FPGA designs with MyHDL

Meher Krishna Patel

Created on : Octorber, 2017

Last updated : October, 2018

Table of contents

Chapter 1

Overview

1.1 Introduction

In this chapter, we will see various keywords of MyHDL which can be used to create the synthesizable FPGA-designs. We assume the familiarity with the various terms of FPGA designs e.g. signals, process block, always block and reg etc. which are discussed in Verilog/SystemVerilog/VHDL tutorials. Please refer to Verilog/SystemVerilog/VHDL tutorials to understand various terms which are used in FPGA designs.

Similar to Verilog/SystemVerilog/VHDL tutorials, in this chapter, we will implement the '2-bit comparator' using different methods. Please see the Chapter "Overview" of Verilog/VHDL tutorial for the details of the designs.

Important:

- In this tutorial, the designs of Verilog/SystemVerilog/VHDL-tutorials are re-implemented using MyHDL.
- In the comments of MyHDL designs, the corresponding verilog/vhdl-file-names are shown, which can be downloaded from the website.
- Please see the Verilog/VHDL tutorials for better understanding of the resultant designs.

1.1.1 Installation and Codes

Important:

- The tutorial is created with MyHDL 1.0 and Python 3.6.2.
- Click here to donwload the Python codes of the tutorial. Run the Python codes to generate the VHDL/Verilog files.

1.1.2 Keywords

Following is the list of MyHDL-keywords which are used in this tutorial. Also relationship between these keywords with Verilog/VHDL are shown in below table,

Keywords	Verilog	VHDL
Signal	reg, wire	signal
bool	reg, wire	std_logic
intbv(0)[N:0]	reg, wire	unsigned
enum	reg, wire	type
if,else	if,else or case	if,else or case
tuple of int	case statement for ROM	case statement for ROM
list of bool	reg	array of std_logic
list of intbv	reg	array of unsigned
def	module	entity
@always	@always	process
@always_comb	@always*	process(all)
@always_seq	@always with initial values of reg/wire	process with initial values of signals
ResetSignal	used with @always_seq to define reset signal	
posedge/negedge	posedge/negedge	rising_edge/falling_edge

1.2 First design

In this section, we will implement a simple 'and gate' using MyHDL, to see the various components of the MyHDL. This simple example will help us to understand the basic interconnection between the VHDL/Verilog-keywords with MyHDL-keywords.

1.2.1 Define 'and' gate

- Below is the Python code which implements the 'and gate',

```
1  # gateEx.py
2  # the resultant Verilog code is same as Listing 'andEx.v of Verilog tutorial
3
4  from myhdl import *
5
6  # Module : and gate
7  @block
8  def andEx(x, y, z):
9      """ input : x, y
10         output : z
11     """
12
13     # behavior : and gate
14     # i.e. implement and gate using combinational logic
15     @always_comb
16     def and_behave():
17         z.next = x & y # and operation
18
19     # return instances individually
20     # return and_behave
21
22     # this is more convenient as it returns all the instances automatically
23     return instances()
```

Note: Above design can be loosely connected with Verilog/VHDL design as follows,
- The first method, i.e. 'andEx', is similar to module/entity of the Verilog/VHDL design.
- The second method, i.e. 'and_behave', defines the implementation method i.e. combinational design, sequential design or latches, which is similar to always/architecture block of the Verilog/VHDL designs.

– 'z.next = x & y' is same as non-blocking assignments i.e. 'z <= x & y' in Verilog or 'z <= x and y' in VHDL.

1.2.2 Instantiate 'and' gate

The above design needs to be instantiated to convert it into VHDL/Verilog code.

- Run following code to convert the Python code to Verilog and VHDL codes,

```
$ python gateEx_convert.py
```

Note:
– The keyword 'Signal' is similar to signal in VHDL or reg/wire in Verilog
– Note that, the 'x', 'y' and 'z' are not signals, but the ports.

```
# gateEx_convert.py
# convert the gateEx.py to VHDL/Verilog code
# the resultant Verilog code is same as Listing 'andEx.v of Verilog tutorial

from myhdl import *
from gateEx import andEx

x = Signal(bool(0)) # signal of type boolean size 1-bit
y = Signal(bool(0))
z = Signal(bool(0))

# convert into Verilog code
andEx_verilog = andEx(x, y, z)

# positional assigments
# andEx_verilog = andEx(x=x, y=y, z=z)

# initial_values = True will initialize the signal
# note that there is no signal in this design (all are ports)
andEx_verilog.convert(hdl="Verilog", initial_values=True)

# convert into VHDL code : convert without instantiation
andEx(x, y, z).convert(hdl="VHDL", initial_values=True)
```

- The generated Verilog code is shown below,

```
// File: andEx.v
// Generated by MyHDL 1.0dev
// Date: Thu Oct 12 20:34:17 2017

`timescale 1ns/10ps

module andEx (
    x,
    y,
    z
);
// input : x, y
// output : z

input x;
input y;
output z;
wire z;
```

(continues on next page)

(continued from previous page)

```
19
20  assign z = (x & y);
21
22  endmodule
```

- The generated VHDL code is shown below. Note that, a package 'pck_myhdl_10.vhd' is also created by MyHDL, which contains various functions in it.

> **Note:**
> – Do not forget to add the 'pck_myhdl_10.vhd' file in the project while synthesizing the code.
> – Or comment the line 'use work.pck_myhdl_10.all;' from the generated VHDL file, if it is not required.

```
1   -- File: andEx.vhd
2   -- Generated by MyHDL 1.0dev
3   -- Date: Thu Oct 12 20:34:17 2017
4
5
6   library IEEE;
7   use IEEE.std_logic_1164.all;
8   use IEEE.numeric_std.all;
9   use std.textio.all;
10
11  use work.pck_myhdl_10.all;
12
13  entity andEx is
14      port (
15          x: in std_logic;
16          y: in std_logic;
17          z: out std_logic
18      );
19  end entity andEx;
20  -- input : x, y
21  -- output : z
22
23  architecture MyHDL of andEx is
24  begin
25
26  z <= (x and y);
27
28  end architecture MyHDL;
```

1.2.3 Multiple designs in one file

- Unlike VHDL/Verilog designs, in MyHDL we can add multiple modules in a single file. In the below code, two modules i.e. 'andEx' and 'xorEx' are added in the single file i.e. 'gateEx.py'.

```
1   # gateEx.py
2   # the resultant Verilog code is same as Listing 'andEx.v of Verilog tutorial
3
4   from myhdl import *
5
6   # Module : and gate
7   @block
8   def andEx(x, y, z):
9       """ input : x, y
10          output : z
11      """
```

(continues on next page)

(continued from previous page)

```
12
13      # behavior : and gate
14      # i.e. implement and gate using combinational logic
15      @always_comb
16      def and_behave():
17          z.next = x & y # and operation
18
19      # return instances individually
20      # return and_behave
21
22      # this is more convenient as it returns all the instances automatically
23      return instances()
24
25  # Module : xor gate
26  @block
27  def xorEx(x, y, z):
28      """ input : x, y
29          output : z
30      """
31
32      # behavior : xor gate
33      # i.e. implement xor gate using combinational logic
34      @always_comb
35      def xor_behave():
36          z.next = x ^ y # xor operation
37
38      # return instances individually
39      # return xor_behave
40
41      # this is more convenient as it returns all the instances automatically
42      return instances()
```

- Then 'gateEx_convert.py' file is used to generate the Verilog code for these two modules.

```
1   # gateEx_convert.py
2   # convert the gateEx.py to VHDL/Verilog code
3   # the resultant Verilog code is same as Listing 'andEx.v of Verilog tutorial
4
5   from myhdl import *
6   from gateEx import andEx, xorEx
7
8   x = Signal(bool(0)) # signal of type boolean size 1-bit
9   y = Signal(bool(0))
10  z = Signal(bool(0))
11
12  # convert into Verilog code
13  andEx_verilog = andEx(x, y, z)
14
15  # initial_values = True will initialize the signal
16  # note that there is no signal in this design (all are ports)
17  andEx_verilog.convert(hdl="Verilog", initial_values=True)
18
19  # convert into VHDL code  : convert without instantiation
20  andEx(x, y, z).convert(hdl="VHDL", initial_values=True)
21
22
23  # convert into Verilog code
24  xorEx_verilog = xorEx(x, y, z)
25
26  # initial_values = True will initialize the signal
27  # note that there is no signal in this design (all are ports)
28  xorEx_verilog.convert(hdl="Verilog", initial_values=True)
```

(continues on next page)

(continued from previous page)

```
29
30 # convert into VHDL code  : convert without instantiation
31 xorEx(x, y, z).convert(hdl="VHDL", initial_values=True)
```

1.2.4 Design and conversion codes in the same files

It is handy to keep the designs and conversion codes in the same file, rather than writing each piece of code in separate files. In this section, we merge the design and conversion code of the 'and gate' and 'xor gate' in one file.

- For writing the design and conversion code in the same file, we need to define one boilerplate, i.e. " if __name__ = '__main__':, which tells the python that the entry point of the code is the 'main()' function, as shown below.
- If we execute the file 'gateEx.py', then the results will be same as previous sections.

```
1  # gateEx.py
2  # the resultant Verilog code is same as Listing 'andEx.v of Verilog tutorial
3
4  from myhdl import *
5
6  # Module : and gate
7  @block
8  def andEx(x, y, z):
9      """ input : x, y
10         output : z
11     """
12
13     # behavior : and gate
14     # i.e. implement and gate using combinational logic
15     @always_comb
16     def and_behave():
17         z.next = x & y # and operation
18
19     # return instances individually
20     # return and_behave
21
22     # this is more convenient as it returns all the instances automatically
23     return instances()
24
25 # Module : xor gate
26 @block
27 def xorEx(x, y, z):
28     """ input : x, y
29         output : z
30     """
31
32     # behavior : xor gate
33     # i.e. implement xor gate using combinational logic
34     @always_comb
35     def xor_behave():
36         z.next = x ^ y # xor operation
37
38     # return instances individually
39     # return xor_behave
40
41     # this is more convenient as it returns all the instances automatically
42     return instances()
43
44
45 def main():
```

(continues on next page)

(continued from previous page)

```
    x = Signal(bool(0)) # signal of type boolean size 1-bit
    y = Signal(bool(0))
    z = Signal(bool(0))

    # convert into Verilog code
    andEx_verilog = andEx(x, y, z)

    # initial_values = True will initialize the signal
    # note that there is no signal in this design (all are ports)
    andEx_verilog.convert(hdl="Verilog", initial_values=True)

    # convert into VHDL code  : convert without instantiation
    andEx(x, y, z).convert(hdl="VHDL", initial_values=True)


    # convert into Verilog code
    xorEx_verilog = xorEx(x, y, z)

    # initial_values = True will initialize the signal
    # note that there is no signal in this design (all are ports)
    xorEx_verilog.convert(hdl="Verilog", initial_values=True)

    # convert into VHDL code  : convert without instantiation
    xorEx(x, y, z).convert(hdl="VHDL", initial_values=True)

if __name__ == '__main__':
    main()
```

1.2.5 Updated port-names for pin assignments using .csv file

In previous section, we implemented the comparator using various methods and converted those designs to Verilog codes. Also, we learn to implement the structural modeling using MyHDL.

Note:

- In this section, we will create the top level design with updated port names (according to pin-assignment .csv file, provided in Verilog/VHDL tutorials), so that we need not to change the pin-names in the csv file according to each new design.
- Also, the different signal values (i.e. SW and output value) are assigned to single output port (i.e. LEDG). For this, we need to create one additional 'always' block as shown in the module 'top_xorEx'.
- Further unlike VHDL/Verilog, for two switches SW[2:0] is used in MyHDL (not SW[1:0]).
- The keyword 'intbv' is used to define the vector-signal instead of 'bool', as the 'bool' keyword is not sub-scriptable.
- Lastly, if we want to change the port name for 'and gate' as well, then we need to write another top level module for andEx.

- In the below code, a top level design is created for the design in Section 1.2. Following port-mapping is done here,
  - For the 'xor gate', the input ports are connect to SW.
  - The inputs and output of 'xor gate' are connected to LEDG.

```
# gateEx.py
# the resultant Verilog code is same as Listing 'andEx.v of Verilog tutorial

from myhdl import *

# Module : and gate
```

(continues on next page)

(continued from previous page)

```
@block
def andEx(x, y, z):
    """ input : x, y
        output : z
    """

    # behavior : and gate
    # i.e. implement and gate using combinational logic
    @always_comb
    def and_behave():
        z.next = x & y # and operation

    # return instances individually
    # return and_behave

    # this is more convenient as it returns all the instances automatically
    return instances()

# Module : xor gate
@block
def xorEx(x, y, z):
    """ input : x, y
        output : z
    """

    # behavior : xor gate
    # i.e. implement xor gate using combinational logic
    @always_comb
    def xor_behave():
        z.next = x ^ y # xor operation

    # return instances individually
    # return xor_behave

    # this is more convenient as it returns all the instances automatically
    return instances()


# top level entity for 'xor' gate
@block
def top_xorEx(SW, LEDG):
    # instantiate xorEx : display only output on green LED (LEDG)
    xorEx_verilog = xorEx(x=SW(0), y=SW(1), z=LEDG)

    return instances()

def main():
    x = Signal(bool(0)) # signal of type boolean size 1-bit
    y = Signal(bool(0))
    z = Signal(bool(0))

    switch = Signal(intbv(0)[2:0]) # 2 switches
    led = Signal(intbv(0)[3:0]) # 3 green LED

    # convert into Verilog code
    andEx_verilog = andEx(x, y, z)

    # initial_values = True will initialize the signal
    # note that there is no signal in this design (all are ports)
    andEx_verilog.convert(hdl="Verilog", initial_values=True)

```

(continues on next page)

(continued from previous page)

```
    # convert into VHDL code  : convert without instantiation
    andEx(x, y, z).convert(hdl="VHDL", initial_values=True)


    # convert into Verilog code
    xorEx_verilog = xorEx(x, y, z)

    # initial_values = True will initialize the signal
    # note that there is no signal in this design (all are ports)
    xorEx_verilog.convert(hdl="Verilog", initial_values=True)

    # convert into VHDL code  : convert without instantiation
    xorEx(x, y, z).convert(hdl="VHDL", initial_values=True)

    # modified port name according to pin-assignment file
    top_xorEx(switch, led).convert(hdl="Verilog", initial_values=True)

if __name__ == '__main__':
    main()
```

- The corresponding Verilog code for "xor gate" is shown below. Now, the below can be loaded on the FPGA with the pin-assignment file provided in the Verilog/VHDL tutorials.

```
// File: top_xorEx.v
// Generated by MyHDL 1.0dev
// Date: Thu Oct 12 21:40:22 2017

`timescale 1ns/10ps

module top_xorEx (
    SW,
    LEDG
);

input [1:0] SW;
output [2:0] LEDG;
wire [2:0] LEDG;

assign LEDG = (SW[0] ^ SW[1]);

endmodule
```

1.3 Comparator

In this section, we will implement the 1-bit comparator and the 2-bit comparator with different modeling styles,

1.3.1 1-bit comparator

- The '@always' decorator along with sensitivity list can be used for implementing the procedural assignments; whereas '@always_comb' without sensitivity list can be used to implement the 'continuous assignment'. In this section, the 1-bit comparator is implemented using procedural and continuous assignments.

Important:

- The continuous or sequential assignments are inferred automatically based on the logics inside the block. More specifically, the combinational design can be implemented by using 'sequential statements' or 'continuous statements' based on the logic inside the 'always_comb' block.

– Further, both the designs, i.e. combinational and procedural, are written in one file (comparator_1_bit.py), but the corresponding Verilog designs are generated as separate files (comparator_1_bit_proc.v and comparator_1_bit_comb.v).

- Below code will generate the Verilog code with continuous and procedural assignments,

```
# comparator_1_bit.py

# the resultant Verilog code is same as Listing 'comparator1Bit.v
# of Verilog tutorial

from myhdl import *

# 1-bit comparator using procedural assignment
@block
def comparator_1_bit_proc(x, y, eq):
    """ x, y : input
        eq : output

        eq = 1 when x = y else 0
    """

    s0 = Signal(bool(0))
    s1 = Signal(bool(0))

    @always(s0, s1)
    def comparator_1_bit_behave():
        s0.next = ~x & ~y
        s1.next = x & y
        eq.next = s0 | s1

    return comparator_1_bit_behave


# 1-bit comparator using continuous assignment
@block
def comparator_1_bit_comb(x, y, eq):
    """ x, y : input
        eq : output

        eq = 1 when x = y else 0
    """

    @always_comb
    def comparator_1_bit_behave():
        eq.next = (~x & ~y) | (x & y)

    return comparator_1_bit_behave

def main():
    x = Signal(bool(0)) # signal of type boolean size 1 bit
    y = Signal(bool(0))
    eq = Signal(bool(0))

    # convert into Verilog code
    comparator_verilog_procedure = comparator_1_bit_proc(x=x, y=y,
                eq=eq).convert(hdl="Verilog", initial_values=True)
    comparator_verilog_continuous = comparator_1_bit_comb(x=x, y=y,
                eq=eq).convert(hdl="Verilog", initial_values=True)

if __name__ == '__main__':
    main()
```

- Two separate verilog files will be generated by above code, which are shown below.

> **Note:** Since we set the "initial_values=True" therefore the signals s0 and s1 are initialized with 0 in the file "comparator_1_bit_proc.v".

```
// File: comparator_1_bit_proc.v
// Generated by MyHDL 1.0dev
// Date: Fri Oct 13 07:52:12 2017


`timescale 1ns/10ps

module comparator_1_bit_proc (
    x,
    y,
    eq
);
// x, y : input
// eq : output
//
// eq = 1 when x = y else 0

input x;
input y;
output eq;
reg eq;

reg s1 = 0;
reg s0 = 0;

always @(s0, s1) begin: COMPARATOR_1_BIT_PROC_COMPARATOR_1_BIT_BEHAVE
    s0 <= ((~x) & (~y));
    s1 <= (x & y);
    eq <= (s0 | s1);
end

endmodule
```

```
// File: comparator_1_bit_comb.v
// Generated by MyHDL 1.0dev
// Date: Fri Oct 13 07:52:12 2017


`timescale 1ns/10ps

module comparator_1_bit_comb (
    x,
    y,
    eq
);
// x, y : input
// eq : output
//
// eq = 1 when x = y else 0

input x;
input y;
output eq;
wire eq;

assign eq = (((~x) & (~y)) | (x & y));
```

(continues on next page)

(continued from previous page)

```
24
25 endmodule
```

1.3.2 2-bit comparator

- Following code implements the two bit comparator,

```
1  # comparator_2_bit.py
2  # the resultant Verilog code is same as Listing 'comparator2Bit.v
3  # of Verilog tutorial
4
5  from myhdl import *
6
7  @block
8  def comparator_2_bit(x, y, eq):
9      """ x, y : input
10         eq : output
11
12         eq = 1 when x = y else 0
13     """
14
15     # 'intbv' is used instead of 'bool' as 'bool' is not subscriptable
16     s = Signal(intbv(0)[4:0])
17
18     @always(s)
19     def comparator_2_bit_behave():
20         s[0].next =  ~x[1] & ~x[0] & ~y[1] & ~y[0]
21         s[1].next =  ~x[1] & x[0] & ~y[1] & y[0]
22         s[2].next =  x[1] & ~x[0] & y[1] & ~y[0]
23         s[3].next =  x[1] & x[0] & y[1] & y[0]
24         eq.next = s[0] | s[1] | s[2] | s[3]
25
26     return comparator_2_bit_behave
27
28 def main():
29     # 'intbv' is used instead of 'bool' as 'bool' is not subscriptable
30     x = Signal(intbv(0)[2:0]) # signal of type boolean size 1-bit
31     y = Signal(intbv(0)[2:0])
32     eq = Signal(bool(0))
33
34     # convert into Verilog code
35     comparator_verilog = comparator_2_bit(x=x, y=y, eq=eq)
36     comparator_verilog.convert(hdl="Verilog", initial_values=True)
37
38 if __name__ == '__main__':
39     main()
```

- The resultant Verilog code is shown below,

```
1  // File: comparator_2_bit.v
2  // Generated by MyHDL 1.0dev
3  // Date: Fri Oct 13 08:21:35 2017
4
5
6  `timescale 1ns/10ps
7
8  module comparator_2_bit (
9      x,
10     y,
11     eq
```

(continues on next page)

(continued from previous page)

```
);
// x, y : input
// eq : output
//
// eq = 1 when x = y else 0

input [1:0] x;
input [1:0] y;
output eq;
reg eq;

reg [3:0] s = 0;

always @(s) begin: COMPARATOR_2_BIT_COMPARATOR_2_BIT_BEHAVE
    s[0] <= ((((~x[1]) & (~x[0])) & (~y[1])) & (~y[0]));
    s[1] <= ((((~x[1]) & x[0]) & (~y[1])) & y[0]);
    s[2] <= (((x[1] & (~x[0])) & y[1]) & (~y[0]));
    s[3] <= (((x[1] & x[0]) & y[1]) & y[0]);
    eq <= (((s[0] | s[1]) | s[2]) | s[3]);
end

endmodule
```

1.3.3 Structural modeling

In structural modeling the predefined designs are connected together to create the larger system. In this section, we will use two *1-bit comparator* to create a 2-bit comparator.

Note: Unlike VHDL/Verilog, the structural model defined in the MyHDL does not instantiate the smaller unit but re-implement it, as shown in this section.

- Two bit comparator design is shown below,

Warning:

- We can not use the 'intbv' to define the **output-signal** to use it in the instantiation by "**slicing it**", i.e. below line can not be synthesized.
- "Signal has multiple driver error" is shown for this, as a part of the slice will be updated by one instantiation, whereas the other part will be updated at other instantiation. In the other words, the output signal will be updated at different 'always-block' of Verilog.
- Also, note that for input signal small brackets are used i.e a(0), b(0).

```
s = Signal(intbv(0)[2:0])
eq0 = comparator_1_bit(x=a(0), y=b(0), eq=s(0)) # this will not work
```

```
# comparator_2_bit_struct.py
# this code is similar to comparator2BitStruct.v of verilog tutorial

from myhdl import *
from comparator_1_bit import comparator_1_bit_proc

@block
def comparator_2_bit_struct(a, b, eq):

    # s = Signal(intbv(0)[2:0]) # this does not work
    s0 = Signal(bool(0))
```

(continues on next page)

(continued from previous page)

```
12      s1 = Signal(bool(0))
13
14      # instantiation : 1-bit comparator
15      # note square brackets are not used i.e. a(0) and b(0)
16      eq0 = comparator_1_bit_proc(x=a(0), y=b(0), eq=s0)
17      eq1 = comparator_1_bit_proc(x=a(1), y=b(1), eq=s1)
18
19      @always(s0, s1)
20      def comparator_2_bit_behave():
21          if ((s0 == 1) & (s1 == 1)) :
22              eq.next = 1
23          else :
24              eq.next = 0
25      return comparator_2_bit_behave, eq0, eq1
26
27  def main():
28      x = Signal(intbv(0)[2:0])
29      y = Signal(intbv(0)[2:0])
30      eq = Signal(bool(0))
31
32      # convert into Verilog code
33      comparator_verilog = comparator_2_bit_struct(a=x, b=y, eq=eq)
34      comparator_verilog.convert(hdl="Verilog", initial_values=True)
35
36  if __name__ == '__main__' :
37      main()
```

- The resultant Verilog code is shown below.

> **Warning:** Note that the 1-bit comparator design is not instantiated here, but reimplemented in the 2-bit comparator (see lines 39-43 and 46-50).

```
1   // File: comparator_2_bit_struct.v
2   // Generated by MyHDL 1.0dev
3   // Date: Fri Oct 13 11:57:42 2017
4
5
6   `timescale 1ns/10ps
7
8   module comparator_2_bit_struct (
9       a,
10      b,
11      eq
12  );
13
14
15
16  input [1:0] a;
17  input [1:0] b;
18  output eq;
19  reg eq;
20
21  reg s1 = 0;
22  reg s0 = 0;
23  reg comparator_1_bit_proc_1_s1 = 0;
24  reg comparator_1_bit_proc_1_s0 = 0;
25  reg comparator_1_bit_proc_2_s1 = 0;
26  reg comparator_1_bit_proc_2_s0 = 0;
27
28
```

(continues on next page)

(continued from previous page)

```
29 always @(s0, s1) begin: COMPARATOR_2_BIT_STRUCT_COMPARATOR_2_BIT_BEHAVE
30     if (((s0 == 1) & (s1 == 1))) begin
31         eq <= 1;
32     end
33     else begin
34         eq <= 0;
35     end
36 end
37
38
39 always @(comparator_1_bit_proc_1_s0, comparator_1_bit_proc_1_s1) begin: COMPARATOR_2_BIT_
   ↪STRUCT_COMPARATOR_1_BIT_PROC_1_COMPARATOR_1_BIT_BEHAVE
40     comparator_1_bit_proc_1_s0 <= ((~a[0]) & (~b[0]));
41     comparator_1_bit_proc_1_s1 <= (a[0] & b[0]);
42     s0 <= (comparator_1_bit_proc_1_s0 | comparator_1_bit_proc_1_s1);
43 end
44
45
46 always @(comparator_1_bit_proc_2_s0, comparator_1_bit_proc_2_s1) begin: COMPARATOR_2_BIT_
   ↪STRUCT_COMPARATOR_1_BIT_PROC_2_COMPARATOR_1_BIT_BEHAVE
47     comparator_1_bit_proc_2_s0 <= ((~a[1]) & (~b[1]));
48     comparator_1_bit_proc_2_s1 <= (a[1] & b[1]);
49     s1 <= (comparator_1_bit_proc_2_s0 | comparator_1_bit_proc_2_s1);
50 end
51
52 endmodule
```

1.4 Conclusion

In this chapter, we saw various features of MyHDL to implement the synthesizable designs. The @always and @always_comb blocks are used for implementing the continuous assignment, procedural assignment and structural designs. Also, we saw basic differences between the pure Verilog design and the Verilog design generated by the MyHDL. Further, we saw the method to create the top level module according to 'pin-assignment-file'. Lastly, we learn the method by which the design and it's conversion functions can be written in the same file.

Chapter 2

Visual verification of designs

2.1 Introduction

Note: In this chapter, a Mod-m counter is implemented and the counting is displayed on the LEDs and seven-segment-display device. Since, we want to visualize the output on these devices, therefore the reduced clock rate is required to visualize the counts.

In this chapter, we will re-implement the designs of the Chapter "Visual verification of designs" in the Verilog/VHDL tutorial. Please see the Verilog/SystemVerilog/VHDL tutorials for the explanation of the designs.

2.2 Flip flops

In this section, two types of flip flops are implemented.

2.2.1 Basic D flip flop

- In the following code, the D-FF is implemented which transfer the input value to the output port after the delay of one clock cycle.

```
1  # basic_dff.py
2  # results is same as "basicDff.v" in the Verilog tutorial
3
4  from myhdl import *
5
6  @block
7  def basic_dff(clk, reset_n, d, q):
8      """ clk, reset, d : input
9          q : output
10     """
11
12     @always(clk.posedge, reset_n.negedge)
13     def dff_behave():
14         if reset_n == 0 :
15             q.next = 0
16         else :
17             q.next = d
18     return dff_behave
19
20
```

(continues on next page)

(continued from previous page)

```
def main():
    clk = Signal(bool(0))
    reset_n = Signal(bool(0))
    d = Signal(bool(0))
    q = Signal(bool(0))

    top_dff = basic_dff(clk, reset_n, d, q)
    top_dff.convert(hdl="Verilog", initial_values=True)

# function main() is the entry point
if __name__ == '__main__':
    main()
```

- The resultant Verilog code is shown below,

```
// File: basic_dff.v
// Generated by MyHDL 1.0dev
// Date: Fri Oct 13 13:56:20 2017


`timescale 1ns/10ps

module basic_dff (
    clk,
    reset_n,
    d,
    q
);
// clk, reset, d : input
// q : output

input clk;
input reset_n;
input d;
output q;
reg q;

always @(posedge clk, negedge reset_n) begin: BASIC_DFF_DFF_BEHAVE
    if ((reset_n == 0)) begin
        q <= 0;
    end
    else begin
        q <= d;
    end
end

endmodule
```

2.2.2 D-FF with enable port

- In the below code, the 'enable' port is added to D-FF, so that input value is transferred to output port when the enable signal is high.

```
# d_ff.py
# results is same as "D_FF.v" in the Verilog tutorial

from myhdl import *

@block
def dff_unit(clk, reset_n, en, d, q):
```

(continues on next page)

(continued from previous page)

```
    """ clk, reset, en, d : input
        q : output
    """

    @always(clk.posedge, reset_n.negedge)
    def dff_behave():
        if reset_n == 0 :
            q.next = 0
        # else :
            # q.next = d
        elif  en == 1 :
            q.next = d
    return dff_behave

# below module is created to change the name of the port
# i.e. clk is changed to CLOCK_50
@block
def d_ff(CLOCK_50, reset_n, en, d, q):
    dff_top = dff_unit(CLOCK_50, reset_n, en, d, q)
    return dff_top

def main():
    clk = Signal(bool(0))
    reset_n = Signal(bool(0))
    en = Signal(bool(0))
    d = Signal(bool(0))
    q = ResetSignal(0, active=0, async=True)

    top_dff2 = d_ff(clk, reset_n, en, d, q)
    top_dff2.convert(hdl="Verilog", initial_values=True)

# function main() is the entry point
if __name__ == '__main__':
    main()
```

- The resultant Verilog code is shown below,

```
// File: d_ff.v
// Generated by MyHDL 1.0dev
// Date: Fri Oct 13 14:05:54 2017


`timescale 1ns/10ps

module d_ff (
    CLOCK_50,
    reset_n,
    en,
    d,
    q
);


input CLOCK_50;
input reset_n;
input en;
input d;
output q;
reg q;


always @(posedge CLOCK_50, negedge reset_n) begin: D_FF_DFF_UNIT_1_DFF_BEHAVE
```

(continues on next page)

(continued from previous page)

```
    if ((reset_n == 0)) begin
        q <= 0;
    end
    else if ((en == 1)) begin
        q <= d;
    end
end

endmodule
```

2.3 Mod-m Counter

In this section, a Mod M counter is implemented. Further, a testbench for the Mod M counter is shown in Section 3.1.

- Following is the Python code for Mod M counter,

Listing 2.1: mod_m_counter.py

```
# mod_m_counter.py
# result is similar to 'modMCounter.v' in the verilog tutorial

from myhdl import *

@block
def mod_m_counter(clk, reset_n, complete_tick, count, M, N):
    """ M = max count
        N = minimum bits required to represent M
    """

    count_reg = Signal(intbv(0)[N:0])
    count_next = Signal(intbv(0)[N:0])

    @always(clk.posedge, reset_n.negedge)
    def logic_reg():
        if reset_n == 0 :
            count_reg.next = 0
        else :
            count_reg.next = count_next

    @always_comb
    def logic_next():
        if count_reg == M-1 :
            count_next.next = 0
            complete_tick.next = 1
        else :
            count_next.next = count_reg + 1
            complete_tick.next = 0

    @always_comb
    def out_val():
        count.next = count_reg

    return out_val, logic_next, logic_reg


def main():
    N = 3
    M = 5
```

(continues on next page)

(continued from previous page)

```
41
42      clk = Signal(bool(0))
43      reset_n = Signal(bool(0))
44      complete_tick = Signal(bool(0))
45      count = Signal(intbv(0)[N:0])
46      mod_m_counter_v = mod_m_counter(clk, reset_n, complete_tick, count, M, N)
47      mod_m_counter_v.convert(hdl="Verilog", initial_values=True)
48
49  if __name__ == '__main__':
50      main()
```

- The resultant Verilog code is shown below,

```
1   // File: mod_m_counter.v
2   // Generated by MyHDL 1.0dev
3   // Date: Fri Oct 13 14:17:35 2017
4
5
6   `timescale 1ns/10ps
7
8   module mod_m_counter (
9       clk,
10      reset_n,
11      complete_tick,
12      count
13  );
14  // M = max count
15  // N = minimum bits required to represent M
16
17  input clk;
18  input reset_n;
19  output complete_tick;
20  reg complete_tick;
21  output [2:0] count;
22  wire [2:0] count;
23
24  reg [2:0] count_reg = 0;
25  reg [2:0] count_next = 0;
26
27
28  assign count = count_reg;
29
30
31  always @(count_reg) begin: MOD_M_COUNTER_LOGIC_NEXT
32      if (($signed({1'b0, count_reg}) == (5 - 1))) begin
33          count_next = 0;
34          complete_tick = 1;
35      end
36      else begin
37          count_next = (count_reg + 1);
38          complete_tick = 0;
39      end
40  end
41
42
43  always @(posedge clk, negedge reset_n) begin: MOD_M_COUNTER_LOGIC_REG
44      if ((reset_n == 0)) begin
45          count_reg <= 0;
46      end
47      else begin
48          count_reg <= count_next;
49      end
```

(continues on next page)

(continued from previous page)

```
50 end
51
52 endmodule
```

2.4 Clock ticks

To visualize the outputs, e.g. counting etc., we need to reduce the clock rate of the system; otherwise the changes will be too fast to observe.

Warning:

- The signal 'count' is not used anywhere in the 'clock_tick.py' but it is defined. Unlike VHDL, we can not keep the port Open in MyHDL. And a warning will be generated for the unused signal.
- But 'open port' can be achieved by using 'None' keyword as shown in Section 2.7; also, it will remove the warning.

- The Python code for generating the clock-ticks with reduced clock-rate are shown below,

```
1  # clock_tick.py
2  # result is similar to 'clockTick.v' in the verilog tutorial
3
4  from myhdl import *
5
6  from mod_m_counter import mod_m_counter
7
8  @block
9  def clock_tick(clk, reset_n, clk_pulse, M, N):
10     count = Signal(intbv(0)[N:0])
11     mod_m_counter_inst = mod_m_counter(clk, reset_n, clk_pulse, count, M, N)
12     return mod_m_counter_inst
13
14 # instantiate clock_tick
15 def main():
16     clk = Signal(bool(0))
17     reset_n = Signal(bool(0))
18     clk_pulse = Signal(bool(0))
19     M = 5000000 # 0.1 sec
20     N = 29
21
22     clock_tick_inst = clock_tick(clk, reset_n, clk_pulse, M, N)
23     clock_tick_inst.convert(hdl="Verilog", initial_values=True)
24
25 if __name__ == '__main__':
26     main()
```

- The corresponding Verilog code is shown below,

```
1  // File: clock_tick.v
2  // Generated by MyHDL 1.0dev
3  // Date: Fri Oct 13 14:23:19 2017
4
5
6  `timescale 1ns/10ps
7
8  module clock_tick (
9      clk,
10     reset_n,
11     clk_pulse
```

(continues on next page)

(continued from previous page)

```
);


input clk;
input reset_n;
output clk_pulse;
reg clk_pulse;

wire [28:0] count;
reg [28:0] mod_m_counter_1_count_reg = 0;
reg [28:0] mod_m_counter_1_count_next = 0;


assign count = mod_m_counter_1_count_reg;


always @(mod_m_counter_1_count_reg) begin: CLOCK_TICK_MOD_M_COUNTER_1_LOGIC_NEXT
    if (($signed({1'b0, mod_m_counter_1_count_reg}) == (5000000 - 1))) begin
        mod_m_counter_1_count_next = 0;
        clk_pulse = 1;
    end
    else begin
        mod_m_counter_1_count_next = (mod_m_counter_1_count_reg + 1);
        clk_pulse = 0;
    end
end


always @(posedge clk, negedge reset_n) begin: CLOCK_TICK_MOD_M_COUNTER_1_LOGIC_REG
    if ((reset_n == 0)) begin
        mod_m_counter_1_count_reg <= 0;
    end
    else begin
        mod_m_counter_1_count_reg <= mod_m_counter_1_count_next;
    end
end

endmodule
```

2.5 Seven segment display

- Below code convert the 'hexadecimal number' into seven-segment-display format,

```
# hex_to_seven_seg.py
# result is same as 'hexToSevenSegment.v' and
# 'hexToSevenSegment_testCircuit.v' in the Verilog tutorial

from myhdl import *

@block
def hex_to_seven_seg(hex_num, seven_seg):

    @always(hex_num)
    def logic():
        if hex_num == 0:
            seven_seg.next = 0b1000000
        elif hex_num == 1 :
            seven_seg.next = 0b1111001
```

(continues on next page)

(continued from previous page)

```
        elif hex_num == 2 :
            seven_seg.next = 0b0100100
        elif hex_num == 3 :
            seven_seg.next = 0b0110000
        elif hex_num == 4 :
            seven_seg.next = 0b0011001
        elif hex_num == 5 :
            seven_seg.next = 0b0010010
        elif hex_num == 6 :
            seven_seg.next = 0b0000010
        elif hex_num == 7 :
            seven_seg.next = 0b1111000
        elif hex_num == 8 :
            seven_seg.next = 0b0000000
        elif hex_num == 9 :
            seven_seg.next = 0b0010000
        elif hex_num == 10 :
            seven_seg.next = 0b0001000
        elif hex_num == 11 :
            seven_seg.next = 0b0000011
        elif hex_num == 12 :
            seven_seg.next = 0b1000110
        elif hex_num == 13 :
            seven_seg.next = 0b0100001
        elif hex_num == 14 :
            seven_seg.next = 0b0000110
        else :
        # elif hex_num == 15 :
            seven_seg.next = 0b0001110

    return logic

# instantiate hex_to_seven_seg with new port names
@block
def hex_to_seven_seg_test(SW, HEX0):
    hex_to_seven_seg_inst = hex_to_seven_seg(SW, HEX0)
    return hex_to_seven_seg_inst

# instantiate hex_to_seven_seg_test
def main():
    hex_num = Signal(intbv(0)[4:0])
    seven_seg = Signal(intbv(0)[7:0])

    hex_ss_v = hex_to_seven_seg_test(hex_num, seven_seg)
    hex_ss_v.convert(hdl="Verilog", initial_values=True)


if __name__ == '__main__':
    main()
```

- Below is the resultant Verilog code.

Note:

- Note that, the if-else statement of Python code is automatically mapped into case-statement in the Verilog code.
- Load this code to FPGA board and change the SW pattern and see the corresponding numbers on LEDG and seven-segment-display device.

```
// File: hex_to_seven_seg_test.v
// Generated by MyHDL 1.0dev
```

(continues on next page)

(continued from previous page)

```
// Date: Fri Oct 13 14:28:15 2017


`timescale 1ns/10ps

module hex_to_seven_seg_test (
    SW,
    HEX0
);


input [3:0] SW;
output [6:0] HEX0;
reg [6:0] HEX0;




always @(SW) begin: HEX_TO_SEVEN_SEG_TEST_HEX_TO_SEVEN_SEG_1_LOGIC
    case (SW)
        'h0: begin
            HEX0 <= 64;
        end
        'h1: begin
            HEX0 <= 121;
        end
        'h2: begin
            HEX0 <= 36;
        end
        'h3: begin
            HEX0 <= 48;
        end
        'h4: begin
            HEX0 <= 25;
        end
        'h5: begin
            HEX0 <= 18;
        end
        'h6: begin
            HEX0 <= 2;
        end
        'h7: begin
            HEX0 <= 120;
        end
        'h8: begin
            HEX0 <= 0;
        end
        'h9: begin
            HEX0 <= 16;
        end
        'ha: begin
            HEX0 <= 8;
        end
        'hb: begin
            HEX0 <= 3;
        end
        'hc: begin
            HEX0 <= 70;
        end
        'hd: begin
            HEX0 <= 33;
```

(continues on next page)

(continued from previous page)

```
        end
        'he: begin
            HEX0 <= 6;
        end
        default: begin
            HEX0 <= 14;
        end
    endcase
end

endmodule
```

2.6 Visual verification of Mod-m counter

In this section, we will connect all the previous designs i.e. mod_m_counter, hex_to_seven_seg and clock_tick to count the number upto 'M' and display the result on LEDs and seven-segment-display device.

- Below is the top level entity which connects all the designs together and convert it into Verilog code,

```
# mod_m_counter_visual_test.py
# result is same as th modMCounter_VisualTest.v of the Verilog tutorial

from myhdl import *

from mod_m_counter import mod_m_counter
from clock_tick import clock_tick
from hex_to_seven_seg import hex_to_seven_seg_test

@block
def mod_m_counter_visual_test(CLOCK_50, reset_n, HEX0, LEDG, M, N):
    count = Signal(intbv(0)[N:0])
    clk_pulse = Signal(bool(0))
    complete_tick = Signal(bool(0))

    # reduced clock rate
    clock_tick_inst = clock_tick(CLOCK_50, reset_n, clk_pulse, 50000000, 29)

    # reduce clock rate is applied to mod_m_counter
    mod_m_counter_inst = mod_m_counter(clk_pulse, reset_n,
            complete_tick, count, M, N)

    # count is send for conversion
    hex_to_seven_seg_test_inst = hex_to_seven_seg_test(count, HEX0)

    # send count to LEDG
    @always_comb
    def led_logic():
        LEDG.next = count

    return instances()

def main():
    CLOCK_50 = Signal(bool(0))
    reset_n = Signal(bool(0))
    HEX0 = Signal(intbv(0)[7:0])
    LEDG = Signal(intbv(0)[4:0])
    M = 12
    N = 4

```

(continues on next page)

(continued from previous page)

```
    mod_m_counter_inst = mod_m_counter_visual_test(CLOCK_50, reset_n, HEX0, LEDG, M, N)
    mod_m_counter_inst.convert(hdl="Verilog", initial_values=True)

if __name__ == '__main__':
    main()
```

- Below is the resultant Verilog code. Load this code to FPGA board and see the counting on the seven-segment-display device and LEDs.

```
// File: mod_m_counter_visual_test.v
// Generated by MyHDL 1.0dev
// Date: Fri Oct 13 15:04:14 2017


`timescale 1ns/10ps

module mod_m_counter_visual_test (
    CLOCK_50,
    reset_n,
    HEX0,
    LEDG
);


input CLOCK_50;
input reset_n;
output [6:0] HEX0;
reg [6:0] HEX0;
output [3:0] LEDG;
wire [3:0] LEDG;

reg complete_tick = 0;
reg clk_pulse = 0;
wire [3:0] count;
wire [28:0] clock_tick_1_count;
reg [28:0] clock_tick_1_mod_m_counter_1_count_reg = 0;
reg [28:0] clock_tick_1_mod_m_counter_1_count_next = 0;
reg [3:0] mod_m_counter_2_count_reg = 0;
reg [3:0] mod_m_counter_2_count_next = 0;




assign clock_tick_1_count = clock_tick_1_mod_m_counter_1_count_reg;


always @(clock_tick_1_mod_m_counter_1_count_reg) begin: MOD_M_COUNTER_VISUAL_TEST_CLOCK_
↪TICK_1_MOD_M_COUNTER_1_LOGIC_NEXT
    if (($signed({1'b0, clock_tick_1_mod_m_counter_1_count_reg}) == (50000000 - 1))) begin
        clock_tick_1_mod_m_counter_1_count_next = 0;
        clk_pulse = 1;
    end
    else begin
        clock_tick_1_mod_m_counter_1_count_next = (clock_tick_1_mod_m_counter_1_count_reg␣
↪+ 1);
        clk_pulse = 0;
    end
end


always @(posedge CLOCK_50, negedge reset_n) begin: MOD_M_COUNTER_VISUAL_TEST_CLOCK_TICK_1_
↪MOD_M_COUNTER_1_LOGIC_REG
```

(continues on next page)

(continued from previous page)

```
    if ((reset_n == 0)) begin
        clock_tick_1_mod_m_counter_1_count_reg <= 0;
    end
    else begin
        clock_tick_1_mod_m_counter_1_count_reg <= clock_tick_1_mod_m_counter_1_count_next;
    end
end



assign count = mod_m_counter_2_count_reg;


always @(mod_m_counter_2_count_reg) begin: MOD_M_COUNTER_VISUAL_TEST_MOD_M_COUNTER_2_
↪LOGIC_NEXT
    if (($signed({1'b0, mod_m_counter_2_count_reg}) == (12 - 1))) begin
        mod_m_counter_2_count_next = 0;
        complete_tick = 1;
    end
    else begin
        mod_m_counter_2_count_next = (mod_m_counter_2_count_reg + 1);
        complete_tick = 0;
    end
end


always @(posedge clk_pulse, negedge reset_n) begin: MOD_M_COUNTER_VISUAL_TEST_MOD_M_
↪COUNTER_2_LOGIC_REG
    if ((reset_n == 0)) begin
        mod_m_counter_2_count_reg <= 0;
    end
    else begin
        mod_m_counter_2_count_reg <= mod_m_counter_2_count_next;
    end
end


always @(count) begin: MOD_M_COUNTER_VISUAL_TEST_HEX_TO_SEVEN_SEG_TEST_1_HEX_TO_SEVEN_SEG_
↪1_LOGIC
    case (count)
        'h0: begin
            HEX0 <= 64;
        end
        'h1: begin
            HEX0 <= 121;
        end
        'h2: begin
            HEX0 <= 36;
        end
        'h3: begin
            HEX0 <= 48;
        end
        'h4: begin
            HEX0 <= 25;
        end
        'h5: begin
            HEX0 <= 18;
        end
        'h6: begin
            HEX0 <= 2;
        end
```

(continues on next page)

(continued from previous page)

```
        'h7: begin
            HEX0 <= 120;
        end
        'h8: begin
            HEX0 <= 0;
        end
        'h9: begin
            HEX0 <= 16;
        end
        'ha: begin
            HEX0 <= 8;
        end
        'hb: begin
            HEX0 <= 3;
        end
        'hc: begin
            HEX0 <= 70;
        end
        'hd: begin
            HEX0 <= 33;
        end
        'he: begin
            HEX0 <= 6;
        end
        default: begin
            HEX0 <= 14;
        end
    endcase
end



assign LEDG = count;

endmodule
```

2.7 Open port in MyHDL

- We can use the 'None' keyword to avoid connection for the unused port. For this, we need to modify the code for the output signal, which is going to be used as 'optional connection'. In the below code, Section 2.6 is reimplemented.
- First modify the 'mod_m_counter.py', in which the output signal 'count' and 'complete_tick' can be the optional connections; therefore these two signals are written inside the 'if' statement with 'None' keyword,

```
# mod_m_counter.py

from myhdl import *

@block
def mod_m_counter(clk, reset_n, complete_tick, count, M, N):
    """ M = max count
        N = minimum bits required to represent M
    """

    count_reg = Signal(intbv(0)[N:0])
    count_next = Signal(intbv(0)[N:0])

    @always(clk.posedge, reset_n.negedge)
```

(continues on next page)

(continued from previous page)

```
    def logic_reg():
        if reset_n == 0 :
            count_reg.next = 0
        else :
            count_reg.next = count_next

    @always_comb
    def logic_next():
        if count_reg == M-1 :
            count_next.next = 0
        else :
            count_next.next = count_reg + 1


    # optional complete_tick
    if complete_tick is not None:
        @always_comb
        def complete_tick_logic():
            if count_reg == M-1 :
                complete_tick.next = 1
            else :
                complete_tick.next = 0

    # optional count
    if count is not None:
        @always_comb
        def out_val():
            count.next = count_reg

    return instances()  # return all instances


def main():
    N = 3
    M = 5

    clk = Signal(bool(0))
    reset_n = Signal(bool(0))
    complete_tick = Signal(bool(0))
    count = Signal(intbv(0)[N:0])

    mod_m_counter_v = mod_m_counter(clk, reset_n, complete_tick, count, M, N)
    mod_m_counter_v.convert(hdl="Verilog", initial_values=True)

if __name__ == '__main__':
    main()
```

- Next, modify the 'clock_tick.py', which is instantiating the 'mod_m_counter.py', but not using the 'count' port. Hence keep if open (using None) as below,

```
# clock_tick.py

from myhdl import *

from mod_m_counter import mod_m_counter

@block
def clock_tick(clk, reset_n, clk_pulse, M, N):
    # count = Signal(intbv(0)[N:0])
    # mod_m_counter_inst = mod_m_counter(clk, reset_n, clk_pulse, count, M, N)

```

(continues on next page)

(continued from previous page)

```
    # None for count-port
    mod_m_counter_inst = mod_m_counter(clk, reset_n, clk_pulse, None, M, N)
    return mod_m_counter_inst

# instantiate clock_tick
def main():
    clk = Signal(bool(0))
    reset_n = Signal(bool(0))
    clk_pulse = Signal(bool(0))
    M = 5000000 # 0.1 sec
    N = 29

    clock_tick_inst = clock_tick(clk, reset_n, clk_pulse, M, N)
    clock_tick_inst.convert(hdl="Verilog", initial_values=True)

if __name__ == '__main__':
    main()
```

- Lastly, update the 'mod_m_counter_visual_test.py', which is instantiating the 'mod_m_counter.py' but not using the port 'complete_tick',

```
# mod_m_counter_visual_test.py

from myhdl import *

from mod_m_counter import mod_m_counter
from clock_tick import clock_tick
from hex_to_seven_seg import hex_to_seven_seg_test

@block
def mod_m_counter_visual_test(CLOCK_50, reset_n, HEX0, LEDG, M, N):
    count = Signal(intbv(0)[N:0])
    clk_pulse = Signal(bool(0))
    # complete_tick = Signal(bool(0))

    # reduced clock rate
    clock_tick_inst = clock_tick(CLOCK_50, reset_n, clk_pulse, 50000000, 29)

    # reduce clock rate is applied to mod_m_counter
    # mod_m_counter_inst = mod_m_counter(clk_pulse, reset_n,
                    # complete_tick, count, M, N)
    mod_m_counter_inst = mod_m_counter(clk_pulse, reset_n, None, count, M, N)

    # count is send for conversion
    hex_to_seven_seg_test_inst = hex_to_seven_seg_test(count, HEX0)

    # send count to LEDG
    @always_comb
    def led_logic():
        LEDG.next = count
    return instances()

def main():
    CLOCK_50 = Signal(bool(0))
    reset_n = Signal(bool(0))
    HEX0 = Signal(intbv(0)[7:0])
    LEDG = Signal(intbv(0)[4:0])
    M = 12
    N = 4

    mod_m_counter_inst = mod_m_counter_visual_test(CLOCK_50, reset_n, HEX0, LEDG, M, N)
```

(continues on next page)

(continued from previous page)

```
41      mod_m_counter_inst.convert(hdl="Verilog", initial_values=True)
42
43  if __name__ == '__main__':
44      main()
```

- Now run the 'mod_m_counter_visual_test.py' and following code will be generated without warning. Load this design on the FPGA board to see the counting on the seven-segment-display device.

```
// File: mod_m_counter_visual_test.v
// Generated by MyHDL 1.0dev
// Date: Fri Oct 13 15:16:53 2017


`timescale 1ns/10ps

module mod_m_counter_visual_test (
    CLOCK_50,
    reset_n,
    HEX0,
    LEDG
);


input CLOCK_50;
input reset_n;
output [6:0] HEX0;
reg [6:0] HEX0;
output [3:0] LEDG;
wire [3:0] LEDG;

reg clk_pulse = 0;
wire [3:0] count;
reg [28:0] clock_tick_1_mod_m_counter_1_count_reg = 0;
reg [28:0] clock_tick_1_mod_m_counter_1_count_next = 0;
reg [3:0] mod_m_counter_2_count_reg = 0;
reg [3:0] mod_m_counter_2_count_next = 0;



always @(posedge CLOCK_50, negedge reset_n) begin: MOD_M_COUNTER_VISUAL_TEST_CLOCK_TICK_1_
↪MOD_M_COUNTER_1_LOGIC_REG
    if ((reset_n == 0)) begin
        clock_tick_1_mod_m_counter_1_count_reg <= 0;
    end
    else begin
        clock_tick_1_mod_m_counter_1_count_reg <= clock_tick_1_mod_m_counter_1_count_next;
    end
end


always @(clock_tick_1_mod_m_counter_1_count_reg) begin: MOD_M_COUNTER_VISUAL_TEST_CLOCK_
↪TICK_1_MOD_M_COUNTER_1_LOGIC_NEXT
    if (($signed({1'b0, clock_tick_1_mod_m_counter_1_count_reg}) == (50000000 - 1))) begin
        clock_tick_1_mod_m_counter_1_count_next = 0;
    end
    else begin
        clock_tick_1_mod_m_counter_1_count_next = (clock_tick_1_mod_m_counter_1_count_reg␣
↪+ 1);
    end
end
```

(continues on next page)

(continued from previous page)

```
always @(clock_tick_1_mod_m_counter_1_count_reg) begin: MOD_M_COUNTER_VISUAL_TEST_CLOCK_
↪TICK_1_MOD_M_COUNTER_1_COMPLETE_TICK_LOGIC
    if (($signed({1'b0, clock_tick_1_mod_m_counter_1_count_reg}) == (50000000 - 1))) begin
        clk_pulse = 1;
    end
    else begin
        clk_pulse = 0;
    end
end


always @(posedge clk_pulse, negedge reset_n) begin: MOD_M_COUNTER_VISUAL_TEST_MOD_M_
↪COUNTER_2_LOGIC_REG
    if ((reset_n == 0)) begin
        mod_m_counter_2_count_reg <= 0;
    end
    else begin
        mod_m_counter_2_count_reg <= mod_m_counter_2_count_next;
    end
end


always @(mod_m_counter_2_count_reg) begin: MOD_M_COUNTER_VISUAL_TEST_MOD_M_COUNTER_2_
↪LOGIC_NEXT
    if (($signed({1'b0, mod_m_counter_2_count_reg}) == (12 - 1))) begin
        mod_m_counter_2_count_next = 0;
    end
    else begin
        mod_m_counter_2_count_next = (mod_m_counter_2_count_reg + 1);
    end
end



assign count = mod_m_counter_2_count_reg;


always @(count) begin: MOD_M_COUNTER_VISUAL_TEST_HEX_TO_SEVEN_SEG_TEST_1_HEX_TO_SEVEN_SEG_
↪1_LOGIC
    case (count)
        'h0: begin
            HEX0 <= 64;
        end
        'h1: begin
            HEX0 <= 121;
        end
        'h2: begin
            HEX0 <= 36;
        end
        'h3: begin
            HEX0 <= 48;
        end
        'h4: begin
            HEX0 <= 25;
        end
        'h5: begin
            HEX0 <= 18;
        end
        'h6: begin
            HEX0 <= 2;
```

(continues on next page)

(continued from previous page)

```
        end
        'h7: begin
            HEX0 <= 120;
        end
        'h8: begin
            HEX0 <= 0;
        end
        'h9: begin
            HEX0 <= 16;
        end
        'ha: begin
            HEX0 <= 8;
        end
        'hb: begin
            HEX0 <= 3;
        end
        'hc: begin
            HEX0 <= 70;
        end
        'hd: begin
            HEX0 <= 33;
        end
        'he: begin
            HEX0 <= 6;
        end
        default: begin
            HEX0 <= 14;
        end
    endcase
end

assign LEDG = count;

endmodule
```

2.8 Conclusion

In this chapter, we learn to reduce the clock rate to visualize the outputs on the display devices. The reduced clock rate is used to display the counting on the seven-segment-display device and LEDs. Further, the structure modeling is used here, which makes the code more manageable and reusable.

Chapter 3

Testbench

In this chapter, we write the testbench for the Listing 2.1. This testbench contains several features of MyHDL which are enough to start writing the testbenches. Also, conversion of MyHDL testbench to HDL testbench is discussed.

3.1 Mod-m counter

- In this section, we will write a testbench for Mod m coutner. For this, the Listing 2.1 is modified as below,

```
# mod_m_counter.py

from myhdl import *

period = 20 # clk frequency = 50 MHz

@block
def mod_m_counter(clk, reset_n, complete_tick, count, M, N):
    """ M = max count
        N = minimum bits required to represent M
    """

    count_reg = Signal(intbv(0)[N:0])
    count_next = Signal(intbv(0)[N:0])

    @always(clk.posedge, reset_n.negedge)
    def logic_reg():
        if reset_n == 0 :
            count_reg.next = 0
        else :
            count_reg.next = count_next

    @always_comb
    def logic_next():
        if count_reg == M-1 :
            count_next.next = 0
        else :
            count_next.next = count_reg + 1


    # optional complete_tick
    if complete_tick is not None:
        @always_comb
        def complete_tick_logic():
```

(continues on next page)

(continued from previous page)

```
            if count_reg == M-1 :
                complete_tick.next = 1
            else :
                complete_tick.next = 0

    # optional count
    if count is not None:
        @always_comb
        def out_val():
            count.next = count_reg

    return instances()  # return all instances


# testbench
@block
def mod_m_counter_tb():
    N = 3
    M = 5

    clk = Signal(bool(0))
    reset_n = Signal(bool(0))
    complete_tick = Signal(bool(0))
    count = Signal(intbv(0)[N:0])

    mod_m_counter_inst = mod_m_counter(clk, reset_n, complete_tick, count, M, N)

    # int is required
    @always(delay(int(period/2)))
    def clk_signal():
        clk.next = not clk

    @instance  # reset signal
    def reset_signal():
        reset_n.next = 0
        yield delay(period)
        reset_n.next = 1

    return instances()


def main():
    N = 3
    M = 5

    clk = Signal(bool(0))
    reset_n = Signal(bool(0))
    complete_tick = Signal(bool(0))
    count = Signal(intbv(0)[N:0])

    mod_m_counter_v = mod_m_counter(clk, reset_n, complete_tick, count, M, N)
    mod_m_counter_v.convert(hdl="Verilog", initial_values=True)

    # test bench
    tb = mod_m_counter_tb()
    tb.config_sim(trace=True)
    tb.run_sim(15*period) # run for 15 clock cycle

if __name__ == '__main__':
    main()
```

- If we run the above code, then a *vcd* file, i.e. mod_m_counter_tb.vcd', will be generated, which can be

open in gtkwave. Following is the results saved in the .vcd file,

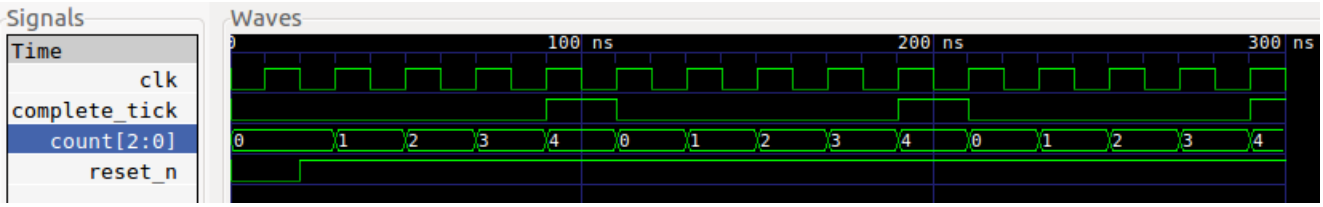

Fig. 3.1: Simulation results of mod_m_counter.py

3.2 Saving data to file

- Sometimes we want to save the testbench-data in the file for further analysis of the data. In the below listing, the code in the above Section 3.1 is slightly modified to save the data in the file and printed on the screen.

Important:
- Note that, the signals at Lines 55-61 are declared as 'intbv' instead of 'bool', because the 'bool' signal is saved as 'True/False' which may not be desirable for the analysis of the data.
- Lines 79-93 prints the data on screen and save the data in the file 'mod_m_counter.csv'.

```
1  # mod_m_counter.py
2
3  from myhdl import *
4
5  period = 20 # clk frequency = 50 MHz
6
7  @block
8  def mod_m_counter(clk, reset_n, complete_tick, count, M, N):
9      """ M = max count
10         N = minimum bits required to represent M
11     """
12
13     count_reg = Signal(intbv(0)[N:0])
14     count_next = Signal(intbv(0)[N:0])
15
16     @always(clk.posedge, reset_n.negedge)
17     def logic_reg():
18         if reset_n == 0 :
19             count_reg.next = 0
20         else :
21             count_reg.next = count_next
22
23     @always_comb
24     def logic_next():
25         if count_reg == M-1 :
26             count_next.next = 0
27         else :
28             count_next.next = count_reg + 1
29
30
31     # optional complete_tick
32     if complete_tick is not None:
33         @always_comb
34         def complete_tick_logic():
```

(continues on next page)

(continued from previous page)

```
            if count_reg == M-1 :
                complete_tick.next = 1
            else :
                complete_tick.next = 0

    # optional count
    if count is not None:
        @always_comb
        def out_val():
            count.next = count_reg

    return instances()  # return all instances


# testbench
@block
def mod_m_counter_tb():
    N = 3
    M = 5

    # 'intbv' is used instead of 'bool' as 'false' will be displayed
    # in the saved-data for 'bool'.
    # clk = Signal(bool(0))
    clk = Signal(intbv(0)[1:0])
    reset_n = Signal(intbv(0)[1:0])
    complete_tick = Signal(intbv(0)[1:0])
    count = Signal(intbv(0)[N:0])

    mod_m_counter_inst = mod_m_counter(clk, reset_n, complete_tick, count, M, N)


    # int is required
    @always(delay(int(period/2)))
    def clk_signal():
        clk.next = not clk


    @instance  # reset signal
    def reset_signal():
        reset_n.next = 0
        yield delay(period)
        reset_n.next = 1


    # print simulation data on screen and file
    file_data = open("mod_m_counter.csv", 'w') # file for saving data
    # print header on screen
    print("{0},{1},{2}".format("reset_n", "complete_tick", "count"))
    # print header to file
    print("{0},{1},{2}".format("reset_n", "complete_tick", "count")
                , file=file_data)
    # print data on each clock
    @always(clk.posedge)
    def print_data():
        # print on screen
        print("{0}, {1}, {2}".format(reset_n, complete_tick, count))
        # print in file
        print("{0}, {1}, {2}".format(reset_n, complete_tick, count)
                , file=file_data)


```

(continues on next page)

(continued from previous page)

```
96      return instances()
97
98
99  def main():
100     N = 3
101     M = 5
102
103     clk = Signal(bool(0))
104     reset_n = Signal(bool(0))
105     complete_tick = Signal(bool(0))
106     count = Signal(intbv(0)[N:0])
107
108     mod_m_counter_v = mod_m_counter(clk, reset_n, complete_tick, count, M, N)
109     mod_m_counter_v.convert(hdl="Verilog", initial_values=True)
110
111     # test bench
112     tb = mod_m_counter_tb()
113     tb.config_sim(trace=True)
114     tb.run_sim(15*period) # run for 15 clock cycle
115
116
117 if __name__ == '__main__':
118     main()
```

- Following is the data which is saved in the file 'mod_m_counter.csv'

```
reset_n,complete_tick,count
0, 0, 0
1, 0, 0
1, 0, 1
1, 0, 2
1, 0, 3
1, 1, 4
1, 0, 0
1, 0, 1
1, 0, 2
1, 0, 3
1, 1, 4
1, 0, 0
1, 0, 1
1, 0, 2
1, 0, 3
```

3.3 Conversion : MyHDL testbench to HDL testbench

- Some changes are required in above listing to convert the MyHDL testbench into HDL testbench, as shown below. Execute the below code to generate the testbench.

Note:
- '@instance' decorator is used with "def clk_signal():", as 'delay' argument is not supported for testbench conversion (Lines 66-70).
- '.format' option of Python3-print statement is not supported by MyHDL (Lines 91-99).
- The conversion statement (Line 119) is placed above the simulation statements (Lines 122-123).

```
1  # mod_m_counter.py
2
3  from myhdl import *
```

(continues on next page)

(continued from previous page)

```
4
5  period = 20 # clk frequency = 50 MHz
6
7  @block
8  def mod_m_counter(clk, reset_n, complete_tick, count, M, N):
9      """ M = max count
10         N = minimum bits required to represent M
11     """
12
13     count_reg = Signal(intbv(0)[N:0])
14     count_next = Signal(intbv(0)[N:0])
15
16     @always(clk.posedge, reset_n.negedge)
17     def logic_reg():
18         if reset_n == 0 :
19             count_reg.next = 0
20         else :
21             count_reg.next = count_next
22
23     @always_comb
24     def logic_next():
25         if count_reg == M-1 :
26             count_next.next = 0
27         else :
28             count_next.next = count_reg + 1
29
30
31     # optional complete_tick
32     if complete_tick is not None:
33         @always_comb
34         def complete_tick_logic():
35             if count_reg == M-1 :
36                 complete_tick.next = 1
37             else :
38                 complete_tick.next = 0
39
40     # optional count
41     if count is not None:
42         @always_comb
43         def out_val():
44             count.next = count_reg
45
46     return instances()  # return all instances
47
48
49 # testbench
50 @block
51 def mod_m_counter_tb():
52     N = 3
53     M = 5
54
55     # 'intbv' is used instead of 'bool' as 'false' will be displayed
56     # in the saved-data for 'bool'.
57     # clk = Signal(bool(0))
58     clk = Signal(intbv(0)[1:0])
59     reset_n = Signal(intbv(0)[1:0])
60     complete_tick = Signal(intbv(0)[1:0])
61     count = Signal(intbv(0)[N:0])
62
63     mod_m_counter_inst = mod_m_counter(clk, reset_n, complete_tick, count, M, N)
64
```

(continues on next page)

(continued from previous page)

```
65
66      @instance
67      def clk_signal():
68          while True:
69              clk.next = not clk
70              yield delay(period//2)
71
72
73      @instance  # reset signal
74      def reset_signal():
75          reset_n.next = 0
76          yield delay(period)
77          reset_n.next = 1
78
79
80      # print simulation data on screen and file
81      file_data = open("mod_m_counter.csv", 'w') # file for saving data
82      # # print header on screen
83      print("{0},{1},{2}".format("reset_n", "complete_tick", "count"))
84      # # print header to file
85      print("{0},{1},{2}".format("reset_n", "complete_tick", "count"),
86                file=file_data)
87      # print data on each clock
88      @always(clk.posedge)
89      def print_data():
90          # print on screen
91          # print.format is not supported in MyHDL 1.0
92          # print("{0}, {1}, {2}".format(reset_n, complete_tick, count))
93          print(reset_n, ",", complete_tick, ",",  count, sep='')
94
95          # print in file
96          # print.format is not supported in MyHDL 1.0
97          # print("{0}, {1}, {2}".format(reset_n, complete_tick, count)
98                  # , file=file_data)
99          print(reset_n, ",", complete_tick, ",",  count, sep='', file=file_data)
100
101
102     return instances()
103
104
105 def main():
106     N = 3
107     M = 5
108
109     clk = Signal(bool(0))
110     reset_n = Signal(bool(0))
111     complete_tick = Signal(bool(0))
112     count = Signal(intbv(0)[N:0])
113
114     mod_m_counter_v = mod_m_counter(clk, reset_n, complete_tick, count, M, N)
115     mod_m_counter_v.convert(hdl="Verilog", initial_values=True)
116
117     # test bench
118     tb = mod_m_counter_tb()
119     tb.convert(hdl="Verilog", initial_values=True)
120     # keep following lines below the 'tb.convert' line
121     # otherwise error will be reported
122     tb.config_sim(trace=True)
123     tb.run_sim(15*period) # run for 15 clock cycle
124
125
```

(continues on next page)

(continued from previous page)

```
126 if __name__ == '__main__':
127     main()
```

- Following is the generated Verilog code for the testbench,

Note: In the above listing, the Python code prints the data on screen and save in the file. But in the resultant Verilog code, the data will not be saved in the file (but will be printed on the screen).

```
// File: mod_m_counter_tb.v
// Generated by MyHDL 1.0dev
// Date: Mon Oct 16 11:44:15 2017


`timescale 1ns/10ps

module mod_m_counter_tb (

);

reg [0:0] reset_n = 0;
wire [2:0] count;
reg [0:0] complete_tick = 0;
reg [0:0] clk = 0;
reg [2:0] mod_m_counter_1_count_reg = 0;
reg [2:0] mod_m_counter_1_count_next = 0;

always @(posedge clk, negedge reset_n) begin: MOD_M_COUNTER_TB_MOD_M_COUNTER_1_LOGIC_REG
    if ((reset_n == 0)) begin
        mod_m_counter_1_count_reg <= 0;
    end
    else begin
        mod_m_counter_1_count_reg <= mod_m_counter_1_count_next;
    end
end


always @(mod_m_counter_1_count_reg) begin: MOD_M_COUNTER_TB_MOD_M_COUNTER_1_LOGIC_NEXT
    if (($signed({1'b0, mod_m_counter_1_count_reg}) == (5 - 1))) begin
        mod_m_counter_1_count_next = 0;
    end
    else begin
        mod_m_counter_1_count_next = (mod_m_counter_1_count_reg + 1);
    end
end


always @(mod_m_counter_1_count_reg) begin: MOD_M_COUNTER_TB_MOD_M_COUNTER_1_COMPLETE_TICK_
↪LOGIC
    if (($signed({1'b0, mod_m_counter_1_count_reg}) == (5 - 1))) begin
        complete_tick = 1;
    end
    else begin
        complete_tick = 0;
    end
end

assign count = mod_m_counter_1_count_reg;

initial begin: MOD_M_COUNTER_TB_CLK_SIGNAL
    while (1'b1) begin
```

(continues on next page)

(continued from previous page)

```
        clk <= (!clk);
        # (20 / 2);
    end
end

initial begin: MOD_M_COUNTER_TB_RESET_SIGNAL
    reset_n <= 0;
    # 20;
    reset_n <= 1;
end

always @(posedge clk) begin: MOD_M_COUNTER_TB_PRINT_DATA
    $write("%h", reset_n);
    $write(" ");
    $write(",");
    $write(" ");
    $write("%h", complete_tick);
    $write(" ");
    $write(",");
    $write(" ");
    $write("%h", count);
    $write("\n");
    $write("%h", reset_n);
    $write(" ");
    $write(",");
    $write(" ");
    $write("%h", complete_tick);
    $write(" ");
    $write(",");
    $write(" ");
    $write("%h", count);
    $write("\n");
end

endmodule
```

- Below is the simulation results for the above testbench, which is same as Fig. 3.1 generated by MyHDL testbench.

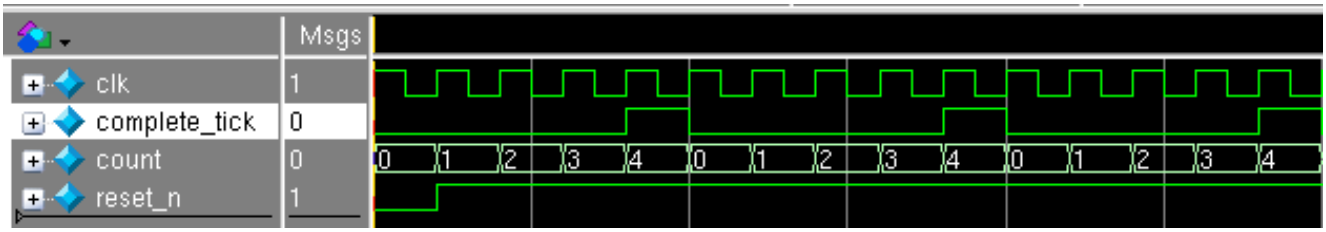

Fig. 3.2: Simulation results of mod_m_counter.v

3.4 Conclusion

In this chapter, we wrote the testbench for the Mod-m counter. First we wrote a testbench which saves the data in the .vcd file. Next, we learn the method to save the data in the file. Lastly, we modified the MyHDL testbench to convert it into HDL testbench.

Chapter 4

Finite state machine

4.1 Introduction

Note:

- In previous chapters, we saw the various examples of combinational and sequential designs. In this chapter, we will learn to use the finite state machines to implement various types of designs.
- The proper methods for implementing the various kinds of FSM designs are disucssed in the Verilog/VHDL tutorials. In this chapter, we will re-implement the designs of the Chapter "Finite state machines" in the Verilog/VHDL tutorial. Please see the Verilog/SystemVerilog/VHDL tutorials for the explanation of the designs.

4.2 Rising edge detector

- In the below code, the rising edge detector is implemented using Mealy and Moore designs,

```
# edge_detector.py
# the results are same as "edgeDetector.v" in the Verilog tutorial

from myhdl import *

@block
def edge_detector(clk, reset_n, level, mealy_tick, moore_tick):

    # mealy states
    mealy_states = enum('zero_mealy', 'one_mealy')
    state_mealy_reg = Signal(mealy_states.zero_mealy)
    state_mealy_next = Signal(mealy_states.zero_mealy)

    # moore states
    moore_states = enum('zero_moore', 'edge_moore', 'one_moore')
    state_moore_reg = Signal(moore_states.zero_moore)
    state_moore_next = Signal(moore_states.zero_moore)

    # current state logic
    @always(clk.posedge, reset_n.negedge)
    def state_current_logic():
        if reset_n == 0 :
            state_moore_reg.next = moore_states.zero_moore
            state_mealy_reg.next = mealy_states.zero_mealy
```

(continues on next page)

(continued from previous page)

```
        else :
            state_moore_reg.next = state_moore_next
            state_mealy_reg.next = state_mealy_next

    # next state logic : Mealy design
    @always_comb
    def state_next_logic_mealy():
        state_mealy_next.next = state_mealy_reg
        mealy_tick.next = 0
        if state_mealy_reg == mealy_states.zero_mealy :
            if level == 1 :
                state_mealy_next.next = mealy_states.one_mealy
                mealy_tick.next = 1
        elif state_mealy_reg == mealy_states.one_mealy:
            if level != 1 :
                state_mealy_next.next = mealy_states.zero_mealy

    # next state logic : Moore design
    @always_comb
    def state_next_logic_moore():

        state_moore_next.next = state_moore_reg
        moore_tick.next = 0
        if state_moore_reg == moore_states.zero_moore :
            if level == 1 :
                state_moore_next.next = moore_states.edge_moore
        elif state_moore_reg == moore_states.edge_moore:
            moore_tick.next = 1
            if level == 1 :
                state_moore_next.next = moore_states.one_moore
            else :
                state_moore_next.next = moore_states.zero_moore
        elif state_moore_reg == moore_states.one_moore:
            if level != 1 :
                state_moore_next.next = moore_states.zero_moore


    return state_current_logic, state_next_logic_moore, state_next_logic_mealy

def main():
    clk = Signal(bool(0))
    reset_n = Signal(bool(0))
    level = Signal(bool(0))
    moore_tick = Signal(bool(0))
    mealy_tick = Signal(bool(0))

    edge_detector_verilog = edge_detector(clk, reset_n, level, mealy_tick, moore_tick)
    edge_detector_verilog.convert(hdl="Verilog", initial_values=True)

if __name__ == '__main__':
    main()
```

- The resultant Verilog code is shown below,

```
// File: edge_detector.v
// Generated by MyHDL 1.0dev
// Date: Fri Oct 13 15:29:53 2017


`timescale 1ns/10ps

module edge_detector (
```

(continues on next page)

(continued from previous page)

```
    clk,
    reset_n,
    level,
    mealy_tick,
    moore_tick
);


input clk;
input reset_n;
input level;
output mealy_tick;
reg mealy_tick;
output moore_tick;
reg moore_tick;

reg [1:0] state_moore_reg = 2'b00;
reg [1:0] state_moore_next = 2'b00;
reg [0:0] state_mealy_reg = 1'b0;
reg [0:0] state_mealy_next = 1'b0;



always @(posedge clk, negedge reset_n) begin: EDGE_DETECTOR_STATE_CURRENT_LOGIC
    if ((reset_n == 0)) begin
        state_moore_reg <= 2'b00;
        state_mealy_reg <= 1'b0;
    end
    else begin
        state_moore_reg <= state_moore_next;
        state_mealy_reg <= state_mealy_next;
    end
end


always @(state_moore_reg, level) begin: EDGE_DETECTOR_STATE_NEXT_LOGIC_MOORE
    state_moore_next = state_moore_reg;
    moore_tick = 0;
    case (state_moore_reg)
        2'b00: begin
            if ((level == 1)) begin
                state_moore_next = 2'b01;
            end
        end
        2'b01: begin
            moore_tick = 1;
            if ((level == 1)) begin
                state_moore_next = 2'b10;
            end
            else begin
                state_moore_next = 2'b00;
            end
        end
        2'b10: begin
            if ((level != 1)) begin
                state_moore_next = 2'b00;
            end
        end
    endcase
end

```

(continues on next page)

(continued from previous page)

```
70
71  always @(level, state_mealy_reg) begin: EDGE_DETECTOR_STATE_NEXT_LOGIC_MEALY
72      state_mealy_next = state_mealy_reg;
73      mealy_tick = 0;
74      case (state_mealy_reg)
75          1'b0: begin
76              if ((level == 1)) begin
77                  state_mealy_next = 1'b1;
78                  mealy_tick = 1;
79              end
80          end
81          1'b1: begin
82              if ((level != 1)) begin
83                  state_mealy_next = 1'b0;
84              end
85          end
86      endcase
87  end
88
89  endmodule
```

4.3 Non overlapping sequence detector : 110

- In the below code, a sequence detector is implement which detects the sequence '110',

```
1   # sequence_detector.py
2   # result is same as the listing "sequence_detector.v" in verilog tutorial
3
4   from myhdl import *
5
6   @block
7   def sequence_detector(clk, reset_n, x, z_mealy_glitch, z_moore_glitch,
8           z_mealy_glitch_free, z_moore_glitch_free):
9
10
11      # mealy states
12      mealy_states = enum('zero_mealy', 'one_mealy', 'two_mealy', 'three_mealy')
13      state_mealy_reg = Signal(mealy_states.zero_mealy)
14      state_mealy_next = Signal(mealy_states.zero_mealy)
15
16      # moore states
17      moore_states = enum('zero_moore', 'one_moore', 'two_moore', 'three_moore')
18      state_moore_reg = Signal(moore_states.zero_moore)
19      state_moore_next = Signal(moore_states.zero_moore)
20
21      z_moore = Signal(bool(0))
22      z_mealy = Signal(bool(0))
23
24      @always(clk.posedge, reset_n.negedge)
25      def current_state_logic():
26          if reset_n == 0 :
27              state_mealy_reg.next = mealy_states.zero_mealy
28              state_moore_reg.next = moore_states.zero_moore
29          else :
30              state_mealy_reg.next = state_mealy_next
31              state_moore_reg.next = state_moore_next
32
33      # next state mealy logic
```

(continues on next page)

(continued from previous page)

```
    @always_comb
    def next_state_logic_mealy():
        z_mealy.next = 0
        state_mealy_next.next = state_mealy_reg
        if state_mealy_reg == mealy_states.zero_mealy:
            if x == 1 :
                state_mealy_next.next = mealy_states.one_mealy
        elif state_mealy_reg == mealy_states.one_mealy :
            if x == 1 :
                state_mealy_next.next = mealy_states.two_mealy
            else :
                state_mealy_next.next = mealy_states.zero_mealy
        elif state_mealy_reg == mealy_states.two_mealy :
            state_mealy_next.next = mealy_states.zero_mealy
            if x == 0 :
                z_mealy.next = 1
            else :
                state_mealy_next.next = mealy_states.two_mealy

    # next state moore logic
    @always_comb
    def next_state_logic_moore():
        z_moore.next = 0
        state_moore_next.next = state_moore_reg
        if state_moore_reg == moore_states.zero_moore:
            if x == 1 :
                state_moore_next.next = moore_states.one_moore
        elif state_moore_reg == moore_states.one_moore :
            if x == 1 :
                state_moore_next.next = moore_states.two_moore
            else :
                state_moore_next.next = moore_states.zero_moore
        elif state_moore_reg == moore_states.two_moore :
            if x == 0 :
                state_moore_next.next = moore_states.three_moore
            else :
                state_moore_next.next = moore_states.two_moore
        elif state_moore_reg == moore_states.three_moore :
            z_moore.next = 1
            if x == 1 :
                state_moore_next.next = moore_states.zero_moore
            else :
                state_moore_next.next = moore_states.one_moore


    @always_comb
    def assign_outputs():
        z_mealy_glitch.next = z_mealy
        z_moore_glitch.next = z_moore

    @always(clk.posedge, reset_n.negedge)
    def glitch_free_output():
        if reset_n == 1 :
            z_mealy_glitch_free.next = 0
            z_moore_glitch_free.next = 0
        else :
            z_mealy_glitch_free.next = z_mealy
            z_moore_glitch_free.next = z_moore


    return current_state_logic, next_state_logic_mealy, \
```

(continues on next page)

(continued from previous page)

```
            next_state_logic_moore, assign_outputs, glitch_free_output

def main():
    clk = Signal(bool(0))
    reset_n = Signal(bool(0))
    x = Signal(bool(0))
    z_mealy_glitch = Signal(bool(0))
    z_moore_glitch = Signal(bool(0))
    z_mealy_glitch_free = Signal(bool(0))
    z_moore_glitch_free = Signal(bool(0))

    sequence_detector_verilog = sequence_detector(clk, reset_n, x,
            z_mealy_glitch, z_moore_glitch,
            z_mealy_glitch_free, z_moore_glitch_free)

    sequence_detector_verilog.convert(hdl="Verilog", initial_values=True)

if __name__ == '__main__':
    main()
```

- The corresponding Verilog design is shown below,

```
// File: sequence_detector.v
// Generated by MyHDL 1.0dev
// Date: Fri Oct 13 15:33:20 2017


`timescale 1ns/10ps

module sequence_detector (
    clk,
    reset_n,
    x,
    z_mealy_glitch,
    z_moore_glitch,
    z_mealy_glitch_free,
    z_moore_glitch_free
);


input clk;
input reset_n;
input x;
output z_mealy_glitch;
wire z_mealy_glitch;
output z_moore_glitch;
wire z_moore_glitch;
output z_mealy_glitch_free;
reg z_mealy_glitch_free;
output z_moore_glitch_free;
reg z_moore_glitch_free;

reg z_moore = 0;
reg z_mealy = 0;
reg [1:0] state_moore_reg = 2'b00;
reg [1:0] state_moore_next = 2'b00;
reg [1:0] state_mealy_reg = 2'b00;
reg [1:0] state_mealy_next = 2'b00;



always @(posedge clk, negedge reset_n) begin: SEQUENCE_DETECTOR_CURRENT_STATE_LOGIC
```

(continues on next page)

(continued from previous page)

```
    if ((reset_n == 0)) begin
        state_mealy_reg <= 2'b00;
        state_moore_reg <= 2'b00;
    end
    else begin
        state_mealy_reg <= state_mealy_next;
        state_moore_reg <= state_moore_next;
    end
end


always @(x, state_mealy_reg) begin: SEQUENCE_DETECTOR_NEXT_STATE_LOGIC_MEALY
    z_mealy = 0;
    state_mealy_next = state_mealy_reg;
    case (state_mealy_reg)
        2'b00: begin
            if ((x == 1)) begin
                state_mealy_next = 2'b01;
            end
        end
        2'b01: begin
            if ((x == 1)) begin
                state_mealy_next = 2'b10;
            end
            else begin
                state_mealy_next = 2'b00;
            end
        end
        2'b10: begin
            state_mealy_next = 2'b00;
            if ((x == 0)) begin
                z_mealy = 1;
            end
            else begin
                state_mealy_next = 2'b10;
            end
        end
    endcase
end


always @(x, state_moore_reg) begin: SEQUENCE_DETECTOR_NEXT_STATE_LOGIC_MOORE
    z_moore = 0;
    state_moore_next = state_moore_reg;
    case (state_moore_reg)
        2'b00: begin
            if ((x == 1)) begin
                state_moore_next = 2'b01;
            end
        end
        2'b01: begin
            if ((x == 1)) begin
                state_moore_next = 2'b10;
            end
            else begin
                state_moore_next = 2'b00;
            end
        end
        2'b10: begin
            if ((x == 0)) begin
                state_moore_next = 2'b11;
```

(continues on next page)

(continued from previous page)

```
            end
            else begin
                state_moore_next = 2'b10;
            end
        end
        2'b11: begin
            z_moore = 1;
            if ((x == 1)) begin
                state_moore_next = 2'b00;
            end
            else begin
                state_moore_next = 2'b01;
            end
        end
    endcase
end



assign z_mealy_glitch = z_mealy;
assign z_moore_glitch = z_moore;


always @(posedge clk, negedge reset_n) begin: SEQUENCE_DETECTOR_GLITCH_FREE_OUTPUT
    if ((reset_n == 1)) begin
        z_mealy_glitch_free <= 0;
        z_moore_glitch_free <= 0;
    end
    else begin
        z_mealy_glitch_free <= z_mealy;
        z_moore_glitch_free <= z_moore;
    end
end

endmodule
```

4.4 Timed FSM - programmable square wave

- In the below code, the square wave is generated with programmable on/off time,

```
# square_wave_ex.py
# results are same as Listing "square_wave_ex.v" in Verilog tutorial

from myhdl import *

@block
def square_wave_ex(clk, reset_n, s_wave, on_time, off_time, N):
    states = enum('on_state', 'off_state')
    state_reg = Signal(states.on_state)
    state_next = Signal(states.on_state)
    t = Signal(intbv(0)[N:0])

    @always(clk.posedge, reset_n.negedge)
    def current_state_logic():
        if reset_n == 0 :
            state_reg.next = states.off_state
        else :
            state_reg.next = state_next
```

(continues on next page)

(continued from previous page)

```
19
20      @always(clk.posedge, reset_n.negedge)
21      def timer_logic():
22          if state_reg != state_next :
23              t.next = 0
24          else :
25              t.next = t + 1
26
27      @always_comb
28      def next_state_logic():
29          if state_reg == states.off_state :
30              s_wave.next = 0
31              if t == off_time - 1 :
32                  state_next.next = states.on_state
33              else :
34                  state_next.next = states.off_state
35          if state_reg == states.on_state :
36              s_wave.next = 1
37              if t == on_time - 1 :
38                  state_next.next = states.off_state
39              else :
40                  state_next.next = states.on_state
41
42      return current_state_logic, timer_logic, next_state_logic
43
44  def main():
45      N = 4
46      on_time = 5
47      off_time = 3
48      clk = Signal(bool(0))
49      reset_n = Signal(bool(0))
50      s_wave = Signal(bool(0))
51
52      square_wave_verilog = square_wave_ex(clk, reset_n,
53              s_wave, on_time, off_time, N)
54
55      square_wave_verilog.convert(hdl="Verilog", initial_values=True)
56
57  if __name__ == '__main__':
58      main()
```

- Following is the generated Verilog code,

```
1   // File: square_wave_ex.v
2   // Generated by MyHDL 1.0dev
3   // Date: Fri Oct 13 15:39:35 2017
4
5
6   `timescale 1ns/10ps
7
8   module square_wave_ex (
9       clk,
10      reset_n,
11      s_wave
12  );
13
14
15  input clk;
16  input reset_n;
17  output s_wave;
18  reg s_wave;
19
```

(continues on next page)

(continued from previous page)

```
reg [3:0] t = 0;
reg [0:0] state_reg = 1'b0;
reg [0:0] state_next = 1'b0;



always @(posedge clk, negedge reset_n) begin: SQUARE_WAVE_EX_CURRENT_STATE_LOGIC
    if ((reset_n == 0)) begin
        state_reg <= 1'b1;
    end
    else begin
        state_reg <= state_next;
    end
end


always @(posedge clk, negedge reset_n) begin: SQUARE_WAVE_EX_TIMER_LOGIC
    if ((state_reg != state_next)) begin
        t <= 0;
    end
    else begin
        t <= (t + 1);
    end
end


always @(t, state_reg) begin: SQUARE_WAVE_EX_NEXT_STATE_LOGIC
    if ((state_reg == 1'b1)) begin
        s_wave = 0;
        if (($signed({1'b0, t}) == (3 - 1))) begin
            state_next = 1'b0;
        end
        else begin
            state_next = 1'b1;
        end
    end
    if ((state_reg == 1'b0)) begin
        s_wave = 1;
        if (($signed({1'b0, t}) == (5 - 1))) begin
            state_next = 1'b1;
        end
        else begin
            state_next = 1'b0;
        end
    end
end

endmodule
```

4.5 Mod-m counter

- In the below code, the Mod-m counter is implemented using FSM method,

```
# counter_ex.py
# results are same as the Listing 'counter_ex.v' in Verilog tutorial

from myhdl import *

```

(continues on next page)

(continued from previous page)

```
@block
def counter_ex(clk, reset_n, out_moore, M, N):
    states = enum('start_moore', 'count_moore')
    state_moore_reg = Signal(states.start_moore)
    state_moore_next = Signal(states.start_moore)

    count_moore_reg = Signal(intbv(0)[N:0])
    count_moore_next = Signal(intbv(0)[N:0])

    @always(clk.posedge, reset_n.negedge)
    def current_state_logic():
        if reset_n == 0 :
            state_moore_reg.next = states.start_moore
            count_moore_reg.next = 0
        else :
            state_moore_reg.next = state_moore_next
            count_moore_reg.next = count_moore_next

    @always_comb
    def next_state_logic():
        if state_moore_reg == states.start_moore :
            count_moore_next.next = 0
            state_moore_next.next = states.count_moore
        elif state_moore_reg == states.count_moore :
            count_moore_next.next = count_moore_reg + 1
            if count_moore_reg + 1 == M - 1 :
                state_moore_next.next = states.start_moore
            else :
                state_moore_next.next = states.count_moore

    @always_comb
    def assign_output_logic():
        out_moore.next = count_moore_reg

    return current_state_logic, next_state_logic, assign_output_logic

def main():
    M = 6
    N = 4
    clk = Signal(bool(0))
    reset_n = Signal(bool(0))
    out_moore = Signal(bool(0))

    counter_ex_verilog = counter_ex(clk, reset_n, out_moore, M, N)
    counter_ex_verilog.convert(hdl="Verilog", initial_values=True)

if __name__ == '__main__':
    main()
```

- Following is the resultant Verilog code,

```
// File: counter_ex.v
// Generated by MyHDL 1.0dev
// Date: Fri Oct 13 15:43:20 2017


`timescale 1ns/10ps

module counter_ex (
    clk,
    reset_n,
    out_moore
```

(continues on next page)

(continued from previous page)

```
);


input clk;
input reset_n;
output out_moore;
wire out_moore;

reg [0:0] state_moore_reg = 1'b0;
reg [0:0] state_moore_next = 1'b0;
reg [3:0] count_moore_reg = 0;
reg [3:0] count_moore_next = 0;



always @(posedge clk, negedge reset_n) begin: COUNTER_EX_CURRENT_STATE_LOGIC
    if ((reset_n == 0)) begin
        state_moore_reg <= 1'b0;
        count_moore_reg <= 0;
    end
    else begin
        state_moore_reg <= state_moore_next;
        count_moore_reg <= count_moore_next;
    end
end


always @(count_moore_reg, state_moore_reg) begin: COUNTER_EX_NEXT_STATE_LOGIC
    case (state_moore_reg)
        1'b0: begin
            count_moore_next = 0;
            state_moore_next = 1'b1;
        end
        1'b1: begin
            count_moore_next = (count_moore_reg + 1);
            if (((count_moore_reg + 1) == (6 - 1))) begin
                state_moore_next = 1'b0;
            end
            else begin
                state_moore_next = 1'b1;
            end
        end
    endcase
end



assign out_moore = count_moore_reg;

endmodule
```

4.6 Conclusion

In this chapter, we implemented various FSM designs using MyHDL. We followed the FSM-design-rules which are discussed in Verilog/SystemVerilog/VHDL tutorials. Further, the resultant Verilog and VHDL codes are exactly same as the Verilog/VHDL tutorial.

Chapter 5

More Design examples

5.1 Introduction

Note: In this chapter, some more useful designs are implemented which can be used to build the large systems. Here, we will re-implement the designs of the Chapter "Design examples" in the Verilog/VHDL tutorial. Please see the Verilog/SystemVerilog/VHDL tutorials for the explanation of the designs.

5.2 Linear Feedback Shift Register (LFSR)

- In the below code, LFSR is implemented which is used to generate the pseudo random numbers. Further, the generated random number is displayed on seven-segment display device and LEDs.

  Note: In the below code, the generated random number will be displayed on the seven-segment display device and LEDs. Since, the seven segment display device can show the number upto 'F' only, therefore the code will work for N = 3 only. While increasing the value of N, please comment the HEX0 connection from the Python code.

```
# rand_num_gen.py
# result is same as 'rand_num_generator.v' in the Verilog tutorial

from myhdl import *

# from mod_m_counter import mod_m_counter
from clock_tick import clock_tick
from hex_to_seven_seg import hex_to_seven_seg

@block
def rand_num_gen(clk, reset_n, q):
    N = 3
    # change the value of feedback_value in feedback_logic according to
    # the value of N
    r_reg = Signal(intbv(0)[N+1:0])
    r_next = Signal(intbv(0)[N+1:0])

    feedback_value = Signal(bool(0))

    @always(clk.posedge, reset_n.negedge)
    def current_state_logic():
```

(continues on next page)

(continued from previous page)

```
        if reset_n == 0 :
            r_reg.next = 1
        else :
            r_reg.next = r_next

    @always_comb
    def feedback_logic():
        # uncomment correct line based on the value of N

        feedback_value.next = r_reg[3] ^ r_reg[2] ^ r_reg[0] # N = 3
        # feedback_value.next = r_reg[4] ^ r_reg[3] ^ r_reg[0] # N = 4
        # feedback_value.next = r_reg[5] ^ r_reg[3] ^ r_reg[0] # N = 5
        # feedback_value.next = r_reg[9] ^ r_reg[5] ^ r_reg[0] # N = 9

    @always_comb
    def next_state_logic(): #
        r_next.next = concat(feedback_value, r_reg[N+1:1])

        # # use above or uncomment below two lines
        # r_next[N].next = feedback_value
        # r_next[N:0].next = r_reg[N+1:1]


    @always_comb
    def assign_output():
        q.next = r_reg

    return current_state_logic, feedback_logic, \
        assign_output, next_state_logic

@block
def rand_num_gen_top(CLOCK_50, reset_n, LEDG, HEX0):
    clk_pulse = Signal(bool(0))
    count = Signal(intbv(0)[4:0])

    # reduced clock rate
    clock_tick_inst = clock_tick(CLOCK_50, reset_n,
            clk_pulse, 50000000, 29)

    # rand_num_gen with new port names
    rand_num_gen_verilog = rand_num_gen(clk=clk_pulse,
            reset_n=reset_n, q=count)

    # below will not work for N > 3 as seven-segment can display upto F
    hex_to_seven_seg_test_inst = hex_to_seven_seg(count, HEX0)

    # send count to LEDG
    @always_comb
    def led_logic():
        LEDG.next = count

    return rand_num_gen_verilog, clock_tick_inst, \
                hex_to_seven_seg_test_inst, led_logic


def main():
    N = 3  # set the value of same as rand_num_gen
    clk = Signal(bool(0))
    reset_n = Signal(bool(0))
    q = Signal(intbv(0)[N+1:0])

```

(continues on next page)

(continued from previous page)

```
    # HEX0 will work for N = 3 only, as seven segment can display till F only
    HEX0 = Signal(intbv(0)[7:0])

    # rand_num_gen for simulation
    rand_num_gen_verilog = rand_num_gen(clk, reset_n, q)
    rand_num_gen_verilog.convert(hdl="Verilog", initial_values=True)

    # rand_num_gen for visual testing
    rand_num_gen_top_verilog = rand_num_gen_top(clk, reset_n, q, HEX0)
    rand_num_gen_top_verilog.convert(hdl="Verilog", initial_values=True)

if __name__ == '__main__':
    main()
```

- Following is the resultant Verilog code generate by the instance "rand_num_gen_top_verilog" in above listing. Load this design on the FPGA board; and the generated random numbers will be displayed on the seven-segment display device.

```
// File: rand_num_gen_top.v
// Generated by MyHDL 1.0dev
// Date: Fri Oct 13 15:57:19 2017


`timescale 1ns/10ps

module rand_num_gen_top (
    CLOCK_50,
    reset_n,
    LEDG,
    HEX0
);


input CLOCK_50;
input reset_n;
output [3:0] LEDG;
wire [3:0] LEDG;
output [6:0] HEX0;
reg [6:0] HEX0;

reg clk_pulse = 0;
wire [3:0] count;
reg [3:0] rand_num_gen_1_r_reg = 0;
wire [3:0] rand_num_gen_1_r_next;
wire rand_num_gen_1_feedback_value;
reg [28:0] clock_tick_1_mod_m_counter_1_count_reg = 0;
reg [28:0] clock_tick_1_mod_m_counter_1_count_next = 0;



always @(posedge clk_pulse, negedge reset_n) begin: RAND_NUM_GEN_TOP_RAND_NUM_GEN_1_
↪CURRENT_STATE_LOGIC
    if ((reset_n == 0)) begin
        rand_num_gen_1_r_reg <= 1;
    end
    else begin
        rand_num_gen_1_r_reg <= rand_num_gen_1_r_next;
    end
end


```

(continues on next page)

(continued from previous page)

```
43
44  assign rand_num_gen_1_feedback_value = ((rand_num_gen_1_r_reg[3] ^ rand_num_gen_1_r_
    ↪reg[2]) ^ rand_num_gen_1_r_reg[0]);
45
46
47
48  assign count = rand_num_gen_1_r_reg;
49
50
51
52  assign rand_num_gen_1_r_next = {rand_num_gen_1_feedback_value, rand_num_gen_1_r_reg[(3 +␣
    ↪1)-1:1]};
53
54
55  always @(posedge CLOCK_50, negedge reset_n) begin: RAND_NUM_GEN_TOP_CLOCK_TICK_1_MOD_M_
    ↪COUNTER_1_LOGIC_REG
56      if ((reset_n == 0)) begin
57          clock_tick_1_mod_m_counter_1_count_reg <= 0;
58      end
59      else begin
60          clock_tick_1_mod_m_counter_1_count_reg <= clock_tick_1_mod_m_counter_1_count_next;
61      end
62  end
63
64
65  always @(clock_tick_1_mod_m_counter_1_count_reg) begin: RAND_NUM_GEN_TOP_CLOCK_TICK_1_MOD_
    ↪M_COUNTER_1_LOGIC_NEXT
66      if (($signed({1'b0, clock_tick_1_mod_m_counter_1_count_reg}) == (50000000 - 1))) begin
67          clock_tick_1_mod_m_counter_1_count_next = 0;
68      end
69      else begin
70          clock_tick_1_mod_m_counter_1_count_next = (clock_tick_1_mod_m_counter_1_count_reg␣
    ↪+ 1);
71      end
72  end
73
74
75  always @(clock_tick_1_mod_m_counter_1_count_reg) begin: RAND_NUM_GEN_TOP_CLOCK_TICK_1_MOD_
    ↪M_COUNTER_1_COMPLETE_TICK_LOGIC
76      if (($signed({1'b0, clock_tick_1_mod_m_counter_1_count_reg}) == (50000000 - 1))) begin
77          clk_pulse = 1;
78      end
79      else begin
80          clk_pulse = 0;
81      end
82  end
83
84
85  always @(count) begin: RAND_NUM_GEN_TOP_HEX_TO_SEVEN_SEG_1_LOGIC
86      case (count)
87          'h0: begin
88              HEX0 <= 64;
89          end
90          'h1: begin
91              HEX0 <= 121;
92          end
93          'h2: begin
94              HEX0 <= 36;
95          end
96          'h3: begin
97              HEX0 <= 48;
```

(continues on next page)

(continued from previous page)

```
        end
        'h4: begin
            HEX0 <= 25;
        end
        'h5: begin
            HEX0 <= 18;
        end
        'h6: begin
            HEX0 <= 2;
        end
        'h7: begin
            HEX0 <= 120;
        end
        'h8: begin
            HEX0 <= 0;
        end
        'h9: begin
            HEX0 <= 16;
        end
        'ha: begin
            HEX0 <= 8;
        end
        'hb: begin
            HEX0 <= 3;
        end
        'hc: begin
            HEX0 <= 70;
        end
        'hd: begin
            HEX0 <= 33;
        end
        'he: begin
            HEX0 <= 6;
        end
        default: begin
            HEX0 <= 14;
        end
    endcase
end



assign LEDG = count;

endmodule
```

5.3 Random Access Memory (RAM)

In this section, the 'single port' and 'dual port' RAMs are implemented.

5.3.1 Single port RAM

- In the below code, a single port RAM is implemented,

```
# single_port_ram.py
# result is same as "single_port_RAM.v" of the Verilog tutorial

```

(continues on next page)

(continued from previous page)

```
from myhdl import *

@block
def single_port_ram(clk, we, addr, din, dout, addr_width=2, data_width=3) :
    ram_single_port = [Signal(intbv(0)[addr_width:0])
                for i in range(data_width)]

    @always(clk.posedge)
    def write_logic():
        """ write data to address 'addr' """
        if we == 1 :
            ram_single_port[addr].next = din

    @always_comb
    def read_logic():
        """ read data from address 'addr' """
        dout.next = ram_single_port[addr]

    return read_logic, write_logic

def main():
    addr_width = 3
    data_width = 3
    clk = Signal(bool(0))
    we = Signal(bool(0))
    addr = Signal(intbv(0)[addr_width:0])
    din = Signal(intbv(0)[data_width:0])
    dout = Signal(intbv(0)[data_width:0])

    single_port_ram_verilog = single_port_ram(clk, we, addr,
                din, dout, addr_width, data_width)

    single_port_ram_verilog.convert(hdl="Verilog", initial_values=True)

if __name__ == '__main__' :
    main()
```

- Following is the resultant Verilog code,

```
// File: single_port_ram.v
// Generated by MyHDL 1.0dev
// Date: Fri Oct 13 15:59:53 2017


`timescale 1ns/10ps

module single_port_ram (
    clk,
    we,
    addr,
    din,
    dout
);


input clk;
input we;
input [2:0] addr;
input [2:0] din;
output [2:0] dout;
wire [2:0] dout;

```

(continues on next page)

(continued from previous page)

```
reg [2:0] ram_single_port [0:3-1];

initial begin: INITIALIZE_RAM_SINGLE_PORT
    integer i;
    for(i=0; i<3; i=i+1) begin
        ram_single_port[i] = 0;
    end
end



// read data from address 'addr'

assign dout = ram_single_port[addr];

// write data to address 'addr'
always @(posedge clk) begin: SINGLE_PORT_RAM_WRITE_LOGIC
    if ((we == 1)) begin
        ram_single_port[addr] <= din;
    end
end

endmodule
```

5.3.2 Dual port RAM

- Following code implements a dual port RAM,

```
# dual_port_ram.py
# result is same as "dual_port_RAM.v" of the Verilog tutorial

from myhdl import *

@block
def dual_port_ram(clk, we, addr_rd, addr_wr, din, dout,
        addr_width=2, data_width=3) :

    ram_dual_port = [Signal(intbv(0)[addr_width:0])
                for i in range(data_width)]

    @always(clk.posedge)
    def write_logic():
        """ write data to address 'addr' """
        if we == 1 :
            ram_dual_port[addr_wr].next = din

    @always_comb
    def read_logic():
        """ read data from address 'addr' """
        dout.next = ram_dual_port[addr_rd]

    return read_logic, write_logic

def main():
    addr_width = 3
    data_width = 3
    clk = Signal(bool(0))
    we = Signal(bool(0))
    addr_rd = Signal(intbv(0)[addr_width:0])
```

(continues on next page)

(continued from previous page)

```
    addr_wr = Signal(intbv(0)[addr_width:0])
    din = Signal(intbv(0)[data_width:0])
    dout = Signal(intbv(0)[data_width:0])

    dual_port_ram_verilog = dual_port_ram(clk, we, addr_rd,
            addr_wr, din, dout, addr_width, data_width)

    dual_port_ram_verilog.convert(hdl="Verilog", initial_values=True)
if __name__ == '__main__' :
    main()
```

- Below is the resultant Verilog code,

```
// File: dual_port_ram.v
// Generated by MyHDL 1.0dev
// Date: Fri Oct 13 16:02:12 2017


`timescale 1ns/10ps

module dual_port_ram (
    clk,
    we,
    addr_rd,
    addr_wr,
    din,
    dout
);


input clk;
input we;
input [2:0] addr_rd;
input [2:0] addr_wr;
input [2:0] din;
output [2:0] dout;
wire [2:0] dout;

reg [2:0] ram_dual_port [0:3-1];

initial begin: INITIALIZE_RAM_DUAL_PORT
    integer i;
    for(i=0; i<3; i=i+1) begin
        ram_dual_port[i] = 0;
    end
end



// read data from address 'addr'

assign dout = ram_dual_port[addr_rd];

// write data to address 'addr'
always @(posedge clk) begin: DUAL_PORT_RAM_WRITE_LOGIC
    if ((we == 1)) begin
        ram_dual_port[addr_wr] <= din;
    end
end

endmodule
```

5.4 Read only memory (ROM)

- In the below code, a ROM is implemented which stores the values for the sevent-segment display device. The switches SW provide the address for the ROM and the ROM returns the corresponding pattern for the seven-segment display device.

```
# rom_seven_segment.py
# the result is same as "ROM_seven_segment.v" in Verilog tutorial
from myhdl import *

@block
def rom_seven_segment(addr, dout):
    # address width = 4
    # data width = 7
    data = (
                0b1000000,
                0b1111001,
                0b0100100,
                0b0110000,
                0b0011001,
                0b0010010,
                0b0000010,
                0b1111000,
                0b0000000,
                0b0010000,
                0b0001000,
                0b0000011,
                0b1000110,
                0b0100001,
                0b0000110,
                0b0001110
            )

    @always_comb
    def read_logic():
        dout.next = data[int(addr)]

    return read_logic

# top level design with updated port names
@block
def rom_ss_visual_test(SW, HEX0, LEDG):
    data = Signal(intbv(0)[7:0])
    rom_seven_segment_inst = rom_seven_segment(SW, data)

    @always_comb
    def display_logic():
        HEX0.next = data
        LEDG.next = data

    return display_logic, rom_seven_segment_inst

# verilog conversion
def main():
    SW = Signal(intbv(0)[4:0])
    LEDG = Signal(intbv(0)[7:0])
    HEX0 = Signal(intbv(0)[7:0])

    rom_seven_segment_verilog = rom_ss_visual_test(SW, HEX0, LEDG)
    rom_seven_segment_verilog.convert(hdl="Verilog", initial_values=True)

```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__' :
    main()
```

- The resultant Verilog code is shown below. Load this design to FPGA board and change the switch pattern, and the corresponding HEX value will be shown on seven-segment device.

```
// File: rom_ss_visual_test.v
// Generated by MyHDL 1.0dev
// Date: Fri Oct 13 16:08:29 2017


`timescale 1ns/10ps

module rom_ss_visual_test (
    SW,
    HEX0,
    LEDG
);


input [3:0] SW;
output [6:0] HEX0;
wire [6:0] HEX0;
output [6:0] LEDG;
wire [6:0] LEDG;

reg [6:0] data = 0;




assign HEX0 = data;
assign LEDG = data;


always @(SW) begin: ROM_SS_VISUAL_TEST_ROM_SEVEN_SEGMENT_1_READ_LOGIC
    case (SW)
        0: data = 64;
        1: data = 121;
        2: data = 36;
        3: data = 48;
        4: data = 25;
        5: data = 18;
        6: data = 2;
        7: data = 120;
        8: data = 0;
        9: data = 16;
        10: data = 8;
        11: data = 3;
        12: data = 70;
        13: data = 33;
        14: data = 6;
        default: data = 14;
    endcase
end

endmodule
```