# Programmable Digital Systems – Exercise 3

Goals:

- introduction to BCD encoding,
- hardware implementation of BIN to BCD conversion.
- hardware implementation of simple BCD adder.

Introduction:

*In computing and electronic systems, binary-coded decimal (BCD) (sometimes called natural binary-coded decimal, NBCD) or, in its most common modern implementation, packed decimal, is an encoding for decimal numbers in which each digit is represented by its own binary sequence. Its main virtue is that it allows easy conversion to decimal digits for printing or display, and allows faster decimal calculations. Its drawbacks are a small increase in the complexity of circuits needed to implement mathematical operations. Uncompressed BCD is also a relatively inefficient encoding—it occupies more space than a purely binary representation.*
*In BCD, a digit is usually represented by four bits which, in general, represent the decimal digits 0 through 9. Other bit combinations are sometimes used for a sign or for other indications (e.g., error or overflow).* The BCD encoding for the number 127 would be: *0001 0010 0111*

*Source: Wikipedia*

Exercise:

1. Run Active-HDL, create a new workspace and a new design.
2. Enter the program below into editor:

*testbench.v file:*

```
module mytestbenchmodule();

reg CLK;
initial CLK  <= 0;
always #50  CLK <= ~CLK;

reg RST;
initial
begin
    RST <= 1;
    RST <= #100 1;
    RST <= #500 0;
end

reg [31:0] i_dat;
reg i_stb;

initial
begin
    i_dat <= 0;
    i_stb <= 0;
    #1000;
    i_dat <= 8;
    i_stb <= 1;
    #100;
    i_stb <= 0;

end

bin_to_bcd_simple test
(
.CLK(CLK),
.RST(RST),

.I_DAT(i_dat),
.I_STB(i_stb),

.O_DAT(),
.O_STB()
);

endmodule
```

*bin_to_bcd_simple.v file:*

```
module bin_to_bcd_simple
(
input wire CLK,
input wire RST,

input wire  [31:0] I_DAT,
input wire         I_STB,

output wire [39:0] O_DAT,
output wire        O_STB
);
assign O_DAT[ 3: 0] = (I_DAT)%10;
assign O_DAT[ 7: 4] = (I_DAT/10)%10;
assign O_DAT[11: 8] = (I_DAT/100)%10;
assign O_DAT[15:12] = (I_DAT/1000)%10;
assign O_DAT[19:16] = (I_DAT/10000)%10;
assign O_DAT[23:20] = (I_DAT/100000)%10;
assign O_DAT[27:24] = (I_DAT/1000000)%10;
assign O_DAT[31:28] = (I_DAT/10000000)%10;
assign O_DAT[35:32] = (I_DAT/100000000)%10;
assign O_DAT[39:36] = (I_DAT/1000000000)%10;

assign O_STB = I_STB;

endmodule
```

3. Compile and run the simulation. Analyze waveforms of the bin_to_bcd_simple module.
4. Does the module work correctly? Check with different test numbers, like:: 16, 127…..
5. Why this module is NOT a good hardware implementation?
6. Enter the programs below:

*bin_to_bcd.v file:*

```
module bin_to_bcd
(
input wire CLK,
input wire RST,

input wire  [31:0] I_DAT,
input wire         I_STB,

output wire [39:0] O_DAT,
output wire        O_STB
);
reg [31:0] bin;
reg [32:0] bst;

always @(posedge CLK or posedge RST)
if (RST) begin
    bin <= 0;
    bst <= 0;
end else begin
    if (I_STB)
    begin
            bin <= I_DAT;
            bst <= 1;
    end else begin
            bin <= (bin<<1);
            bst <= (bst<<1);
    end
end

wire [3:0] bcd9,bcd8,bcd7,bcd6,bcd5,bcd4,bcd3,bcd2,bcd1,bcd0;
wire [9:0] ovr;

bcd_shl_1 b0 (.CLK(CLK), .RST(RST), .ADD1(bin[31]), .DAT(bcd0), .OVERFLOW(ovr[0]) );
bcd_shl_1 b1 (.CLK(CLK), .RST(RST), .ADD1(ovr[0]), . DAT(bcd1), .OVERFLOW(ovr[1]) );
bcd_shl_1 b2 (.CLK(CLK), .RST(RST), .ADD1(ovr[1]), . DAT(bcd2), .OVERFLOW(ovr[2]) );
. . . .
bcd_shl_1 b9 (.CLK(CLK), .RST(RST), .ADD1(ovr[8]), . DAT(bcd9), .OVERFLOW(ovr[9]) );

assign O_DAT = {bcd9,bcd8,bcd7,bcd6,bcd5,bcd4,bcd3,bcd2,bcd1,bcd0};
assign O_STB = bst[32];

endmodule
```

*bcd_shl_1.v file:*

```
module bcd_shl_1
(
    input wire CLK,
    input wire RST,

    input wire       ADD1,
    output reg [3:0] DAT,
    output reg       OVERFLOW
);

always @(posedge CLK or posedge RST)
if (RST) begin
    DAT <= 0;
    OVERFLOW <= 0;
end else begin
    DAT <=
            (DAT==0) ? {3'd0, ADD1} : // 0-1        no overflow
            (DAT==1) ? {3'd1, ADD1} : // 2-3        no overflow
            (DAT==2) ? {3'd2, ADD1} : // 4-5        no overflow
            (DAT==3) ? {3'd3, ADD1} : // 6-7        no overflow
            (DAT==4) ? {3'd4, ADD1} : // 8-9        no overflow
            (DAT==5) ? {3'd0, ADD1} : // 0-1
            (DAT==6) ? {3'd1, ADD1} : // 2-3
            (DAT==7) ? {3'd2, ADD1} : // 4-5
            (DAT==8) ? {3'd3, ADD1} : // 6-7
            (DAT==9) ? {3'd4, ADD1} : // 8-9
            {3'd7, ADD1};   // when error
    OVERFLOW <= ((DAT>=0) && (DAT<=4)) ? 1'b0 : 1'b1;
end
endmodule
```

7. Patch testbench.v file, so that is use bin_to_bcd module instead of bin_to_bcd_simple.
8. Compile and run the simulation. Analyze waveforms of the bin_to_bcd module.
9. Does the module work correctly? Check with different test numbers, like:: 16, 127…..
10. Look closer at „bcd0"…."bcd9" wires and find why the module isn't working correctly. Fix the problem and check again.
11. Update the program with „handshake"-style control lines - STB and ACK. Header of the bin_to_bcd file should include the following ports::

```
input CLK,
input RST,

input [31:0] I_DAT,
input        I_STB,
output       I_ACK,

output [39:0] O_DAT,
output        O_STB,
input         O_ACK
);
```

The simplest scheme of doing so, is to include busy/ready register:

```
reg ready;
```

which would be reset to „1", set to „0" when new data arrives, and turned to "1" after completion of job:

```
always @(posedge CLK or posedge RST)
if (RST) ready <= 1;
else if (O_ACK&O_STB) ready <= 1;
else if (I_STB) ready <= 0;

assign I_ACK = I_STB && ready;
```

and by changing control mechanism of data arrival:

`if (I_STB)` to a correct one: `if (I_STB && ready)` or alternatively `if (I_ACK)`

a)

If the module is to wait for output Signac O_ACK, it is crucial to store the result somewhere. It can be done with changing O_DAT and O_STB into registers (reg) and employing the following code instead of assignment „assign O_...="

```
always @(posedge CLK or posedge RST)
if (RST) begin
```

```
    O_STB <= 0;
    O_DAT <= 0;
end else if (bst[32]) begin
    O_STB <= 1;
    O_DAT <= {bcd9,bcd8,bcd7,bcd6,bcd5,bcd4,bcd3,bcd2,bcd1,bcd0};
end else if (O_ACK) O_STB <= 0;
```

b) alternatively, bcd_shl_1 module can be modified with addition of port:

```
    input wire       ENABLE,
```

which would be control non-blocking assignment of DAT register:

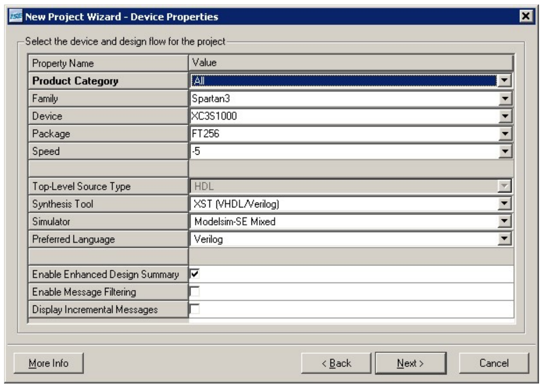
```
    if (ENABLE) DAT <=
```

Port ENABLE of bin_shl_1 modules should be connected to a new wire: „en”:

```
    wire en = ~O_STB || ready;
```

which also would control bit-shifting mechanism in bin_to_bcd module:

```
    end else if (en) begin
            bin <= (bin<<1);
            bst <= (bst<<1);
    end
```



New Project Wizard - Device Properties

Select the device and design flow for the project

| Property Name | Value |
|---|---|
| **Product Category** | All |
| Family | Spartan3 |
| Device | XC3S1000 |
| Package | FT256 |
| Speed | -5 |
| | |
| Top-Level Source Type | HDL |
| Synthesis Tool | XST (VHDL/Verilog) |
| Simulator | Modelsim-SE Mixed |
| Preferred Language | Verilog |
| | |
| Enable Enhanced Design Summary | ✓ |
| Enable Message Filtering | ☐ |
| Display Incremental Messages | ☐ |

12. Please add missing lines in testbench. Provide I_STB, I_ACK, O_STB i O_ACK controlling:

```
initial
begin
    i_dat <= 0;
    i_stb <= 0;
    #1000;
    i_dat <= 16;
    i_stb <= 1;
    #100;
    i_stb <= 0;

end
```

13. Compile and run the simulation. Analyze waveforms of the bin_to_bcd module.
14. With use of Xilinx ISE, compare synthesis results of two alternative implementations (a and b) and check whether program from point 2 can be synthesized.
15. Run Xilinx – ISE Project Navigator.
16. Create new project (File/New Project). Give it a name e.g. „bcd_test”. Set type of the project to „HDL”.
17. The settings of the project should be match those on figure above. Do not add new source files. As “existing files”, add files of only analyzed module e.g. „*bin_to_bcd_simple.v”*. Do NOT add testbench files!
18. From left panel select „Synthesize – XST”. Read maximum frequency and HDL synthesis results e.g.

```
Minimum period: 4.174ns (Maximum Frequency: 239.572MHz)
 . . .
Macro Statistics
# Registers                                            : 106
 Flip-Flops                                            : 106
# Comparators                                          : 10
 4-bit comparator lessequal                            : 10
=======================================================
```

Select „Implement design”. Read the real size of the circuit:

```
Logic Utilization:
Number of Slice Flip Flops:          106 out of  15,360    1%
Number of 4 input LUTs:              110 out of  15,360    1%
Logic Distribution:
  Number of occupied Slices:            56 out of   7,680    1%
```

19. Help for point 9:
    - is OVERFLOW set on time?
    - is bcd_shl_1 module reseted (cleared) properly upon new data arrival?