Experiments with FPGA DE2-115

P443 Integrated Experimental Lab I

Submitted By: Nisarg Vyas
Roll No. 1711091

School of Physical Sciences
National Institute of Science Education and Research

February 7, 2021

Aknowledgement

First and foremost thanks to Dr. Santosh Babu Gunda, Scientific officer-E, SPS NISER. His encouragement made it possible for me to complete the lab targets before time and with his guidance I sought to apply the device to a physics experiment. His enthusiam for the lab is unparalleled and he shares the joys of troubleshooting with each of the lab students.
Special thanks to Dr. Kartikeswar Senapati for asking good questions in lab viva, they helped me think and understand some of the important aspects of the experiments. Finally many due thanks to the department for providing us such an opportunity to explore physics in the lab.

Preface

This report has been written in a way that it can be used to reproduce the lab exercises. Throughout the report, the lab exercises are complimented with a 'remarks' section that warns the reader of possible mistakes and provides insights, if there is a need. By the end of the report, one can expect the reader to have learned basics of fpga programming in VHDL and a general understanding of the pin connections on the boards and how to use them. All the exercises were implemented on DE2-115 fpga board and written in VHDL -an industry standard hardware description language. Quartus (edition 20.1) was used as the user-interface application to compile and write the codes on the fpga board.

Contents

1 Introduction to FPGA . . . 3
1.1 Altera DE2 Board . . . 3

2 Bacics of VHDL . . . 6

3 Experiments with FPGA . . . 7
3.1 Getting started . . . 7
3.2 Implimenting logic gates and building a multiplexer . . . 8
3.3 Adder and multiplier . . . 8
3.4 Counters . . . 10
3.5 Internal memory . . . 11

4 Application: Gamma-gamma correlation spectrometry . . . 11

5 Conclusion . . . 13

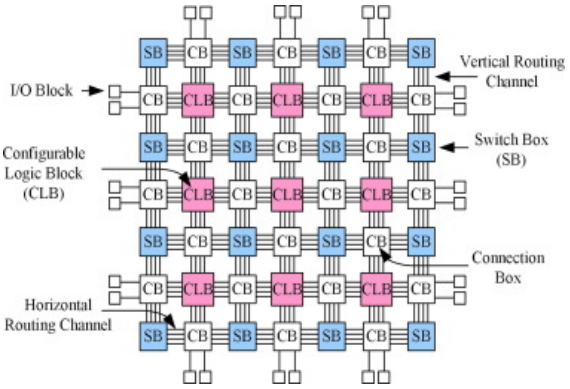
Appendix: Codes . . . 14

1 Introduction to FPGA

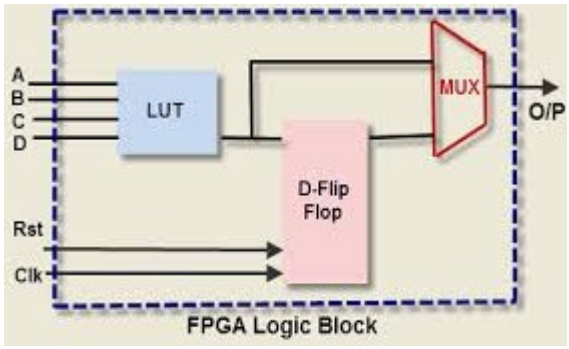
The first commercial fpga was launched in around 1985 with fewer than 9000 gates. The versatility of the product quickly founded its diverse applications and since then, the industry has grown to produce latest FPGAs with upto 50 million gates. In the Large-Hadron-Collider experiment, with upto one billion proton-proton collisions, the wealth of the data produced is immense and needs to be analysed quickly. For this and for possible future upgrades in the experiment where even higher number of collisions would be achieved, the state-of-the-art technology in fpga has also replaced the traditional CPUs.

Dive-in into the name. FPGA stands for Field-Programmable Gate Array. The term 'field-programmable' comes from its greatest and one-of-a-kind feature: it is designed to be configured post-manufacturing, by any customer with a PC/laptop. The term 'gate array' tells about its architecture: the device is composed of an array of logic blocks and programmable interconections that can be re-configured by the user. The fpga chip is connected to external hardware via I/O blocks which are provided towards the outer edge of a FPGA board.

The Logic Block is compact and provides advanced features with efficient logic utilization. Each Logic Block features: (a) a four-input look-up table (LUT), which is a function generator that can implement any function of four or fewer variables, (b) a programmable register, (c) a carry chain connection, (d) a register chain connection, (e) the ability to drive all types of interconnects: local, row, column, register chain, and direct link interconnects, (f) support for register packing, and (g) support for register feedback



(a) Architecture of FPGA. Labelled.
Image credit: Sciencedirect.com



(b) A single logic block. Image credit: ElProcus.com

1.1 Altera DE2 Board

The Altera DE2-115 board has several hardware connected on it. These are:

1. **Altera Cyclone® IV 4CE115 FPGA device**
2. **Altera Serial Configuration device – EPCS64**
3. **USB Blaster (on board) for programming; both JTAG and Active Serial (AS) programming modes are supported**
4. **2MB SRAM**, two 64MB SDRAM, and 8MB Flash memory
5. SD Card socket
6. 4 Push-buttons

7. **18 Slide switches**
8. **18 Red user LEDs and 9 green user LEDs**
9. **50MHz oscillator for clock sources**
10. 24-bit CD-quality audio CODEC with line-in, line-out, and microphone-in jacks
11. VGA DAC (8-bit high-speed triple DACs) with VGA-out connector
12. TV Decoder (NTSC/PAL/SECAM) and TV-in connector
13. 2 Gigabit Ethernet PHY with RJ45 connectors
14. USB Host/Slave Controller with USB type A and type B connectors
15. RS-232 transceiver and 9-pin connector
16. PS/2 mouse/keyboard connector and an IR Receiver
17. 2 SMA connectors for external clock input/output
18. **One 40-pin Expansion Header with diode protection**
19. One High Speed Mezzanine Card (HSMC) connector
20. 16x2 LCD module and **8 7-segment displays**

Where the ones used for my purposes in the experiments have been typed in bold.

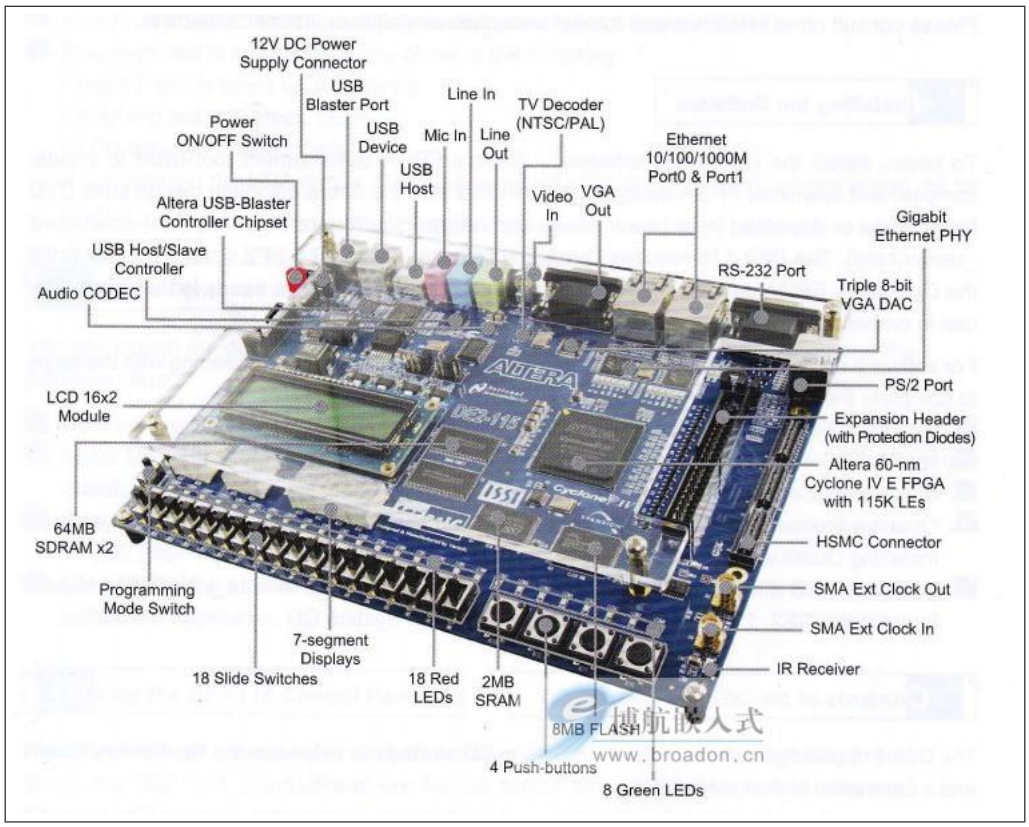


Figure 1.2: Labeled FPGA board. Image credits: broadon.com

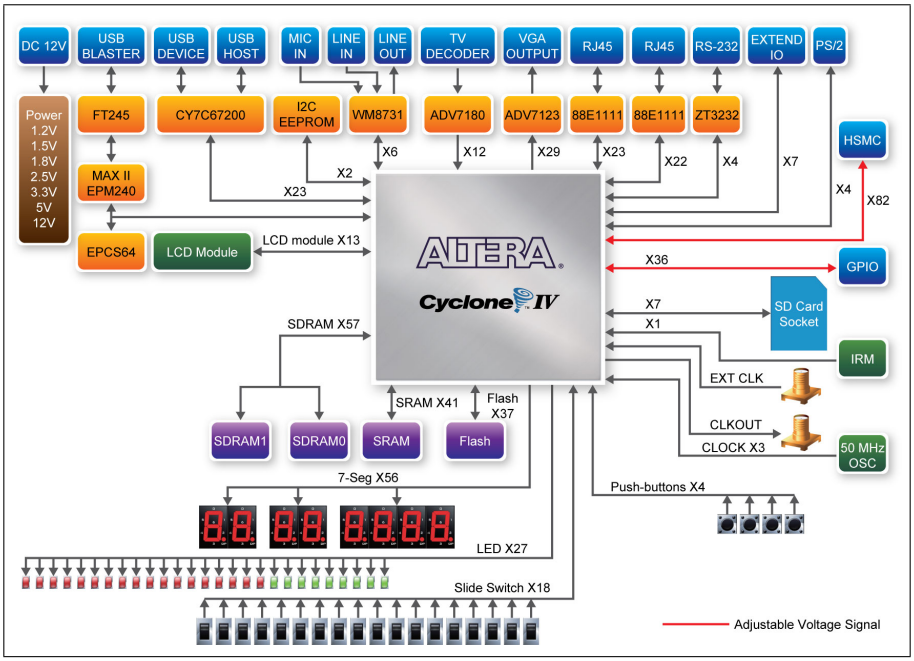


Figure 1.3: Schematic diagram of an FPGA board. Image courtesy: intel.in

Active serial and JTAG

The board has a toggle switch which shifts the mode of operation of FPGA between ”prog” and ”run”. The board can be programmed in one of the two modes: active serial programming or JTAG programming. If you choose to compile a code and write it on the board using JTAG mode the configuration shall stay on the board only as long as it is powered. The board forgets the configuration on switching off and has to be reconfigured. JTAG mode is implimented with the switch in ”run” position. The active serial programming is done by compiling the code and generating an executable file for this particular mode (look up the manuals for more details) and pushing the switch in ”prog” position. This mode writes the code on the FPGA memory and the configuration is retained even after the board is powered off.

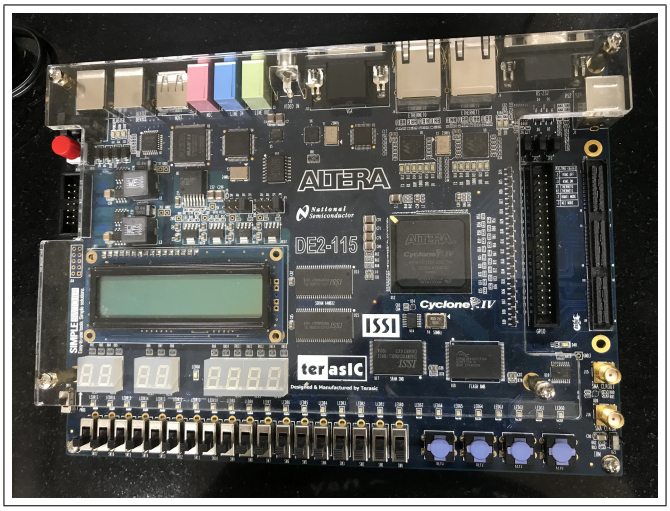


Figure 1.4: The FPGA board from the lab. Courtesy: SPS, NISER

2 Bacics of VHDL

The full-form is mouthful: Very High Speed Integrated Circuit Hardware Description Language. It is used to specify design logic for electonic systems with signals and hardware interfaces. The language is revised and standardised by Institute of Electrical and Electronics Engineers (IEEE). VHDL, being quite a popular industrially used language, a multitude of tutorials and books are available online. However, I present a small introduction here for completeness.
VHDL contains 6 basic data-types: (1) Boolean, (2) Integer, (3) Floating point (use with keyword 'real'), (4) Character, (5) Time and (6) Bit. Standard library also contains definition for 'std_logic' which is prefered over the Bit. Several composite data-types can be constructed from these basic ones; for example: an array containing multiple instances of one type of data-type, or a record which can contain different data-types.
Several operators that can be used are listed in the following table aranged in the order of preference given to them while evaluating expressions.

Preference	**Operator type**	**Operations**
1	Arithmetic operators	+, -, /, * & -concatenation rem -remainder mod -modulus abs -absolute value ** -exponential
2	Shift operators	sll -shift logical left slr -shift logical right sla -shift arithmetic left sra -shift arithmetic right rol -rotate left ror -rotate right
3	Relational operators	= equal to /= not equal to <, <=, >, >=
4	Logic operators	and, or, nand, nor, xor, xnor, not

Table 2.1

Implementing VHDL codes using integrated development environments (IDEs) like the open source Quartus is easy and can be learned through the plenty of documents and tutorials available over the internet. As all the experiments were done in VHDL using Qurtus, and all the codes presented in the appendice are built on it, some remarks are neccesary here.
The language is NOT case specific when it comes to keywords. Quartus codes show all the indentifiers pertaining to the structure of the code in blue. The built-in data-type identifiers are shown in pink, user defined variable names are in black and everything that follows '–' is commented out and shown in green.
There is a basic structure to VHDL codes. Every program starts with declaration of a named "entity' (with entity keyword) which contains all the Input-output connections from the fpga baord that the code is going to use. After the entity is declared, the circuit to be implemented on the board is mentioned under the keyword-"architecture". Once the code representing the circuit is compiled, the pin connections corresponding to each of the "port" mentioned in the "entity" are to be made using pin-planner or pin-assignment editor found in the "assignments" options on the top left of the Quartus window. If a given "port" is of input (or output) type the corresponding pin would bring in (or take out) the input (or output) from the I/O block of fpga assigned to it.

multiplier.vhd | Assignment Editor

<<new>> | Filter on node names: * | Category: All

	:atı	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
1	✓		prod[0]	Location	PIN_E21	Yes			
2	✓		prod[1]	Location	PIN_E22	Yes			
3	✓		prod[2]	Location	PIN_E25	Yes			
4	✓		prod[3]	Location	PIN_E24	Yes			
5	✓		num2[1]	Location	PIN_Y23	Yes			
6	✓		num2[0]	Location	PIN_Y24	Yes			
7	✓		num1[0]	Location	PIN_AB28	Yes			
8	✓		num1[1]	Location	PIN_AC28	Yes			
9	✓		led_1[0]	Location	PIN_G19	Yes			
10	✓		led_1[1]	Location	PIN_F19	Yes			
11	✓		led_2[1]	Location	PIN_H15	Yes			
12	✓		led_2[0]	Location	PIN_G16	Yes			
13	✓		clk	Location	PIN_Y2	Yes			
14	✓		num2_1[0]	Location	PIN_AA17	Yes			
15	✓		num2_1[1]	Location	PIN_AB16	Yes			
16	✓		num2_1[2]	Location	PIN_AA16	Yes			
17	✓		num2_1[3]	Location	PIN_AB17	Yes			
18	✓		num2_1[4]	Location	PIN_AB15	Yes			
19	✓		num2_1[5]	Location	PIN_AA15	Yes			
20	✓		num2_1[6]	Location	PIN_AC17	Yes			

Figure 2.1: Typical pin assignment as done on Quartus software.

3 Experiments with FPGA

The main goal of the lab exercises being to implement basic electronic circuits on fpga, I make a head start into it right away. Later in the report I will elaborate more on a basic applications of the device.

3.1 Getting started

A "hello world"-equivalent of a program code in VHDL would be to use LEDs on the fpga board. The programs used are given in the appendix xection in the order in which they appear here. A switch and a LED were connected via simple assignment operator " <= " in the code. The state (on\off) of the toggle switch decides the state of the LED on the board. Since it would be used in the coming lectures, the use of *for loop* was also learnt at this point (code in appendix).
Remarks: The for loop implimentation in VHDL can be difficult to handle. It was learned that any dynamic code which needs to be updated in the run-time must be mentioed in the architecture under a "process" (keyword). A process has a dependency list, as the name justifies, the process updates itself whenever the variables in the dependency list change their values.

3.2 Implimenting logic gates and building a multiplexer

First various logic operators (2.1) were used; the code for xor gate can be found in the appendix. Then, using the logic gates, a multiplexer was also implemented. A $2 \longrightarrow 1$ multiplexer circuit (3.1) uses the state of a toggle switch to decide between its two inputs. The corresponding circuit diagram which can be used to achieve this action has been shown in the figure (3.1).

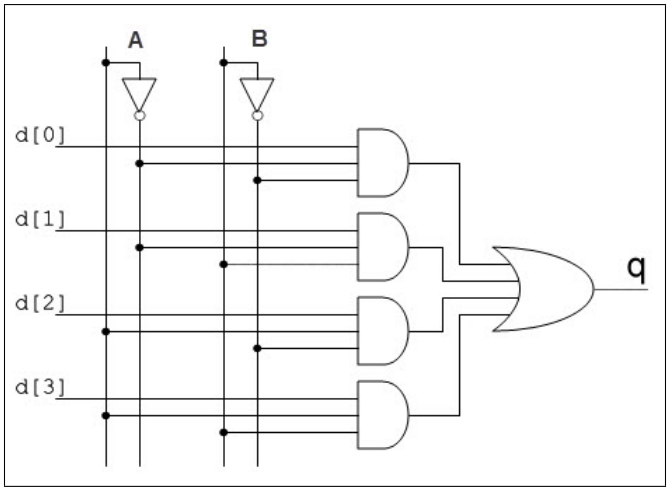
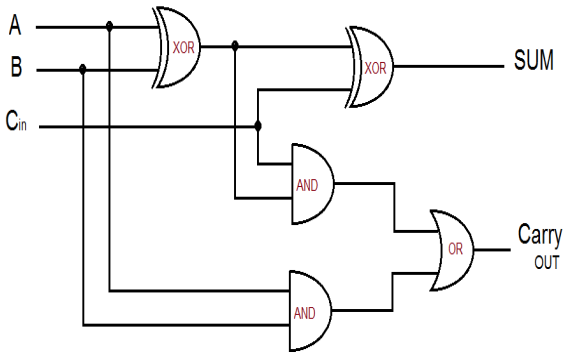


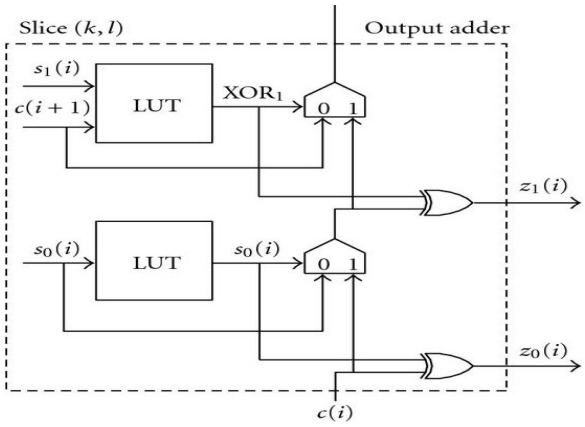
Figure 3.1: Logic gate diagram of a 4 to 1 multiplexer circuit. Image credits: Gate-overflow.com

3.3 Adder and multiplier

Typically the adder and multiplier circuits are implemented in electronics lab using opamps. The FPGA architecture allows us to directly implement the operation of adding\multiplying by breaking them into smaller boolean expressions which can be done with the help of the logic operators that the board comes equipped with.



(a) Adder circuit



(b) Adder, as implimented on fpga. Image courtesy: ResearchGate.

Figure 3.2: Caption for this figure with two images

The code for addition of two numbers was made upto arbitrary specified bits by declaring a "generic" variable 'n' which determines the size of the input numbers. A code for $n = 2$ is shown in the appendix. Two numbers are obtaind as binary inputs from toggle switches, and their summed output (binary) is shown by glowing the LEDs the correspond to their place holder. In the code, this is achieved by having rwo arrays of the std_logic-type, of length n; the operator $'+'$ sums up the binary numbers and stores the output in an array of length $n + 1$.

The code for multiplication of two numbers was similarly made arbitrary upto any specified bits (subject to limitation of the number of input-ports on the fpga board). A typical electronic circuit achieves multiplication by repeatedly shifting and adding gated inputs (3.3). Here the two imputs were taken in binary through designated toggle switches, these inputs were converted to their binary-coded-decimal (BCD) form and displayed on the 7-segment displays. The product of the two numbers (say, num1 and num2) is achieved (in binary form) by a for loop which ANDs two binary digits at a time, does it for the all the digits of num1 for a given digit of num2; and then shifts the resultant array of binary logic and adds the next one to be computed with other digit of num2.

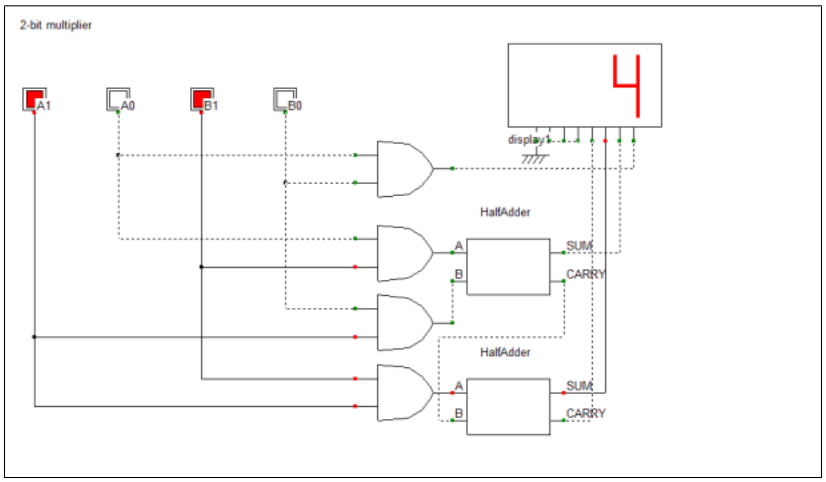
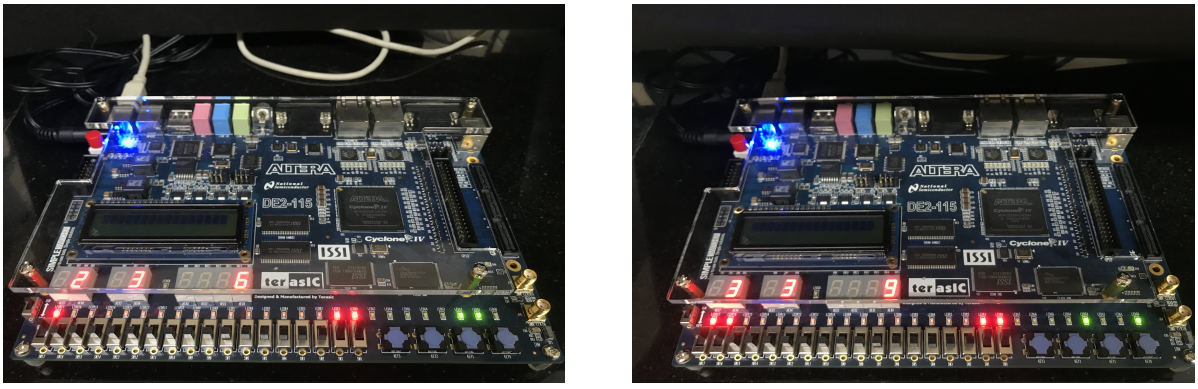


Figure 3.3: 2 bit multiplier. Image courtesy technobyte.com



(a) $2 * 3 = 6$ (b) $3 * 3 = 9$

Figure 3.4: FPGA as a multiplier

3.4 Counters

As we have learned, the fpga board consists of several logical units each of which also has flip-flops and registers. The counter circuit can be implemented by taking advantage of such clocked circuits which can detect falling edge or rising edge in a signal pulse. The Altera DE2-115 fpga board comes with two internal quartz crystals which oscillate at frequencies of 28MHz and 50MHz; these two provide an internal clock circuitry which can be easily used (making a suitable pin assignment) to time and clock circuits.

An external clock can also be used for the same, however its synchronisation with the internal clock has to be taken care of. If instead, an external signal pulse is taken as an input through 3.3V input pins (collectively called general purpose input-output or GPIO pins), the falling or rising edge can be detected for this too. Thereby extending the use of the board in counting any external signal, provided it is converted to a digital signal compatible with the fpga.

Two codes for counter circuits are given in the appendix. One uses an integer variable to decide when to stop counting and the other uses a time type variable for the same. This is done just to demonstrate how to use time type variables and also the convenience they offer in implementing clocked circuits on fpga. Time type variables come handy if the external clock being used is made synchronous with the internal one. Otherwise counting an integer variable can be used as a condition for stopping the code. See the codes in appendix for clarification. The codes are well commented and readable.
At first, having counted the internal pulse and displayed the count on the 7-segment display, I experimented with an external TTL pulse given to thr fpga board using GPIOs. The code for counting this remains the same and the display shows correct counts for any given TTL pulse frequency. **Remarks:** Please double check the output voltage from TTL pulse generator and make sure it is compatible with the board's input requirements.

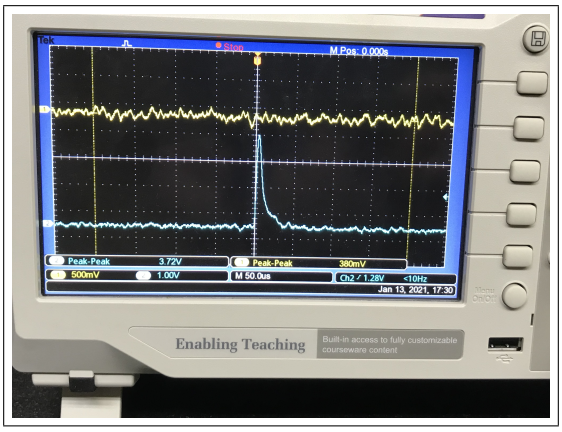
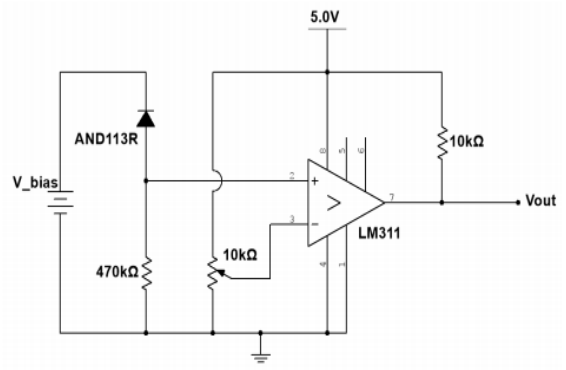
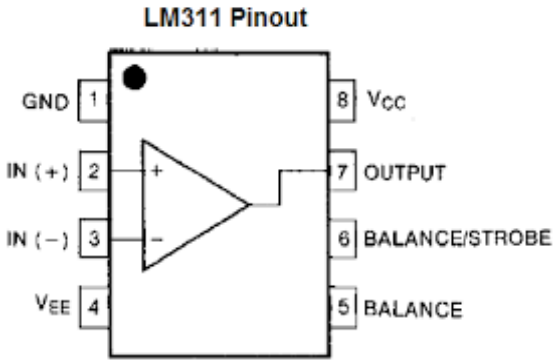


Figure 3.5: Pulse from a gamma spectrometer. Note that this is essentially incompatible with the FPGA input ports.

Next goal was to design an electronic circuit with can take in any type of pulse and generate a digital signal compatible with the fpga input requirements. LM 311 IC (figure: ??) was used to make a comparator circuit with a threshold of a few hundred mV and output of 3.3 V. The internal clock of fpga has a frequency of 50MHz which means there is a time gap of about 20ns between two pulses. The LM 311 IC has a response time between 150 and 200 ns which means many clock cyles elapse between by the time the output pulse's edge rises or falls. To take care of this, I reduced (see the code in appendix) the frequency at which the external signal is sampled by factoring the internal clock frequency according to the rising time of external signal. For example if a sine wave converted to TTL pulse using the comparator circuit has a rising time of $1\mu s$ then $\frac{1\mu s}{20ns} = 50$ number of clock cylces elapse during the rising time. Hence the internal clock is factored by 50.



(a) Image credit: "LEDs as Single-Photon Avalanche Photodiodes" by Jonathan Newport, American University



(b) Image credit: learning-about-electronics.com

3.5 Internal memory

FPGA board come equipped with internal memories. DE2-115 has one SRAM and a SDRAM which can be accessed via designated pins. The numbers input by user in binary using toggle switches, were written on the SRAM and read back and displayed on the 7-segment display. The code for the same can be found in the appendix. The purpose of this exercise was to prepare a base code so that while applying of fpga in real experiments, data acquisition can be done and at much faster rate than reading off from displays. The data on SRAM can be written at rates as fast as order of the internal clock frequency.

4 Application: Gamma-gamma correlation spectrometry

Having a counter circuit at hand and an electronic circuit to convert analog signals to digital, we deliberated on what possible applications we can use it for. Counter circuits are abound in nuclear physics experiments where various detectors produce voltage/current pulses on detection of an event. This pulse then goes through preamplifier, get converted into a large voltage pulse, and a shaping amplifier if needed, and then finaly to a counter circuit (or a multichannel analyser) via ADC (analog to digita) circuit. It was therefore thought to count pulses from a detector and hence count nuclear decay events.

Cspark Research is an indeginous company which manufactures detectors for use in physics teaching labs. One such product by them is gamma spectrometer. Two such spectrometers can be put face-to-face for measuring gamma-gamma coincidence from a source which emits 2 gamma photons. The idea was to put Na^{22} sounce in the middle of two gamma spectrometers facing each other (as shown in figure **??**).

This setup ensures that the two photons corresponding to an actual decay event in the source reach the two detectors simultaneously. In other words, only the pulses coincident in the two detectors should be counted as a decay even and others discarded as background radiation. This approach has too many approximations and care must be taken to avoid too much deviation from them. For example, it is not guaranteed that when two gamma photons strike the detectors in their paths, simultaneously, the pulse coming out of the pre-amplifier and shaping amplifier would remain simultaneous. In general the electronic circuits can produce different time delays in different pulses. Hence, the radiation source must have radioactivity lesser than the delay time of all the electonic component added together.

An FPGA code was written much alike the previous counter codes. This code has a ”process” for each of the two incoming pulses and a ”process” to count only when both the pulses are simultaneaous (see appendix). However, while performing the experiment in the lab several technical issues were encountered. I managed to solve several of them but new problems emerged and the 5 weeks assigned for this lab got over. If given

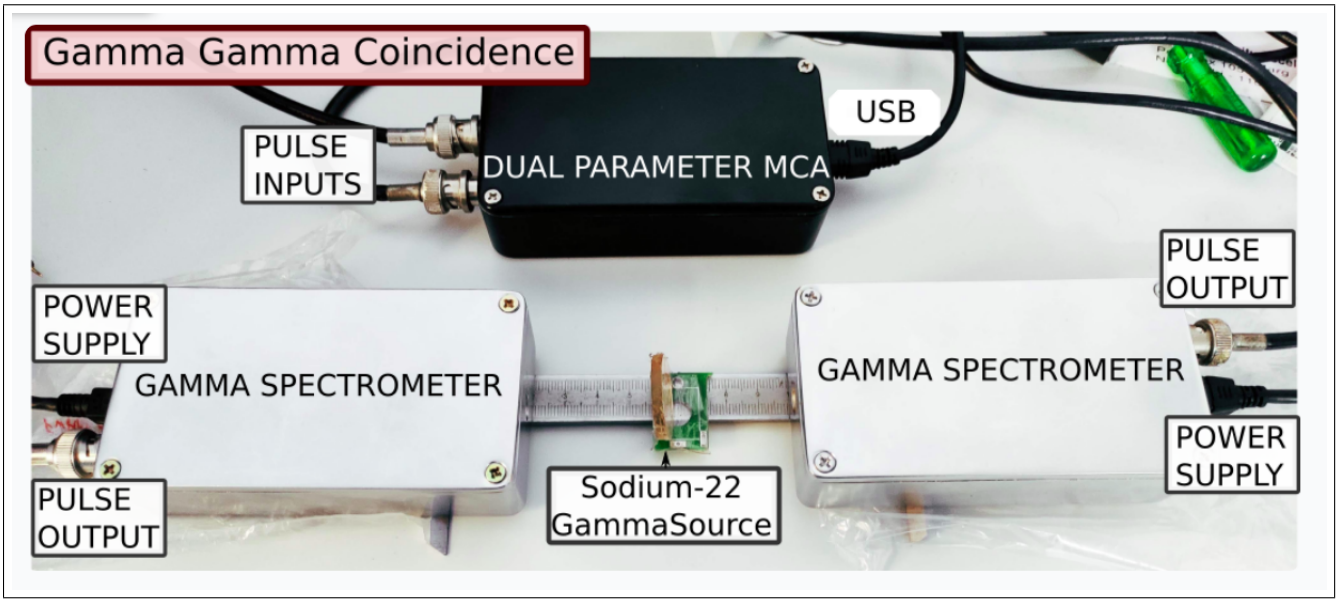


Figure 4.1: Setup for gamma-gamma coincidence count using cspark gamma detectors. Image credits: csparkresearch.com

a chance again, I would like to proceed with the same agenda and complete the application part. Next, I breifly discuss some of the issues (resolved and unresolved).

The pulse output from the shaping amplifier of the gamma spectrometers when have different hieghts. This means that the comparator circuit output would stay on or off for a variable amount of time depending on the input pulse from the spectrometer. In general, even if there is a coincident event from the two detectors, the comparator output for the same pulses might not coincide in terms of coinciding falling edge or rising edge of the signal. This is a serious issue because the actual events might never be counted unless the two pulses have same hieght. This issue was only partially resolved by increasing the threshold which disregards low energy counts. Certainly this is not the desired solution. I shall have to make a new circuit which produces digital pulses of nearly equal width, irrespective of the input analog pulse. The pulse output from the comparator is shown in figure 4.3

Caution: The FPGA board has a feedback in the circuit that can cause jitters in the detected pulse when given as input to the board (say, via GPIO). To nullify this, keep the board connected to the PC\ laptop.

Moreover, other than the challenge mentioned above, I also faced some difficulty when the rising time of the TTL pulse from the comparators, changed abrubptly during the experiment. As the reader would remenber, while counting the external pulses I need to modify the internal clock cycle to make it in sync with the external signal's rising (or falling) time. A pulse with widely fluctuating rising times is therefore uncountable with the present algorithmic setup. I need to improvise on this aspect too.

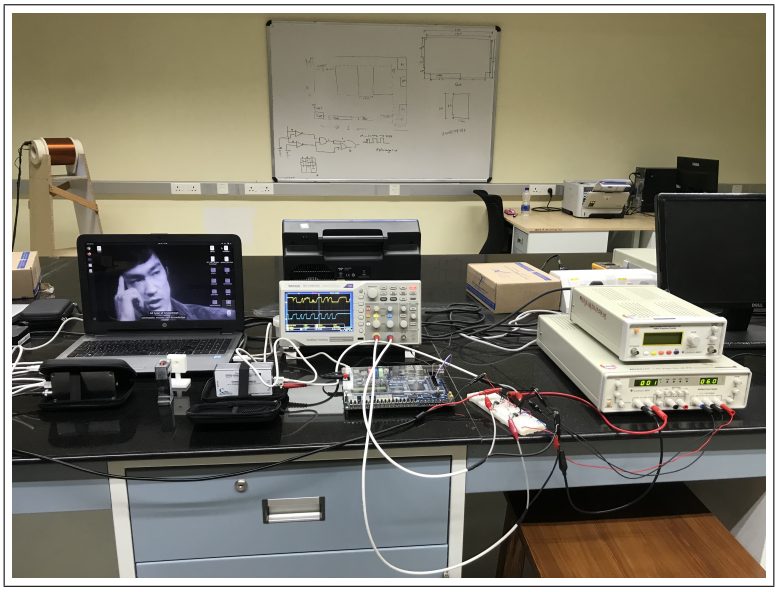


Figure 4.2: Setup made in the lab for gamma-gamma coincidence. The power generator is used to power the NM311-IC. The detectors are plugged to the laptop for power. The outputs from gamma spectrometers are fed to the comparator circuits and the output from the comparators are the one shown in the oscilloscope. A close up of the oscilloscope output in the absence of a gamma source in front of the detectors, is shown in the next image.

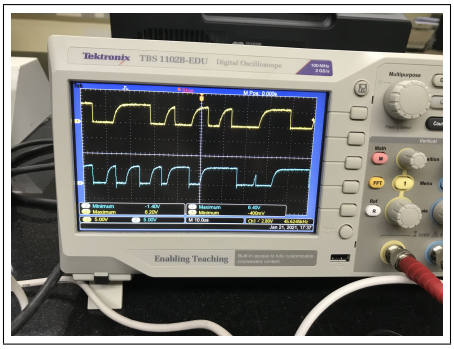


Figure 4.3: A close up of oscilloscope which shows the output of comparator circuits.

5 Conclusion

FPGA being a flexibly configurable device can be applied to a variety of places. The only limitation is the boards input-output hardware, which are plenty enough to allow diverse applications. I started learning the VHDL and did hands-on basic experiments on the board. What was achieved in the given 5 weeks is significant in terms of getting started with the device. I would definitely like to work more with the board and see if I can apply it to automate some of the physics experiments conduxted in the department's labs.

Appendix: Codes

The various codes in the appendix are listed below. They can be found in the pages succeeding the bibliography in the following order.

1. XOR gate and for-loop implimentation
2. Multiplexer
3. Adder circuit
4. Multipier
5. Internal clock counter and external pulse counter (the difference is just in their pin assignments)
6. Using FPGA memory block: SRAM
7. External correlation counter
8. External correlation counter using time type variable

Bibliography

[1] https://www.intel.in/content/www/in/en/products/programmable/fpga.html

[2] https://csparkresearch.in/gammacoincidence

[3] "LEDs as Single-Photon Avalanche Photodiodes" by Jonathan Newport, American University

[4] "LM111, LM211, LM311 Differential Comparators" by Texas Instruments

[5] "FPGA Based Frequency Measurement for The Purpose of Synchronization" by Communications Laboratory – University of Kassel

[6] "Energy-resolved coincidence counting using an FPGA for nuclear lifetime experiments" Mario Vretenar, Nataša Erceg, and Marin Karuza; American Journal of Physics 87, 997 (2019); doi:10.1119/1.5122744

[7] Altera DE2-115 User-Manual

[8] FPGA for Dummies; 2nd Intel Edition.

```
1   LIBRARY  ieee;
2   USE ieee.std_logic_1164.all;
3   ----------------------------------------------------------------------------------
    ---------
4   -- vhdl code to implement XOR Gate on inputs given by two switches.
5   ----------------------------------------------------------------------------------
    ---------
6   ENTITY xor_gate IS
7      PORT
8      (
9         x1, x2 : IN STD_LOGIC;  -- two inputs
10        f : OUT STD_LOGIC       -- the XOR-gated output
11     );
12  END xor_gate;
13  ----------------------------------------------------------------------------------
    ---------
14  ARCHITECTURE logicFunction OF xor_gate IS
15  BEGIN
16     f <= (x1 AND NOT x2) OR (NOT x1 AND x2); -- A XOR B = A.B` + A`.B
17  END logicFunction;
18  ----------------------------------------------------------------------------------
    ---------
19
```

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
----------------------------------------------------------------------------------
------
-- a program to test the working of a 'for loop' in vhdl.
----------------------------------------------------------------------------------
------
entity test is
   port
      (
         x: in std_logic_vector(0 to 3);     -- input
         y: out std_logic_vector(0 to 3)     -- output
      );
end test;
----------------------------------------------------------------------------------
------
architecture logic of test is
begin

process(x)
variable a: std_logic_vector(0 to 3) := (3 => '1', others => '0');
begin
   for i in 0 to 3 loop
      y(i) <= x(i) AND a(i);
   end loop;
end process;

end logic;
```

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3   USE ieee.std_logic_arith.all;
4   ------------------------------------------------------------------------------
    ----------------------
5   -- A multiplexer code chooses between it's input variables based on the state of
    a toggle switch
6
7   ENTITY multiplexer2_1 IS
8      GENERIC
9      (
10        constant n: integer := 3
11     );
12     PORT
13     (
14        s: IN STD_LOGIC;                        -- s is the toggle switch
15        x,y : IN STD_LOGIC_VECTOR(n downto 0); -- x and y are the two input
    variables to the multiplexer 's' which chooses between them
16        r: OUT STD_LOGIC_VECTOR(n  downto 0);  -- r is the resultant
17        s_led: OUT STD_LOGIC;                   -- s_led is the LED that indicates
    the state of s
18        x_led, y_led: OUT STD_LOGIC_VECTOR(n downto 0)
19        -- x_led (or y_led) glows to represent state of x (or y) variables
    depending on state of 's'
20     );
21  END multiplexer2_1;
22  ------------------------------------------------------------------------------
    ----------------------
23
24  ARCHITECTURE logicFunction OF multiplexer2_1 IS
25     BEGIN                                        -- the logic circuit of a
    multiplxer is implemented here.
26        r(0) <= ((x(0) AND s) OR (y(0) AND NOT s));
27        r(1) <= ((x(1) AND s) OR (y(1) AND NOT s));
28        r(2) <= ((x(2) AND s) OR (y(2) AND NOT s));
29        r(3) <= ((x(3) AND s) OR (y(3) AND NOT s));
30        -- the corresponding LEDs are assigned values so that the user can infer
    the circuit output
31        s_led <= s;
32        x_led <= x;
33        y_led <= y;
34  END logicFunction;
35
```

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3   USE ieee.std_logic_arith.all;
4   USE ieee.std_logic_unsigned.all;
5
6   ------------------------------------------------------------
7   ENTITY adder IS
8      GENERIC
9         (
10           n: integer := 2   -- this is a two-bit adder. Just change the 'n' value
    to make it work for additional bits
11        );
12     PORT
13        (
14           x  :   IN STD_LOGIC_VECTOR(n-1 downto 0);
15           y  :   IN STD_LOGIC_VECTOR(n-1 downto 0);
16           carry :  OUT STD_LOGIC;
17           sum   :  OUt STD_LOGIC_VECTOR(n-1 downto 0);
18           input1   :  BUFFER STD_LOGIC_VECTOR(n-1 downto 0);
19           input2   :  BUFFER STD_LOGIC_VECTOR(n-1 downto 0)
20
21        );
22  END adder;
23
24  ------------------------------------------------------------
25
26  ARCHITECTURE logicFunction OF adder IS
27  SIGNAL result  :  STD_LOGIC_VECTOR(n downto 0);
28
29  BEGIN
30     input1 <= x;
31     input2 <= y;
32     result <= ('0' & x) + ('0' & y); --append 0 at the beginning to make the
    addition of vectors compatible.
33     sum <= result(n-1 downto 0);
34     carry <= result(n);              --the last bit goes as carry.
35
36  END logicFunction;
37
```

```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3   USE ieee.std_logic_arith.all;
4   USE ieee.std_logic_unsigned.all;
5   ---------------------------------------------------------------------------
6   -- multiplier code takes two numbers as inputs (in binary) and gives their
    product as an output (both binary and decimal)
7   -- to learn: the use of seven segment displays and to output a given number in
    BCD format (binary coded decimal)
8   ---------------------------------------------------------------------------
9
10  ENTITY multiplier IS
11     GENERIC
12     (
13        constant n: integer := 2   --this is a two-bit multiplier. Can easily be
    extended by chaning 'n' value.
14     );
15     PORT
16     (
17        num1, num2: IN STD_LOGIC_VECTOR(n-1 downto 0);  --input numbers
18        prod: INOUT STD_LOGIC_VECTOR(2*n-1 downto 0);   --product
19        num1_1,  num2_1, prod_1 : out std_LOGIC_VECTOR(6 downto 0); --pin
    connections to show the numbers on 7-segment display
20        led_1, led_2: OUT STD_LOGIC_VECTOR(n-1 downto 0); -- pin connections to
    show the numbers (input and output) on LEDs
21        clk : IN std_logic         -- internal clock input
22     );
23  END multiplier;
24
25  ---------------------------------------------------------------------------
26  ---------------------------------------------------------------------------
27
28  ARCHITECTURE logicFunction OF multiplier IS
29
30  shared variable temp_prod: STD_LOGIC_VECTOR(2*n downto 0);
31  signal hex_coding_11, hex_coding_21, hex_coding_p1: STD_LOGIC_VECTOR(7 downto 0)
    := (others => '1');
32
33  BEGIN
34  led_1 <= num1;
35  led_2 <= num2;
36
37
38     process(num1, num2)
39     variable num1_alt: STD_LOGIC_VECTOR(n downto 0) := '0' & num1;
40     variable num2_alt: STD_LOGIC_VECTOR(n downto 0) := '0' & num2;
41
42     begin
43     temp_prod := "00000";
44     for i in 0 to n-1 loop
45        if (num2_alt(i)='1') then
46           temp_prod(2*n downto n) := temp_prod(2*n downto n) + num1_alt(n downto 0
    );
47        end if;
48        temp_prod(2*n downto 0) := '0' & temp_prod(2*n downto 1);
49     end loop;
50     end process;
51
```

```
52    prod <= temp_prod(2*n-1 downto 0);
53    ----------------------------------------------------------------------------
54    -- the following processes (on num1, num2, and prod) operate at each clock edge
      to assign the 7-segment displays to show the correct number.
55       process(num1)
56       variable num1_alt: STD_LOGIC_VECTOR(n+1 downto 0) := "00" & num1;
57       begin
58       if rising_edge(clk) then
59       case num1_alt is
60          when "0000" =>
61             hex_coding_11 <= "10000001";
62          when "0001" =>
63             hex_coding_11 <= "11001111";
64          when "0010" =>
65             hex_coding_11 <= "10010010";
66          when "0011" =>
67             hex_coding_11 <= "10000110";
68          when "0100" =>
69             hex_coding_11 <= "11001100";
70          when "0101" =>
71             hex_coding_11 <= "10100100";
72          when "0110" =>
73             hex_coding_11 <= "10100000";
74          when "0111" =>
75             hex_coding_11 <= "10001111";
76          when "1000" =>
77             hex_coding_11 <= "10000000";
78          when "1001" =>
79             hex_coding_11 <= "10000100";
80          when "1010" =>
81             hex_coding_11 <= "10000010";
82          when "1011" =>
83             hex_coding_11 <= "11100000";
84          when "1100" =>
85             hex_coding_11 <= "10110001";
86          when "1101" =>
87             hex_coding_11 <= "11000010";
88          when "1110" =>
89             hex_coding_11 <= "10110000";
90          when "1111" =>
91             hex_coding_11 <= "10111000";
92       end case;
93       end if;
94       end process;
95          num1_1(0) <= hex_coding_11(6);
96          num1_1(1) <= hex_coding_11(5);
97          num1_1(2) <= hex_coding_11(4);
98          num1_1(3) <= hex_coding_11(3);
99          num1_1(4) <= hex_coding_11(2);
100         num1_1(5) <= hex_coding_11(1);
101         num1_1(6) <= hex_coding_11(0);
102
103      process(num2)
104      variable num2_alt: STD_LOGIC_VECTOR(n+1 downto 0) := "00" & num2;
105      begin
106      if rising_edge(clk) then
107      case num2_alt is
108         when "0000" =>
```

```
109            hex_coding_21 <= "10000001";
110         when "0001" =>
111            hex_coding_21 <= "11001111";
112         when "0010" =>
113            hex_coding_21 <= "10010010";
114         when "0011" =>
115            hex_coding_21 <= "10000110";
116         when "0100" =>
117            hex_coding_21 <= "11001100";
118         when "0101" =>
119            hex_coding_21 <= "10100100";
120         when "0110" =>
121            hex_coding_21 <= "10100000";
122         when "0111" =>
123            hex_coding_21 <= "10001111";
124         when "1000" =>
125            hex_coding_21 <= "10000000";
126         when "1001" =>
127            hex_coding_21 <= "10000100";
128         when "1010" =>
129            hex_coding_21 <= "10000010";
130         when "1011" =>
131            hex_coding_21 <= "11100000";
132         when "1100" =>
133            hex_coding_21 <= "10110001";
134         when "1101" =>
135            hex_coding_21 <= "11000010";
136         when "1110" =>
137            hex_coding_21 <= "10110000";
138         when "1111" =>
139            hex_coding_21 <= "10111000";
140      end case;
141      end if;
142      end process;
143         num2_1(0) <= hex_coding_21(6);
144         num2_1(1) <= hex_coding_21(5);
145         num2_1(2) <= hex_coding_21(4);
146         num2_1(3) <= hex_coding_21(3);
147         num2_1(4) <= hex_coding_21(2);
148         num2_1(5) <= hex_coding_21(1);
149         num2_1(6) <= hex_coding_21(0);
150
151      process(prod)
152      variable prod_alt: STD_LOGIC_VECTOR(2*n-1 downto 0) := prod;
153      begin
154      if rising_edge(clk) then
155      case prod_alt is
156         when "0000" =>
157            hex_coding_p1 <= "10000001";
158         when "0001" =>
159            hex_coding_p1 <= "11001111";
160         when "0010" =>
161            hex_coding_p1 <= "10010010";
162         when "0011" =>
163            hex_coding_p1 <= "10000110";
164         when "0100" =>
165            hex_coding_p1 <= "11001100";
166         when "0101" =>
```

```
167             hex_coding_p1 <= "10100100";
168          when "0110" =>
169             hex_coding_p1 <= "10100000";
170          when "0111" =>
171             hex_coding_p1 <= "10001111";
172          when "1000" =>
173             hex_coding_p1 <= "10000000";
174          when "1001" =>
175             hex_coding_p1 <= "10000100";
176          when "1010" =>
177             hex_coding_p1 <= "10000010";
178          when "1011" =>
179             hex_coding_p1 <= "11100000";
180          when "1100" =>
181             hex_coding_p1 <= "10110001";
182          when "1101" =>
183             hex_coding_p1 <= "11000010";
184          when "1110" =>
185             hex_coding_p1 <= "10110000";
186          when "1111" =>
187             hex_coding_p1 <= "10111000";
188       end case;
189       end if;
190       end process;
191          prod_1(0) <= hex_coding_p1(6);
192          prod_1(1) <= hex_coding_p1(5);
193          prod_1(2) <= hex_coding_p1(4);
194          prod_1(3) <= hex_coding_p1(3);
195          prod_1(4) <= hex_coding_p1(2);
196          prod_1(5) <= hex_coding_p1(1);
197          prod_1(6) <= hex_coding_p1(0);
198
199    END logicFunction;
200
201
```

```
 1
 2  library ieee ;
 3  use ieee.std_logic_1164.all;
 4  use ieee.std_logic_unsigned.all;
 5
 6  ----------------------------------------------------
 7
 8  entity counter is
 9     generic(n: natural :=13);
10     port
11     (
12        clock:   in std_logic;     -- internal clock signal 50 MHz
13        clear:   in std_logic;     -- toggle switch to reset
14        count:   in std_logic;     -- toggle switch function: enable count
15        Q: out std_logic_vector(n-1 downto 0);
16        dig1: out std_logic_vector(6 downto 0);
17        dig2: out std_logic_vector(6 downto 0);
18        dig3: out std_logic_vector(6 downto 0);
19        dig4: out std_logic_vector(6 downto 0)    -- dig 1 to 4 are to be used for
   display on 4 7-segment displays.
20     );
21  end counter;
22
23  ----------------------------------------------------
24
25  architecture behv of counter is
26
27     signal Pre_Q: std_logic_vector(n-1 downto 0);
28     shared variable num: integer := 0;
29     signal dig1_code, dig2_code, dig3_code, dig4_code: STD_LOGIC_VECTOR(7 downto 0
   ) := (others => '1');
30
31  begin
32
33      -- behavior describe the counter
34     process(clock, count, clear)
35     variable cnter: integer range 0 to 2500000:= 0;
36     begin
37     if clear = '1' then
38         Pre_Q <= Pre_Q - Pre_Q;         -- reset the counter if clear is on
39     elsif count = '1' then              -- stop counting of count is off
40        if (clock='1' and clock'event) then
41        if(cnter < 2500000) then
42           cnter := cnter+1;
43        elsif (cnter = 2500000) then
44           cnter := 0;
45           Pre_Q <= Pre_Q + '1';         -- count only every 2500000th of the
   internal clock pulse.
46  -- NOTE: this way, the internal clock can be used to produce any other clock of
   frequency lesser than 50 MHz
47        end if;
48        end if;
49     end if;
50     end process;
51     -- concurrent assignment statement
52      Q <= Pre_Q;
53  ----------------------------------------------------------
54     process(clock)
```

```
55      begin
56      num:=0;
57      for i in 0 to n-1 loop
58         if Pre_Q(i) = '1' then
59            num := num + (2**i);
60         end if;
61      end loop;
62      end process; --you could have used the built-in function conv_integer() to
    convert std_logic_vector to integer.
63
64  --the below processes break 'num' into its digits and then display them on
    7-segment displays.
65      process(clock)
66      variable num1: integer range 0 to 9;
67      begin
68      num1:= num mod 10;
69      if rising_edge(clock) then
70      case num1 is
71         when 0 =>
72            dig1_code <= "10000001";
73         when 1 =>
74            dig1_code <= "11001111";
75         when 2 =>
76            dig1_code <= "10010010";
77         when 3 =>
78            dig1_code <= "10000110";
79         when 4 =>
80            dig1_code <= "11001100";
81         when 5 =>
82            dig1_code <= "10100100";
83         when 6 =>
84            dig1_code <= "10100000";
85         when 7 =>
86            dig1_code <= "10001111";
87         when 8 =>
88            dig1_code <= "10000000";
89         when 9 =>
90            dig1_code <= "10000100";
91         when others =>
92            dig1_code <= "10000000";
93      end case;
94      end if;
95      end process;
96      dig1(0) <= dig1_code(6);
97      dig1(1) <= dig1_code(5);
98      dig1(2) <= dig1_code(4);
99      dig1(3) <= dig1_code(3);
100     dig1(4) <= dig1_code(2);
101     dig1(5) <= dig1_code(1);
102     dig1(6) <= dig1_code(0);
103
104     process(clock)
105     variable num_alt1, num2: integer;
106     begin
107     num_alt1 := num/10;
108     num2 := num_alt1 mod 10;
109     if rising_edge(clock) then
110     case num2 is
```

```
111          when 0 =>
112             dig2_code <= "10000001";
113          when 1 =>
114             dig2_code <= "11001111";
115          when 2 =>
116             dig2_code <= "10010010";
117          when 3 =>
118             dig2_code <= "10000110";
119          when 4 =>
120             dig2_code <= "11001100";
121          when 5 =>
122             dig2_code <= "10100100";
123          when 6 =>
124             dig2_code <= "10100000";
125          when 7 =>
126             dig2_code <= "10001111";
127          when 8 =>
128             dig2_code <= "10000000";
129          when 9 =>
130             dig2_code <= "10000100";
131          when others =>
132             dig2_code <= "10000000";
133       end case;
134       end if;
135       end process;
136       dig2(0) <= dig2_code(6);
137       dig2(1) <= dig2_code(5);
138       dig2(2) <= dig2_code(4);
139       dig2(3) <= dig2_code(3);
140       dig2(4) <= dig2_code(2);
141       dig2(5) <= dig2_code(1);
142       dig2(6) <= dig2_code(0);
143
144       process(clock)
145       variable num_alt2, num3: integer;
146       begin
147       num_alt2 := num/100;
148       num3 := num_alt2 mod 10;
149       if rising_edge(clock) then
150       case num3 is
151          when 0 =>
152             dig3_code <= "10000001";
153          when 1 =>
154             dig3_code <= "11001111";
155          when 2 =>
156             dig3_code <= "10010010";
157          when 3 =>
158             dig3_code <= "10000110";
159          when 4 =>
160             dig3_code <= "11001100";
161          when 5 =>
162             dig3_code <= "10100100";
163          when 6 =>
164             dig3_code <= "10100000";
165          when 7 =>
166             dig3_code <= "10001111";
167          when 8 =>
168             dig3_code <= "10000000";
```

```
169         when 9 =>
170            dig3_code <= "10000100";
171         when others =>
172            dig3_code <= "10000000";
173      end case;
174      end if;
175      end process;
176      dig3(0) <= dig3_code(6);
177      dig3(1) <= dig3_code(5);
178      dig3(2) <= dig3_code(4);
179      dig3(3) <= dig3_code(3);
180      dig3(4) <= dig3_code(2);
181      dig3(5) <= dig3_code(1);
182      dig3(6) <= dig3_code(0);
183
184      process(clock)
185      variable num_alt3, num4: integer;
186      begin
187      num_alt3 := num/1000;
188      num4 := num_alt3 mod 10;
189      if rising_edge(clock) then
190      case num4 is
191         when 0 =>
192            dig4_code <= "10000001";
193         when 1 =>
194            dig4_code <= "11001111";
195         when 2 =>
196            dig4_code <= "10010010";
197         when 3 =>
198            dig4_code <= "10000110";
199         when 4 =>
200            dig4_code <= "11001100";
201         when 5 =>
202            dig4_code <= "10100100";
203         when 6 =>
204            dig4_code <= "10100000";
205         when 7 =>
206            dig4_code <= "10001111";
207         when 8 =>
208            dig4_code <= "10000000";
209         when 9 =>
210            dig4_code <= "10000100";
211         when others =>
212            dig4_code <= "10000000";
213      end case;
214      end if;
215      end process;
216      dig4(0) <= dig4_code(6);
217      dig4(1) <= dig4_code(5);
218      dig4(2) <= dig4_code(4);
219      dig4(3) <= dig4_code(3);
220      dig4(4) <= dig4_code(2);
221      dig4(5) <= dig4_code(1);
222      dig4(6) <= dig4_code(0);
223
224   end behv;
225
```

```
1   library ieee ;
2   library work;
3   use ieee.std_logic_1164.all;
4   use ieee.std_logic_unsigned.all;
5   use work.all;
6   ------------------------------------------------------------------------------
    -----
7   -- writing and reading data on sram
8   ------------------------------------------------------------------------------
    -----
9
10  ENTITY sram IS
11     GENERIC
12     (
13        breadth: integer:=4;
14        depth:   integer:=4;
15        addr: integer:=2
16     );
17
18     PORT
19     (
20        enable: in std_logic;
21        read_switch: in std_logic;
22        write_switch: in std_logic;
23
24        clock: in std_logic;
25        oe_n : in std_logic;
26        we_n: in std_logic;
27        ce_n: in std_logic;
28        ub: in std_logic;
29        lb: in std_logic;       -- these are the various signals the SRAM needs
    for various operations.
30
31        Read_Addr:  in std_logic_vector(addr-1 downto 0);
32        Write_Addr:    in std_logic_vector(addr-1 downto 0);
33        data_in : inout std_logic_vector(15 downto 0);
34        data_out : out std_logic_vector(15 downto 0)
35        user_read: out std_logic_vector(7 downto 0); -- for 7 segment display
36     );
37  END sram;
38  ------------------------------------------------------------------------------
    --------
39  ------------------------------------------------------------------------------
    --------
40  ARCHITECTURE working OF sram IS
41
42  signal bcd: std_logic_vector(7 downto 0) := (others => '1');
43  signal num: integer := 0;
44  type ram_type is array (0 to depth-1) of
45     std_logic_vector(breadth-1 downto 0);
46  signal tmp_ram: ram_type;
47
48  BEGIN
49
50     -- Read Function
51     process(Clock, read_switch)
52     begin
53     if (Clock'event and Clock='1') then
```

```
54          if Enable='1' then
55             if read_switch='1' then
56                data_out <= tmp_ram(conv_integer(Read_Addr));
57             else
58                data_out <= (data_out'range => 'N');
59             end if;
60          end if;
61       end if;
62       end process;
63
64       -- Write Function
65       process(Clock, write_switch)
66       begin
67       if (Clock'event and Clock='1') then
68          if Enable='1' then
69             if write_switch='1' then
70                tmp_ram(conv_integer(Write_Addr)) <= data_in;
71             end if;
72          end if;
73       end if;
74       end process;
75
76    num <= data_out;
77    ----------------------------------------------------------------------------------
78    -- below function displays the data on 7-segment display.
79    ----------------------------------------------------------------------------------

80       process(clock)
81       begin
82       if rising_edge(clock) then
83       case num is
84          when 0 =>
85             bcd <= "10000001";
86          when 1 =>
87             bcd <= "11001111";
88          when 2 =>
89             bcd <= "10010010";
90          when 3 =>
91             bcd <= "10000110";
92          when 4 =>
93             bcd <= "11001100";
94          when 5 =>
95             bcd <= "10100100";
96          when 6 =>
97             bcd <= "10100000";
98          when 7 =>
99             bcd <= "10001111";
100         when 8 =>
101            bcd <= "10000000";
102         when 9 =>
103            bcd <= "10000100";
104         when 10 =>
105            bcd <= "10001000";
106         when 11 =>
107            bcd <= "11100000";
108         when 12 =>
109            bcd <= "10110001";
110         when 13 =>
```

```
111             bcd <= "11000010";
112          when 14 =>
113             bcd <= "10110000";
114          when 15 =>
115             bcd <= "10111000";
116          when others =>
117             bcd <= "11111111";
118       end case;
119       end if;
120       end process;
121       user_read(0) <= bcd(6);
122       user_read(1) <= bcd(5);
123       user_read(2) <= bcd(4);
124       user_read(3) <= bcd(3);
125       user_read(4) <= bcd(2);
126       user_read(5) <= bcd(1);
127       user_read(6) <= bcd(0);
128    ------------------------------------------------------------------------------
       -
129
130    END working;
```

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4   ------------------------------------------------------------------------------
    ------------------------------------------------------------------------------
5   ENTITY count_correlate IS
6   GENERIC
7   (
8      n: natural :=13
9   );
10  PORT
11  (
12     clock: IN std_logic;
13     enable: IN std_logic;
14     reset: IN std_logic;
15     pulse: IN std_logic;
16     grnd: IN std_logic;
17     Q: OUT std_logic_vector(n-1 downto 0);
18  );
19  END ENTITY;
20
21  ------------------------------------------------------------------------------
    ------------------------------------------------------------------------------
22  ARCHITECTURE logicFunction OF count_correlate IS
23     constant stop_time: time := 10*20*50000000 ns; -- in sec;
24     constant sampling_time: time := 20*20 ns; --that is, sample the pulse once
    every "sampling_time" number of internal clock cycles
25     shared variable real_time: time := 0 ns;
26     shared variable keep_counting: std_logic := '0';
27     signal Pre_Q: std_logic_vector(n-1 downto 0);
28
29  BEGIN
30
31     process(clock)
32     variable sub_timer: time := 0 ns;
33     begin
34     if rising_edge(clock) then
35        if(sub_timer < sampling_time) then
36           sub_timer := sub_timer +20 ns;
37           keep_counting := '0';
38           real_time := real_time + 20 ns;
39        elsif(sub_timer = sampling_time) then  -- sample the pulse only at the
    frequency determined by the sampling rate.
40           if(real_time < stop_time) then      -- count only if timer is lesser
    than the time for which the counts are to be taken
41              sub_timer := 0 ns;
42              real_time := real_time + 20 ns;
43              keep_counting := '1';
44           else
45              keep_counting := '0';
46           end if;
47        end if;
48     end if;
49     end process;
50
51     process(pulse, reset, enable)
52     begin
53     if (reset = '1') then
```

```
54          pre_Q <= pre_Q - pre_Q;
55       elsif(enable = '1' and keep_counting = '1') then
56          if falling_edge(pulse) then
57             pre_Q <= pre_Q + '1';
58          end if;
59       end if;
60       end process;
61
62    Q <= pre_Q;           -- assign the count value to a port for display purposes.
63
64    END logicFunction;
65
```

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4   ----------------------------------------------------------------------------
5   --Program to count coincidence of two external pulses.
6   ----------------------------------------------------------------------------
7   ENTITY count_correlate2 IS
8   GENERIC
9   (
10     n: natural :=13
11  );
12  PORT
13  (
14     clock: IN std_logic;
15     enable: IN std_logic;
16     reset: IN std_logic;
17     pulse1: IN std_logic;            --input pulse from signal 1
18     pulse2: IN std_logic;            --input pulse from signal 2
19     grnd: IN std_logic;
20
21     Q: out std_logic_vector(n-1 downto 0);
22     dig1: out std_logic_vector(6 downto 0);
23     dig2: out std_logic_vector(6 downto 0);
24     dig3: out std_logic_vector(6 downto 0);
25     dig4: out std_logic_vector(6 downto 0)
26  );
27  END ENTITY;
28
29  ----------------------------------------------------------------------------
30  ARCHITECTURE logicFunction OF count_correlate2 IS
31     constant stop_time: integer := 100; -- in sec;
32     constant sampling_time: integer := 100; --that is, sample the pulse once
    every "sampling_time" number of internal clock cycles
33     shared variable keep_counting: std_logic := '0';
34     shared variable flag0, flag1, flag2: std_logic := '0';
35     shared variable counter: integer:= (50000000 * stop_time)/sampling_time;
36     signal Pre_Q: std_logic_vector(n-1 downto 0);
37     shared variable num: integer := 0;
38     signal dig1_code, dig2_code, dig3_code, dig4_code: STD_LOGIC_VECTOR(7 downto 0
    ) := (others => '1');
39
40  BEGIN
41
42     process(clock)
43     variable sub_counter: integer := 0;
44     begin
45     if rising_edge(clock) then
46        if(sub_counter < sampling_time) then
47           sub_counter := sub_counter +1;
48           keep_counting := '0';
49        elsif (sub_counter = sampling_time) then
50           if (counter > 0) then
51              sub_counter := 0;
52              counter := counter -1;
53              keep_counting := '1';
54           else
55              keep_counting := '0';
56           end if;
```

```
57          end if;
58       end if;
59       end process;
60
61       process(reset, enable)
62       begin
63       if (reset = '1') then
64          flag0 := '0';
65       elsif(enable = '1' and keep_counting = '1') then
66          flag0 := '1';
67       end if;
68       end process;
69       process(pulse1)
70       begin
71       if falling_edge(pulse1) then
72          flag1 := '1';
73       end if;
74       end process;
75       process(pulse2)
76       begin
77       if falling_edge(pulse2) then
78          flag2 := '1';
79       end if;
80       end process;
81       process(clock)
82       begin
83       if (clock='1' and clock'event) then
84          if(flag0='1' and flag1='1' and flag2='1') then
85             pre_Q <= pre_Q + '1';
86          elsif(flag0 = '0') then
87             pre_Q <= pre_Q - pre_Q;
88          end if;
89       end if;
90       end process;
91
92
93       process(clock)
94       begin
95       num:=0;
96       for i in 0 to n-1 loop
97          if Pre_Q(i) = '1' then
98             num := num + (2**i);
99          end if;
100      end loop;
101      end process; --you could have used the built-in function conv_integer() to
     convert std_logic_vector to integer.
102
103      ------------------------------------------------------------------------------
     -----------------------------
104      --the below processes break 'num' into its digits and then display them on
105
106      7-segment displays.
107         process(clock)
108         variable num1: integer range 0 to 9;
109         begin
110         num1:= num mod 10;
111         if rising_edge(clock) then
112         case num1 is
```

```
113          when 0 =>
114             dig1_code <= "10000001";
115          when 1 =>
116             dig1_code <= "11001111";
117          when 2 =>
118             dig1_code <= "10010010";
119          when 3 =>
120             dig1_code <= "10000110";
121          when 4 =>
122             dig1_code <= "11001100";
123          when 5 =>
124             dig1_code <= "10100100";
125          when 6 =>
126             dig1_code <= "10100000";
127          when 7 =>
128             dig1_code <= "10001111";
129          when 8 =>
130             dig1_code <= "10000000";
131          when 9 =>
132             dig1_code <= "10000100";
133          when others =>
134             dig1_code <= "10000000";
135       end case;
136       end if;
137       end process;
138       dig1(0) <= dig1_code(6);
139       dig1(1) <= dig1_code(5);
140       dig1(2) <= dig1_code(4);
141       dig1(3) <= dig1_code(3);
142       dig1(4) <= dig1_code(2);
143       dig1(5) <= dig1_code(1);
144       dig1(6) <= dig1_code(0);
145
146       process(clock)
147       variable num_alt1, num2: integer;
148       begin
149       num_alt1 := num/10;
150       num2 := num_alt1 mod 10;
151       if rising_edge(clock) then
152       case num2 is
153          when 0 =>
154             dig2_code <= "10000001";
155          when 1 =>
156             dig2_code <= "11001111";
157          when 2 =>
158             dig2_code <= "10010010";
159          when 3 =>
160             dig2_code <= "10000110";
161          when 4 =>
162             dig2_code <= "11001100";
163          when 5 =>
164             dig2_code <= "10100100";
165          when 6 =>
166             dig2_code <= "10100000";
167          when 7 =>
168             dig2_code <= "10001111";
169          when 8 =>
170             dig2_code <= "10000000";
```

```
171         when 9 =>
172            dig2_code <= "10000100";
173         when others =>
174            dig2_code <= "10000000";
175      end case;
176      end if;
177      end process;
178      dig2(0) <= dig2_code(6);
179      dig2(1) <= dig2_code(5);
180      dig2(2) <= dig2_code(4);
181      dig2(3) <= dig2_code(3);
182      dig2(4) <= dig2_code(2);
183      dig2(5) <= dig2_code(1);
184      dig2(6) <= dig2_code(0);
185
186      process(clock)
187      variable num_alt2, num3: integer;
188      begin
189      num_alt2 := num/100;
190      num3 := num_alt2 mod 10;
191      if rising_edge(clock) then
192      case num3 is
193         when 0 =>
194            dig3_code <= "10000001";
195         when 1 =>
196            dig3_code <= "11001111";
197         when 2 =>
198            dig3_code <= "10010010";
199         when 3 =>
200            dig3_code <= "10000110";
201         when 4 =>
202            dig3_code <= "11001100";
203         when 5 =>
204            dig3_code <= "10100100";
205         when 6 =>
206            dig3_code <= "10100000";
207         when 7 =>
208            dig3_code <= "10001111";
209         when 8 =>
210            dig3_code <= "10000000";
211         when 9 =>
212            dig3_code <= "10000100";
213         when others =>
214            dig3_code <= "10000000";
215      end case;
216      end if;
217      end process;
218      dig3(0) <= dig3_code(6);
219      dig3(1) <= dig3_code(5);
220      dig3(2) <= dig3_code(4);
221      dig3(3) <= dig3_code(3);
222      dig3(4) <= dig3_code(2);
223      dig3(5) <= dig3_code(1);
224      dig3(6) <= dig3_code(0);
225
226      process(clock)
227      variable num_alt3, num4: integer;
228      begin
```

```
229       num_alt3 := num/1000;
230       num4 := num_alt3 mod 10;
231       if rising_edge(clock) then
232       case num4 is
233          when 0 =>
234             dig4_code <= "10000001";
235          when 1 =>
236             dig4_code <= "11001111";
237          when 2 =>
238             dig4_code <= "10010010";
239          when 3 =>
240             dig4_code <= "10000110";
241          when 4 =>
242             dig4_code <= "11001100";
243          when 5 =>
244             dig4_code <= "10100100";
245          when 6 =>
246             dig4_code <= "10100000";
247          when 7 =>
248             dig4_code <= "10001111";
249          when 8 =>
250             dig4_code <= "10000000";
251          when 9 =>
252             dig4_code <= "10000100";
253          when others =>
254             dig4_code <= "10000000";
255       end case;
256       end if;
257       end process;
258       dig4(0) <= dig4_code(6);
259       dig4(1) <= dig4_code(5);
260       dig4(2) <= dig4_code(4);
261       dig4(3) <= dig4_code(3);
262       dig4(4) <= dig4_code(2);
263       dig4(5) <= dig4_code(1);
264       dig4(6) <= dig4_code(0);
265
266
267    END logicFunction;
268
```