= '1';
            c_enable <= '0';
            m_enable <= '0';
            q_enable <= '0';
            q_mode   <= '1';
        WHEN initialize =>
            a_enable <= '1';
            a_reset  <= '0';
            a_mode   <= '1';
            c_enable <= '1';
            m_enable <= '1';
            q_enable <= '1';
            q_mode   <= '1';
        WHEN test =>
            a_enable <= '0';
            a_reset  <= '1';
            a_mode   <= '1';
            c_enable <= '0';
            m_enable <= '0';
            q_enable <= '0';
            q_mode   <= '1';
        WHEN add =>
            a_enable <= '1';
            a_reset  <= '1';
            a_mode   <= '1';
            c_enable <= '1';
            m_enable <= '0';
            q_enable <= '0';
            q_mode   <= '0';
        WHEN shift =>
            a_enable <= '1';
            a_reset  <= '1';
            a_mode   <= '0';
            c_enable <= '0';
            m_enable <= '0';
            q_enable <= '1';
            q_mode   <= '0';
```

```
        WHEN OTHERS =>
            a_enable <= '0';
            a_reset  <= '1';
            a_mode   <= '1';
            c_enable <= '0';
            m_enable <= '0';
            q_enable <= '0';
            q_mode   <= '1';
        END CASE;
    END PROCESS OUTPUT;

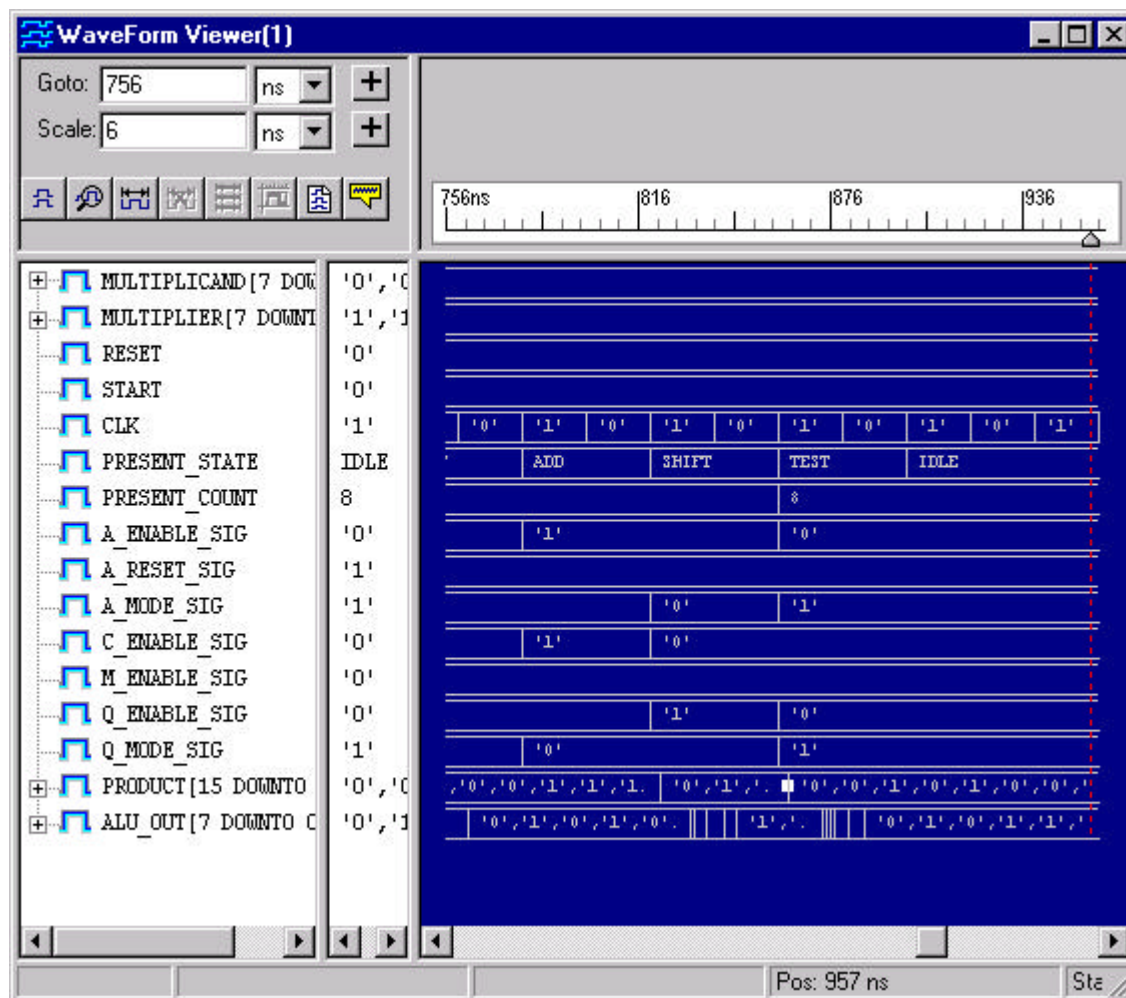
END state_machine;
```

- 2.2. Before the **mult\_controller\_behav.vhdl** model will compile correctly, some configuration of the libraries is needed. Notice the library clause "LIBRARY gate\_lib;" at the beginning of the **mult\_controller\_behav.vhdl** file. This tells the compiler that a library called "gate\_lib" is available for use.
- 2.3. As in the Module 11 lab, the "gate\_lib" library must be mapped to a library stored on disk. Create a library mapping from the logical library name "gate\_lib" to the working library created in module 10.
- 2.4. Once the library mapping is correctly created, compile the **mult\_controller\_behav.vhdl** model. The model should compile without any errors.
- 2.5. Open the file **full\_mult\_str.vhdl** using a text editor or a VHDL editing environment. This is a structural description of an 8-bit multiplier.
- 2.6. Notice that the "mult\_lib" library is used in this model. This library must be mapped to a library stored on disk. Create a library mapping from the logical library name "mult\_lib" to the working library created in module 11.
- 2.7. Once the library mapping is correctly created, compile the **full\_mult\_str.vhdl** model. The model should compile without any errors.
- 2.8. A test bench for the structural multiplier model is provided in the file **full\_mult\_str\_test.vhdl**. Open and compile this test bench file.

### 3. Simulate the multiplier structural model

- 3.1. Start the VHDL simulator. Remember to select the **testbnch** entity as the design root, if required by your simulator.
- 3.2. Open a waveform window, then add the signals from the full multiplier component.

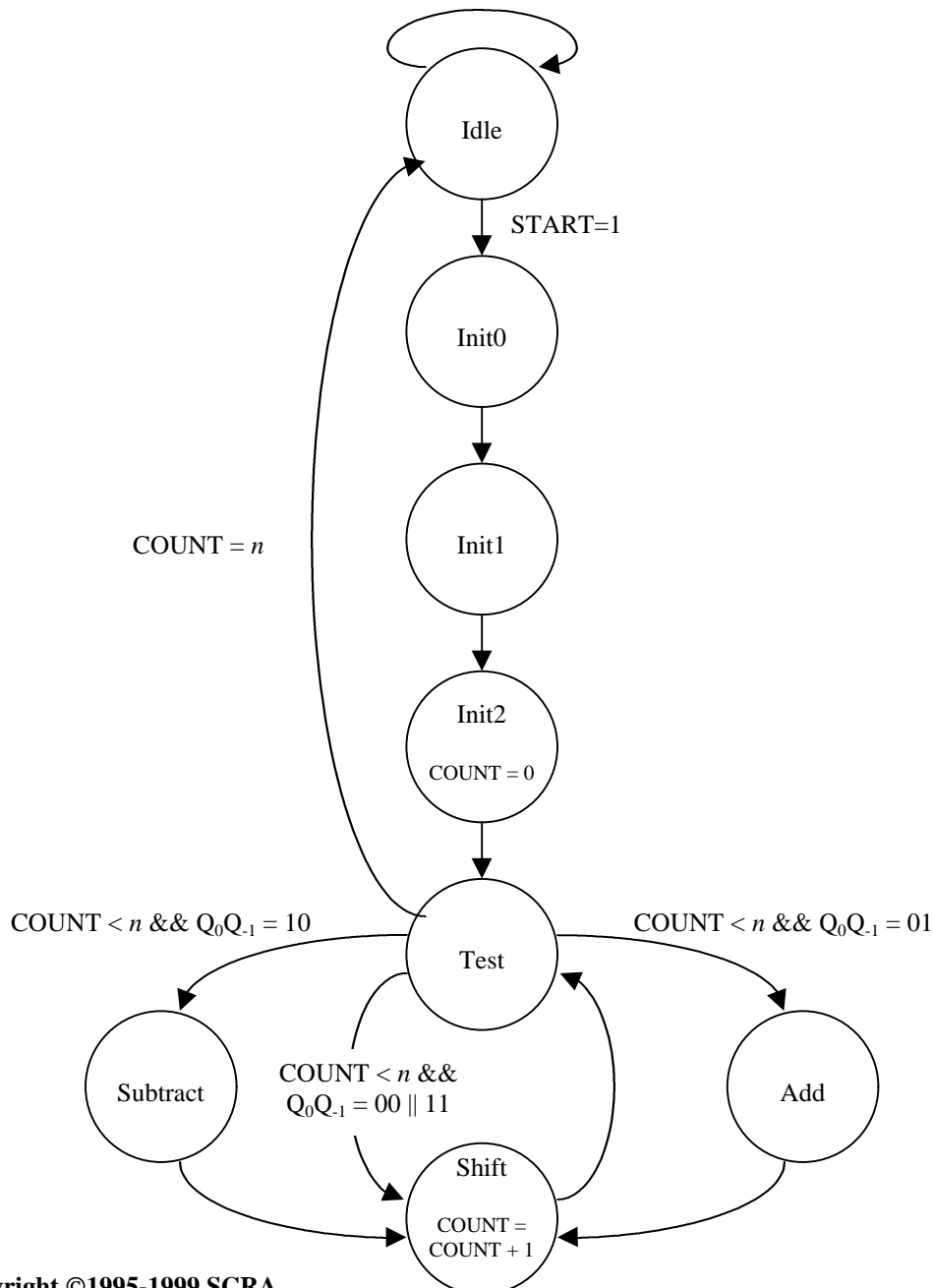
- 3.3. Run the simulator for sufficient time to exercise the circuit (at least 900 nanoseconds in this example). Note that the **present\_state** signal in the control unit (CU) makes many transitions through the ADD, SHIFT, and TEST states before finally reaching IDLE. The signal waveforms should look something like this:



## Module 12 Exercise

### Assignment:

Develop a VHDL behavioral description of a control unit for the Booth's algorithm datapath you designed for Module 11. Note that your datapath will probably require several initialization states before it is ready to start the main loop of the algorithm. A generalized state diagram for the control unit with the outputs eliminated for clarity is shown below:



Develop a structural description for the full Booth's algorithm multiplier that instantiates the datapath and control unit together. Compile and simulate your design.

### Module 12 Tutorial Continued (optional if time permits)

#### 4. Copy, Compile and run the C source code for the quicksort example

4.1. You will begin this example by compiling and running a version of the quicksort routine coded in C. Copy the **qsort.c** source code now.

4.2. Using your favorite text editor, examine the **qsort.c** file:

```

/*****
/*      Quicksort Algorithm Behavioral Example - C Test Code      */
/*      RASSP E&F Module # 12 Behavioral VHDL                    */
/*      Robert Klenke UVa 28 May 1996                            */
*****/

#include <stdio.h>

/* use static allocation */
#define MAXARRAY 100

void quicksort(a,l,r)
int *a;
int l;
int r;
{
    int v, t;
    int i, j;

    if(r>l) {
        v = a[r];
        i = l-1;
        j = r;
        do {
            do {
                i = i + 1;
            } while(a[i]<v);
            do {
                j = j - 1;
            } while(a[j]>v);
            t = a[i];
            a[i] = a[j];
            a[j] = t;
        } while(j>i);
    }
}

```



## Module 12 – Behavioral VHDL Lab Tutorial

```
        a[j] = a[i];
        a[i] = a[r];
        a[r] = t;
        quicksort(a,l,i-1);
        quicksort(a,i+1,r);
    }
}

main(argc,argv)
int argc;
char *argv[];
{
    int nelements, i, tempint, temppointer, min, iarray[MAXARRAY];
    FILE *in_fd, *out_fd;

    if(argc < 2) {
        fprintf(stderr,"Usage:qsort <infile> <outfile>\n");
        exit(0);
    }

    if ((in_fd = fopen(argv[1], "r")) == 0) {
        fprintf(stderr, "Usage: qsort <input_file> <output_file>\n");
        exit(0);
    }
    if ((out_fd = fopen(argv[2], "w")) == 0) {
        fprintf(stderr, "Usage: qsort <input_file> <output_file>\n");
        exit(0);
    }

    /* read in file and set number of elements */
    nelements = 0;
    while((fscanf(in_fd,"%d\n",&iarray[nelements])) != EOF)
        && (nelements < MAXARRAY))
        nelements++;

    /* find mininum element and place in element zero for sentinel */
    tempint = iarray[0];
    temppointer = 0;
    for(i=1;i<nelements;i++)
        if(iarray[i] < tempint) {
            tempint = iarray[i];
            temppointer = i;
        }
    if(temppointer != 0) {
        iarray[temppointer] = iarray[0];
        iarray[0] = tempint;
    }

    /* do the quicksort! */
    quicksort(iarray,0,nelements-1);

    /* write out results */
}
```

## Module 12 – Behavioral VHDL Lab Tutorial

```
    for(i=0;i<nelements;i++)  
        fprintf(out_fd,"%d\n",iarray[i]);  
}
```

- 4.3. Compile *qsort.c* using a command-line C compiler (such as the GNU C compiler). The command line used will depend on the specific compiler, though a common example would look like this:

```
cc -o qsort qsort.c
```

- 4.4. Create an unsorted input file for the quicksort routine using your favorite text editor, for example:

```
40
56
987
9
928374
89273
743
823
52
7
127638
83
2
187
1879
897234
3483
84502
284734
2
8729
947
```

- 4.5. Run the quicksort routine on the **test.in** file:

```
qsort test.in test.out
```

The result will be a **test.out** file like this one:

```
2
2
7
9
40
52
56
```

83  
187  
743  
823  
947  
987  
1879  
3483  
8729  
84502  
89273  
127638  
284734  
897234  
928374

## 5. Copy, Compile and simulate the VHDL source code for the quicksort example

- 5.1. Open the quicksort package file, **qsort\_pkg.vhdl** using a text editor or a VHDL editing environment. Notice the similarity between the quicksort routine in VHDL and the same routine in C. The major difference is the way in which the loops are constructed because of the differences in the available loop constructs in the two languages.
- 5.2. Open the quicksort architecture file, **qsort.vhdl** using a text editor or a VHDL editing environment. Notice that the filenames for the input and output are declared here. Also note that the architecture is coded as a single process statement that executes one time at the beginning of the simulation and then stops at a *WAIT* statement at the end of the process. The VHDL code for this process is also very similar to the corresponding C code for the *main* program.
- 5.3. Compile both **qsort\_pkg.vhdl** and **qsort.vhdl**.
- 5.4. Run the VHDL quicksort routine by starting the VHDL simulation. Remember to run the simulation for some period of time (at least 1 nanosecond).
- 5.5. Verify that the **test.out2** file generated by the VHDL quicksort routine is the same as the **test.out** file produced by the C implementation. You can look at the files using a text editor.