THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

ELECTENG 304
Computer Systems 2E

Implementation of a Simple Microcontroller

Authors : Thusitha Mabotuwana (9790416)
Jaisheel Singh (9801736)

Department: Computer Systems Engineering.

Date: October 03, 2003.

Signed: _____

_____

We hereby declare that the following work is our own and not done in collaboration with, or with assistance from another person or party

Summary

The main aim of the ELECTENG304 - Computer Systems 2E Assignment 1B was to implement a simple microcontroller based on the SIMPX microprocessor using Verilog HDL. It was required in the assignment that four modules be developed with SIMPX, 256 bytes of RAM, two parallel I/O ports and one Serial I/O port using UART, which were then to be appropriately combined together to form the final microcontroller.

The ISA explained in lectures consists of 62 instructions but 3 more new instructions have been added in the development process to accommodate UART and parallel port reception and transmission. Although the total number of instructions in lecture notes was slightly over 60, only 45 of them have been implemented due to very close similarities between instructions within the same group.

FSMs have been used in the design implementation since different states had to be used for different types of implementations. The final design has been implemented using the FLEX10KE EPF10K200SFC672-1X device and uses up 15% of the available resources.

Glossary of Terms

HDL	Hardware Description Language
ALU	Arithmetic Logical Unit
UART	Universal Asynchronous Receiver Transmitter
IR	Instruction Register
SP	Stack Pointer
PC	Program Counter
RAM	Random Access Memory
I/O	Input / Output
MC1	Microcontroller
CLK	Clock
ISA	Instruction Set Architecture

Table of Contents

1.0 Introduction

The aim of this assignment was to design and implement a simple microcontroller based on the SIMPX microprocessor using Verilog HDL. The microcontroller was to have four main modules, SIMPX, 256 bytes of RAM, two parallel I/O Ports and one Serial I/O port using UART. These four separate modules have successfully been implemented in the final design and connected together to form the microcontroller.
This report briefly describes the breakdown of all the individual tasks involved and how each task was designed and implemented to meet the assignment specifications [1]. It also includes relevant algorithms, finite state machines, simulations and other related design issues considered in the final implementation. The Verilog code written can be found in appendices as referenced.

2.0 Assignment Specifications

The HDL to be used was Verilog and Altera MaxPlus was to be used as the EDA tool. The microcontroller was to have four main modules, the SIMPX Processor, Serial I/O, RAM and Parallel I/O.
The SIMPX processor to be designed was to be able to execute ALU instructions on 16-bit operands, include four 16-bit registers for ALU operations, access memory for read and/or write operations from/to register-memory and stack based operations, address 4K 16-bit word memory, and have one interrupt input. The machine code to test the processor were to be loaded to RAM first.
The Serial I/O was to be implemented as an I/O mapped port. It was suggested that the UART code given in lecture 10 [4] be modified to include an odd parity bit before the stop bit.
Memory was to have 256 bytes and ".mif" files were to be used to initialise the memory to test the microcontroller.
Two 8-bit parallel I/O ports were to be designed and the option for them to be either programmable or one port be input and the other be output was given to us. We were also given the option to make them either memory mapped or I/O mapped.

3.0 Development of the Design

The initial stage of the development process was deciding on a sensible breakdown of all the individual tasks involved and making a rough schedule. This was done in coordination with the group members, Mr. Jaisheel Singh and Mr. Thusitha Mabotuwana of Auckland University (UPIs: jsin065 and tmab001 respectively). Since most of the code for Serial I/O was given, it was decided to finish that module off first. The SIMPX processor was to be done next followed by RAM and parallel I/O. Although it was decided that both the group members work on the same code in order for both members to get a better understanding of the different modules, the work load was split up and the serial and parallel I/Os were to be developed by Mr. Mabotuwana while Mr. Singh was to develop RAM, due to time constraints. However the SIMPX module was developed by both members. All these individual modules have been combined to give the final MC1 implementation (as shown in Appendix I).
The device used to implement the MC1 was the FLEX10KE EPF10K200SFC672-1X.

4.0 Implementation of the Final Design

This section outlines the individual tasks involved along with a brief description of the actual implementation of each task. Relevant simulation screenshots are also included.

4.1 SIMPX Processor

The ISA of this processor consists of the following basic group of instructions:

- Load/store instructions for memory access
- ALU operations
- Data movement operations
- Control Transfer Operations

Majority of the given instructions with at least 6 instructions from each of the above four classes have been successfully implemented in the final design (Please refer to Appendix II for a list of instructions that have been implemented). The control path and datapath for this have been implemented as separate modules which have then been combined together to form the SIMPX processor module as shown below.

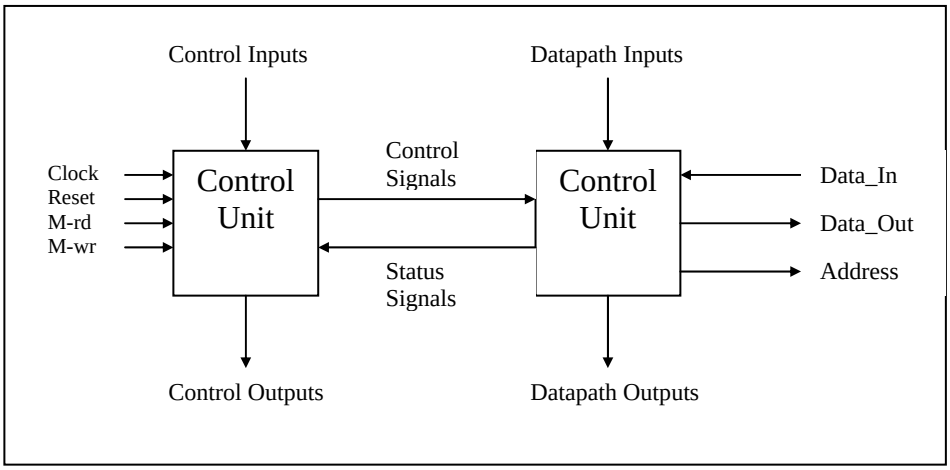
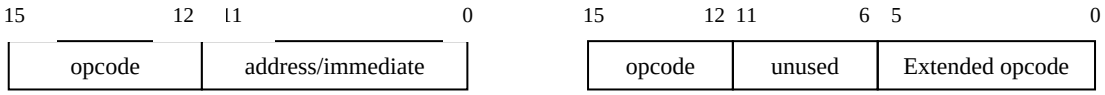


Figure 01: Initial partitioning of SIMPX into Datapath and Control Unit

4.1.1 Control Unit

The main task of the control unit is to decode instructions and set the control signals appropriately. 16 bits are used to encode the different instruction. For simplicity and efficiency, these instructions have been classified into two general groups G1 and G2, which have the following format.



G1 Type G2 Type

All the G2 type instructions that belong to the same instruction class have the same opcode while all the G1 type instructions have their own unique opcode. However all the G2 type instructions have got their own unique extended opcode thus giving each instruction its own identity. The control unit has four states, INIT, FETCH, DECODE and EXEC.

The INIT state is the initialisation state and all the control signals are initialised here. It also sets the next state to be FETCH.
The instruction is fetched from memory in the FETCH state so that it can be decoded in the next state. Memory is read using PC as the index and *register IR* loaded with the instruction read. PC is also incremented and the next state set to DECODE.
This DECODE state checks the opcode and extended_opcode fields of the instruction and sets the appropriate control signals. This state is responsible for setting all the required control signals used by the datapath.
The actual instruction execution happens in the EXEC state which sets the next state to INIT after execution.

4.1.2 Datapath

The datapath checks the control signals generated by the control unit and executes the instruction. It outputs addressbus to memory and databus with any loaded values from registers.

4.1.2.1 Implementation of General, Special and Instruction Registers and DataBus and AddressBus

Four 16-bit general purpose registers, three 12-bit special purpose registers and one 16-bit instruction register have been implemented along with 16-bit databus and a 12-bit address bus. The special purpose registers are the *PC, SP* and the *flag* register.
The values loaded to databus can be from registers A,B,C,D,IR,PC,SP or flag or from the ALU, hence is determined by *dataBus_sel* which is set according to the instruction by the control unit. Likewise, the values loaded to addressbus are determined by *addressBus_sel*, another control signal set by the control unit to select between the lower 12 bits of registers A,B,C,D,IR or registers PC or SP.

4.1.2.2 Implementation of ALU

The type of ALU operation is determined by *ALU_func*, another control signal sent by the control unit which selects between addition, subtraction, AND, OR or XOR operations between register A and another register. This second register is selected using *sel_BCD* signal which selects the input to the ALU to be from registers B,C or D.

4.2 RAM

The RAM module uses *M_rw* signal as its inclock. The outclock is “Unregistered” hence memory can be read when the inclock is high or low. However, RAM can be written to only on a positive clock edge, therefore when data is to be stored into memory the signal *M_wr*, which enables data to be written into memory has to be asserted. The contents of *dataBusOut* will be used as the input data that will be stored at the memory location specified by addressBus.
When a memory read operation is to be performed, the memory location to be read is sent on the addressBus resulting in the data at that memory location being placed on *Data_Bus_In*.

4.3 Serial I/O

A new G2 type of instruction with opcode (0110xxxxxxxxxxx) has been implemented for this with the last 2 bits of the don’t care field used to distinguish between a transmit and a receive. The Serial I/O consists of a transmitter and a receiver and these two

components have been implemented separately, which have then been combined into one main module to give the final UART implementation. The control path in SIMPX will send appropriate control signals to this main module which make UART perform either a transmit or a receive.
This section gives a brief overview of the transmitter and the receiver but the FSMs used and the waveform simulations and Verilog HDL implementation of this can be found in Appendix IV.

4.3.1 UART Transmitter

When the *byteReady* control signal in this module is asserted, the values in 'dataBus' are loaded into 'XMTdataReg'. These values are later copied into the 'XMTshiftReg' when the *trByte* signal is set to high along with the start, odd-parity and the stop bits at the appropriate bit positions. The transmitter then transmits these 11 bits, 1 start bit, 8 data bits, 1 odd-parity bit and the stop bit at each positive clock edge, one bit at a time on the external output pins. A simple flowchart showing this sequence is shown in Figure 02.

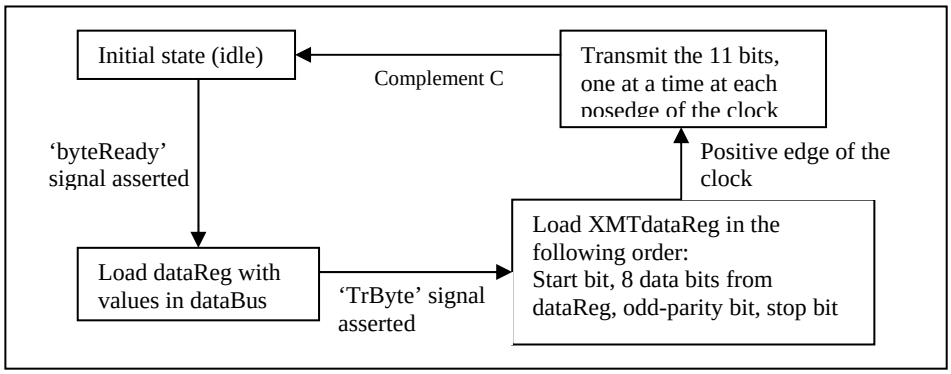


Figure 02: Flowchart showing data flow in UART transmitter

4.3.2 UART Receiver

This receives data from the external input pins. If a start bit is received (ie. a (0) signal) reception begins and the next bits will correspond to 8 data bits, 1 odd-parity bit and 1 stop bit. The start bit is sampled 4 times and all the other bits are sampled 8 times with 8 times faster sampling clock to ensure that the received value was at the centre position of the main clock and not a mere noise signal. Figure 03 shows how these bits are sampled.

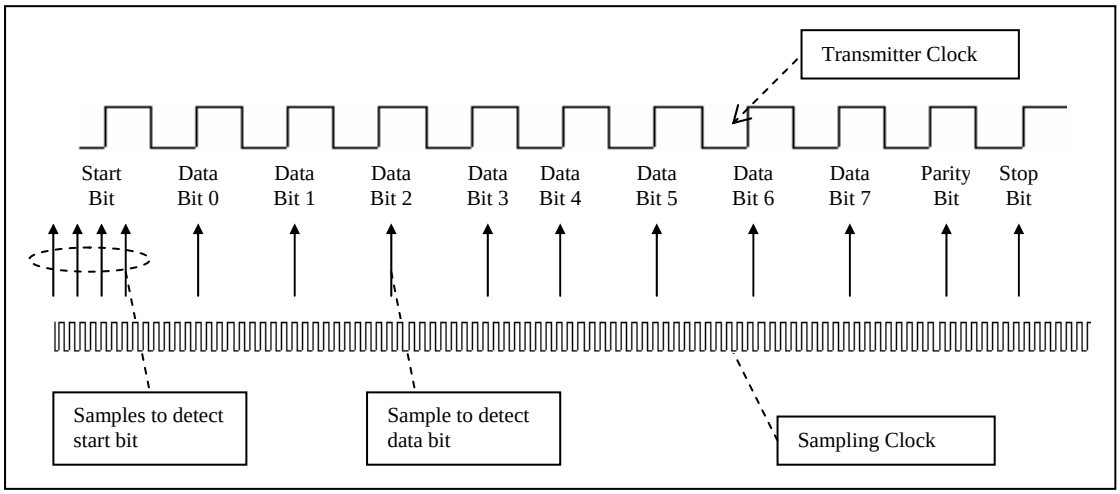


Figure 03: UART Receiver receiving 1 byte of data

The received bits are stored in the 'REC_shftreg' register and after all the 11 bits have been received, the parity bit is checked and if this is correct, the 'REC_datareg' is loaded with 'REC_shftreg' provided that the 'host_ready' signal is set to high. The contents of 'REC_datareg' is then transferred to memory via databus and stored at location 0xFB in RAM. If 'host_ready' was not asserted, the 'hostError' signal is driven high. Also, the 'error1' signal is asserted if the stop bit was missing and/or 'error2' signal asserted if the parity bit was incorrect.

4.4 Parallel I/O

Two new instructions called 'load_par1' and 'load_par2' have been added to the basic instruction set that was provided in lectures [4] to implement the two parallel I/O ports required as memory mapped ports. The two new instructions have been implemented as G2 type with opcodes (0001xxxxxxxxxxx) and (0101xxxxxxxxxxx) respectively. The last 2 bits of the don't care terms are used to detect whether the operation is a transmit or a receive. (00) is used for a transmit and (01) for a receive.

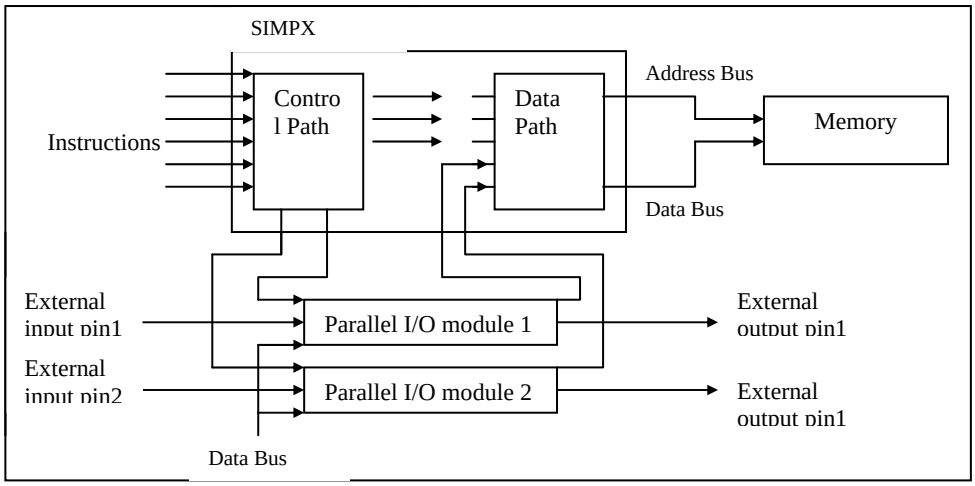


Figure 04: Block diagram showing parallel I/O implementation

As shown above, the control path in SIMPX decodes the two relevant instructions and sets appropriate control signals that control the datapath. These controls are used by the datapath to set the 'Addressbus' value and select the input to 'Databus' to be from parallel I/O module 1 or 2 if the instruction was a receive. A fixed memory location (locations 0xFC and 0xFD) is used to store the values of the two parallel I/O registers, which will also be the value address bus is set to, after the control signals have been decoded by datapath.
For a transmit instruction, the lower 8 bits of databus are used as the input to the parallel module which will then be transmitted via the external output pins.

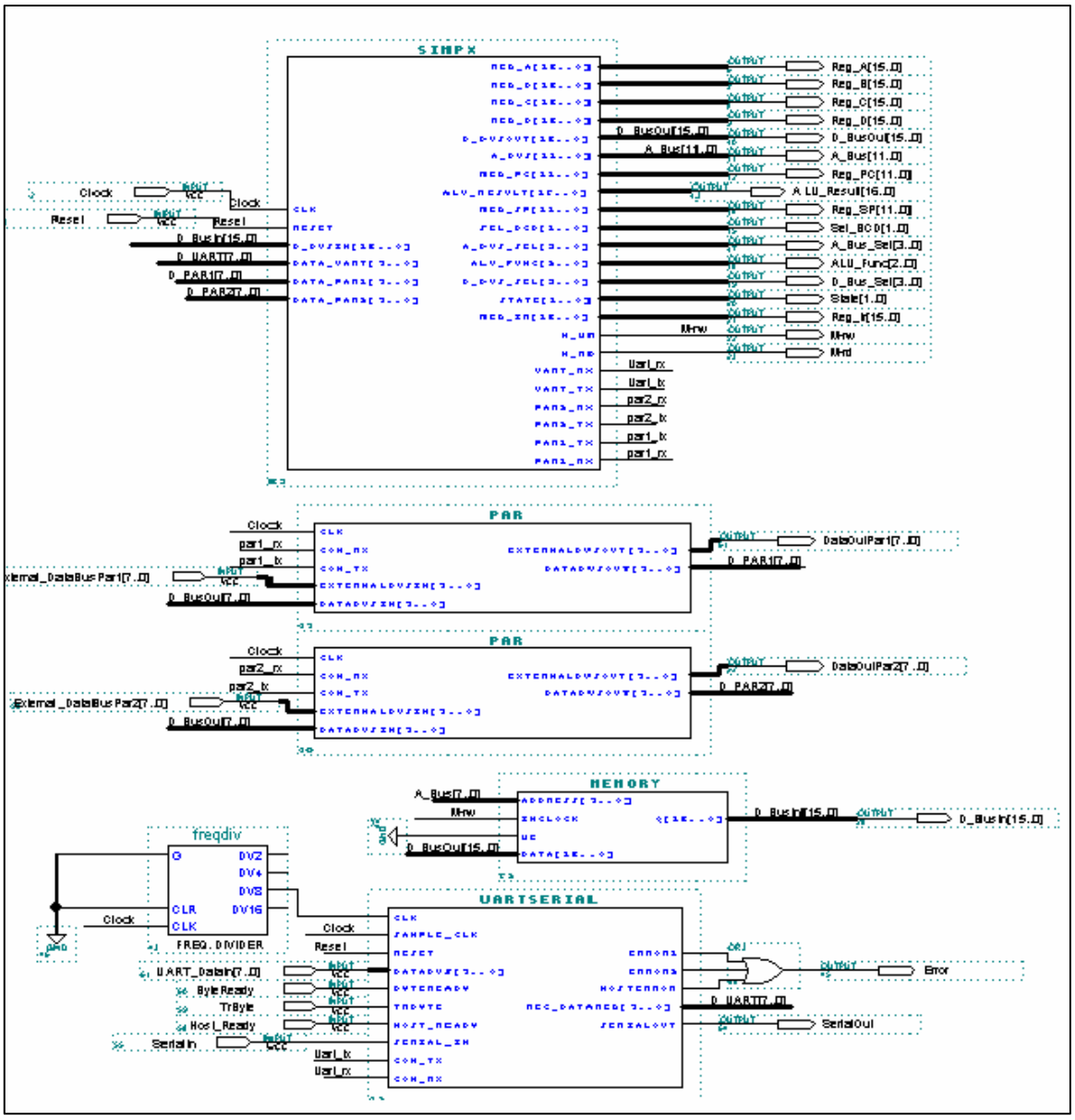
5.0 Conclusions

Based on the assignment specifications and deadlines, the MC1 has been developed as above. All the algorithms have been developed based on efficiency of the software and the feasibility of programming the algorithms in Verilog HDL.
The required microcontroller with SIMPX, 256 bytes RAM, two parallel I/O ports and one Serial I/O port using UART, which has successfully been implemented meets all the assignment specifications and uses up only 15% of the FLEX10KE EPF10K200SFC672-1X device.

References

1. ELECTENG 304 – Computer Systems 2E Assignment 1B handout given out on August 22th 2003.
2. Stephen Brown and Zvonko Vranesic, *Fundamentals of Digital Logic with Verilog Design*, McGraw Hill, 2003,
3. Michael D.C., *Modelling, Synthesis and Rapid Prototyping with the Verilog HDL*, Prentice-Hall, 2001.
4. Lecture handouts provided by Dr. Morteza Biglari-Abhari – *Digital System Synthesis and Simulation using Verilog Hardware Description Language.*

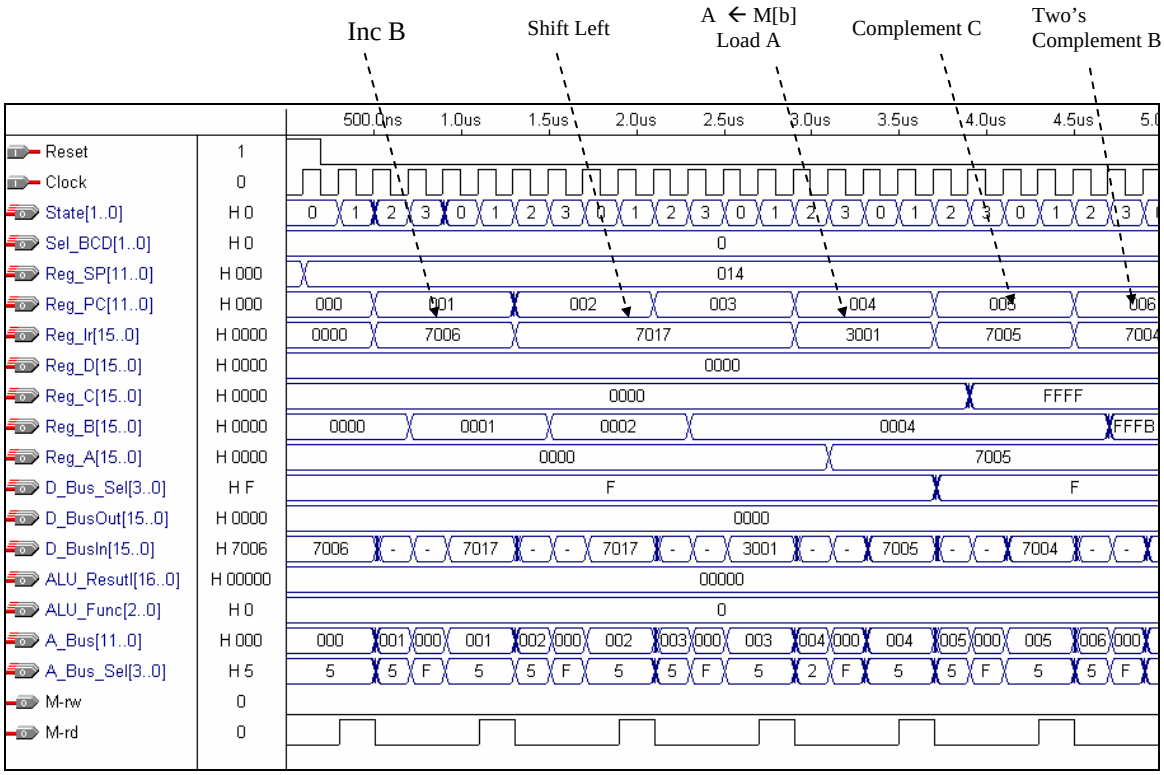
Appendix I: Combination of individual modules to give the final MC1 implementation

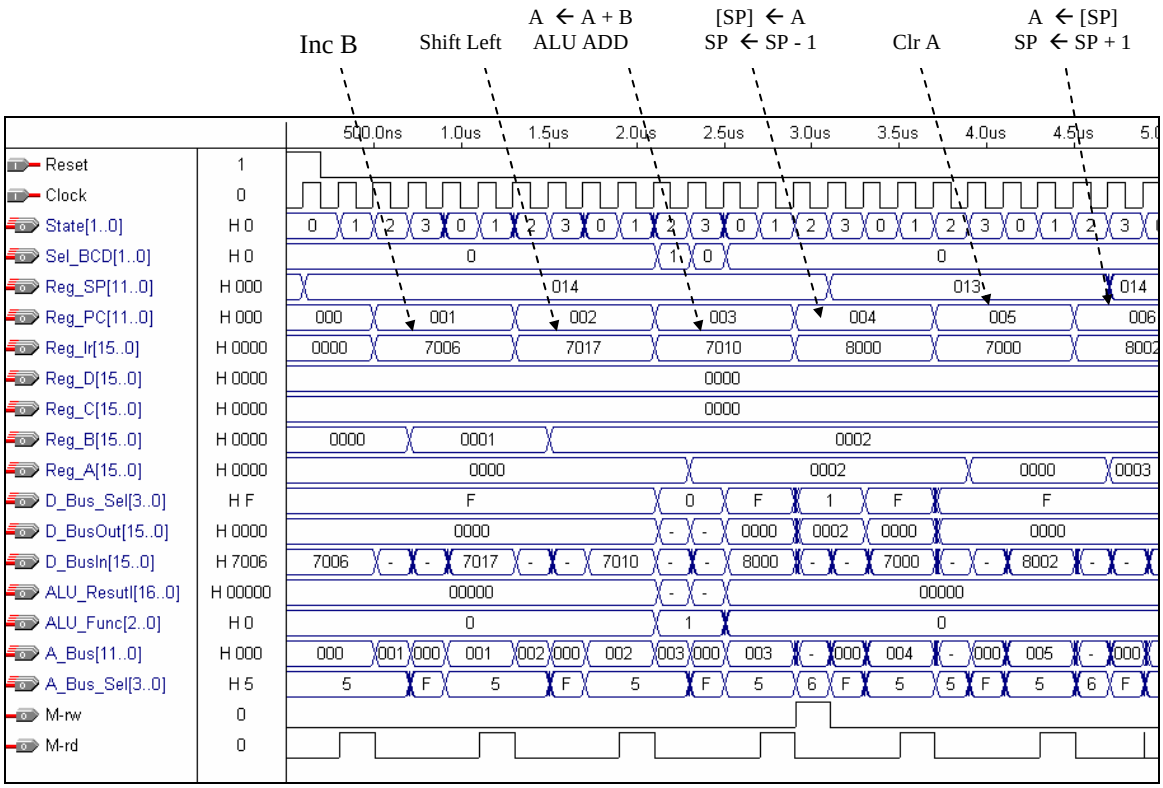


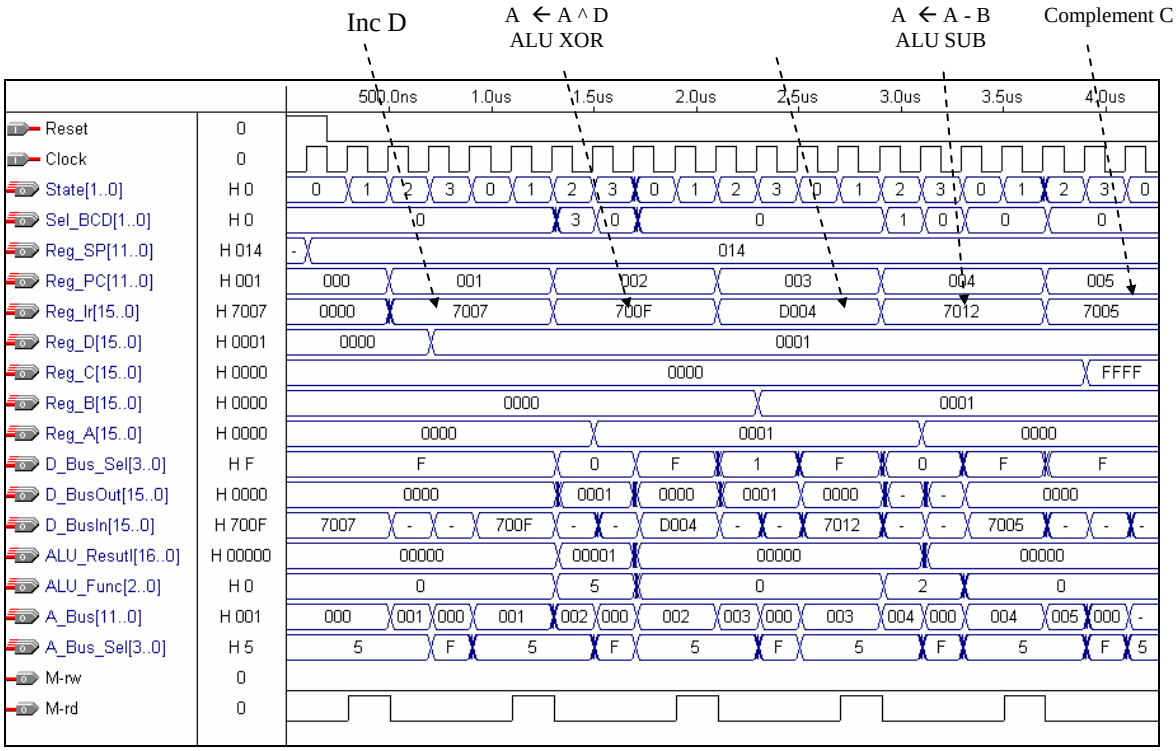
Appendix II: List of instructions that have been implemented

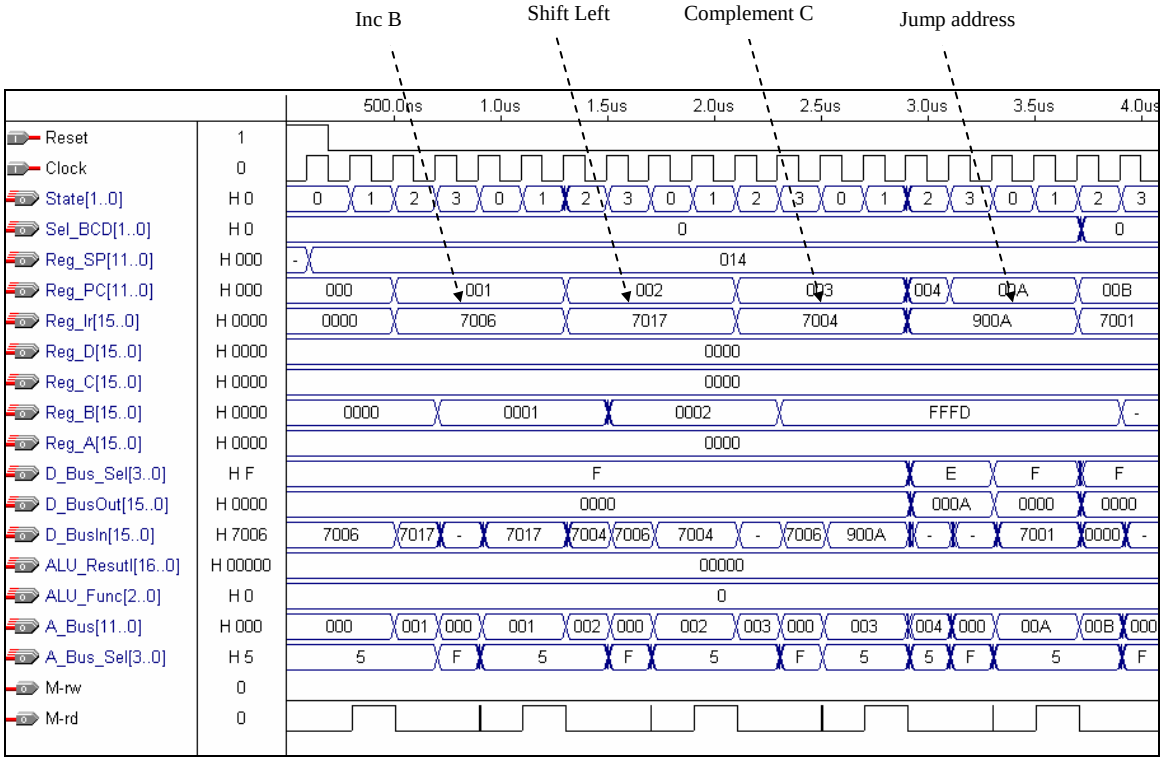
1. JMP (immed)
2. LDA (immed)
3. LDA
4. LDB
5. LDC
6. LDD
7. STA
8. STB
9. STC
10. STD
11. CLA
12. CLB
13. CLC
14. CLD
15. CMB
16. CMC
17. INCB
18. INCD
19. DECD
20. SLLA
21. SLLB
22. SRLA
23. SRLB
24. SRAA
25. SRAB
26. ANDB
27. ANDD
28. ORB
29. ORD
30. XORB
31. ADDB
32. ADDC
33. SUBB
34. TCB
35. CLCF
36. CLCZ
37. PSHA
38. PSHB
39. PULA
40. PULB
41. MVBC
42. MVAD
43. MVBA
44. MVCB
45. MVDA

Appendix III: Waveform simulations related to SIMPX





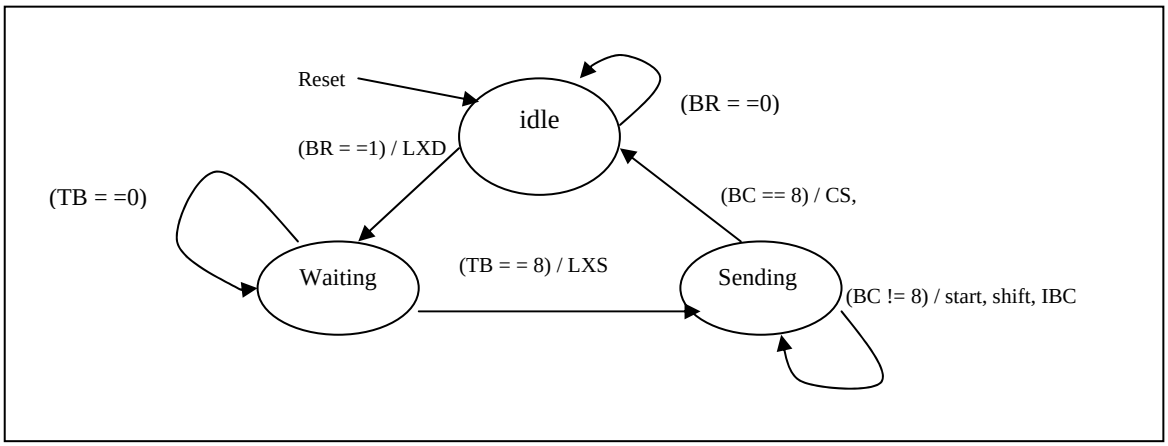

Inc D
A ← A ^ D
ALU XOR
A ← A - B
ALU SUB
Complement C
500.0ns
1.0us
1.5us
2.0us
2.5us
3.0us
3.5us
4.0us
Reset
Clock
State[1..0]
Sel_BCD[1..0]
Reg_SP[11..0]
Reg_PC[11..0]
Reg_Ir[15..0]
Reg_D[15..0]
Reg_C[15..0]
Reg_B[15..0]
Reg_A[15..0]
D_Bus_Sel[3..0]
D_BusOut[15..0]
D_BusIn[15..0]
ALU_Resutl[16..0]
ALU_Func[2..0]
A_Bus[11..0]
A_Bus_Sel[3..0]
M-rw
M-rd


Inc B
Shift Left
Complement C
Jump address
500.0ns
1.0us
1.5us
2.0us
2.5us
3.0us
3.5us
4.0us
Reset
Clock
State[1..0]
Sel_BCD[1..0]
Reg_SP[11..0]
Reg_PC[11..0]
Reg_Ir[15..0]
Reg_D[15..0]
Reg_C[15..0]
Reg_B[15..0]
Reg_A[15..0]
D_Bus_Sel[3..0]
D_BusOut[15..0]
D_BusIn[15..0]
ALU_Resutl[16..0]
ALU_Func[2..0]
A_Bus[11..0]
A_Bus_Sel[3..0]
M-rw
M-rd

Appendix IV: Implementation of UART Transmitter and Receiver

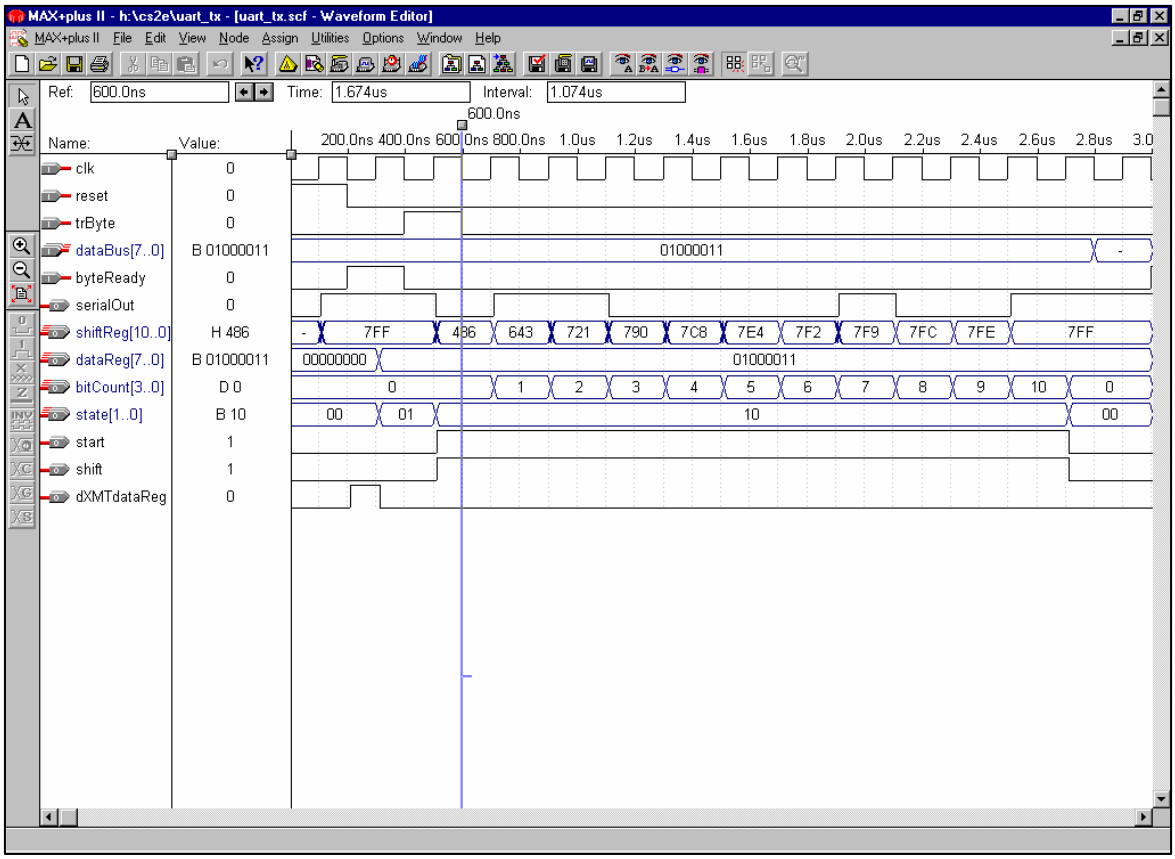
A(IV).1 UART Transmitter

Shown below is FSM used for in this module.



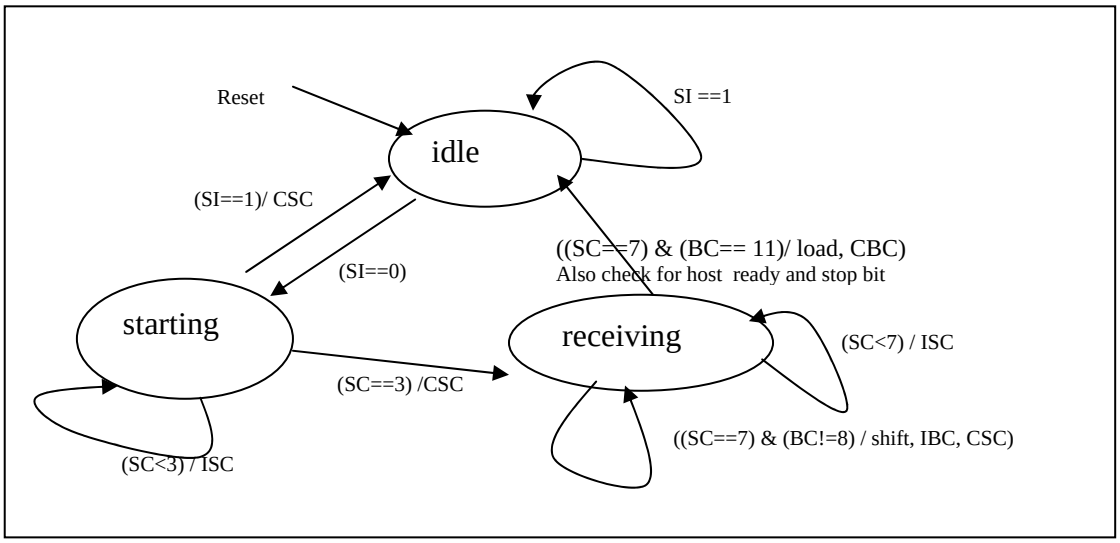
Note: (BR (Byte_Ready), BC (bit_counter), TB (Tr_Byte), CS (start =0), CBC (clear bit_counter), IBC (increment bit_counter), LXD (load XMT_datareg), LXS (load XMT_shftreg)

This waveform shows all the inputs, outputs and control signals related to the UART transmitter.



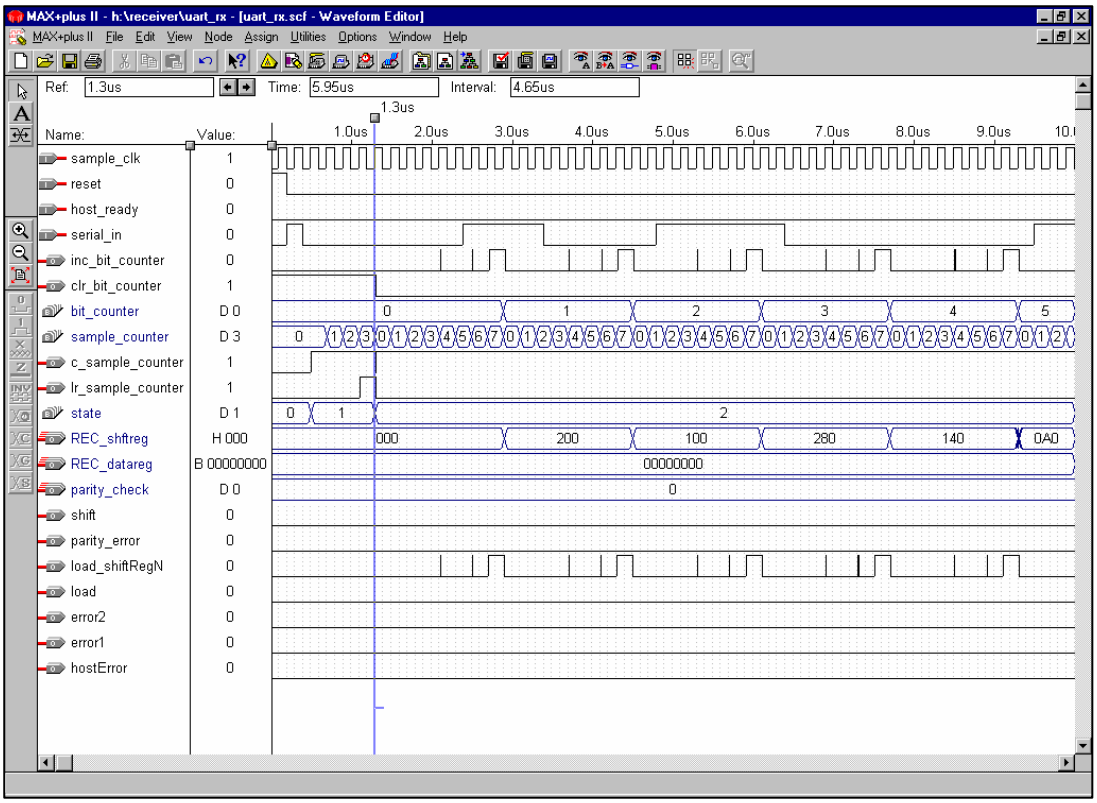
A(IV).2 UART Receiver

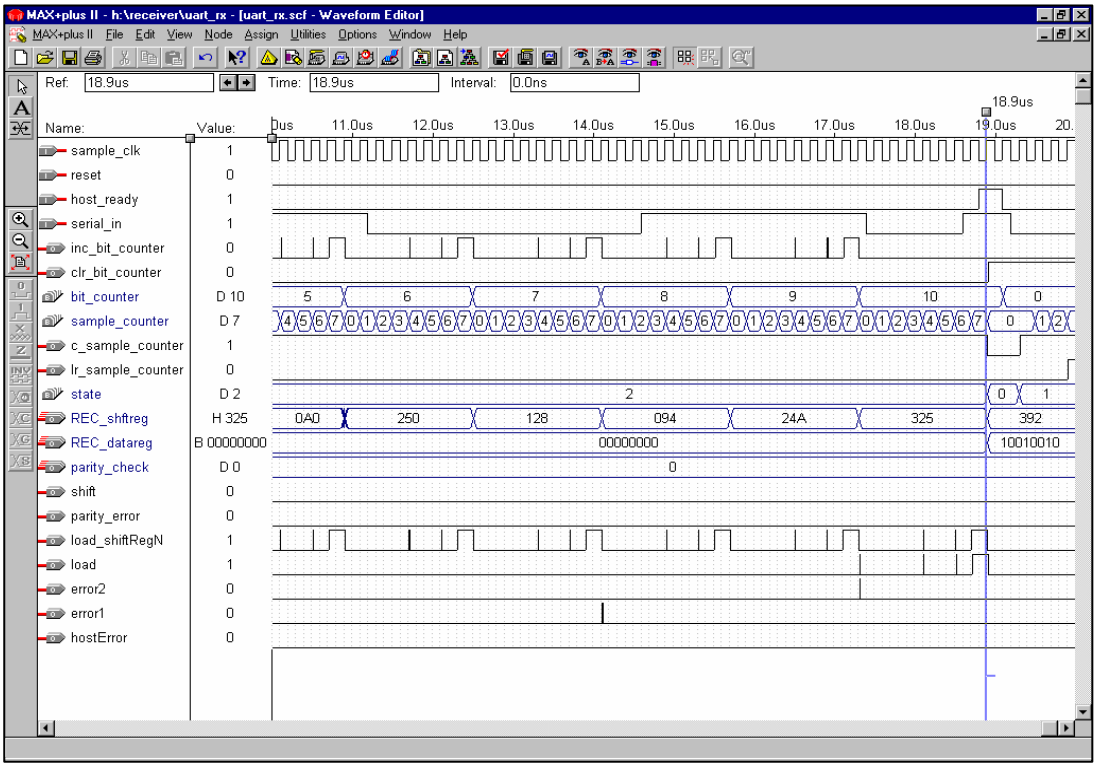
Shown below is FSM used the UART receiver



Note: SC (sample_counter), BC (bit_counter), SI (serial_in), CSC (activate clr_sample_counter), CBC (activate clr_bit_counter), IBC (activate inc_bit_counter), ISC (activate inc_sample_counter)

The waveforms below show all the related input, output and control signals for UART receiver. The first figure shows the first five bits of transmission and the one below shows the rest of the 11 bits.




MAX+plus II - h:\receiver\uart_rx - [uart_rx.scf - Waveform Editor]
MAX+plus II File Edit View Node Assign Utilities Options Window Help
Ref: 18.9us
Time: 18.9us
Interval: 0.0ns
Name:
Value:
sample_clk 1
reset 0
host_ready 1
serial_in 1
inc_bit_counter 0
clr_bit_counter 0
bit_counter D 10
sample_counter D 7
c_sample_counter 1
lr_sample_counter 0
state D 2
REC_shftreg H 325
REC_datareg B 00000000
parity_check D 0
shift 0
parity_error 0
load_shiftRegN 1
load 1
error2 0
error1 0
hostError 0

Appendix V: Verilog code written to implement SIMPX

```
/*ELECTENG304 COMPSYS2E Assignment 1b
Assignment: Implementation of a simple microcontroller
Date: 03.10.03
Developers: Jaisheel Singh - 9801736
                                        Thusitha Mabotuwana - 9790416

*/
//SIMPX

module simpx(clk,reset,D_BusIn,reg_A,reg_B,reg_C,reg_D,D_BusOut,A_Bus, reg_PC,ALU_result,
reg_SP,sel_BCD,A_Bus_sel,ALU_func, D_Bus_sel,state,reg_IR, M_wr,M_rd, uart_rx,uart_tx,par2_rx,par2_tx,par1_tx,par1_rx,
Data_UART, Data_PAR1, Data_PAR2);

        // all the outputs and the inputs required for the simpx
        input clk,reset;
        input [15:0] D_BusIn;
        input [7:0] Data_UART, Data_PAR1, Data_PAR2;

        output [15:0] reg_A,reg_B,reg_C,reg_D;
        output [11:0] A_Bus, reg_PC, reg_SP;
        output [15:0] D_BusOut;
        output [16:0] ALU_result;
        output uart_rx,uart_tx,par2_rx,par2_tx,par1_tx,par1_rx;
        output [15:0] reg_IR;
        output M_wr,M_rd;
        output [2:0] ALU_func;
        output [3:0] A_Bus_sel, D_Bus_sel;
        output [1:0] state,sel_BCD ;

        // wires required to connect the control to the datapath
        wire clr_A,sll_A,srl_A, sra_A,load_AL,load_AH,load_A,clr_B,inc_B,dec_B,sll_B,inc_SP, dec_SP,
        srl_B,sra_B,cmb_B,load_B,TCB_B,clr_C,cmc_C,load_C, clr_D,inc_D,dec_D,load_D, load_PC,CLCF,CLCZ,
        DBus_load_D, DBus_load_C, DBus_load_B, DBus_load_A;

        wire [15:0] reg_IR;
        wire [2:0] ALU_func;
        wire [1:0] state,sel_BCD ;
        wire [3:0] A_Bus_sel, D_Bus_sel;

        // calling simpxcontrolpath using Naming Association
        simpxcontrolPath control (.clk(clk),.reset(reset),.D_BusIn(D_BusIn),.clr_A(clr_A),.sll_A(sll_A),.srl_A(srl_A),.sra_A(sra_A),
        .load_AL(load_AL),.load_AH(load_AH),.load_A(load_A),.clr_B(clr_B),.inc_B(inc_B),.dec_B(dec_B),.sll_B(sll_B),
        .srl_B(srl_B),.sra_B(sra_B),.cmb_B(cmb_B),.load_B(load_B),.TCB_B(TCB_B),.clr_C(clr_C),.cmc_C(cmc_C), .inc_PC(inc_PC),
        .load_C(load_C),.clr_D(clr_D),.inc_D(inc_D),.dec_D(dec_D),.load_D(load_D),.load_PC(load_PC),.CLCF(CLCF),
        .CLCZ(CLCZ),.D_Bus_sel(D_Bus_sel),.A_Bus_sel(A_Bus_sel),.ALU_func(ALU_func),.sel_BCD(sel_BCD), .state(state),.reg_IR(reg_IR),
        .M_wr(M_wr),.M_rd(M_rd) ,.inc_SP(inc_SP), .dec_SP(dec_SP) ,
        .uart_rx(uart_rx), .uart_tx(uart_tx) , .par2_rx(par2_rx) ,.par2_tx(par2_tx) , .par1_tx(par1_tx) , .par1_rx(par1_rx),
        .DBus_load_D(DBus_load_D), .DBus_load_C(DBus_load_C), .DBus_load_B(DBus_load_B), .DBus_load_A(DBus_load_A));

        // calling simpxdatapath using Naming Association
        simpxdataPath data (.clk(clk),.reset(reset),.D_BusIn(D_BusIn),.clr_A(clr_A),.sll_A(sll_A),.srl_A(srl_A),.sra_A(sra_A),
        .load_AL(load_AL),.load_AH(load_AH),.load_A(load_A),.clr_B(clr_B),.inc_B(inc_B),.dec_B(dec_B),.sll_B(sll_B),
        .srl_B(srl_B),.sra_B(sra_B),.cmb_B(cmb_B),.load_B(load_B),.TCB_B(TCB_B),.clr_C(clr_C),.cmc_C(cmc_C),
        .load_C(load_C),.clr_D(clr_D),.inc_D(inc_D),.dec_D(dec_D),.load_D(load_D),.load_PC(load_PC),.CLCF(CLCF),
        .CLCZ(CLCZ),.D_Bus_sel(D_Bus_sel),.A_Bus_sel(A_Bus_sel),.ALU_func(ALU_func),.sel_BCD(sel_BCD), .inc_PC(inc_PC),
        .reg_A(reg_A),.reg_B(reg_B),.reg_C(reg_C),.reg_D(reg_D),.A_Bus(A_Bus), .reg_PC(reg_PC),
        .reg_SP(reg_SP),.D_BusOut(D_BusOut),.ALU_result(ALU_result), .reg_IR(reg_IR),.inc_SP(inc_SP), .dec_SP(dec_SP),
        .DBus_load_D(DBus_load_D), .DBus_load_C(DBus_load_C), .DBus_load_B(DBus_load_B), .DBus_load_A(DBus_load_A),
        .Data_UART(Data_UART), .Data_PAR2(Data_PAR2), .Data_PAR1(Data_PAR1));

endmodule
```

SIMPX controlpath

```
module simpxcontrolPath(clk,reset,D_BusIn,clr_A,sll_A,srl_A,sra_A,load_AL,load_AH,load_A,clr_B,inc_B,dec_B,
sll_B,srl_B,sra_B,cmb_B,load_B,TCB_B,clr_C,cmc_C,load_C,clr_D,inc_D,dec_D,load_D,load_PC,CLCF,CLCZ, inc_SP, dec_SP, DBus_load_D,
DBus_load_C, DBus_load_B, DBus_load_A, M_wr,M_rd, A_Bus_sel,A_Bus_ sel,ALU_func,sel_BCD,inc_PC,state,reg_IR, uart_rx, uart_tx, par2_rx,
par2_tx, par1_tx, par1_rx);

        input clk,reset;
        input [15:0] D_BusIn;
        output clr_A,sll_A,srl_A,sra_A,load_AL,load_AH,load_A;
        output clr_B,inc_B,dec_B,sll_B,srl_B,sra_B,cmb_B,load_B,TCB_B;
        output clr_C,cmc_C,load_C;
        output clr_D,inc_D,dec_D,load_D;
        output DBus_load_D, DBus_load_C, DBus_load_B, DBus_load_A;
        output uart_rx, uart_tx, par2_rx, par2_tx, par1_tx, par1_rx;
        output load_PC,inc_PC;
        output CLCF,CLCZ;
        output [3:0] D_Bus_sel,A_Bus_sel;
        output [2:0] ALU_func;
        output [1:0] sel_BCD;
        output M_wr,M_rd;
        output [15:0] reg_IR;
        output inc_SP, dec_SP;
        output[1:0] state;
        reg inc_SP, dec_SP;
        reg clr_A,sll_A,srl_A,sra_A,load_AL,load_AH,load_A;
        reg clr_B,inc_B,dec_B,sll_B,srl_B,sra_B,cmb_B,load_B,TCB_B;
        reg clr_C,cmc_C,load_C;
        reg clr_D,inc_D,dec_D,load_D;
        reg DBus_load_D, DBus_load_C, DBus_load_B, DBus_load_A;
        reg uart_rx, uart_tx, par2_rx, par2_tx, par1_tx, par1_rx;
        reg load_PC,inc_PC,load_IR;
        reg CLCF,CLCZ;
        reg [3:0] D_Bus_sel,A_Bus_sel;
        reg [2:0] ALU_func;
        reg [1:0] sel_BCD;
        reg M_wr,M_rd,IACK;
        reg [15:0] reg_IR;

        //register selection and datapath inputs
        parameter ALU_result_sel= 4'b0000, reg_A_sel= 4'b0001,reg_B_sel = 4'b0010, reg_C_sel=
        4'b0011,reg_D_sel = 4'b0100,
         reg_PC_sel = 4'b0101, reg_SP_sel = 4'b0110, reg_IR_sel = 4'b1110,          No_Change = 4'b1111;
        // for UART and parallel
        parameter UART_sel = 4'b1000, PAR1_sel = 4'b1001, PAR2_sel = 4'b1010;

        //instruction opcodes for G1
        parameter  STA_OP=4'b0100, LDA_OP = 4'b0010,
        LDAL_OP = 4'b1110,LDAH_OP = 4'b1111,JMP_OP = 4'b1001,JPC_OP = 4'b1010,JPZ_OP = 4'b1011,JSR_OP
        = 4'b1100;

        //instruction opcodes for G2
        parameter G2_ALU=4'b0111,G2_CTI=4'b1000,G2_DTI=4'b1101, G2_LDST = 4'b0011;
        parameter G2_UART = 4'b0110, G2_PAR1 = 4'b0001, G2_PAR2 = 4'b010;

        //ALU functions
        parameter ADD_func=3'b001,SUB_func=3'b010,AND_func=3'b011,OR_func=3'b100,XOR_func=3'b101;

// ---------------------- ALL the OpCodes for G1-----------------------------------------------------
        parameter G1_JMP = 4'b1001, G1_LOAD = 4'b0010;

// ---------------------- ALL the Extended OpCodes for G2 -----------------------------------------------
        //Extended opcodes for G2 : for LSI
        parameter LDA_EOC = 6'b0000xx,LDB_EOC=6'b0001xx,LDC_EOC=6'b0010xx,LDD_EOC=6'b0011xx,
        STA_EOC=6'b1000xx,STB_EOC=6'b1001xx,STC_EOC=6'b1010xx,STD_EOC=6'b1011xx;

        // Extended OpCode for G2 on Slide 7 : ALU
        parameter CLA_EOC = 6'b000000, CLB_EOC = 6'b000001, CLC_EOC = 6'b000010, CLD_EOC = 6'b000011,
        CMB_EOC = 6'b000100,
        CMC_EOC = 6'b000101, INCB_EOC = 6'b000110, INCD_EOC = 6'b000111, DECB_EOC = 6'b001000,
        DECD_EOC = 6'b001001;

        // Extended OpCode for G2 on Slide 8 : ALU
        parameter ANDB_EOC = 6'b001010, ANDD_EOC = 6'b001011, ORB_EOC = 6'b001100,
        ORD_EOC = 6'b001101, XORB_EOC = 6'b001110,
        XORD_EOC = 6'b001111, ADDB_EOC = 6'b010000, ADDC_EOC = 6'b010001, SUBB_EOC = 6'b010010, TCB_EOC = 6'b010011,
        CLCF_EOC = 6'b010100;

        // Extended OpCode for G2 on Slide 9 : ALU
        parameter CLCZ_EOC = 6'b010101, SLLA_EOC = 6'b010110, SLLB_EOC = 6'b010111, S
        RLA_EOC = 6'b011000, SRLB_EOC = 6'b011001,
        SRAA_EOC = 6'b011010, SRAB_EOC = 6'b011011;

        // Extended OpCode for G2 on Slide 10  : CTI
        parameter PSHA_EOC = 6'b000000, PSHB_EOC = 6'b000001, PULA_EOC = 6'b000010,
        PULB_EOC = 6'b000011, PSHF_EOC = 6'b000100,
        PULF_EOC = 6'b000101, SZ_EOC = 6'b000110, SC_EOC = 6'b000111, ION_EOC = 6'b001000,
        IOF_EOC = 6'b001001, RET_EOC = 6'b001010;
```

```
// Extended OpCode for G2 on Slide 12 : DTI
parameter NOP_EOC = 6'b000000, MVAB_EOC = 6'b000001, MVBC_EOC = 6'b000010,
MVAD_EOC = 6'b000011, MVBA_EOC = 6'b000100,
MVCB_EOC = 6'b000101,MVDA_EOC = 6'b000110;

// Extened OpCode for G2 : UART and Parellel
parameter TRANS = 2'b00, RECV = 2'b01;



// state encoding..
parameter INIT = 3'b00, FETCH = 3'b01, DECODE = 3'b10, EXEC = 3'b11;
reg[1:0] state, next_state;

//-------------------------- Control path----------------------------------------

always @ (posedge clk) begin
        if (reset == 1'b1)
                reg_IR = {16{1'b0}};
        else if (load_IR == 1'b1)
                reg_IR = D_BusIn;    // loads reg_IR with the contents of DataBusIn
        end

always@(posedge clk)  // changes states on each posedge clock
        if(reset == 1'b1)  // if reset is asserted then state will be INIT
                state <= INIT;
        else
                state <= next_state;

always @ (state)  // control path begins which is senstive to state.
begin
case(state)
        INIT:begin
                // clear all signals
        DBus_load_D = 1'b0; DBus_load_B = 1'b0; DBus_load_C = 1'b0; DBus_load_A = 1'b0;
        clr_A = 1'b0;sll_A = 1'b0;srl_A = 1'b0;sra_A = 1'b0;load_AL = 1'b0;
        load_AH = 1'b0;load_A = 1'b0;clr_B = 1'b0;inc_B = 1'b0;dec_B = 1'b0;
        sll_B = 1'b0;srl_B = 1'b0;sra_B = 1'b0;cmb_B = 1'b0;load_B = 1'b0;
        clr_C = 1'b0;cmc_C = 1'b0;load_C = 1'b0;
        clr_D = 1'b0; inc_D = 1'b0;dec_D = 1'b0;load_D = 1'b0;
        load_PC = 1'b0;load_IR = 1'b0; inc_PC = 1'b0;
        M_wr = 1'b0;M_rd = 1'b0;IACK = 1'b0;inc_SP = 1'b0; dec_SP = 1'b0;
        uart_tx = 1'b0; uart_rx = 1'b0;          par1_rx = 1'b0;par1_tx = 1'b0;
        par2_rx = 1'b0;par2_tx = 1'b0;
        D_Bus_sel <= No_Change; A_Bus_sel <= reg_PC_sel; ALU_func <= 3'b000; sel_BCD = 2'b00;
        TCB_B = 1'b0; CLCF = 1'b0;          CLCZ = 1'b0;
        next_state <= FETCH;
        end

        FETCH: begin
                M_rd = 1'b1; //Memery read
                A_Bus_sel <= reg_PC_sel; // Address <- PC--
                load_IR = 1'b1;  // loads reg_IR wit new instruction
                inc_PC = 1'b1;   // increments PC in datapath
                D_Bus_sel <= No_Change; sel_BCD = 2'b00; ALU_func <= 3'b000;
                next_state <= DECODE;
        end

        DECODE:begin
                //deactive output signales from pervious state and resets all signals to //remove conflicts between instructions
        load_IR = 1'b0; inc_PC = 1'b0;          M_rd = 1'b0;
        DBus_load_D = 1'b0; DBus_load_B = 1'b0; DBus_load_C = 1'b0; DBus_load_A = 1'b0;
        clr_A = 1'b0;sll_A = 1'b0;srl_A = 1'b0;sra_A = 1'b0;load_AL = 1'b0;
        load_AH = 1'b0;load_A = 1'b0; clr_B = 1'b0;inc_B = 1'b0;dec_B = 1'b0;
        sll_B = 1'b0;srl_B = 1'b0;sra_B = 1'b0;cmb_B = 1'b0;load_B = 1'b0;
        clr_C = 1'b0;cmc_C = 1'b0;load_C = 1'b0;
        clr_D = 1'b0; inc_D = 1'b0;dec_D = 1'b0;load_D = 1'b0;
        load_PC = 1'b0;load_IR = 1'b0; inc_PC = 1'b0; M_wr = 1'b0;IACK = 1'b0;
        sel_BCD = 2'b00;          inc_SP = 1'b0; dec_SP = 1'b0;TCB_B = 1'b0; CLCF = 1'b0;
        CLCZ = 1'b0;uart_tx = 1'b0; uart_rx = 1'b0;par1_rx = 1'b0;par1_tx = 1'b0;
        par2_rx = 1'b0;par2_tx = 1'b0;
        A_Bus_sel <= reg_PC_sel; D_Bus_sel <= No_Change; ALU_func <= 3'b000; sel_BCD = 2'b00;

                // begining to decode the instruction
                case(reg_IR[15:12])
                // checkin for GI type opcode
                        G1_JMP:
                                begin
                                        D_Bus_sel <= reg_IR_sel;
                                        load_PC = 1'b1;
                                end
                        G1_LOAD:
                                begin
                                        D_Bus_sel <= reg_IR_sel;
                                        load_A = 1'b1;
                                end
```

```
// checkin for G2 type opcode
        G2_LDST:
                begin
                case(reg_IR[5:2])
                        LDA_EOC: begin
                                load_A = 1'b1;
                                case(reg_IR[1:0]) // to check with
                                //register should be loaded as the address
                                        00: A_Bus_sel <= reg_A_sel;
                                        01: A_Bus_sel <= reg_B_sel;
                                        10: A_Bus_sel <= reg_C_sel;
                                        11: A_Bus_sel <= reg_D_sel;
                                        default:  A_Bus_sel <= No_Change;
                                endcase
                                end
                        LDB_EOC: begin
                                load_B = 1'b1;
                                case(reg_IR[1:0]) // to check with
                                //register should be loaded as the address
                                        00: A_Bus_sel <= reg_A_sel;
                                        01: A_Bus_sel <= reg_B_sel;
                                        10: A_Bus_sel <= reg_C_sel;
                                        11: A_Bus_sel <= reg_D_sel;
                                        default: A_Bus_sel <= No_Change;
                                endcase
                                end
                        LDC_EOC: begin
                                load_C = 1'b1;
                                case(reg_IR[1:0]) // to check with
                                //register should be loaded as the address
                                        00: A_Bus_sel <= reg_A_sel;
                                        01: A_Bus_sel <= reg_B_sel;
                                        10: A_Bus_sel <= reg_C_sel;
                                        11: A_Bus_sel <= reg_D_sel;
                                endcase
                                end
                        LDD_EOC: begin
                        load_D = 1'b1; // enabled so can write to memory
                                case(reg_IR[1:0]) // to check with
                                //register should be loaded as the address
                                        00: A_Bus_sel <= reg_A_sel;
                                        01: A_Bus_sel <= reg_B_sel;
                                        10: A_Bus_sel <= reg_C_sel;
                                        11: A_Bus_sel <= reg_D_sel;
                                endcase
                                end
                        STA_EOC: begin
                                D_Bus_sel <= reg_A_sel;
                        M_wr = 1'b1;// enabled so can write to memory
                                case(reg_IR[1:0]) // to check with register should
                                //be loaded as the address
                                        00: A_Bus_sel <= reg_A_sel;
                                        01: A_Bus_sel <= reg_B_sel;
                                        10: A_Bus_sel <= reg_C_sel;
                                        11: A_Bus_sel <= reg_D_sel;
                                        default:  A_Bus_sel <= No_Change;
                                endcase
                                end
                        STB_EOC:begin
                                D_Bus_sel <= reg_B_sel;
                        M_wr = 1'b1;// enabled so can write to memory
                                case(reg_IR[1:0]) // to check with
                                //register should be loaded as the address
                                        00: A_Bus_sel <= reg_A_sel;
                                        01: A_Bus_sel <= reg_B_sel;
                                        10: A_Bus_sel <= reg_C_sel;
                                        11: A_Bus_sel <= reg_D_sel;
                                        default:  A_Bus_sel <= No_Change;
                                endcase
                                end
                        STC_EOC:begin
                                D_Bus_sel <= reg_C_sel;
                        M_wr = 1'b1; // enabled so can write to memory
                                case(reg_IR[1:0]) // to check with
                                //register should be loaded as the address
                                        00: A_Bus_sel <= reg_A_sel;
                                        01: A_Bus_sel <= reg_B_sel;
                                        10: A_Bus_sel <= reg_C_sel;
                                        11: A_Bus_sel <= reg_D_sel;
                                        default:  A_Bus_sel <= No_Change;
                                endcase
                                end
                        STD_EOC:begin
                                D_Bus_sel <= reg_D_sel;
                        M_wr = 1'b1;  // enabled so can write to memory
                                case(reg_IR[1:0]) // to check with
                                //register should be loaded as the address
```

```
                                        00: A_Bus_sel <= reg_A_sel;
                                        01: A_Bus_sel <= reg_B_sel;
                                        10: A_Bus_sel <= reg_C_sel;
                                        11: A_Bus_sel <= reg_D_sel;
                                        default:  A_Bus_sel <= No_Change;
                                endcase
                                end
                        default         next_state <= state;
        endcase

        end
G2_ALU:
        begin
        case(reg_IR[5:0])
                        CLA_EOC: clr_A = 1'b1;   // sets control signals to
                        //clears the selected register
                        CLB_EOC: clr_B = 1'b1;
                        CLC_EOC: clr_C = 1'b1;
                        CLD_EOC: clr_D = 1'b1;
                        CMB_EOC: cmb_B = 1'b1;  // sets control signals to
                        //complement's B or C
                        CMC_EOC: cmc_C = 1'b1;
                        INCB_EOC: inc_B = 1'b1;  // sets control signals to
                        //increments reg B or C
                        INCD_EOC: inc_D = 1'b1;
                        DECB_EOC: dec_B = 1'b1;  // sets control signals to
                        //decrements reg B or C
                        DECD_EOC: dec_D = 1'b1;

        SLLA_EOC: sll_A = 1'b1;  //sets control signals to shift
                        SLLB_EOC: sll_B = 1'b1;
                        SRLA_EOC: srl_A = 1'b1;
                        SRLB_EOC: srl_B = 1'b1;
                        SRAA_EOC: sra_A = 1'b1;
                        SRAB_EOC: sra_B = 1'b1;

                        // sets control signals to ALU functions.
                        // depending to the type of ALU function the
                        //ALU_func is given wit a value
                        // and depending on the register sel_BCD is given wit a value
                                ANDB_EOC:begin
                                ALU_func <= AND_func;
                                sel_BCD = 2'b01; // sel reg B
                                D_Bus_sel <= ALU_result_sel;
                                DBus_load_A = 1'b1;
                        end

                        ANDD_EOC:begin
                                ALU_func <= AND_func;
                                sel_BCD = 2'b11; // sel reg D
                                D_Bus_sel <= ALU_result_sel;
                                DBus_load_A = 1'b1;
                        end

                        ORB_EOC:begin
                                ALU_func <= OR_func;
                                sel_BCD = 2'b01; // sel reg B
                                D_Bus_sel <= ALU_result_sel;
                                DBus_load_A = 1'b1;
                        end

                        ORD_EOC:begin
                                ALU_func <= OR_func;
                                sel_BCD = 2'b11; // sel reg D
                                D_Bus_sel <= ALU_result_sel;
                                DBus_load_A = 1'b1;
                        end

                        XORB_EOC:begin
                                ALU_func <= XOR_func;
                                sel_BCD = 2'b01; // sel reg B
                                D_Bus_sel <= ALU_result_sel;
                                DBus_load_A = 1'b1;
                        end

                        XORD_EOC:begin
                                ALU_func <= XOR_func;
                                sel_BCD = 2'b11; // sel reg D
                                D_Bus_sel <= ALU_result_sel;
                                DBus_load_A = 1'b1;
                        end

                        ADDB_EOC:begin
                                ALU_func <= ADD_func;
                                sel_BCD = 2'b01; // sel reg B
                                D_Bus_sel <= ALU_result_sel;
                                DBus_load_A = 1'b1;
```

```
                        end

                        ADDC_EOC:begin
                                ALU_func <= ADD_func;
                                sel_BCD = 2'b10; // sel reg B
                                D_Bus_sel <= ALU_result_sel;
                                DBus_load_A = 1'b1;
                        end

                        SUBB_EOC:begin
                                ALU_func <= SUB_func;
                                sel_BCD = 2'b01; // sel reg B
                                D_Bus_sel <= ALU_result_sel;
                                DBus_load_A = 1'b1;
                        end

                        TCB_EOC:TCB_B = 1'b1; // sel reg B
                        CLCF_EOC:CLCF = 1'b1;
                        CLCZ_EOC :CLCZ = 1'b1;

                        default  next_state <= state;
                        endcase
                end

// PUSH and PULL registers A and B from memory!!
// while increasing or decreasing by 1 stack pointer
G2_CTI:
                begin
                case(reg_IR[5:0])
                        PSHA_EOC: begin
                                                A_Bus_sel <= reg_SP_sel;
                                                D_Bus_sel <= reg_A_sel;
                                                M_wr = 1'b1;
                                                dec_SP = 1'b1;
                                                end
                        PSHB_EOC: begin
                                                A_Bus_sel <= reg_SP_sel;
                                                D_Bus_sel <= reg_B_sel;
                                                M_wr = 1'b1;
                                                dec_SP = 1'b1;
                                                end
                        PULA_EOC: begin
                                                inc_SP = 1'b1;
                                                load_A = 1'b1;
                                                A_Bus_sel <= reg_SP_sel;
                                                end
                        PULB_EOC: begin
                                                inc_SP = 1'b1;
                                                load_B = 1'b1;
                                                A_Bus_sel <= reg_SP_sel;
                                                end
                        default next_state <= state;
                endcase
                end
// move data from one register to another register !!
G2_DTI:
                begin
                case(reg_IR[5:0])
                        MVAB_EOC: begin
                                D_Bus_sel <= reg_B_sel; //move from B to A
                                DBus_load_A = 1'b1;
                                end
                        MVBC_EOC:begin
                                D_Bus_sel <= reg_C_sel; //move from B to C
                                DBus_load_B = 1'b1;
                                end
                        MVAD_EOC: begin
                                D_Bus_sel <= reg_D_sel; //move from A to D
                                DBus_load_A = 1'b1;
                                end

                        MVBA_EOC: begin
                                D_Bus_sel <= reg_A_sel; //move from B to A
                                DBus_load_B = 1'b1;
                                end

                        MVCB_EOC: begin
                                D_Bus_sel <= reg_B_sel; //move from C to B
                                DBus_load_C = 1'b1;
                                end
                        MVDA_EOC: begin
                                D_Bus_sel <= reg_A_sel; //move from D to A
                                DBus_load_D = 1'b1;
                                end
                        default next_state <=state;
                endcase
                end
```

```
                                        // for uart
                                        G2_UART:
                                                begin
                                                        case(reg_IR[1:0])
                                                        TRANS: uart_tx = 1'b1;
                                                        RECV: begin
                                                                uart_rx = 1'b1;
                                                                D_Bus_sel <= UART_sel;
                                                                A_Bus_sel <= UART_sel;
                                                                end
                                                        endcase
                                                end
                                        // for parellel
                                        G2_PAR1:
                                                begin
                                                        case(reg_IR[1:0])
                                                        TRANS: par1_tx = 1'b1;
                                                        RECV: begin
                                                                par1_rx = 1'b1;
                                                                D_Bus_sel <= PAR1_sel;
                                                                A_Bus_sel <= PAR1_sel;
                                                                end
                                                        endcase
                                                end
                                        G2_PAR2:
                                                begin
                                                        case(reg_IR[1:0])
                                                        TRANS: par2_tx = 1'b1;
                                                        RECV: begin
                                                                par2_rx = 1'b1;
                                                                D_Bus_sel <= PAR2_sel;
                                                                A_Bus_sel <= PAR2_sel;
                                                                end
                                                        endcase
                                                end
                                        default         next_state <= state;
                                endcase
                                next_state <= EXEC;
                        end
                EXEC: begin
                        // this is when all executation happens.
                        // all the signals are installised back
                        clr_A = 1'b0;sll_A = 1'b0;srl_A = 1'b0;sra_A = 1'b0;load_AL = 1'b0;
                        load_AH = 1'b0;load_A = 1'b0;clr_B = 1'b0;inc_B = 1'b0;dec_B = 1'b0;
                        sll_B = 1'b0;srl_B = 1'b0;sra_B = 1'b0;cmb_B = 1'b0;load_B = 1'b0;
                        clr_C = 1'b0;cmc_C = 1'b0;load_C = 1'b0;clr_D = 1'b0; inc_D = 1'b0;
                        dec_D = 1'b0;load_D = 1'b0;DBus_load_D = 1'b0; DBus_load_B = 1'b0;
                        DBus_load_C = 1'b0; DBus_load_A = 1'b0;load_PC = 1'b0;load_IR = 1'b0; inc_PC = 1'b0;
                        M_wr = 1'b0;M_rd = 1'b0;IACK = 1'b0;sel_BCD = 2'b00;inc_SP = 1'b0; dec_SP = 1'b0;
                        TCB_B = 1'b0; CLCF = 1'b0;          CLCZ = 1'b0;
                        uart_tx = 1'b0; uart_rx = 1'b0;         par1_rx = 1'b0;par1_tx = 1'b0;
                        par2_rx = 1'b0;par2_tx = 1'b0;A_Bus_sel <= No_Change;
                        next_state <= INIT;
                end
                default next_state <= state;
                endcase
                end

endmodule


SIMPX datapath
module simpxdataPath(clk,reset,D_BusIn,clr_A,sll_A,srl_A, sra_A,load_AL,load_AH,load_A,clr_B,inc_B,dec_B,sll_B,
srl_B,sra_B,cmb_B,load_B,TCB_B,clr_C,cmc_C,load_C, clr_D,inc_D,dec_D,load_D, load_PC,CLCF,CLCZ,
sel_BCD, A_Bus_sel, D_Bus_sel, ALU_func, inc_PC, reg_IR,inc_SP, dec_SP,
DBus_load_D, DBus_load_C, DBus_load_B, DBus_load_A, Data_UART, Data_PAR1, Data_PAR2,
reg_A,reg_B,reg_C,reg_D,D_BusOut,A_Bus, reg_PC, reg_Flag, ALU_result, reg_SP);

                input clk,reset;
                input [15:0] D_BusIn;
                input [7:0] Data_UART, Data_PAR1, Data_PAR2;
                input clr_A,sll_A,srl_A,sra_A,load_AL,load_AH,load_A;
                input clr_B,inc_B,dec_B,sll_B,srl_B,sra_B,cmb_B,load_B,TCB_B;
                input clr_C,cmc_C,load_C;
                input clr_D,inc_D,dec_D,load_D;
                input DBus_load_D, DBus_load_C, DBus_load_B, DBus_load_A;
                input load_PC,inc_PC;
                input CLCF,CLCZ;
                input [3:0] D_Bus_sel,A_Bus_sel;
                input[1:0] sel_BCD;
                input [2:0] ALU_func;
                input [15:0] reg_IR;
                input inc_SP, dec_SP;
                output [15:0] D_BusOut;
                output [11:0] A_Bus;
                output [16:0] ALU_result;
```

```
		reg C_flag, Z_flag;
		output [15:0] reg_A,reg_B,reg_C,reg_D;
		output [11:0] reg_PC,reg_Flag, reg_SP;
		reg [15:0] reg_A,reg_B,reg_C,reg_D;
		reg [16:0] ALU_result;
		reg [11:0] reg_PC,reg_Flag, reg_SP;
		reg[11:0] A_Bus;
		reg [15:0] D_BusOut;

// ------------------------------------------------------------------------------------------

		//ALU functions
		parameter ADD_func=3'b001,SUB_func=3'b010,AND_func=3'b011,OR_func=3'b100,XOR_func=3'b101;
		parameter ALU_ADD = 3'b001, ALU_SUB = 3'b010, ALU_AND = 3'b011, ALU_OR = 3'b100, ALU_XOR = 3'b101;

//register selection and datapath inputs
		parameter ALU_result_sel= 4'b0000, reg_A_sel= 4'b0001,reg_B_sel = 4'b0010, reg_C_sel= 4'b0011,reg_D_sel = 4'b0100, reg_PC_sel =
4'b0101,
		reg_SP_sel = 4'b0110, reg_IR_sel = 4'b1110,	No_Change = 4'b1111; //Data_in_sel = 4'b1000,
		parameter UART_sel = 4'b1000, PAR1_sel = 4'b1001, PAR2_sel = 4'b1010;



//---------------------------- Register IR and PC -----------------------------------------
		always @ (posedge clk)
			//begin
			//if (state == 2'b11)
			begin

			if (reset == 1'b1)
				begin					// resets all the registers.
				reg_A = {16{1'b0}};
				reg_B = {16{1'b0}};
				reg_C = {16{1'b0}};
				reg_D = {16{1'b0}};
				end

		// executes the control signal that is high, set from the decode state of the controlpath.
		// includes all the control signals for regA, regB, regC and regD

//---------------------------- Register A -----------------------------------------
			else if (clr_A == 1'b1)
				reg_A = {16{1'b0}};
			else if (sll_A == 1'b1)
				reg_A = {reg_A[14:0],1'b0};
			else if (srl_A == 1'b1)
				reg_A = {1'b0,reg_A[15:1]};
			else if (sra_A == 1'b1)
				reg_A [14:0] = reg_A[15:1];
			else if (load_AL == 1'b1) // load the LSB part of A
				reg_A [7:0] = D_BusIn[7:0];
			else if (load_AH == 1'b1) //load the MSB part of A
				reg_A [15:8] = D_BusIn[15:8];
			else if (load_A == 1'b1) // load A from D_Bus
				reg_A = D_BusIn;

//---------------------------- Register B -----------------------------------------

			else if (clr_B == 1'b1)
				reg_B= {16{1'b0}};
			else if (inc_B == 1'b1)
				reg_B = reg_B + 1;
			else if (dec_B == 1'b1)
				reg_B= reg_B - 1;
			else if (sll_B == 1'b1)
				reg_B= {reg_B[14:0],1'b0};
			else if (srl_B == 1'b1)
				reg_B= {1'b0, reg_B[15:1]};
			else if (sra_B == 1'b1)
				reg_B[14:0] = reg_B[15:1];
			else if (cmb_B == 1'b1)
				reg_B= ~reg_B;
			else if (load_B == 1'b1) // load B from D_Bus
				reg_B= D_BusIn;
			else if (TCB_B == 1'b1)
				begin
				reg_B= ~reg_B;
				reg_B= reg_B + 1;
				end

//---------------------------- Register C -----------------------------------------

			else if (clr_C == 1'b1)
				reg_C= {16{1'b0}};
			else if (cmc_C == 1'b1)
				reg_C = ~reg_C;
			else if (load_C == 1'b1) // load C from D_Bus
```

```
                                        reg_C = D_BusIn;

//--------------------------- Register D ------------------------------------------

                                else if (clr_D== 1'b1)
                                        reg_D = {16{1'b0}};
                                else if (inc_D == 1'b1)
                                        reg_D = reg_D + 1;
                                else if (dec_D == 1'b1)
                                        reg_D= reg_D - 1;
                                else if (load_D == 1'b1) // load D from D_Bus
                                        reg_D = D_BusIn;

//--------------------------- Flag register ----------------------------------------

                                else if (CLCF == 1'b1)
                                        reg_Flag = reg_Flag && 4094;
                                else if (CLCZ == 1'b1)
                                        reg_Flag = reg_Flag && 4093;

//--------------------------- Move between registers -------------------------------

                                else if (DBus_load_D == 1'b1)
                                        reg_D = D_BusOut;
                                else if (DBus_load_C== 1'b1)
                                        reg_C = D_BusOut;
                                else if (DBus_load_B== 1'b1)
                                        reg_B = D_BusOut;
                                else if (DBus_load_A== 1'b1)
                                        reg_A = D_BusOut;

// ---------------------- default---------------------------------------------------

                                else                                                    // just in case
                                        begin
                                        reg_A = reg_A;
                                        reg_B = reg_B;
                                        reg_C = reg_C;
                                        reg_D = reg_D;
                                        reg_Flag = reg_Flag;
                                        end
                                end


                        always @ (posedge clk)   // implements all the SP pointer related executions.
                        begin
                                if (reset == 1'b1)   // reseting SP
                                        reg_SP = 12'b000000010100;
                                else if (inc_SP == 1'b1)   // inc SP
                                        reg_SP = reg_SP + 1;
                                else if (dec_SP== 1'b1)  // dec SP
                                        reg_SP = reg_SP - 1;
                                else reg_SP = reg_SP;
                                end

//--------------------------- PC ----------------------------------------

                        always @ (posedge clk)begin  // implements all the PC related executions
                                if (reset == 1'b1)
                                        reg_PC = {12{1'b0}};
                                else if (inc_PC == 1'b1)  // inc PC
                                        reg_PC = reg_PC + 1;
                                else if (load_PC == 1'b1)   // for JUMP instructions
                                        reg_PC = D_BusOut[11:0];
                                else reg_PC = reg_PC;
                                end

//--------------------------- D Bus ------------------------------------------
                                                                                        // D_Bus_sel is assigned a value in the controlpath
                        always @ (D_Bus_sel)  // depending on D_Bus_sel a value is assigned to D_BusOut
                                case(D_Bus_sel)                         // everytime D_Bus_sel changes
                                        ALU_result_sel:D_BusOut <= ALU_result[15:0];
                                        reg_A_sel:D_BusOut <= reg_A;
                                        reg_B_sel:D_BusOut <= reg_B;
                                        reg_C_sel:D_BusOut <= reg_C;
                                        reg_D_sel:D_BusOut <= reg_D;
                                        UART_sel: D_BusOut <= Data_UART;
                                        PAR1_sel: D_BusOut <= Data_PAR1;
                                        PAR2_sel: D_BusOut <= Data_PAR2;
                                        reg_IR_sel: D_BusOut[11:0] <= reg_IR[11:0];
                                        default D_BusOut <= {16{1'bz}};
                                endcase


//--------------------------- A Bus-------------------------------------------
                                                                                        // A_Bus_sel is assigned a value in the controlpath
                        always@(A_Bus_sel)                      // depending on A_Bus_sel a value is assigned to A_Bus
```

```
		case(A_Bus_sel)				// everytime A_Bus_sel changes
			reg_A_sel: A_Bus <= reg_A[11:0];
			reg_B_sel: A_Bus <= reg_B[11:0];
			reg_C_sel: A_Bus <= reg_C[11:0];
			reg_D_sel: A_Bus <= reg_D[11:0];
			reg_PC_sel: A_Bus <= reg_PC;
			reg_SP_sel: A_Bus <= reg_SP;
			reg_IR_sel: A_Bus <= reg_IR[11:0];
			UART_sel: A_Bus <= 12'b11111011;
			PAR1_sel: A_Bus <= 12'b11111100;
			PAR2_sel: A_Bus <= 12'b11111101;
			default A_Bus <= {12{1'bz}};
		endcase

		                                                                                // ALU_func is assigned a value in the controlpath
	always @ (ALU_func)   // depending on ALU_func a value is assigned to ALU_result
		begin                                                   // everytime ALU_func changes
		case(ALU_func)
			ALU_ADD: begin  //adds
				 if (sel_BCD == 2'b01)   // sel_BCD, is assigned in the controlpath
					ALU_result = reg_A + reg_B;
				else if (sel_BCD == 2'b10)
					ALU_result = reg_A + reg_C;
				end
			ALU_SUB: begin  //subs
				if (sel_BCD == 2'b01)
					ALU_result = reg_A -  reg_B;
				else
					ALU_result = {16{1'bz}};
				end
			ALU_AND: begin
				if (sel_BCD == 2'b01)
					ALU_result = reg_A &  reg_B;
				else if (sel_BCD == 2'b11)
					ALU_result = reg_A &  reg_D;
				end
			ALU_OR: begin
				if (sel_BCD == 2'b01)
					ALU_result = reg_A | reg_B;
				else if (sel_BCD == 2'b11)
					ALU_result = reg_A | reg_D;
				end
			ALU_XOR: begin
				if (sel_BCD == 2'b01)
					ALU_result = reg_A ^ reg_B;
				else if (sel_BCD == 2'b11)
					ALU_result = reg_A ^ reg_D;
				end
			default ALU_result = {17{1'bz}};
		endcase

		if (ALU_result == {17{1'b0}})   // checks for carry and the zero flag.
			Z_flag = 1'b1;
		else
			Z_flag = 1'b0;

		if (ALU_result[16] == 1'b1)
			C_flag = 1'b1;
		else
			C_flag = 1'b0;
	end

endmodule
```

Appendix VI: Verilog code written to implement UART

```
/*ELECTENG304 COMPSYS2E Assignment 1b
Assignment: Implementation of a simple microcontroller
Date: 03.10.03
Developers: Jaisheel Singh - 9801736
                                        Thusitha Mabotuwana - 9790416

*/

/*This module is used to transmit data via the serial interface. When byteReady is high the data in databus gets
loaded into the shift register and when trByte is high this value gets passed onto the dataRegister which will
get transmitted one bit at a time
*/
module UART_TX(clk,reset,dataBus,byteReady,trByte,serialOut, con_tx);
        parameter N = 8, M = N + 2;
        input clk, reset;
        input [N-1:0] dataBus;
        input byteReady,trByte, con_tx;
        output serialOut;

        reg [M:0] XMTshiftReg;
        reg [N-1:0] XMTdataReg;
        reg start,shift,loadXMTdataReg,loadXMTshiftReg;
        reg [3:0] bitCount;
        reg inc_bitCount;
        reg count_start,reset_count,reset_bitCount;
        wire clk,count_start_wire,reset_count_wire;
//Different states used
        parameter idle = 2'b00,    waiting = 2'b01, sending = 2'b10;
        reg             [1:0] state,nextState;

        //Datapath

        always @ (posedge clk)
                begin
                if (reset == 1'b1 || con_tx == 1'b0) //seond condition is needed since this will be a
//contrl signal when transmit is combined with receiver
                        begin
                        XMTshiftReg <= {11{1'b1}};
                        bitCount <= 0;
                        end
                else if (loadXMTshiftReg == 1)
                                begin
                                XMTshiftReg <= {1'b1,XMTdataReg, 1'b0}; //1'b0 is start bit
                                XMTshiftReg[9] <= !(XMTdataReg[0] ^  XMTdataReg[1] ^ XMTdataReg[2] ^
                                XMTdataReg[3] ^ XMTdataReg[4] ^ XMTdataReg[5] ^ XMTdataReg[6] ^ XMTdataReg[7]); //parity
bit
                                XMTshiftReg[10] <= 1'b1; //send stop bit
                                end
                if (start == 1 && shift == 1)
                        XMTshiftReg <= {1'b1, XMTshiftReg[N+2:1]}; // shift right, fill 1's
                if (bitCount == 10) begin
                        bitCount <=0;
                        reset_bitCount <= 1;
                        end
                else if (inc_bitCount == 1)
                        bitCount <= bitCount + 1;
                else
                        reset_bitCount <= 0;
                if (loadXMTdataReg == 1) //load data reg with databus
                        XMTdataReg <= dataBus;

                end
        assign serialOut = XMTshiftReg[0];    //this bit has to be transmitted

        //Control path. This is where the required control signals are set
        always @ (posedge clk)
                if (reset == 1'b1 || con_tx == 1'b0)
                        state <= idle;
                else
                        state <= nextState;

        always @ (state or byteReady or trByte)
                begin
                case (state)
                        idle: begin
                                inc_bitCount<=0;
                                shift <= 0;
                                start <= 0;
                                loadXMTshiftReg <= 0;
                                loadXMTdataReg <= 0;
                                if (byteReady == 1) //byteready signal high
                                        begin
                                        loadXMTdataReg <= 1;
```

```
                                        nextState <= waiting;
                                        end
                                else
                                        nextState <= idle;
                                end
                        waiting: begin
                                loadXMTdataReg <= 0;
                                if (trByte == 1) //start transmitting
                                        begin
                                        loadXMTshiftReg <= 1;
                                        nextState <= sending;
                                        end
                                else
                                        nextState <= waiting;
                                end
                        sending:
                                begin
                                start <= 1;
                                shift <= 1;
                                if (bitCount < 10) //have to send 1 start,8data,1 parity, 1 stop bit
                                        begin
                                        inc_bitCount<=1;
                                        nextState <= sending;
                                        end
                                if (reset_bitCount == 1) begin
                                        start <= 0;
                                        shift <= 0;
                                        nextState <= idle;
                                        end
                                if (bitCount == 10)         //finishd sending all bits go back to idle
                                        nextState <= idle;
                                end
                        default nextState <= idle;
                endcase
                end

endmodule


/*ELECTENG304 COMPSYS2E Assignment 1b
Assignment: Implementation of a simple microcontroller
Date: 03.10.03
Developers: Jaisheel Singh - 9801736
                        Thusitha Mabotuwana - 9790416

*/

/*This module is used to receive data received on the external input pins. Later on these values
will be written into a fixed position in memory. 11 bits (1 start,8data,1 parity,1stop)
are received and when hostready is high these values are sent to databus which will write to memory*/


module uart_rx(sample_clk,reset,serial_in,host_ready,REC_datareg,error1,error2,hostError, con_rx);
parameter N=9;
input sample_clk,reset,serial_in, host_ready, con_rx;
output hostError,error1,error2;
output [N-2:0] REC_datareg;

//variables (and constatnts values) for implementing control part
parameter idle=2'b00,starting=2'b01,receiving = 2'b10;
reg[1:0] state,next_state;
// some control signals
reg load,shift,error1,error2;

// datapath
reg parity,hostError;
reg[N-2:0] REC_datareg;
reg[N:0] REC_shftreg;
reg[3:0] bit_counter;
reg[2:0] sample_counter;
reg load_shftreg;
reg data;
// control signals (from control part)
reg clr_sample_counter,inc_sample_counter,bit_count_zero;
reg clr_bit_counter,inc_bit_counter,load_shiftRegN;

// implimanting control path

always @ (posedge sample_clk)
        begin
        if (reset== 1'b1 || con_rx == 1'b0) //second signal is reqd in the main module as 2 control signals
//, 1 for Tx and 1 for Rx will be send n in this module we worry only abt rx
                begin
                sample_counter<=0;
                bit_counter<=0;
                REC_shftreg<=0;
                REC_datareg<=0;
```

```
                end
        else
                begin
                hostError<=0;
                if (clr_sample_counter==1)
                        sample_counter<=0;
                else if (inc_sample_counter==1)
                        sample_counter <= sample_counter+1'b1;
                if(clr_bit_counter==1)
                        bit_counter<=0;
                else if (inc_bit_counter==1)
                        bit_counter <= bit_counter+1'b1;
                if(load==1 && host_ready==1)          //valid data available. send recvd data to host
                        REC_datareg<=REC_shftreg[N:1];
                else if (load==1 && host_ready == 0) //host not ready
                        hostError <=1;
                if(load_shftreg==1)
                        REC_shftreg<=0;
                if (load_shiftRegN==1)   //load 1 recvd bit
                        REC_shftreg<={serial_in,REC_shftreg[N:1]};
                if(shift==1)
                        REC_shftreg<={1'b0,REC_shftreg[N:1]};
                end
    end

        always @ (posedge sample_clk) begin
                if (reset==1'b1  || con_rx == 1'b0)
                        state <=idle;
                else
                        state <=next_state;
                end

//the sampling clk is 8 times faster than the tr clk as we want to sample bits
        always @ (state or serial_in or sample_counter or bit_counter)
                begin
                case(state)
                        idle:
                                begin
                                load_shftreg<=0;load <=0;shift<=0;
                                clr_sample_counter<=0;inc_sample_counter<=0;clr_bit_counter<=1;          inc_bit_counter<=0;
                                error1<=0;error2<=0;load_shiftRegN<=0;
                                if(serial_in==0) //start bit received
                                        next_state<=starting;
                                else
                                        next_state<=idle;          //idle line
                                end

                        starting:begin
                                shift<=0;
                                if (serial_in==1)
                                        begin
                                        clr_sample_counter<=1;
                                        next_state<=idle;
                                        end
                                else
                                        begin
                                        if (sample_counter==3) //start bit received properly. goto next state
                                                begin
                                                clr_sample_counter<=1;
                                                next_state<=receiving;
                                                end
                                        else
                                                begin
                                                inc_sample_counter<=1;
                                                next_state<=starting;
                                                end
                                        end
                                end

                        receiving: begin
                                clr_sample_counter<=0;
                                inc_sample_counter<=1;
                                clr_bit_counter<=0;error1<=0;

                                if(sample_counter==7) //received 1 bit
                                        begin
                                        load_shiftRegN<=1;
                                        if (bit_counter!=9 && bit_counter<9) //havent recvd whole byte yet
                                                begin
                                                inc_bit_counter<=1;
                                                next_state<=receiving;
                                                end

                                        else if (bit_counter==9) //check for parity bit
                                                begin
                                                inc_bit_counter<=1;
                                                if (REC_shftreg[9]^REC_shftreg[1]^REC_shftreg[2]^REC_shftreg[3]^
```

```
REC_shftreg[4]^REC_shftreg[5]^REC_shftreg[6]^REC_shftreg[7]^REC_shftreg[8]==0)
                                                                        error2<=1;                  //parity error
                                                                next_state<=receiving;
                                                                end
                                                        else
                                                                begin
                                                                inc_bit_counter<=0;
                                                                if (REC_shftreg[9] == 1 && error2==0) //stop bit received
                                                                        begin
                                                                        load<=1;

                                                                        end
                                                                else            //stop bit missing
                                                                        begin
                                                                        load<=0;
                                                                        error1<=1;
                                                                        end
                                                                next_state<=idle;           //go back to idle
                                                                end
                                                        end
                                                else
                                                        begin
                                                        shift<=0;load<=0;error2<=0;
                                                        inc_bit_counter<=0;
                                                        load_shiftRegN<=0;
                                                        next_state<=receiving;
                                                        end
                                                end
                                        default next_state<=idle;
                                        endcase
                                end

endmodule
```

Appendix VII: Verilog code written to implement RAM

```
/*ELECTENG304 COMPSYS2E Assignment 1b
Assignment: Implementation of a simple microcontroller
Date: 03.10.03
Developers: Jaisheel Singh - 9801736
                                        Thusitha Mabotuwana - 9790416

*/

//memory

module memory(address, inclock, we ,data, q);

input [7:0] address;
input inclock;
input we;
input [15:0] data;
output [15:0] q;
wire [15:0] sub_wire0;
wire [15:0] q = sub_wire0[15:0];



LPM_RAM_DQ LMP_RAM_DQ_component (.inclock(inclock), .address(address), .data(data), .we(we), .q(sub_wire0));

defparam
        LMP_RAM_DQ_component.LPM_WIDTH = 16,
        LMP_RAM_DQ_component.LPM_WIDTHAD = 8,
        LMP_RAM_DQ_component.LPM_INDATA = "REGISTERED",
        LMP_RAM_DQ_component.LPM_ADDRESS_CONTROL= "UNREGISTERED",
        LMP_RAM_DQ_component.LPM_OUTDATA = "UNREGISTERED",
        LMP_RAM_DQ_component.LPM_FILE = "abc.mif",
        LMP_RAM_DQ_component.USE_EAB = "ON";

Endmodule
```

Appendix VIII: Verilog code written to implement Parallel I/O

```
/*ELECTENG304 COMPSYS2E Assignment 1b
Assignment: Implementation of a simple microcontroller
Date: 03.10.03
Developers: Jaisheel Singh - 9801736
                                        Thusitha Mabotuwana - 9790416

*/

//PAR
module par(clk, con_rx, con_tx, ExternalBusIn, ExternalBusOut, DataBusIn, DataBusOut);

        input clk, con_rx, con_tx;
        input [7:0] ExternalBusIn, DataBusIn;
        output [7:0] ExternalBusOut, DataBusOut;

        reg [7:0] ExternalBusOut, DataBusOut;

        always @ (posedge clk)
                begin  // depending on the control signals sent from the control path
                        if (con_rx == 1'b1)  // either data is assigned to the External Pins from DataBusOut
                                DataBusOut = ExternalBusIn;

                        if(con_tx == 1'b1)  // or data from external Pins is assinged to DataBusIn
                                ExternalBusOut = DataBusIn;
                end
endmodule
```