MIDDLE EAST TECHNICAL UNIVERSITY

ELECTRICAL-ELECTRONICS ENGINEERING DEPARTMENT

EE400 Summer Practice Report

Student Name: Zeynepnur ŞAHİNEL

Student ID: 2305399

SP Date: 16.08.2021-13.09.2021

SP Company Name: ASELSAN

Company Division: Avionic System Design Department

Company Location: Akyurt, Ankara

Related Field: Computer

TABLE OF CONTENTS

1. INTRODUCTION

I have performed my summer practise in ASELSAN which is the well-known electronics company of Turkey. ASELSAN designs, develops and manufactures electronics systems for military and professional customers. My practice lasted 4 weeks, 20 work days which starts on 16.08.2021 ends on 13.09.2021. The division in which I have performed my work is the Avionic System Design Department (ASTM). ASTM is the part of Microelectronic Guidance and Electro-Optical Group (MGEO). Avionic Systems Department develops and produces Airborne Electro-Optical Systems, Helmet Integrated Cueing Systems, Inertial Navigation Systems, Flight & Mission Management Systems, Cockpit Management Systems, Airborne Communication Systems, Identification, Friend or Foe Systems. In ASTM, I have worked under the supervision of System Design Engineer Samet Karakaş. In the first week, I have read the literature about common communication protocols like UART, SPI, I2C and test protocols for military avionics such as MIL-STD-1553, 461, 810 and DO160. In the second week, I have learned the basics of VHDL programming. In the third and the fourth weeks I have worked on coding the UART Protocol for Temren Grip Control Unit (TGCU) using VHDL. The TGCU project is distributed in three parts for three interns including me. Other interns have worked on the other parts of the projects. In this report, a detailed description of what I have accomplished during the summer practice is included. It starts with the description of the company and it is followed by the main part of the report in which work done is detailly given related to the Temren Grip Control Unit. Finally, the report is completed with a conclusion part. Also references and appendices are given at last.

2. DESCRIPTION OF THE COMPANY

Company Name: ASELSAN ELEKTRONİK SANAYİ ve TİCARET A.Ş.
Company Location: ASELSAN Akyurt Tesisleri Balıkhisar Mahallesi Çankırı Bulvarı 7. km. No:89, 0675016 Akyurt/ANKARA
Phone: (0312) 847 53 00
Website: https://www.aselsan.com.tr/

ASELSAN is a company of the Turkish Armed Forces Foundation, established in 1975 to meet the communication needs of the Turkish Armed Forces by national means. ASELSAN is the largest defense electronics company of Turkey whose capability/product portfolio comprises communication and information technologies, radar and electronic warfare, electro-optics, avionics, unmanned systems, land, naval and weapon systems, air defense and missile systems, command and control systems, transportation, security, traffic, automation and medical systems.
Facilities:
● Macunköy Facility: Macunköy is home to the CEO, Communications and Information Technologies Business Sector and Defence System Technologies Business Sector and Transportation, Security, Energy, Automation and Healthcare Systems Business Sector.
● Akyurt Facility: Microelectronic Guidance and Electro-Optical Group is at Aselsan Akyurt facilities.
● Gölbaşı Facility: The operations of land, air, sea, space, radar and electronical warfare systems for unmanned vehicles is being sited.
● Teknokent Facility: Research and Developments Activities of Communication and Information Technologies Group are carried out at Aselsan Teknokent Facility.
Fig. 1 shows the organizational structure of ASELSAN.

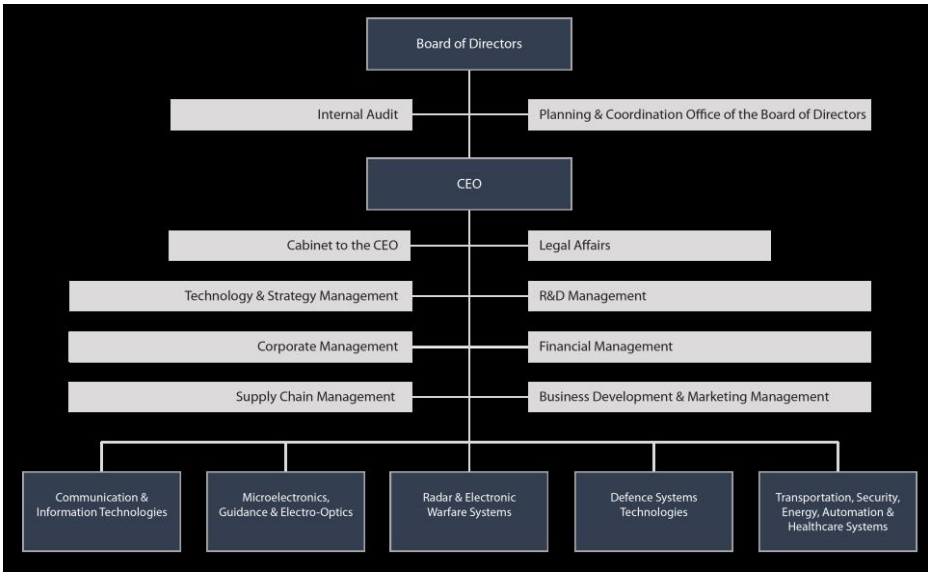


Figure 1. The organizational structure of ASELSAN

3. TEMREN GRIP CONTROL UNIT

In helicopters, pilots use controllers like joysticks to determine the target through a screen on the helicopter panel. Thus, Temren Grip Unit Controller (TGCU) refers to the Joystick Controller Unit in this project. The objective is to design a Printed Circuit Board (PCB) for the joystick controller and construct communication protocols to provide connection between a joystick and a computer. Since we were three people working on the same project, tasks were distributed by our responsible engineer. The project was divided into three parts: PCB layout, SPI and UART. Ayberk Acar was assigned to the PCB part in which he created the PCB layout using Altium. Bertay Eren was responsible for coding SPI protocol using VHDL. Whereas, I am assigned to code UART protocol using VHDL.

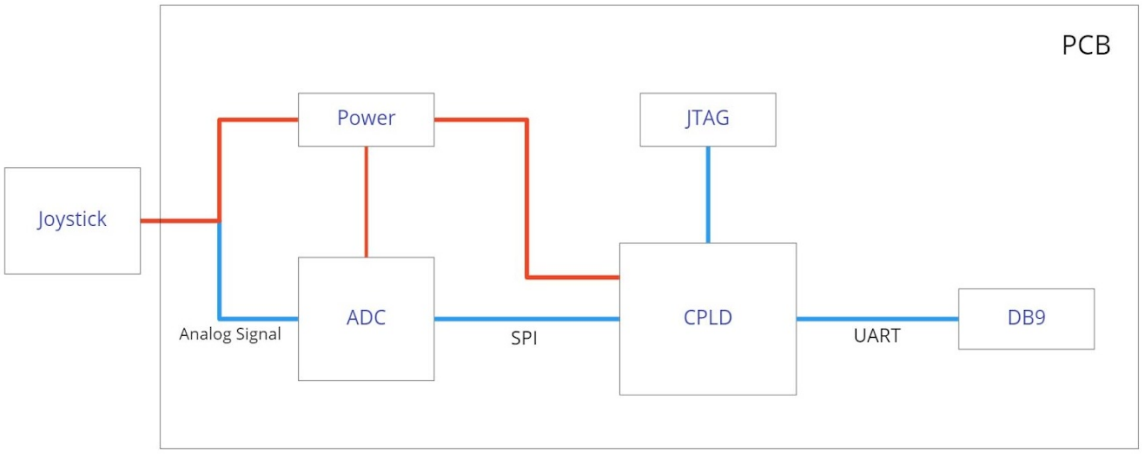


Figure 2. The schematic for the Temren Grip Control Unit

Fig. 2 demonstrates the schematic of the Joystick Controller. Here, the joystick generates analog signals depending on the position of the joystick. Inside the joystick, there is a potentiometer. The resistance of the potentiometer changes with altering the location of the joystick, which yields a change in output voltage. The joystick has two analog output voltage signals. One signal refers to the x direction, the other signal represents the y direction. These analog signals become inputs of the Analog Digital Converter (ADC). As the name implies, ADC converts analog data to digital data. Afterwards this digitized data is given to the Complex Programmable Logic Device (CPLD). CPLD constructs the brain of the system. It converts VHDL to machine language to complete tasks. Joint Test Access Group (JTAG) is a hardware interface that enables a computer to communicate directly with the chips on the PCB. Lastly, UART protocol is used for data transmission serially to the computer through DB9, which is a common connector type for serial peripherals.

4. LED FLIP MODULE

Before coding the UART Protocol, the first task that I implemented by using VHDL is the Led Flip module. In this module, a basic frequency divider is constructed. It is a 2 input and 1 output entity. It takes clk, reset as serial inputs and led as serial output.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ledflip is
port ( clk, reset : in std_logic;
       led : out std_logic);
end ledflip;

architecture struct of ledflip is

constant cnt_1HZ: natural := 12500000;
signal counter : integer := 1;
signal tmp : std_logic := '0';

begin

flipping: process (clk,reset) is
begin
    if (reset = '0') then
        counter <=1;
        tmp <= '0';
    elsif rising_edge(clk) then
        counter <= counter +1;
        if (counter=cnt_1Hz) then
            tmp <= not(tmp);
            counter <=1;
        end if;
    end if;
led <= tmp;
end process;
end struct;
```

Figure 3. Led flip module in VHDL

Fig. 3 shows the whole code to create the Led flip module. In the entity statement module, inputs and output are defined. Afterwards, architecture is constructed in which the function of the entity is determined. Here, it is asked to design a flipping led with 1 Hz, meaning that at every one second the led is going to turn on. For this purpose cnt_1Hz is created, the calculation is given below for 25 Mhz clk:

$$cnt\ 1Hz = \frac{clk\ frequency}{desired\ frequency} \times Duty\ Cycle = \frac{25\ MHz}{1\ Hz} \times 0.5 = 12500000$$

Moreover, counter and tmp signals are introduced before the begin statement. In the flipping process, firstly initialization of the counter and the tmp signal is done by active low reset. Then at each rising edge of the clk, the counter increases until it reaches the cnt_1Hz value. At that point, the tmp signal gets its complement of it and counter set to 1 again. After if statements, the tmp signal is assigned to the led signal. The process continues whenever clk or reset changes. clk and reset constitutes the sensitivity list, which will be explained in detail in Section 5.

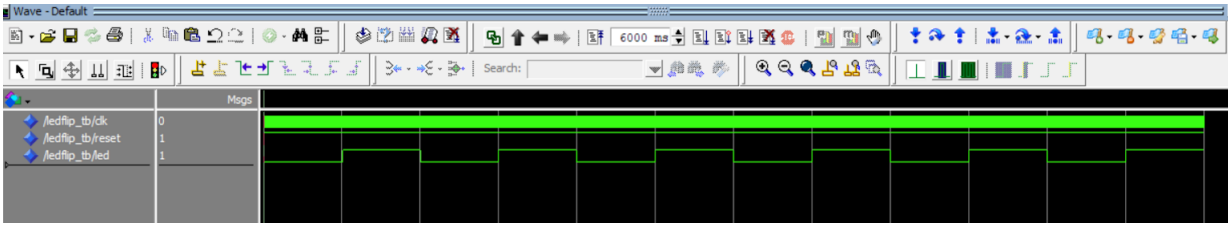


Figure 4. The simulation result of the Led Flip module

Fig. 4 demonstrates the simulation result of the Led Flip Module. To construct this simulation, test bench is written in VHDL which is given in Appendix A. Here, 25 Mhz clk is created, and the simulation runtime is arranged as 6 sec. Therefore, clk signal can not be observed clearly. Whereas, the led output can be interpreted obviously. For 6 sec runtime, there are six pulses as expected.

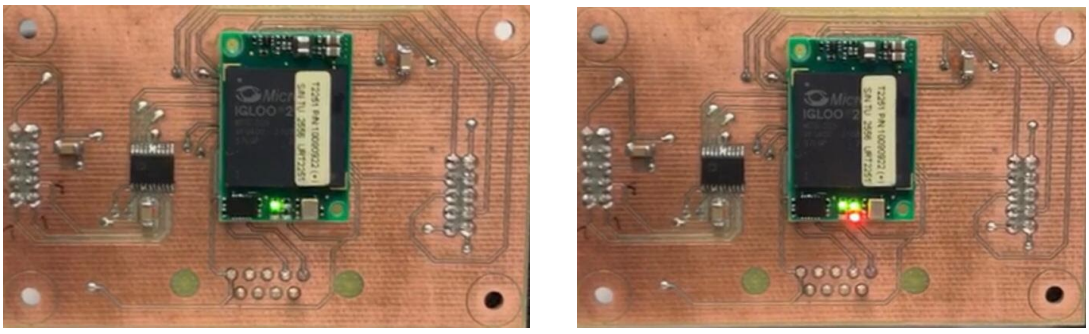


Figure 5. Demonstration of the Ledflip module using designed PCB

Fig. 5 shows the designed PCB and flipping leds on it. To test whether the pcb works or not, this simple led flip code is uploaded to the CPLD. As can be seen from this figure the card and the CPLD passed the test. The left part shows when the leds are turned off, while the right part shows when led are turned on. Using two of this Led Flip module, one can manipulate two leds. To upload the codes written in VHDL, Libero is used. Libro is one of the comprehensive FPGA design and development software.
In this PCB, the small integrated circuit is the Analog Digital Converter and the bigger integrated one is the CPLD. While making this card, Ayberk paid attention not to intersect the lines. Smaller circuits like for this case, even though in the layout in Altium no intersection occurs, short circuit paths can emerge during the printing phase. To determine if a short circuit exists, scanning with a multimeter is employed. These multimeters include a short-circuit test part. Whenever it detects a short circuit in a circuit, it starts beeping.

5. UART PROTOCOL

UART, or universal asynchronous receiver-transmitter, is a device-to-device hardware communication protocol. UART uses solely two wires for its receiving and transmitting ends. As its name implies, UART is a communication protocol that employs asynchronous serial communication with configurable speed. This speed can be also referred to as baud rate, which means how many bits per second is transmitted. UART can take one of several baud rate values such as 9600, 19200, 38400, 57600, 115200, 230400, 460800, 921600, 1000000, 1500000. [1]

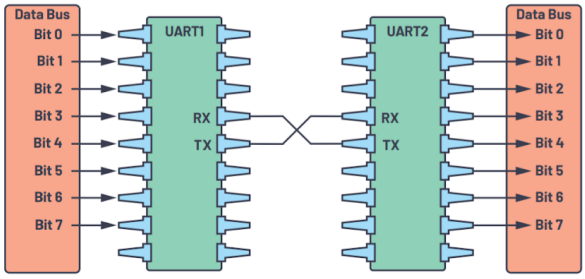


Figure 6. UART with Data Bus [1]

Fig. 6 illustrates communicating two UART with data buses. Here, the transmitting UART takes parallel input from the data bus, then transmits these bits serially through TX pin to the RX pin of the receiver UART. To provide the communication, the baud rate must be set the same for both transmitting and receiving devices. Having the same baud rate, the synchronization is managed. In this figure, UART lines construct the communication medium. Bits are sent serially one to another.

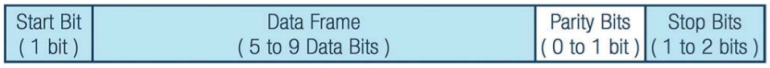


Figure 7. UART Data transmission pattern [1]

Fig. 7 shows the data transmission pattern used in UART. A transmission packet consists of a start bit, a data frame, a parity bit and a stop bit. Normally, the UART data transmission line is kept at a high voltage level when it is not sending data. To initiate the transmission, UART pulls the transmission line to low voltage for one clock cycle. The data frame consists of the actual data being sent. It can take five bits to eight bits long if a parity bit is used. Parity refers to the evenness or oddness of a number. It is a way for realizing whether any changes occur in the data frame or not. Bits can be altered by mismatched baud rates, electromagnetic radiation or long distance transfers. After reading the data frame, it counts the number of 1s and checks if the total is an even or odd number. If odd parity is employed, for the case where the total number of 1s of the data frame is even, parity bit takes 0. If the parity bit matches with the actual data, UART understands that the transmission occurs without errors. To indicate that the data packet transmission finished, the data transmission line is drived to high voltage for one or two bits duration.

5.1 Bit Controller

UART consists of two sub-blocks, Bit Controller and Uart Controller. In this section, the Bit Controller entity is examined. Bit Controller converts 8-bit parallel input into serial output. This serial output transmits data with the baud rate. (57600 Hz)

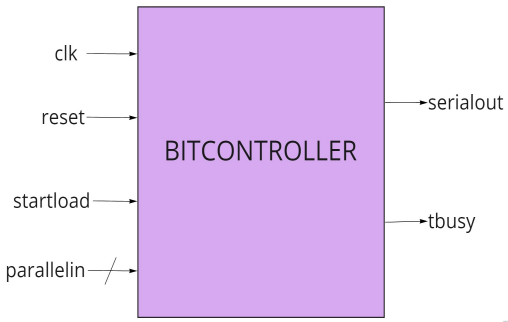


Figure 8. The block diagram of the BITCONTROLLER entity

As Fig. 8 indicates, BITCONTROLLER has four input terminals being clk, reset, startload, parallelin, and two output terminals being serialout and tbusy. While parallelin is an 8-bit vector input, others are 1-bit logic values. Since this entity is a synchronous sequential circuit, clk is used as an input. Moreover, reset is introduced to make proper initializations. startload input determines when the data parallelin contains starts to send via serialout. tbusy shows whether the entity is sending data or is in idle mode.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity BITCONTROLLER is
port
(clk: in std_logic;
reset: in std_logic;
startload: in std_logic;
parallelin: in std_logic_vector(7 downto 0);
serialout: out std_logic;
tbusy: out std_logic
);
end BITCONTROLLER;
```

Figure 9. The BITCONTROLLER entity declaration in VHDL

Fig. 9 shows how inputs and outputs are indicated within an entity block. In VHDL, the entity defines the interface. Here, the BITCONTROLLER block diagram and the entity declaration have a direct correlation. Inside the entity, ports are defined as being in and out. In addition to in and out statements, the type of ports should be written. For this case, std_logic represents one data bit; whereas, std_logic_vector represents several data bits. The std_logic_1164 library is added at the top, to make std_logic and std_logic_vector available.

```
architecture STRUCT of BITCONTROLLER is

constant period: integer:= 278; ----to obtain 57600 Hz-----

signal timecounter: integer:=1;
signal cntx: integer:=0;

type vhdlarray is array (0 to 11) of  std_logic;
signal bitarray: vhdlarray;

type StateType is (Idle, DataSent, WaitPeriod);
signal State: StateType;
```

Figure 10. The architecture of the BITCONTROLLER.vhdl

Fig. 10 demonstrates the interior signals in the bit controller entity and a constant to construct transmission at the baud rate. The *architecture* part of the code describes the function of the block. The processes and some assignments are conducted inside the architecture after begin statement. In Fig. 10 before processes, required signals are introduced:

- *timecounter* is used to count until period. The baud rate is arranged to 57600 Hz, and clk is set up to 16 MHz in CPLD. period constant is found dividing clk frequency into baud rate. Even though its name is period, it is not the baud rate period.

$$(constant)\ period = \frac{clk\ frequency}{baud\ rate} = \frac{16\ MHz}{57600\ Hz} = 277.7 \simeq 278$$

- *cntx* is introduced to count bitarray elements.
- *bitarray* signal is constructed for one data packet having 8-bit input data, start, parity, and stop bits. Thus, one data packet contains 11 bits. Before the construction of the bitarray signal, *vhdlarray* is created as an array object. This object has 12 elements having 1-bit logic data.
- *State* signal is a corresponding signal to the *StateType.* This type is called the enumeration type. Instead of fixed bit patterns, the symbolic names of the data type values are used which will be mapped to a bit-level representation automatically during synthesis. [2] Idle, DataSent, WaitPeriod are state names.

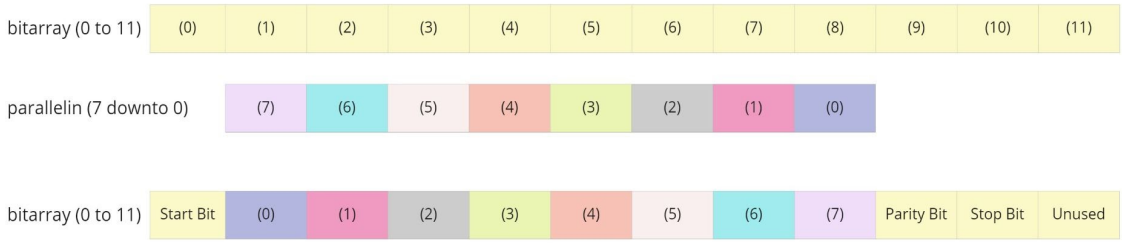


Figure 11. Illustration of bitarray and parallelin vectors

As Fig. 11 shows, bitarray is a 12-bit and parallein is an 8-bit logic vector. The following bitarray is the final version with start, parity, and stop bits. bitarray(1) to bitarray(8) are allocated for the parallelin in reverse order. Since UART transmits the least significant bit first. The last bit of the bitarray is unused yet it is defined for the synthesis process.

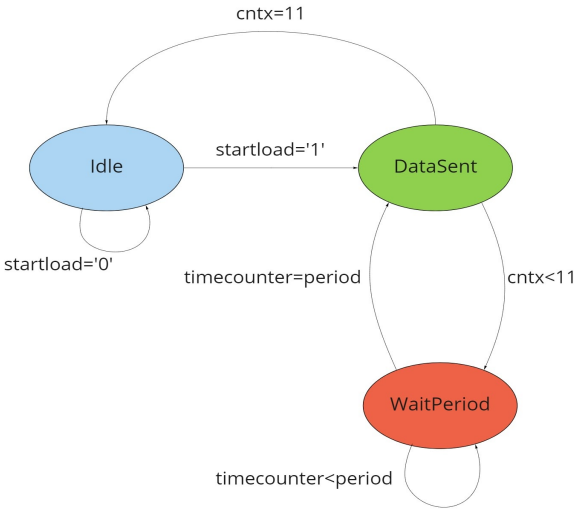


Figure 12. State diagram of the Bit Controller

In Fig 12, the state diagram is shown. In Idle state, the Bit Controller stays in the same state until startload becomes 1. Later, the state becomes DataSent. In DataSent until cntx becomes 11, the state changes periodically between DataSent and WaitPeriod. When WaitPeriod is the state, timecounter is started. Up to the period value, defined in Fig. 10, the state remains in WaitPeriod. At this period, the state comes back to DataSent. If cntx reaches 11, Idle becomes the new state. After the startload turns into 1, the same process repeats itself.

```
begin
---------------------Finite State Machine----------------------------
FSM: process (clk, State, startload) is
begin
if reset='0' then------Reset is active low
    State<=Idle;
    serialout<='1';
    bitarray<="000000000000";
    tbusy<='0';
```

Figure 13. Initialization of the Bit Controller Finite State Machine

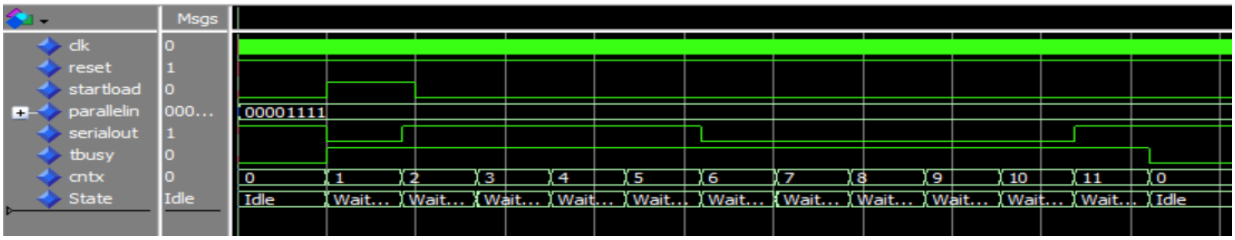
In Fig. 13, the finite state machine process is introduced. In VHDL, the process statement contains a set of sequential statements that assign values to signals. With the process statement, combinational and sequential logic can be created. After the process statement, the sensitivity list is identified. The sensitivity list includes the signals causing the process statement to execute if their values change. Here, clk, State, and startload constitute the sensitivity list. Inside the process statement, signal initializations are made with reset. Since reset is active low, when it becomes 0 assignments are made. In Fig. 13., initially, the state is Idle, serialout is 1, tbusy and bitarray is 0.

```
else
    if rising_edge(clk) then

        case State is

        when Idle=>

            tbusy<='0';

            cntx<=0;

            if(startload='1')then

                State<= DataSent;-----next state----

                bitarray(0)<='0'; -----start bit-----
                bitarray(10)<='1';-----stop bit------
                bitarray(1)<=parallelin(0);
                bitarray(2)<=parallelin(1);
                bitarray(3)<=parallelin(2);
                bitarray(4)<=parallelin(3);
                bitarray(5)<=parallelin(4);
                bitarray(6)<=parallelin(5);
                bitarray(7)<=parallelin(6);
                bitarray(8)<=parallelin(7);
                bitarray(9)<=parallelin(7) xor parallelin(6) xor
parallelin(5) xor parallelin(4) xor parallelin(3) xor parallelin(2) xor
parallelin(1) xor parallelin(0);---------even parity bit--------

                bitarray(11)<='0';---unused bit---
            end if;

when DataSent=>

            tbusy<='1';---------set tbusy--------

            if(cntx<11) then
                serialout<=bitarray(cntx);
                cntx<=cntx+1;
                State<=WaitPeriod;
            else
                cntx<=0;
                State<=Idle;
            end if;

when WaitPeriod=>

            if(timecounter<period) then
                State<=WaitPeriod;
                timecounter<=timecounter+1;
            else
                timecounter<=1;
                State<=DataSent;
            end if;
        end case;
    end if;
end if;
end process;
end STRUCT;
```

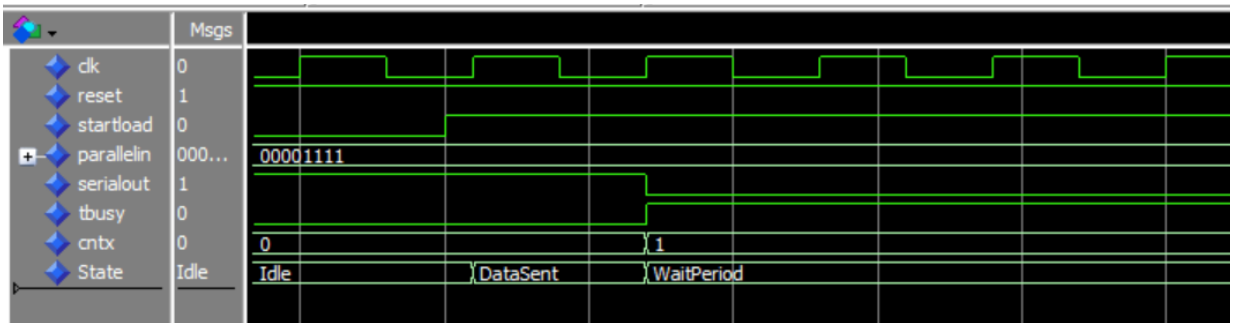
Figure 14. State Assignments of Bit Controller

Fig. 14 shows the finite state machine in detail. After initializing the state and outputs, operations are held inside the *rising_edge (clk)* statement. Every assignment is done according to the rising edge of the clk to have a synchronous circuit. Inside this clk statement, different states are described. *case* statement is used to construct more than one state. In VHDL, a variable or signal having many options is written inside the case statement. (case <signal,variable> is). Case statements are generally used to implement multiplexers. In Idle state, tbusy is 0 since no data transmission occurs. cntx is initialized as 0 before DataSent state. If startload is 1, meaning data comes to the Bit Controller; bitarray is assigned according to Fig. 11. The first index of the bitarray is attained zero as the start bit. This bit implies the start of the incoming data. Starting from the second bit up to the ninth bit, parallelin bits are transferred respectively. The tenth bit is the even parity check bit. Taking xor of all parallelin bits yields even parity. For instance, if the parallelin has an even number of 1s, even parity is 0; whereas parallelin has an odd number of 1s, even parity is 1. The eleventh bit is the stop bit which is 1. The twelfth bit is an unused bit, which is required for the synthesis part. In DataSent state, tbusy becomes 1 meaning Bit Controller starts to transmit. In this state, bitarray elements are sent through serialouts. At each increment, the state is updated to WaitPeriod until cntx becomes 11. When cntx reaches 11, cntx is set to be zero and the state will be Idle. And the same process repeats itself after startload turns into 1. Until this event, seriolut remains 1 and tbusy is 0. In WaitPeriod state, serialout keeps its value during the baud period. Baud period is calculated and predefined as *period* in Fig. 10. Period is found approximately 278, meaning that for 278 clk rising edge the state remains at WaitPeriod and counter increases by 1 . When timecounter reaches that value, it returns to DataSent, and timecounter is set to 1.

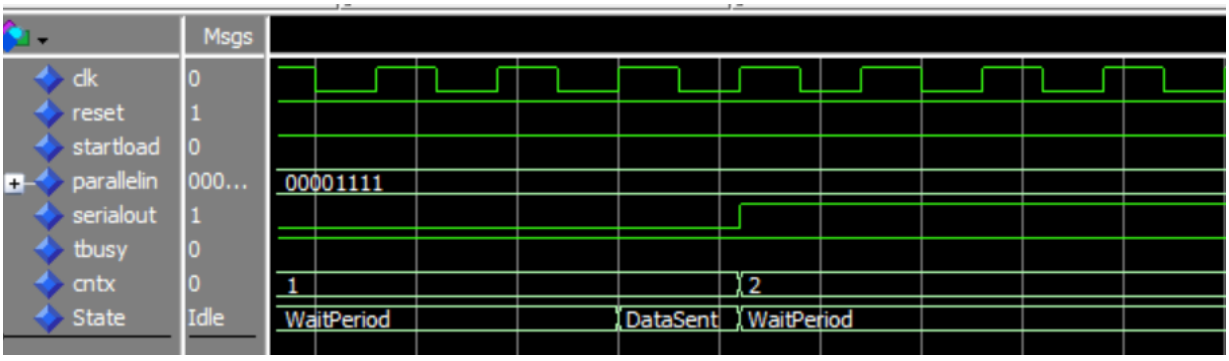
(a)



(b)



(c)



(d)

Figure 15. a) Simulation result where parallelin is 00001111 and 11110001, respectively. b) Zoomed simulation result where parallelin is 0000111. c) Zoomed simulation result where parallelin is 00001111 and cntx is 0 and 1. d) Zoomed simulation result where parallelin is 00001111 and cntx is 1 and 2.

Fig. 15 demonstrates the simulation results of the Bit Controller. These simulations are done by using ModelSim. For that, Bit Controller test bench is created. This test bench is given in the Appendix B. In the Bit Controller Test Bench, inputs are given manually. clk is created by a process whose period is approximately 60 ns. In Fig. 15a, simulation is run for 700 us; thus the clk signal seems constant blurry line. reset is assigned as 0 for 20 ns, then it becomes 1. Since the reset signal is active low, the Bit Controller enters to the finite state machine process. Moreover, to differentiate the even parity; two parallelin data are given with different numbers of 1. Two parallelin data are transmitted in a way that their least significant bits are sent first. For instance, for the first parallelin data; transmission should be 01111000001. Here, the first bit is the start bit, from the second to the ninth bit parallelin is sent conversely. The tenth bit is the even parity bit, which is 0; since, there are four 1s in the input. The last bit is the stop bit, which is 1. Till startload becomes 1, serialout value is latched at 1. Fig. 15b shows the first parallel in data simulation in detail. Here, tbusy is 1 during data transmission; elsewhere it is 0. cntx is increased to 11 while tbusy is 1. Fig. 15c and Fig. 15d state transition shows clearly. Initially, state is Idle, when startload is 1 the state becomes DataSent. During this transition cntx remains as 0. At next rising edge clk, cntx becomes 1 and the state changes to WaitPeriod. After baud period is completed, WaitPeriod state goes to DataSent. The state remains at DataSent for one clk cycle and cntx keep its value at 1. Following rising edge, cntx updates to 2 and the state become WaitPeriod again. This process goes up to cntx reaches to 11.

5.2 UART Controller

In this section, the UART Controller entity is investigated. UART Controller reads 11 parallel inputs periodically, and then transmits these parallel inputs sequentially as outputs. It also creates a serial output to turn on the Bit Controller in the UART module, which will be explained later in the Section 5.3

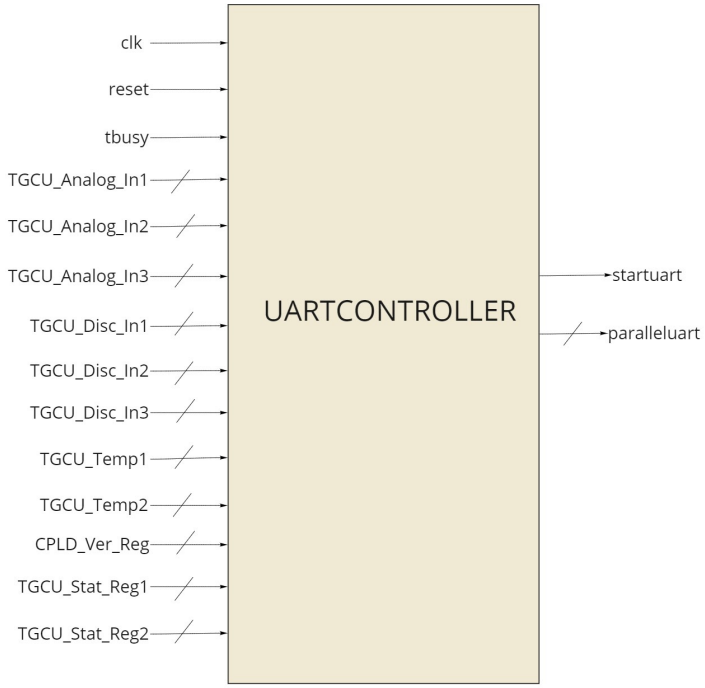


Figure 16. The block diagram of the UARTCONTROLLER entity

As Fig 16. indicates, UARTCONTROLLER has 14 inputs, being 3 serial and 11 parallel inputs. clk is used to provide synchronous circuit and reset is introduced for the initialization purposes. tbusy is related to sending data or being in idle mode. It is the same signal that is used in the Bit Controller. TGCU Analog Inputs come from ADC Unit, these analog inputs are used to show the position of the joystick on the computer screen. In this project, TGCU discrete inputs and temperature values are not read; thus, these values are given manually. Here, CPLD Version Register is employed to show whether a UART module works or not by printing the version number on the screen. CPLD and TGCU Stat Register values are also determined arbitrarily. The paralleluart is an output which is the same as the parallel inputs. Here, paralleluart takes input values from TGCU_Analog_In1 to TGCU_Start_Reg2 respectively. The startuart is an output which is later used to start the Bit Controller. Also in this module, all parallel signals are 8-bit logic vectors.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

```
entity UARTCONTROLLER is
port
(clk: in std_logic;
reset: in std_logic;
tbusy: in std_logic;
data1: in std_logic_vector(7 downto 0);
data2: in std_logic_vector(7 downto 0);
data3: in std_logic_vector(7 downto 0);
parallelout: out std_logic_vector(7 downto 0);
startuart: out std_logic
);
end;

architecture STRUCT of UARTCONTROLLER is

constant period: integer:=160000;-------for trigger------
signal timecounter: integer:=1;
constant pwmval: integer:=2;-----for pwm----------
signal trig: std_logic;

signal cntx: integer:=0;

type vhdlarray is array (0 to 16) of  std_logic_vector(7 downto 0);
signal data_array: vhdlarray;

type StateType is (Idle, DataArrange, DataSent, WaitBusy);
signal State: StateType;
```

Figure 17. The UARTCONTROLLER entity and architecture declaration in VHDL

Fig. 17 shows inputs and outputs declarations inside the UARTCONTROLLER entity. Here, TGCU Analog inputs are introduced as data1, data2, data3 respectively for short assignment. Other parallel inputs are not taken as inputs inside this block since they are initialized later inside the data_array. Similarly to the Bit Controller, serial inputs are indicated with std_logic; whereas, parallel inputs and outputs are shown with 8-bit std_logic_vector. The architecture part demonstrates the interior signals in the Uart Controller entity and constants to create periodic trigs later. Below, introduced signals are explained:

- *period* is a constant which is used to create trig signals at each 10 ms. It is found by dividing clk frequency into trig frequency (100 Hz).

$$(constant)\ period = \frac{clk\ frequency}{trig\ frequency} = \frac{16\ MHz}{100\ Hz} = 160000$$

- *timecounter* is a signal which counts up at each rising clk edge. This signal is used to make comparisons with period and pwmval.
- *pwmval* is a constant which is used to create one clk pulse width pwm.
- *trig* represents the trigger which is used for the data read at each 10 ms in the Uart Controller.
- *cntx* signal counts through data_array elements.

- *data_array* signal is constructed to store the values of 8-bit parallel inputs and other defined values. Before the construction of the data_array signal, *vhdlarray* is created as an array object. This object consists of 17 elements where each element has 8-bit logic data.
- *State* signal is a corresponding signal to the *StateType.* Likewise Bit Controller, StateType is an enumeration type. In Uart Controller there are four states being Idle, DataArrange, DataSent, WaitBusy.

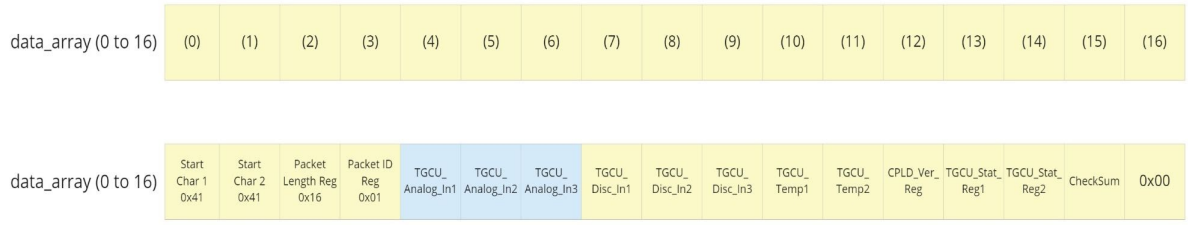


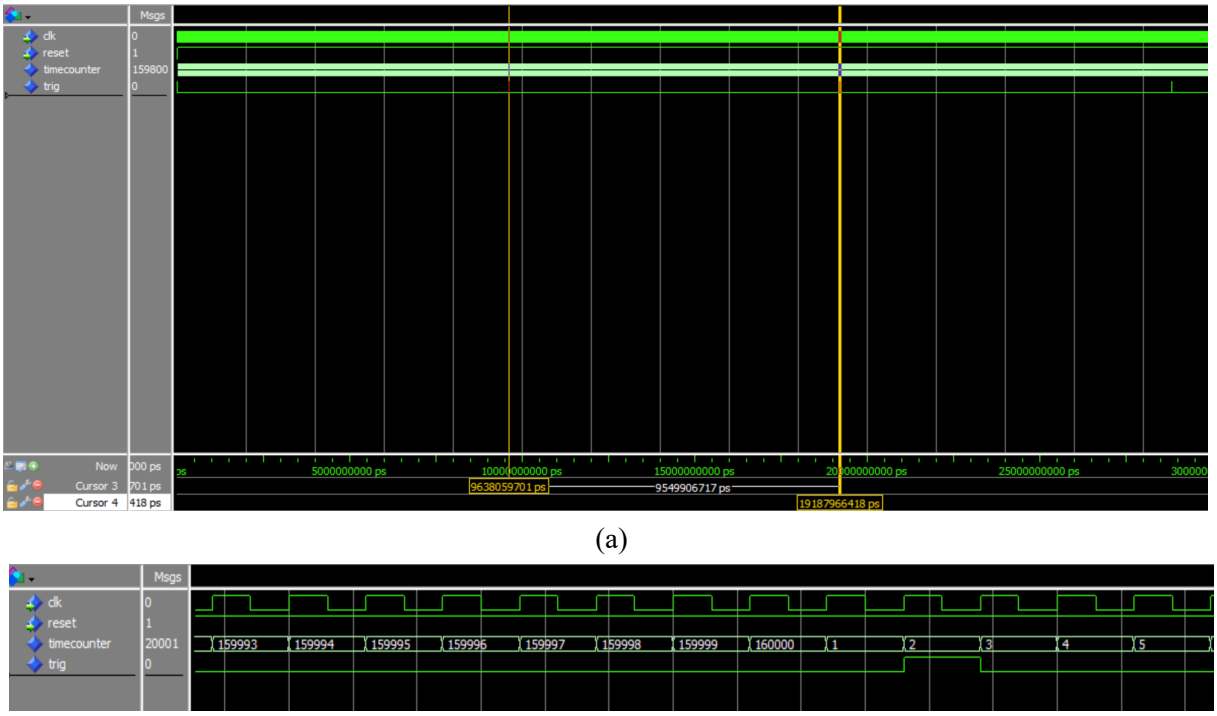
Figure 18. Illustration of the data_array

As Fig. 18. shows, data_array has 17 elements. In addition to inputs which are indicated in Fig x9. Start Char 1 and 2, Packet Length Reg, Packet ID Reg are elements which are predetermined. Moreover, after storing parallel inputs to the data_array, data_array(15) is reserved for CheckSum. CheckSum is a control element which ensures that the summation of the all data_array elements is zero. The last element of the array is unused, it is defined for the synthesis process. Inside one of the if blocks in FSM, data_array takes its 16th value due to the cntx value. Thus, for the synthesis one should consider this extra element.

```
-----------------Creating Trigs at each 10 ms (PWM)-----------------
trigprocess: process(clk) is
begin

if reset='0' then
    trig<='0';

else
if rising_edge(clk) then

    if(timecounter<period) then
        timecounter<=timecounter+1;
    else
        timecounter<=1;
    end if;

    if(timecounter<pwmval) then
        trig<='1';
    else
        trig<='0';
    end if;
end if;
end if;
end process;
```

Figure 19. Trigger Process of the Uart Controller

Fig. 19. demonstrates the trigger process at each 10 ms. Firstly, *process* is defined with clk sensitivity meaning that when the clk value changes, the process wakes up again and starts to execute. For initialization, trig is set to zero; when, reset becomes zero. Here, reset is active low. If reset is not activated, another if block is introduced. At each rising edge of the clk, this if block starts to execute. Nested if statements are written to create a pwm pulse. The first nested if increases *timecounter* until it reaches to the *period* value; otherwise, it is set to 1. Simultaneously, the second if statement starts and it assignes trig to 1 when timecounter is less than pwmval, elsewhere trig is assigned to 0.



(a)

(b)

Figure 20. (a) The simulation result of the trig process for 30 ms. (b) Zoomed simulation result of the trig process where timecounter finishes its loop and startes executing again.

Fig. 20 demonstrates the simulation results of the trig process. In part a,30 ms result is shown, there are four trig can be observed where each trig pulse has approximately 10 ms interval. In part b, one trig is shown. Here, timecounter finishes its one cycle upto period value (160000) and starts to execute again. Since timecounter is initialized as 1 in the Fig. 17, trig becomes 1 at the next clk cycle. Also, it should be mentioned that, unlike C programming, in VHDL multiple if blocks in the same process are executed simultaneously. Thus, one can achieve a pulse when the timecounter is 2.

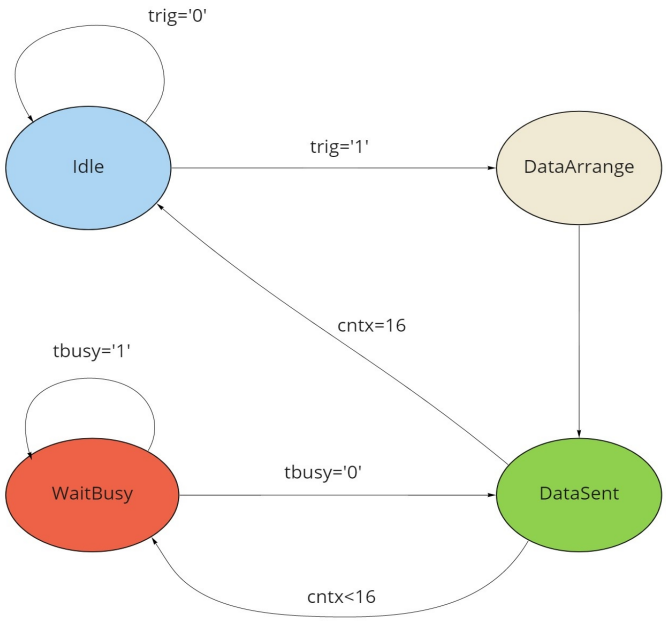


Figure 21. State Diagram of the UART Controller

In Fig. 21 the state diagram is shown which has four states being Idle, DataArrange, DataSent and WaitBusy. Initially, the Uart Controller is in the Idle state. It stays in this state until trig becomes 1. When the trig pulse is 1, Uart Controller should start to read the data. Thus, the state changes to DataArrange in which all the required data is kept in the data array. After one clk cycle the state directly becomes the DataSent. Here, the data transmission starts through *parallelout.* Parallelout refers to the paralleluart in Fig. x9. According to the cntx value, the state turns into WaitBusy or Idle. WaitBusy state provides a hold during one data transmit period. This period is determined by the tbusy which is discussed detailly in the Section 4.1. This hold continues periodically until cntx reaches to 16. Meaning that all the data inside the array is transmitted.When the transmission is over, the state returns to the Idle State. Therefore, at each trig pulse a similar process repeats.

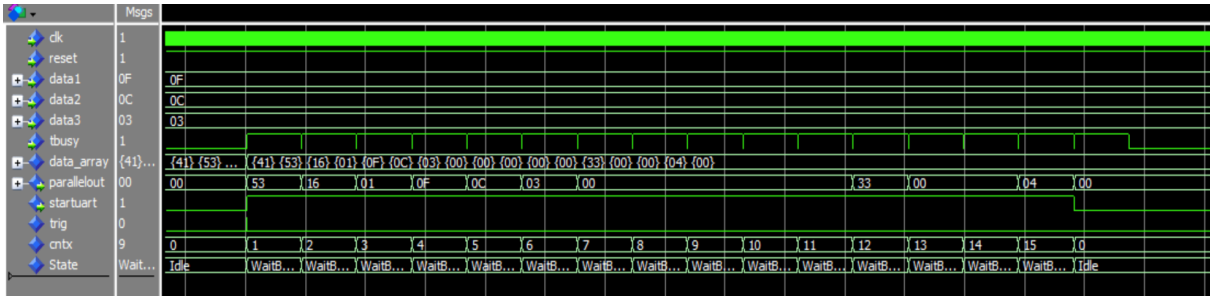
```
----------------Finite State Machine--------------------------------
FSM:process(clk, State, tbusy, cntx) is
begin

if reset='0' then
    State<=Idle;
    parallelout<="00000000";
    data_array(16)<="00000000";

else
    if rising_edge(clk) then

        case State is

        when Idle=>

            startuart<='0';

            if (trig='1') then
```

```
                State<= DataArrange;------Next State-------
            end if;

          when DataArrange=>

            data_array(0)<="01000001"; ---start character 1 0x41
            data_array(1)<="01010011"; ---start character 2 0x53
            data_array(2)<="00010110"; ---packet length reg 0x16
            data_array(3)<="00000001";----packet ID reg     0x01
            data_array(4)<=data1;
            data_array(5)<=data2;
            data_array(6)<=data3;
            data_array(7)<="00000000";-----TGCU Disc_In1;
            data_array(8)<="00000000";-----TGCU Disc_In2;
            data_array(9)<="00000000";-----TGCU Disc_In3;
            data_array(10)<="00000000";-----TGCU Temp1;
            data_array(11)<="00000000";-----TGCU Temp2;
            data_array(12)<="00110011";-----CPLD Version   0x33
            data_array(13)<="00000000";-----TGCU_Stat_Reg1
            data_array(14)<="00000000";------TGCU_Stat_Reg2

            data_array(16)<="00000000";----dummy-----

            State<=DataSent;-------Next State-----

           when DataSent=>

             data_array(15)<= "00000000"- (data_array(0)+ data_array(1)+
data_array(2)+data_array(3)+data_array(4)+data_array(5)+data_array(6)+data
_array(7)+data_array(8)+data_array(9)+data_array(10)+data_array(11)+data_a
rray(12)+data_array(13)+data_array(14)); ----CheckSum----

            startuart<='1';

            parallelout<=data_array(cntx);

            if(cntx<16) then
                State<=WaitBusy;
            else
                cntx<=0;
                State<=Idle;
            end if;

          when WaitBusy=>

            if(tbusy='1') then
                State<=WaitBusy;
            else
                State<=DataSent;
                cntx<=cntx+1;
            end if;
        end case;
    end if;
end if;
end process;
end struct;
```

Figure 22. The State Assignments of the UART Controller

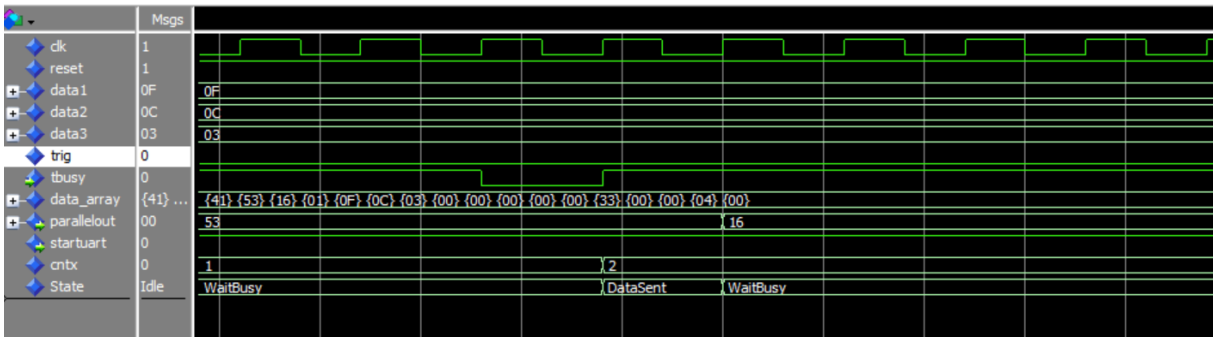
Fig. 22 shows the Finite State Machine of the Uart Controller. The sensitivity list of this process consists of clk, State, tbusy, cntx. When one of these changes, the process starts to execute again. After defining the sensitivity list, initializations are made with the reset signal. State is set to Idle, parallelout and the last element of the data array is assigned to zero. When reset is deactivated, then at each rising clk edge, different cases of State are executed.

In the Idle state, startuart is set to zero since no data transmission occurs in this state. When trig becomes 1, the next state will be DataArrange. Otherwise, it stays in the Idle state. In the DataArrange state, data_array elements are assigned. Here, the first and the second element should be the start character being 0x41 and 0x53. The third element is related to the packet length. Since data_array has 16 used elements, it should be taken as 16. The fourth element is a packet id value. Up to the sixteenth element, parallel inputs are assigned in the order of Fig. 15 Then the next state becomes DataSent. In this state firstly the CheckSum is calculated. This calculation is not done in the DataArrange state since in VHDL value assignments to the data_array occur concurrently. After the calculation of the CheckSum, the startuart is set to one. Meaning that data transmission starts. And the parallelout begins to take the values from data_array. Here, cntx is used to determine which element should be sent through parallelout. As long as cntx is smaller than 16, the next state becomes WaitBusy; otherwise, it goes to the Idle state. In the WaitBusy state, as the name implies tbusy is expected to be one. Until tbusy becomes zero, the state remains in WaitBusy. Else, it comes back to the DataSent state to send other elements of the data array. While doing this, it increments the cntx by one.



(a)

(b)



(c)

Figure 23. a)The simulation result of the UART Controller. b) The zoomed simulation result of the UART Controller where one trig pulse is observed. c) The continuation of part b where cntx takes values 1, 2, and more.

Fig. 23 shows the simulation results of the UART Controller. In part a, an extensive display of this controller is given with clk, reset, TGCU Analog inputs (data1-3), data_array, parallel output, startuart, and internal signals like cntx and State. Here, data1, data2, data3 is determined arbitrarily being 0x0F, 0x0C and 0x03. Given this data and internal values data_array is created as shown in this figure. The purpose of this controller module is to pass data_array elements consecutively to the parallelout. However, looking at the part a, the first element of the data_array and the full state transition can not be seen. Thus, the zoomed version is given in part b. In this part, Until trig comes the state remains in the Idle state; when trig becomes 1, in the next clk cycle the state becomes DataArrange. In this state no transmission occurs, in the next clk parallelout becomes 0x41, the state changes to WaitBusy, and startuart becomes 1. Here, tbusy does not yet become 1, after one clk cycle the state becomes DataSent. And the next element of the data array (0x53) passed through parallelout. It remains at the WaitBusy until tbusy becomes zero. It should be noted that these simulation results are taken from the UART_TB. Therefore tbusy is the output of the Bit Controller. Since Bit Controller takes startuart as input until startuart becomes 1, tbusy stays at 0. Thus, 0x41 can remain for 2 clk cycle. However, it won't be a problem in the UART simulation results. (section 4.3) In part c, the continuation of the cntx and state transition can be seen. Here WaitBusy and DataSent change periodically as expected.

5.3 UART

In this section, the UART entity is investigated. The UART module takes the same inputs as the UART Controller. Except for tbusy, UART takes tbusy input from Bit Controller. The outputs of UART are the same as the Bit Controller. Here, the tbusy output of the Bit Controller is given as input of the UART Controller, as mentioned previously.

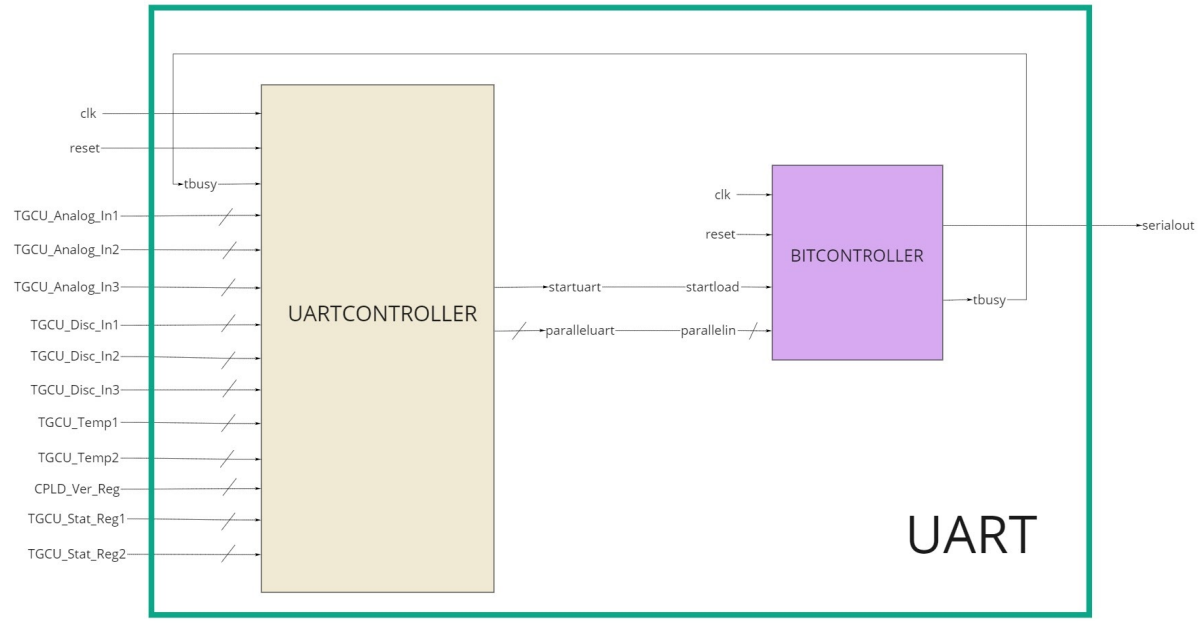


Figure 24.. The block diagram of the UART entity

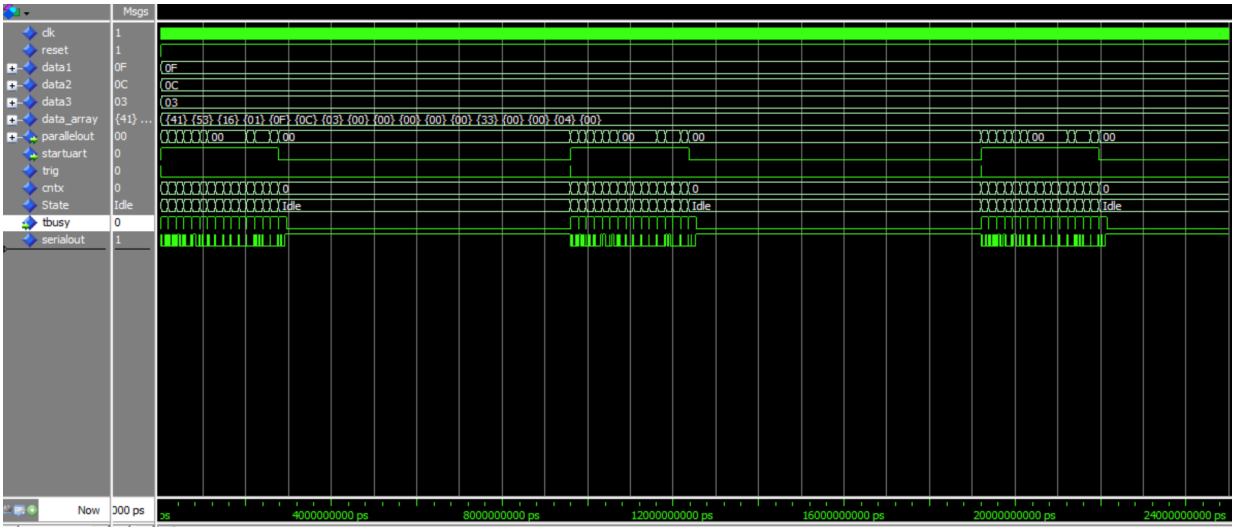
As Fig. 24 indicates, UART has 13 inputs coming externally and 1 input from the Bit Controller entity. Here, the outputs of the UART Controller are used as inputs for the Bit Controller. The serial output startuart creates the startload input of the Bit Controller. This signal is used for indication to start the data transmission through serialout. Whereas, paraleluart, the parallel output of the UART Controller, forms the parallelin input of the BitController.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity UART is
port
(clk: in std_logic;
reset: in std_logic;
data1: in std_logic_vector(7 downto 0);
data2: in std_logic_vector(7 downto 0);
data3: in std_logic_vector(7 downto 0);
serialout: out std_logic
);
end UART;

architecture STRUCT of UART is

component UARTCONTROLLER
port
(clk: in std_logic;
reset: in std_logic;
tbusy: in std_logic;
data1: in std_logic_vector(7 downto 0);
data2: in std_logic_vector(7 downto 0);
data3: in std_logic_vector(7 downto 0);
```

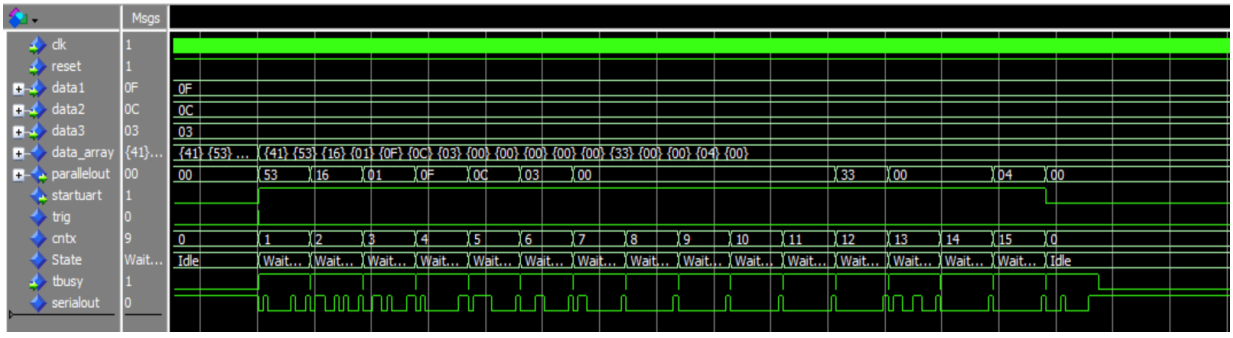
```
parallelout: out std_logic_vector(7 downto 0);
startuart: out std_logic
);
end component;

component BITCONTROLLER
port
(clk: in std_logic;
reset: in std_logic;
startload: in std_logic;
parallelin: in std_logic_vector(7 downto 0);
serialout: out std_logic;
tbusy: out std_logic
);
end component;

signal paralleluart: std_logic_vector(7 downto 0);
signal startuart: std_logic;
signal tbusy2: std_logic;

begin

A1: UARTCONTROLLER port map(clk, reset, tbusy2, data1, data2, data3,
paralleluart, startuart);
A2: BITCONTROLLER port map (clk, reset, startuart, paralleluart,serialout,
tbusy2);

end STRUCT;
```

Figure 25. UART Module construction in VHDL

Fig. 25 shows the UART Module construction. Likewise Bit and Uart Controller, firstly entity declarations are made. As mentioned above, the inputs of UART except tbusy are the same as the Uart Controller and the output is the same as the Bit Controller except tbusy. Afterward, the architecture structure is established. Here, both Uart Controller and Bit Controller are defined as *components*. Components in VHDL declare a virtual design entity interface. These component declarations are the same as their entity declarations. Thus these declarations are directly taken from the Controller Module codes. After defining the components, internal signal assignments are done. These internal signals are paralleluart, startuart, and tbusy. Then, inside the begin statement, two components are created with a port map. In VHDL, port map is the part of the module instantiation in which local signal declaration and connections of the outputs and inputs are done. A1 and A2 are arbitrary names to represent these components.



(a)



(b)

Figure 26. a) The simulation result of the UART module within 25 ms b) The zoomed simulation result of one data packet transmission

Fig. 26 demonstrates the UART simulation results. For this simulation, the UART_TB testbench is used. (Appendix). In part a, for 25 ms range, three times data is read, and transmitted through serial out. For this example, similar to Fig. x17 parallelout transmits the data_array elements consecutively. The serialout transmits the bits with the desired baud rate. It can be easily seen that when trig pulse comes, startuart turns on and bits of these 16 elements of the data_array are sent through serialout respectively. Since TGCU Analog inputs are kept the same, these three data transmission patterns repeat themselves. In part b, one data packet transmission is focused. Even though the first element 0x41 can not be observed in paralleluart, in the serialout output it can be seen as 01000001001 data packet. Likewise, the Bit Controller, the first bit represents start bit '0', after 8 bit comes from 0x41 in a way that lsb is sent first, then even parity bit '0' (since there are two 1, in the 0x41) and lastly stop bit '1'. During this transmission tbusy remains 1, at the end of the first element transmission it becomes 0. Shortly after the second element 0x53 is passed through serialout. This packet consists of 01100101001, the same operation applied in the case of 0x41. This transmission process continues until all the data array elements are sent.

6. TOP PROJECT MODULE

In this section, the Top Project Module is investigated. This module consists of two sub modules which are SPI-ADC Interface and UART. As previously mentioned, Bertay worked on the SPI protocol and I was responsible for the UART protocol. After finishing our modules, we merged these modules together in a Top Project Module. Later this module was used in Libero to upload codes to the CPLD.

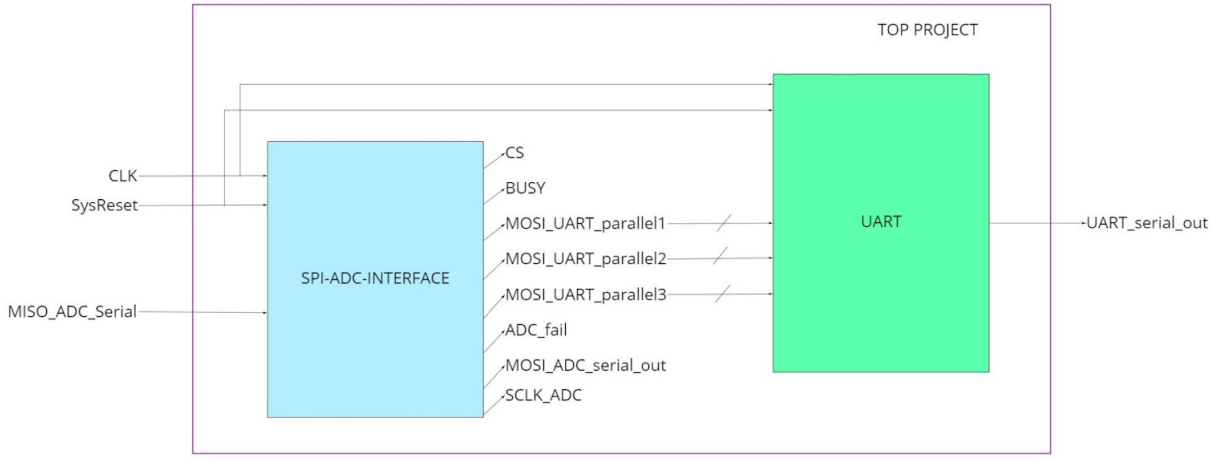


Figure 27. The block diagram of the Top Project Module

Fig. 27 shows the Top Project Module schematic. Here, inputs of the Top Project Module are the same as the SPI-ADC-Interface and the output is the same as the UART Module. SPI (Serial Peripheral Interface) is a synchronous serial communication interface. SPI has one SPI Master and one SPI Slave. In the SPI module written by Bertay, the input MISO_ADC_Serial represents the Master In Slave Out (MISO), which is the input for the SPI Master. This SPI module generates MOSI_UART_parallelx outputs that are inputs for the slave, here the slave is UART. It also checks ADC with ADC_fail output. Moreover, this module has CS (Chip Select for slave), BUSY, SCLK_ADC (serial clk) outputs that are not used in the Top Project Module. MOSI (Master Out Slave In) outputs are 8-bit logic vectors and constitute the inputs of the UART module. In Section 5.3, UART is explained detailly.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.NUMERIC_STD.all;

entity TOP_PROJECT is
port
(clk: in STD_LOGIC; --CLOCK
MISO_ADC_serial_in: in STD_LOGIC; --serial input from ADC
SysReset: in STD_LOGIC;
CS: out STD_LOGIC;
ADC_fail: out STD_LOGIC;
MOSI_ADC_serial_out: out STD_LOGIC;
SCLK_ADC: out STD_LOGIC;
UART_serial_out:out STD_LOGIC; --this will sent to pc
```

```
BUSY: out STD_LOGIC);
end;

architecture STRUCTURE of TOP_PROJECT is

    component SPI_ADC_INTERFACE
    port( CLK: in STD_LOGIC; --CPLD Clock
    SysReset: in STD_LOGIC;
    MISO_ADC_serial: in STD_LOGIC;
    CS: out STD_LOGIC;
    BUSY: out STD_LOGIC;
    MOSI_UART_parallel_1: out STD_LOGIC_VECTOR(7 downto 0);
    MOSI_UART_parallel_2: out STD_LOGIC_VECTOR(7 downto 0);
    MOSI_UART_parallel_3: out STD_LOGIC_VECTOR(7 downto 0);
    ADC_fail: out STD_LOGIC;
    MOSI_ADC_serial_out: out STD_LOGIC;
    SCLK_ADC: out STD_LOGIC);
end component;

    component UART
    port(clk: in std_logic;
    reset: in std_logic;
    data1: in std_logic_vector(7 downto 0);
    data2: in std_logic_vector(7 downto 0);
    data3: in std_logic_vector(7 downto 0);
    serialout: out std_logic
    );
end component;

signal UART_parallel_A, UART_parallel_B, UART_parallel_C:
STD_LOGIC_VECTOR(7 downto 0);

begin

ADC_component: SPI_ADC_INTERFACE port map ( clk, SysReset,
MISO_ADC_serial_in, CS, BUSY,UART_parallel_A, UART_parallel_B,
UART_parallel_C ,ADC_fail, MOSI_ADC_serial_out, SCLK_ADC);

UART_component: UART port map(clk, SysReset, UART_parallel_A,
UART_parallel_B, UART_parallel_C, UART_serial_out);

end STRUCTURE;
```

Figure 28. Top Project Module construction in VHDL

Fig. 28 demonstrates the Top Project Module construction. Likewise in the UART module case, initially entity declarations are made. As mentioned previously, inputs of the Top Project Module are the same as SPI Module, being CLK, SysReset, MISO_ADC_Serial. All of them are serial 1-bit logic signals. Here, including UART_serial_out other outputs of the SPI_ADC_Interface are determined as outputs. Inside the architecture, both SPI and UART modules are given as components with their full input and output description. After the component declarations, ADC_component and UART_component are constructed with a port map statement. While constructing these components, UART inputs coming from the SPI are defined as separate signals being UART_parallel_x. These signals are 8-bit logic

vectors. It should also be noted that as making the port map declaration all inputs and outputs should be arranged in the order of component declaration. We were in the simulation phase of the JoyStick Controller at the end of the internship. At last, we encountered some issues related to the Libero and tried to solve them. Even though the modules worked on the ModelSim simulations, it didn't work as expected in the uploading phase for some synthesis problems that we overlooked in VHDL.

7. CONCLUSION

I completed my internship at ASELSAN under the Avionic System Design Department. I believe that it has been a beneficial experience for me for several reasons. Firstly, I improved my technical and soft skills such as time management, teamwork, and networking. Throughout this period, I have learned VHDL and implemented UART Protocol using this programming language. Moreover, I have had a chance to work with other interns who are students at METU EEE as well. We worked on the Joystick Controller Unit as a team. The task distribution was done by our responsible engineer Samet Karakaş. Every team member took their responsibilities seriously and completed the tasks according to the distribution. Afterwards, each component designed by each member was combined. This term has passed fruitful for all members. Since I have completed my first internship online and by myself, this summer practice has been productive in terms of teamwork and physical implementation. Likewise our team compatibility, in ASELSAN all the engineers pay significant attention to their duties, they can manage their time and work as a team professionally. In addition, during the internship, our responsible engineer showed us the test center in which the parts of a system are put to the several tests such as dust test, extreme condition tests in terms of pressure, temperature etc. Finally, I recommend this SP location to other students who want to develop their technical and soft skills. I believe, ASELSAN will remain one of the important research and development companies in Turkey in later years.

8. REFERENCES

[1] Peña, E & Legaspi M 2020, 'UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/ Transmitter', *Analog Dialogue*, vol. 54, no. 4, pp. 51-55.

[2] VHDL Online, accessed 01.10.2021, <https://www.vhdl-online.de/courses/system_design/vhdl_language_and_syntax/extended_data_types/enumeration_type>

9. APPENDICES

APPENDIX A: LEDFLIP_TB.vhdl

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ledflip_tb is
end ledflip_tb;


architecture bench of ledflip_tb is

component ledflip
port (clk, reset : in std_logic;
       led : out std_logic);
end component;

signal clk: std_logic;
signal reset: std_logic;
signal led: std_logic;

begin

dut: ledflip port map (clk, reset, led);

clk_process: process
begin
clk <= '0';
wait for 20 NS;
clk <= '1';
wait for 20 NS;
end process;

reset <= '0', '1' after 100 NS;
end bench;
```

APPENDIX B: BITCONTROLLER_TB.vhdl

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity BITCONTROLLER_TB is
end;

architecture BENCH of BITCONTROLLER_TB is

component BITCONTROLLER
port
(clk: in std_logic;
reset: in std_logic;
startload: in std_logic;
parallelin: in std_logic_vector(0 to 7);
serialout: out std_logic;
tbusy: out std_logic
);
end component;

signal clk:  std_logic;
signal reset:  std_logic;
signal startload:  std_logic;
signal parallelin:  std_logic_vector(0 to 7);
signal serialout:  std_logic;
signal tbusy: std_logic;

begin

dut: BITCONTROLLER port map (clk, reset, startload, parallelin, serialout,
tbusy);

clkprocess: process
begin
clk<='0';
wait for 30 NS;
clk<='1';
wait for 30 NS;
end process;

reset <= '0','1' after 20 NS;
startload<= '0', '1' after 20 us, '0' after 40 us, '1' after 450 us, '0'
after 480 us;
parallelin<="00001111","11110001" after 400 us;

end BENCH;
```

APPENDIX C: UART_TB.vhdl

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity UART_TB is
end;

architecture BENCH of UART_TB is

component UART
port
(clk: in std_logic;
reset: in std_logic;
data1: in std_logic_vector(7 downto 0);
data2: in std_logic_vector(7 downto 0);
data3: in std_logic_vector(7 downto 0);
serialout: out std_logic
);
end component;

signal clk, reset: std_logic;
signal data1, data2, data3: std_logic_vector(7 downto 0);
signal serialout: std_logic;

begin

dut: UART port map (clk, reset, data1, data2,data3, serialout);

clkprocess: process
begin
clk<='0';
wait for 30 NS;
clk<='1';
wait for 30 NS;
end process;

reset <= '0','1' after 20 NS;
data1<="00001111";
data2<="00001100";
data3<="00000011";

end BENCH;
```