Summer Project

NAME: SIVAPRAKASAM D

ROLL NO: 2020105578

COURSE/DEPARTMENT: BE/ECE

PROJECT: 12 HR FORMAT DIGITAL CLOCK USING BASYS-3 FPGA BOARD

DOMAIN: Digital VLSI

AIM :

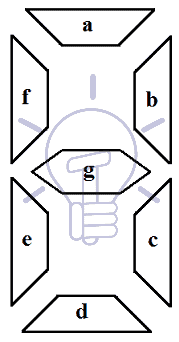
To design a 12 hr format digital clock using FPGA with the help of Verilog coding in vivado software because a better way to chck whether a designed circuit is working properly or not is by simulating it in a software which is why I have chosen vivado.

SOFTWARE REQUIRED: vivado

**Construction of 7-Segment Display**

There are seven small rectangular LEDs are present in a seven segment display and each led is called as segment. An additional led is also used in some displays for displaying decimal point to. Position of each led is set and named from A to G and this combination is used to display characters.

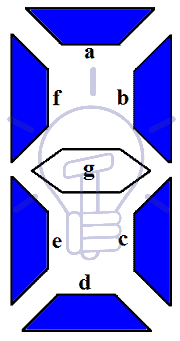
One pin from each led is brought out to give the signal to glow. The other pins of the LEDs are connected together to form a common pin. Led glows only when it is forward biased therefore if we make particular pin in forward bias mode we can make the characters of desired choice.

[](https://www.electricaltechnology.org/wp-content/uploads/2018/05/7-Segment-display.png)

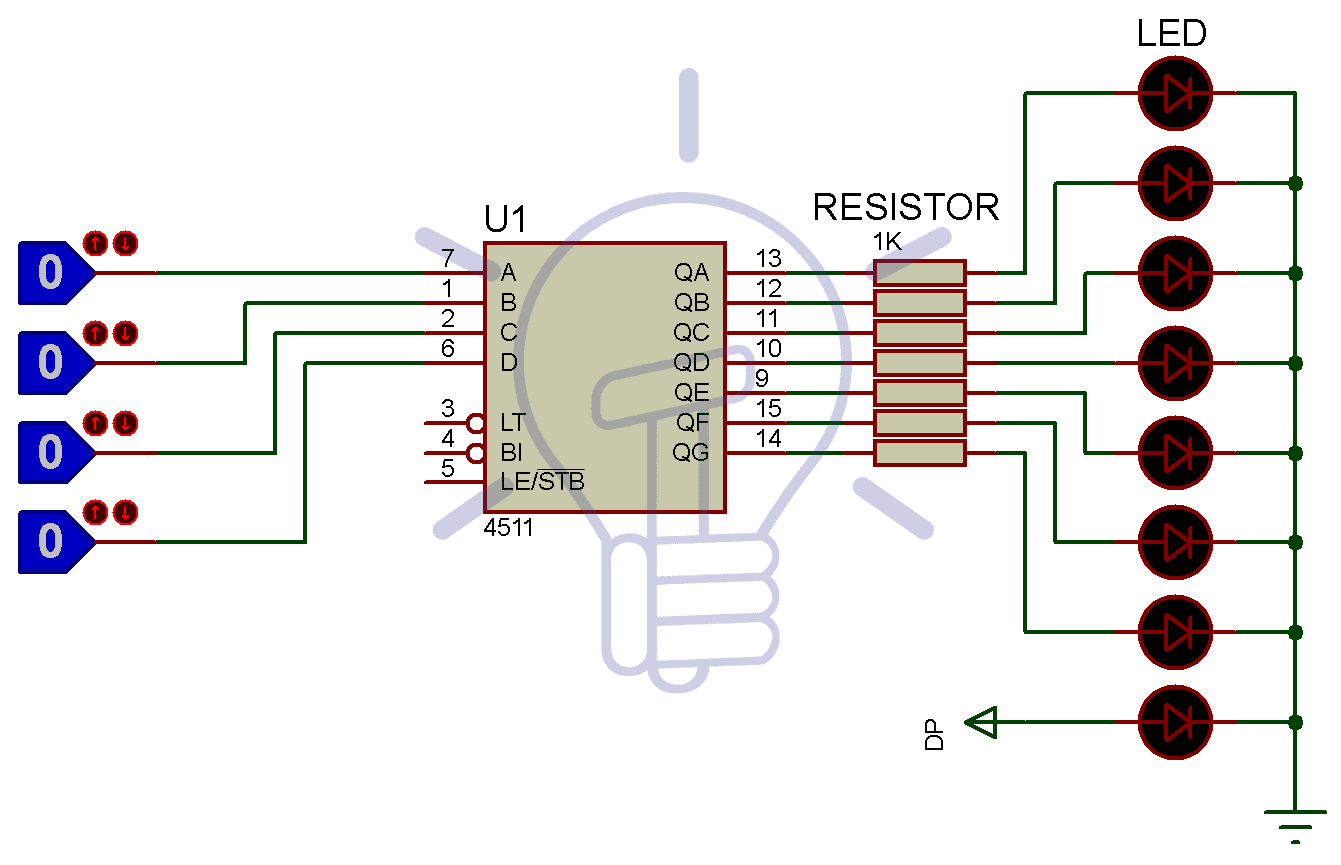
### ****Working of Seven Segment Display****

**Seven LED segments** of the display and their pins are “**a”, “b”, “c”, “d”, “e”, “f”**&**“g”** as shown in the figure given below. Each of the pins will illuminate the specific segment only.

We assume common cathode LED segment as our example. Suppose we want to display digit ‘0’, in order to display 0, we need to turn on “a”, “b”, “c”, “d”, “e”, “f” and turn-off the “g”. which would look like the figure given below.

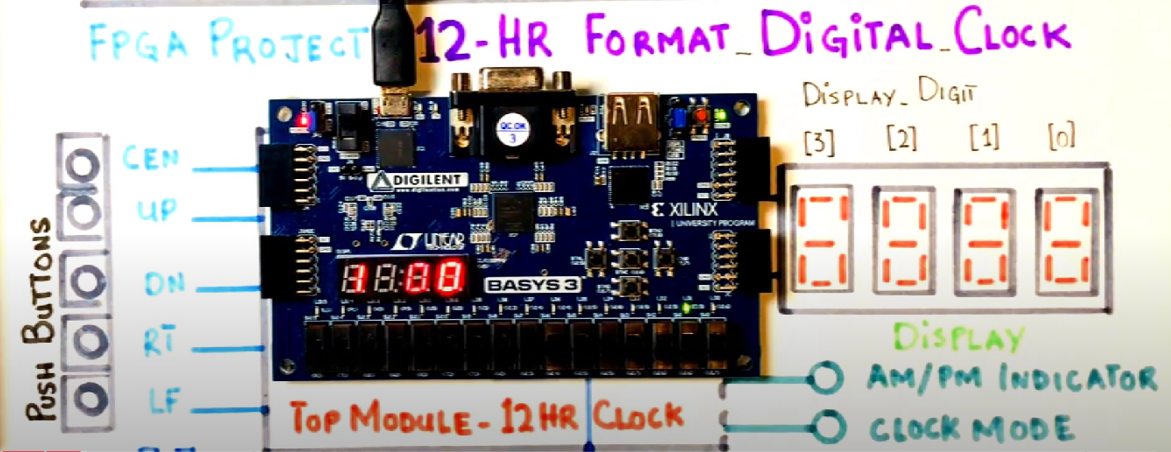
[](https://www.electricaltechnology.org/wp-content/uploads/2018/05/7-LED-segments-of-display-and-their-pins-a-b-c-d-e-f-g.png)

It is working the same for other numbers like 0, 1, 2, 4, 5, 6, 7, 8 and 9 as shown in the above fig 1. The following section will show how the different types of seven segment display works with their circuit diagrams.

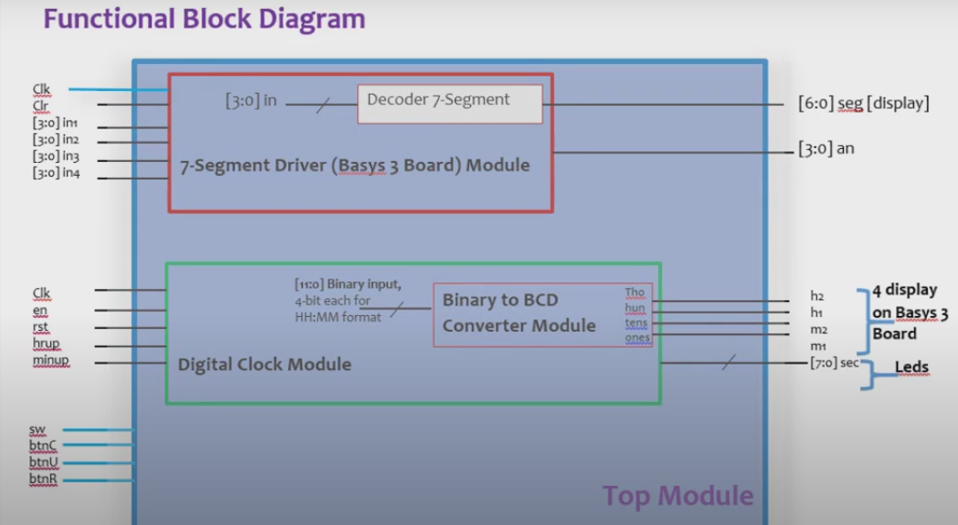


**BINARY TO BCD CONVERTER:**

The Binary to BCD Converter is used to convert a binary (Base-2) number to a BCD (Binary-coded decimal).

****

FUNCTIONAL DIAGRAM:



## What is an FPGA

We can think of an FPGA as if it were a massive collection of unconnected digital components.

This includes basic components such as [multiplexors](https://www.electronics-tutorials.ws/combination/comb_2.html)and [logic gates](https://en.wikipedia.org/wiki/Logic_gate) as well as more complex components like[DSP cores](https://www.xilinx.com/products/technology/dsp.html).

When we program an FPGA, we are actually creating connections between these different components to create a complex system.

All of this means that we are fundamentally designing hardware when we create an FPGA based design.

As a result of this, we can design a number of circuits which run in [parallel](https://www.allaboutcircuits.com/video-lectures/parallel-circuits/)to each other.

This means that FPGAs are capable of performing a large number of different operations at the same time. This is a major advantage over software approaches, which must be run [sequentially by a CPU](http://www.mathcs.emory.edu/~cheung/Courses/170/Syllabus/01/intro-computer2.html).

In addition, we also have much more control over the timing of our design in an FPGA. We can estimate to within a few nanoseconds how long operations will take to complete in an FPGA. Again, we could not do this if we used a CPU to implement our design.

As a result of these features, FPGA designs can be much quicker than the equivalent implementation in a microcontroller.

The drawback is that they tend to be more difficult to work with.

As we will come to see, this is not because designing FPGAs is inherently more difficult.

The major difference is that there is a much smaller community of people who develop FPGAs. As a result, we have much less libraries and open source code available to us.

The exact structure of a CLB depends on the actual chip vendor. More information about the structure of the CLB in a Xilinx FPGA is available [here](https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf)while the structure for an Intel FPGA is available [here](https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01003.pdf).

#### Configurable Logic Blocks

The configurable logic blocks are the main building block of an FPGA. We can think of CLBs as if they were a reconfigurable logic gate which we can configure to perform different logical functions in our design.

A typical CLB consist of a few inputs, [look-up tables](https://en.wikipedia.org/wiki/Lookup_table) (LUT), multiplexors and some RAM. LUTs are simply small blocks of memory which are programmed to implement a given logical function.

#### IO Blocks

In an FPGA, the IO blocks provide an connection to circuits which are external to the FPGA. They are connected directly to the physical pins of an FPGA.

We can program the IO blocks in an FPGA so that they function as inputs, outputs or a mixture of both. We can also select different logic standards to apply to our IO, such as 3V3 or 1V8 [LVCMOS](https://en.wikipedia.org/wiki/LVCMOS) or [LVDS](https://semiengineering.com/knowledge_centers/communications-io/off-chip-communications/i-o-enabling-technology/lvds-low-voltage-differential-signaling/).

In modern FPGAs it is also common for us to have dedicated pins for special functions, such as high speed data transfers.

The exact structure of an IO block in an FPGA varies slightly between different vendors.

However, they typically always include simple registers, such as D type flip flops, and tri-state buffers.

#### FPGA Program Memory

Once we have finished with our FPGA design, we create a programming file which tells the FPGA how it should be configured.

The FPGA uses this to configure the internal LUTs and interconnections, as well as other internal components such as PLLs and DSP cores.

Most modern devices use SRAM type memory to store this information due to its speed.

One downside of this is that we need an external memory chip to store our program. This is because SRAM is volatile memory, meaning it can't retain its state when it is powered off.

Some vendors offer FPGAs which use flash memory instead of SRAM, as this memory type is non-volatile. However, SRAM based FPGAs can typically run with higher clock speeds which is why they are more popular.

## FPGA Development Process

We can broadly divide the FPGA development process into three different stages – design, verification and implementation.

In the design phase, we focus on transferring our initial concept or idea into an actual FPGA device. This normally involves architecting the chip, or breaking it down into smaller blocks to form a complete design. We then implement each of these smaller blocks using a HDL language, or some other approach.

The implementation stage takes our HDL design and converts this into a programming file for our FPGA.

The verification process feeds into both of these processes. This involves testing and analysing our design and implementation to ensure that it functions correctly.

The image below shows the stages in the development life cycle and how they are interlinked.

### FPGA Design Process

The first stage in the development of an FPGA is the design.

We normally start this by architecting the chip in some way. This involves breaking the design into a number of smaller blocks to simplify the coding.

This may be a formal process involving block diagrams and discussions with other engineers. This is especially likely if we are working on a complex design in a professional capacity.

Similarly, we may just use pseudo code to create a basic idea of how our design will look if we are working on a small project.

The next stage is the creation of a function model of our design. We normally use one of the major HDL languages for this purpose.

There are two main styles of modelling which we use for this process. These are commonly referred to as Register Transfer Level (RTL) and Gate Level.

When we use the RTL approach we are describing how data flows between [flip flops](https://www.electronicsforu.com/technology-trends/learn-electronics/flip-flop-rs-jk-t-d) in the FPGA.

This means that we are writing code to explicitly describe the behaviour of our FPGA in terms of logic, RAM and state machines.

We use gate level modelling to define the interconnection of different pre-existing components. These components are instances of integrated FPGA elements, such as PLLs, LUTs or register cells.

Although we use RTL for the majority of our FPGA design work, most projects feature a mixture of both approaches.

### FPGA Verification Process

After writing a model of our design, we then need to prove that it works. The process which we use for this is known as verification.

The first stage of this simulation of our design. For this purpose, we create a [test bench](https://fpgatutorial.com/how-to-write-a-basic-verilog-testbench/) which generates a number of inputs to our design.

We then check that the FPGA outputs are what we expect them to be, either through manual inspection or through self checking code.

We can repeat this process on our functional code and on our post place and route netlists. This is a model of the FPGA which is created by our software tools when we implement the FPGA. This model includes information about the internal timing of our FPGA and so is more representative of the final implementation.

Typically, simulation is the main process that is involved the verification of our design. We also normally complement this with hardware testing to ensure that the FPGA interfaces as expected with all external circuity.

However, as FPGA designs have become more complex, other techniques have become popular.

More modern verification activities include [hardware in the loop](https://www.mathworks.com/videos/fpga-based-hardware-in-the-loop-hil-simulation-for-power-electronics-1561644009561.html) (HiL) and [emulation](https://www.aldec.com/en/company/blog/154--emulation-in-fpga).

In both cases, this involves running code on our target device and feeding back data to simulation software. This allows us to run specific, structured tests on our device in near real time.

This is an advantage as post place and route netlists are extremely slow to simulate in comparison to function code. A simulation which takes an hour to run with functional code can easily take a day or more to run with a post place and route netlist.

### FPGA Implementation Process

Once we have written our code and proved that it works, we then need to program this design on our FPGA.

There are actually three stages involved in this process - synthesis, place and route and programming file generation.

The synthesis process transforms our functional code into a number of interconnected gate level macros. These are models of the internal FPGA cells.

This process creates a netlist which describes what the contents of the programmed FPGA should be. We can think of this as being equivalent to a circuit diagram in traditional circuit design.

After we synthesise our design, we then map the netlist to the actual FPGA resources.

The first part of this is mapping the macros to the physical cells in the FPGA using a process known as placement. We can think of this as being equivalent to placing components on a circuit board when we design PCBs in traditional electronic design.

The second stage involves routing the interconnections between the different cells in a process known as routing. This process is equivalent to routing traces in a PCB.

It is normally necessary to run the place and route process several times in order to meet the timing requirements of our design. The place and route tool is responsible for scheduling these multiple runs based on our configuration.

We also perform a static timing analysis (STA) as part of the place and route process. This analysis calculates the delays for all of the timing paths in our FPGA and ensures that our design meets with our timing requirements.

If our design fails the STA then we can't guarantee that our FPGA will work reliably. When this happens we either have to run the implementation process again with different settings or we must change our design.

The final stage in implementation process is generating the programming file which configures the FPGA.

The block diagram below gives an overview of the entire FPGA implementation flow.

#### Synthesis and Place and Route Software Tools

There are a number of different software tools which we can use to implement our design.

Both of the major FPGA vendors ([Xilnix](https://www.xilinx.com/) and [Intel](https://www.intel.com/content/www/us/en/products/programmable.html)) offer free synthesis tools which are suitable for most projects.

In addition to this, there are also a number of [open source](https://opensource.com/resources/what-open-source) synthesis tools which we can use. The most popular of these tools is [yosys](http://www.clifford.at/yosys/) which is frequently used with [Lattice FPGAs](https://www.latticesemi.com/fpga).

We can also use paid tools such as [Synplify Pro](https://www.synopsys.com/implementation-and-signoff/fpga-based-design/synplify-pro.html)which typically deliver more optimized netlists.

There are no third party place and route tools for Xilinx or Intel parts, meaning we must use the vendor specific tools. These are freely available for download, although paid versions are also available.

**VERILOG CODES:**

module Decoder\_7\_segment(

input [3:0] in, //4 bits going into the segment

output reg [6:0] seg //display the BCD number on a 7-segment

);

always @(in)

begin

case(in)

4'b0000: seg=7'b0000001; //active low logic here, this displays zero on the seven segment

4'b0001: seg=7'b1001111; //"1"

4'b0010: seg=7'b0010010;//"2"

4'b0011: seg=7'b0000110;//3

4'b0100: seg=7'b1001100;//4

4'b0101: seg=7'b0100100;//5

4'b0110: seg=7'b0100000;//6

4'b0111: seg=7'b0001111;//7

4'b1000: seg=7'b0000000;//8

4'b1001: seg=7'b0001100;//9

4'b1010: seg=7'b0001000; //A

4'b1011: seg=7'b0000011;//B

4'b1100: seg=7'b1000110;//C

4'b1101: seg=7'b0100001;//D

4'b1110: seg=7'b0000110;//E

4'b1111: seg=7'b0001110;//F

endcase

end

endmodule

module sevenseg\_driver(

input clk,

input clr,

input [3:0] in1,

input [3:0] in2,

input [3:0] in3,

input [3:0] in4,

output reg [6:0] seg,

output reg [3:0] an

);

wire [6:0] seg1, seg2, seg3, seg4;

reg [12:0] segclk; //for turning segment displays one by one on the board, 8192, 0-8191

localparam LEFT=2'b00, MIDLEFT=2'b01, MIDRIGHT=2'b10, RIGHT =2'b11;

reg [1:0] state=LEFT;

//instantiating the seven segment decoder four times

Decoder\_7\_segment disp1(in1,seg1);

Decoder\_7\_segment disp2(in2,seg2);

Decoder\_7\_segment disp3(in3,seg3);

Decoder\_7\_segment disp4(in4,seg4);

always @(posedge clk)

segclk<= segclk+1'b1; //counter goes up by 1

always @(posedge segclk[12] or posedge clr)

begin

if (clr==1)

begin

seg<=7'b0000000;

an<=4'b0000;

state<=LEFT;

end

else

begin

case(state)

LEFT:

begin seg<=seg1;

an<=4'b0111;

state<=MIDLEFT;

end

MIDLEFT:

begin

seg<=seg2;

an<=4'b1011;

state<=MIDRIGHT;

end

MIDRIGHT:

begin

seg<=seg3;

an<=4'b1101;

state<=RIGHT;

end

RIGHT: begin

seg<=seg4;

an<=4'b1110;

state<=LEFT;

end

endcase

end

end

endmodule

module binarytoBCD(

input [11:0] binary, //12 bit input data that would come-in

output reg [3:0] thos, //outputs thousands

output reg [3:0] huns,//hundreds,

output reg [3:0] tens,//tens

output reg [3:0] ones//ones

);

reg [11:0] bcd\_data=0;

always @(binary) //1250

begin

bcd\_data=binary; //1250

thos=bcd\_data/1000;//1250/1000=1, ""1""

bcd\_data=bcd\_data%1000;//1250/1000= 250

huns =bcd\_data/100;//250/100 =2, ""2""

bcd\_data=bcd\_data%100;//250/100, remainder here is 50

tens = bcd\_data/10;//50/10 = ""5""

ones = bcd\_data%10; //5/10 = ""0""

end

endmodule

module digital\_clock(

input clk,

input en,

input rst, //all of these are our inputs and outputs

input hrup,

input minup,

output [3:0] s1,

output [3:0] s2,

output [3:0] m1,

output [3:0] m2,

output [3:0] h1,

output [3:0] h2

);

//time display

//h2 h1 : m2 m1

reg [5:0] hour=0, min=0, sec=0;//60 for min/sec count, you will require 6 bits, 64, 0-63

integer clkc=0;

localparam onesec=100000000; //1second

always @ (posedge clk)

begin

//reset clock

if (rst==1'b1)

{hour,min,sec}<=0;

//set clock

else if (minup==1'b1)//minup button is pressed

if (min==6'd59)

min<=0;

else min<=min+1'd1;

else if (hrup ==1'b1)

if (hour==6'd23)

hour<=0;

else hour<=hour+1'd1;

//count

else if (en==1'b1)

if (clkc==onesec)

begin

clkc<=0;

if(sec==6'd59)

begin

sec<=0;

if (min==6'd59)

begin min<=0;

if (hour==6'd23)

hour<=0;

else

hour<=hour+1'd1;

end

else

min<=min+1'd1;

end

else

sec<=sec+1'd1;

end

else

clkc<=clkc+1;

end

//instantiating the binarytoBCD module here to convert the numbers and display the on the 7-segment

binarytoBCD secs(.binary(sec), .thos(), .huns(), .tens(s2), . ones(s1));

binarytoBCD mins(.binary(min), .thos(), .huns(), .tens(m2), . ones(m1));

binarytoBCD hours(.binary(hour), .thos(), .huns(), .tens(h2), . ones(h1));

endmodule

module Top\_Module(

input clk,//fpga clokc

input sw, //switch[0] to enable the clock

input btnC, //reset the clock

input btnU,//hour increment

input btnR, //min increment

output [6:0] seg,

output [3:0] an,

output [7:0] led //display seconds,

);

wire [3:0] s1, s2, m1, m2, h1, h2;

reg hrup, minup;

wire btnCclr, btnUclr, btnRclr;

reg btnCclr\_prev, btnUclr\_prev, btnRclr\_prev;

//instantiate the Debounce Module that I just added

debounce dbC(clk,btnC,btnCclr);

debounce dbU(clk,btnU,btnUclr);//hour up

debounce dbR(clk,btnR,btnRclr);// min up

//instantiate seven segmren t driver and digital clock modules

sevenseg\_driver seg7(clk,1'b0, h2, h1,m2,m1, seg, an);//HH:MM

digital\_clock clock(clk,sw,btnCclr, hrup,minup,s1,s2,m1,m2,h1,h2);

//detting up the logic for the clock, hrup and minup using the pushbuttons

always @(posedge clk)

begin

btnUclr\_prev <= btnUclr;//hrup

btnRclr\_prev <= btnRclr; //minup

if (btnUclr\_prev ==1'b0 && btnUclr ==1'b1) hrup <=1'b1; else hrup<=1'b0;

//hrup button is zero and clr button is high then hrup is pressed, active

if (btnRclr\_prev ==1'b0 && btnRclr ==1'b1) minup <=1'b1; else minup<=1'b0;

//minup button is zero and clr button is high then minup is pressed, active

end

assign led[7:0]={s2,s1};

endmodule

module D\_FF(

input clk,

input D,

output Q,

output Qbar

);

wire clk, D;

reg Q, Qbar;

always @ (posedge clk)

begin

Q<=D;

Qbar<=!Q;

end

endmodule

module debounce(

input btn,clk,

output btn\_clr);

wire clk\_out;

wire Q1, Q2, Q2\_bar;

Slow\_Clock\_4Hz u1(clk, clk\_out);

D\_FF d1(clk\_out, btn, Q1);

D\_FF d2(clk\_out, Q1, Q2);

assign Q2\_bar = ~Q2;

assign btn\_clr = Q1 & Q2\_bar;

endmodule

module Slow\_Clock\_4Hz(

input clk, //input clock of the basys 3 board 100 MHz

output clk\_out //4Hz slow clock

);

reg [25:0] count=0;//2^25 equals a number which is greater than 12.5 million

reg clk\_out;

always @(posedge clk)

begin

count<=count+1;

if (count==12\_500\_000)//equals

begin

count<=0;//reset itself to zero

clk\_out=~clk\_out;//inverts the clock

end

end

endmodule

**Constraint file :**

**## Clock signal**

**set\_property PACKAGE\_PIN W5 [get\_ports clk]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports clk]**

**create\_clock -add -name sys\_clk\_pin -period 10.00 -waveform {0 5} [get\_ports clk]**

**set\_property CLOCK\_DEDICATED\_ROUTE FALSE [get\_nets btnC\_IBUF]**

**set\_property CLOCK\_DEDICATED\_ROUTE FALSE [get\_nets btnR\_IBUF]**

**set\_property CLOCK\_DEDICATED\_ROUTE FALSE [get\_nets btnU\_IBUF]**

**## Switches**

**set\_property PACKAGE\_PIN V17 [get\_ports {sw}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {sw}]**

**#7 segment display**

**set\_property PACKAGE\_PIN W7 [get\_ports {seg[6]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {seg[6]}]**

**set\_property PACKAGE\_PIN W6 [get\_ports {seg[5]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {seg[5]}]**

**set\_property PACKAGE\_PIN U8 [get\_ports {seg[4]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {seg[4]}]**

**set\_property PACKAGE\_PIN V8 [get\_ports {seg[3]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {seg[3]}]**

**set\_property PACKAGE\_PIN U5 [get\_ports {seg[2]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {seg[2]}]**

**set\_property PACKAGE\_PIN V5 [get\_ports {seg[1]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {seg[1]}]**

**set\_property PACKAGE\_PIN U7 [get\_ports {seg[0]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {seg[0]}]**

**#set\_property PACKAGE\_PIN V7 [get\_ports dp]**

**# set\_property IOSTANDARD LVCMOS33 [get\_ports dp]**

**set\_property PACKAGE\_PIN U2 [get\_ports {an[0]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {an[0]}]**

**set\_property PACKAGE\_PIN U4 [get\_ports {an[1]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {an[1]}]**

**set\_property PACKAGE\_PIN V4 [get\_ports {an[2]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {an[2]}]**

**set\_property PACKAGE\_PIN W4 [get\_ports {an[3]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {an[3]}]**

**#leds**

**# LEDs**

**set\_property PACKAGE\_PIN U16 [get\_ports {led[0]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {led[0]}]**

**set\_property PACKAGE\_PIN E19 [get\_ports {led[1]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {led[1]}]**

**set\_property PACKAGE\_PIN U19 [get\_ports {led[2]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {led[2]}]**

**set\_property PACKAGE\_PIN V19 [get\_ports {led[3]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {led[3]}]**

**set\_property PACKAGE\_PIN W18 [get\_ports {led[4]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {led[4]}]**

**set\_property PACKAGE\_PIN U15 [get\_ports {led[5]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {led[5]}]**

**set\_property PACKAGE\_PIN U14 [get\_ports {led[6]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {led[6]}]**

**set\_property PACKAGE\_PIN V14 [get\_ports {led[7]}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports {led[7]}]**

**#Pushbuttons**

**set\_property PACKAGE\_PIN U18 [get\_ports btnC]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports btnC]**

**set\_property PACKAGE\_PIN T18 [get\_ports {btnU}]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports btnU]**

**set\_property PACKAGE\_PIN T17 [get\_ports btnR]**

**set\_property IOSTANDARD LVCMOS33 [get\_ports btnR]**

**OUTPUT:**

