Digital Systems Laboratory 4/MSc
ELEE10023/PGEE10009

Course Organiser: Dr. Alister Hamilton
School of Engineering, University of Edinburgh
Email: Alister.Hamilton@ed.ac.uk
January 2016

Course Description

Aim: The course aims to produce students who are capable of developing hardware-software digital systems from high level functional specifications and prototyping them on to FPGA hardware using a standard hardware description language and software programming language.

Pre-requisites: Digital Systems Laboratory 3 (ELEE09018) or Digital Systems Laboratory A (PGEE10017) or equivalent in other schools and outside institutions (see below). Engineering Software 3 or equivalent is advisable but not necessary.

Co-requisites: Undergraduate students must take Digital System Design 4 (ELEE10007)

Prohibited Combinations: None

Visiting Students Pre-requisites: Digital design using Verilog, and embedded system programming.

Keywords: Embedded Digital System Design, Embedded Processor Programming, Verilog, Data path and Control Path design, Hardware-Software Co-design

Default Course Mode of Study: **Lab only, 10 weekly 3-hr lab sessions**

Default Delivery Period: Semester 2, starting in Week 2.

Learning Outcomes:

1. Knowledge and understanding of:
   - I. Data paths and Control paths and number of ways of designing them;
   - II. Instruction-set based control path design;
   - III. Control and data path integration;
   - IV. Capture the design of hardware-software digital systems in a standard hardware description language;

2. Intellectual
   - I. Ability to use and choose between different techniques for digital system design and capture;
   - II. Ability to evaluate implementation results (e.g. speed, area, power) and correlate them with the corresponding high level design and capture;
3. Practical
   - I. Ability to use a commercial digital system development tool suite to develop hardware-software digital systems and prototype them on to FPGA hardware;

Lab Content, What You Are Required To Do

You are required to develop a microprocessor-based system on FPGA with a demo application, written in software running on the microprocessor, which controls toy race-cars remotely. Undergraduate students will be split into groups of three students each to tackle this problem, postgraduate students into groups of two. The final system will allow a user to control cars remotely using a mouse, a VGA screen, and the BASYS 3 FPGA board as illustrated in the figure below.

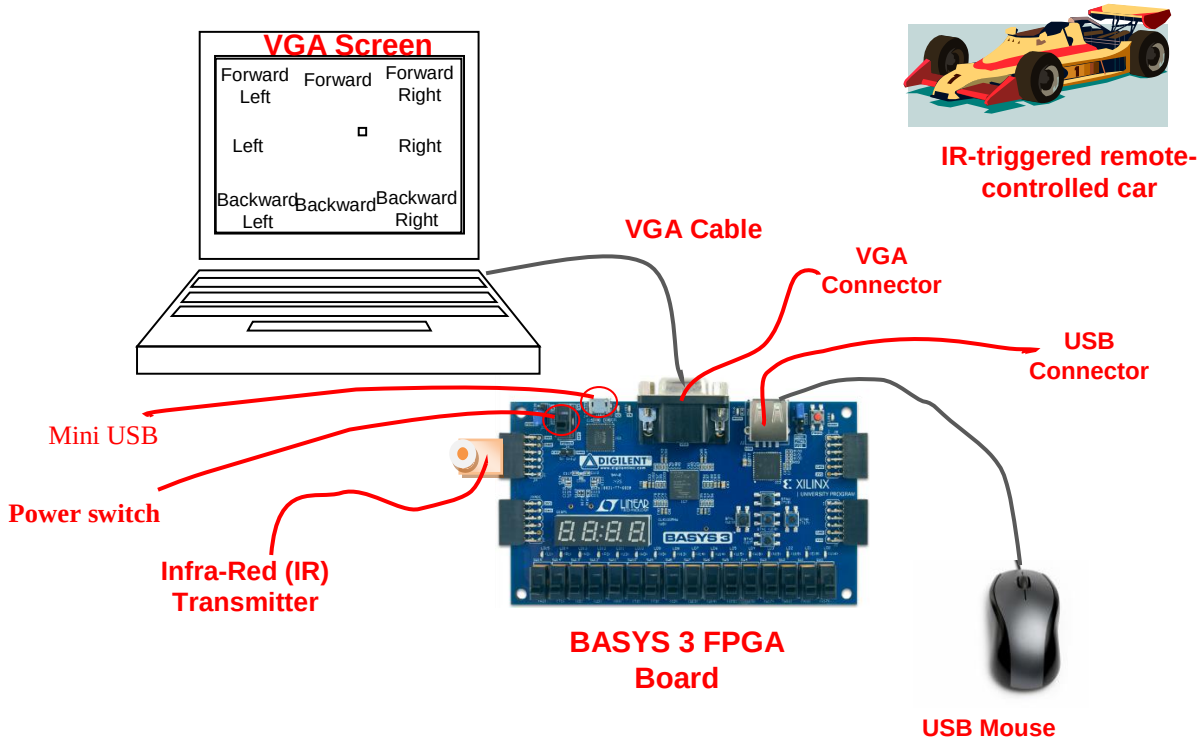


Figure 1. Proposed System for FPGA-Based Remote Car Control

A user will be able to hover a mouse pointer over a VGA screen with the position of the mouse pointer on screen commanding the movement of the remote car as illustrated in the following figure.

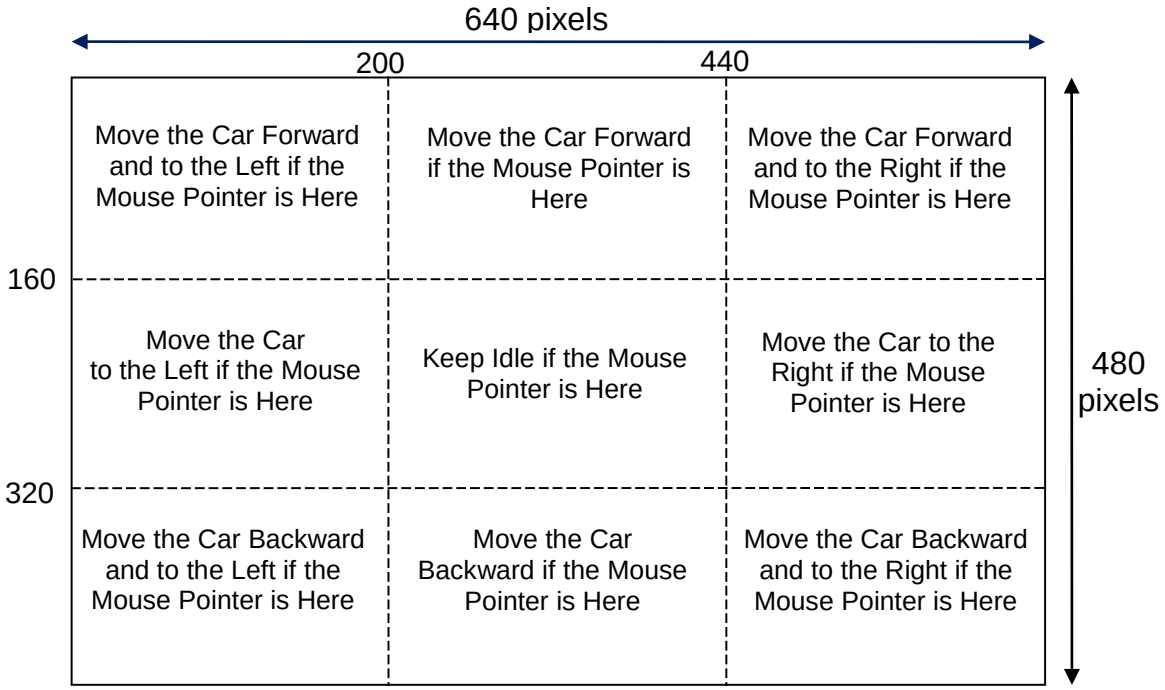


Figure 2. Car movement (command) depending on mouse pointer position

The FPGA-based system will consist of a simple microprocessor, a VGA interface, a mouse interface, and an Infra-Red (IR) Transmitter interface, in addition to other peripherals which will be detailed later on this document. **For undergraduate students**, each team member will first develop one of the following three peripherals, individually: VGA interface, mouse interface, or IR Transmitter interface. **For postgraduate students**, the whole group will co-operate to develop the IR Transmitter interface, then each team member will develop either the VGA interface or the mouse interface separately. **For both undergraduate and postgraduate students**, the whole group will then collaborate to develop the microprocessor at the heart of the proposed system with all necessary peripherals, as well as the complete demo application for remote car control as specified above.

Assessment: The lab will be assessed during lab sessions through a number of checkpoints. The overall lab mark will be split into individual (60%) and group components (40%) for both undergraduate and postgraduate students. **All students should keep a lab book**.

Assessment (undergraduate students): The individual component will consist of two checkpoints:

1. First individual assessment in Week 6, when every team member will be assessed on the particular peripheral interface they developed i.e. mouse driver, VGA interface or IR Transmitter interface. Code should be uploaded to Learn prior to the timetabled laboratory in Week 6 for assessment. This will account for 25% of the overall lab mark.
2. Second individual assessment in Week 9, when every team member will demonstrate a working microprocessor + peripheral demo software application. For instance, the team member in charge of mouse driver development will present a demo software application running on the microprocessor with the mouse peripheral. Similarly, the team member in charge of VGA interface development will present a demo software application running on the microprocessor with the VGA interface peripheral, and finally the team member in charge of IR Transmitter interface development will

present a demo software application running on the microprocessor with the IR transmitter peripheral. The specification of the individual demo software application will be specified later on this document. This second individual assessment will account for 35% of the overall lab mark. Note that while this assessment is individual, it requires prior design of the same microprocessor architecture by all team members. Code should be uploaded to Learn prior to the timetabled laboratory in Week 9 for assessment.

The final assessment will be a group assessment in Week 11, where the entire team will demonstrate the complete demo software application running on the complete microprocessor-based system on the BASYS 3 board. There will be an element of peer assessment in this final group based assignment where individuals will be asked to evaluate the individual performance of colleagues within the group. More details of this process will be given nearer the time of assessment. Again all design files should be uploaded to Learn prior to the start of the scheduled laboratory.

Assessment (postgraduate students): The first group assessment will be in Week 5 when each group will be assessed on the IR Transmitter interface. Code should be uploaded to Learn prior to the timetabled laboratory in Week 5 for assessment. This will account for 10% of the overall lab mark.

The individual component will consist of two checkpoints:

1. First individual assessment in Week 6, when every team member will be assessed on the particular peripheral interface they developed i.e. mouse driver or VGA interface. Code should be uploaded to Learn prior to the timetabled laboratory in Week 6 for assessment. This will account for 25% of the overall lab mark.
2. Second individual assessment in Week 9, when every team member will demonstrate a working microprocessor + peripheral demo software application. Each team member will demonstrate a demo software application running on the microprocessor with the IR transmitter peripheral. This may be developed in associated with the other group member. The team member in charge of mouse driver development will present a demo software application running on the microprocessor with the mouse peripheral. Similarly, the team member in charge of VGA interface development will present a demo software application running on the microprocessor with the VGA interface peripheral. The mouse and VGA applications should be developed individually. The specification of the individual demo software applications may be found later in this document. This second individual assessment will account for 35% of the overall lab mark. Note that while this assessment is mostly individual, it requires prior design of the same microprocessor architecture by all team members. Code should be uploaded to Learn prior to the timetabled laboratory in Week 9 for assessment.

The final assessment will be a second group assessment in Week 11 which is the same as for undergraduate students (see above), but counts for 30% of the overall assessment.

Details of the university semester structure can be found on the university web pages. Note that the week between weeks 5 and 6 is an unnumbered week, Innovative Learning Week.

The remainder of this document will present the detailed specification of each component of the proposed system. The details of the assessment components will also be presented where appropriate.

1. PS/2 Mouse Driver

The USB connector on the BASYS 3 board can accommodate a USB mouse. Internally, the signals are converted to PS/2-like signals via a microcontroller as discussed on page 7 and 8 of the BASYS 3 reference manual. Hence, all we need do is implement a driver for a PS/2 mouse as the USB connector could be seen as just a wrapper which has already been implemented.

The PIC24 drives several signals into the FPGA – two are used to implement a standard PS/2 interface for communication with a mouse or keyboard (see figure 3 below).

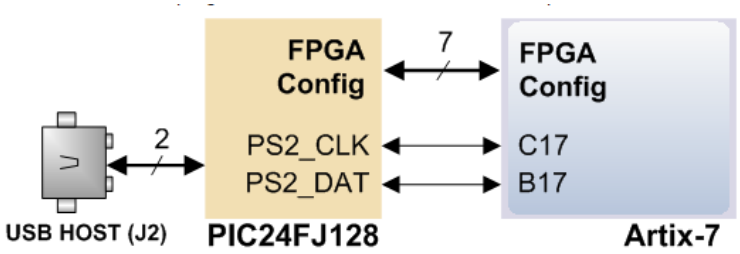


Figure 3: USB Host signals to PS/2 signal conversion

PS/2 devices use a two-wire serial bus (clock and data) to communicate with a host device

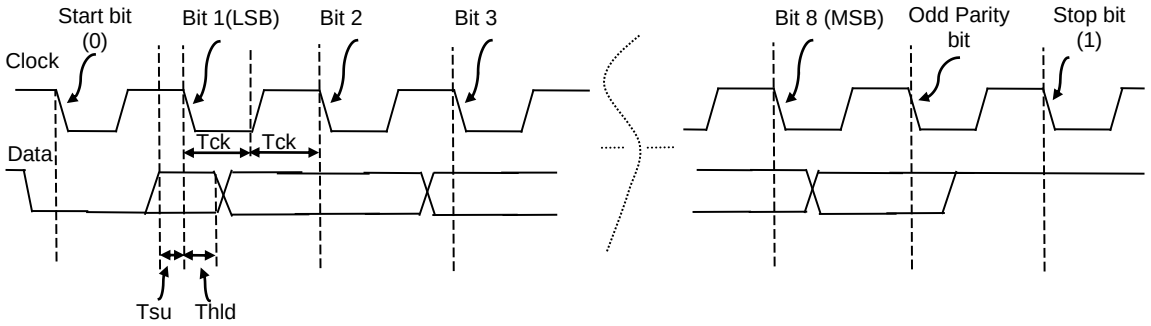
Communication is bidirectional and performed in packets of 11-bit words, with each word containing a start, stop and odd parity bit. The following describes the PS/2 mouse protocol in detail (NB. the PS/2 keyboard protocol can be found in the BASYS 3 user manual if you are interested in developing a keyboard interface but this is not required in this lab).

The mouse device only outputs a clock and data signal when it is moved. Otherwise, the clock and data lines remain at logic high (i.e. '1'). Open-Collector drivers are usually used to drive the two-wire bus between the mouse and host.

The device can send data to the host only when both data and clock lines are high. Because the host is the bus master, the device must check whether the host is sending data before driving the bus. The clock line is used for this purpose as a "clear to send" signal; if the host pulls the clock line low, the device must not send any data until the clock is released.

Communication is performed in 11-bit words, where each word consists of a '0' start bit, followed by 8 bits of data (LSB first), followed by an odd parity bit (i.e. a bit that is set to '1' if the number of 1's in the 8 bits of data is even, and '0' otherwise), and terminated with a '1' stop bit. The odd-parity bit is used for error detection.

Data sent from a PS/2 device to a host is read on the falling edge of the clock signal, whereas data sent from a host to a PS/2 device is read on the rising edge of the clock signal. The following figure shows PS/2 signal timing for a device to host communication. Note the timing requirements which must be strictly adhered to. The clock frequency, for instance, must lie between 10 and 16.7 KHz.



Tck: Clock Time, should be between 30us and 50us
Tsu: Data-to-clock setup time, should be between 5us and 25us
Thld: Clock-to-data hold time, should be between 5us and 25us

Figure 3. PS/2 Device to Host Signal Timing

The following figure shows PS/2 signal timing for a host to device communication. The host brings the clock line low first, for at least 100μs. It then brings the data line low and releases the clock line. The host then waits for the PS/2 device to bring the clock line low. After that, it sets or resets the data line with the first data bit, and waits for the device to bring clock line high. It then waits for the device to bring the clock line low before it sets/rests the data line with the second data bit. This process is repeated until all eight data bits are sent as well as the odd-parity bit. Next, the host releases the data line, and waits for the device to bring the data line low, and then the clock line low. Finally, the host waits for the device to release data and clock lines.

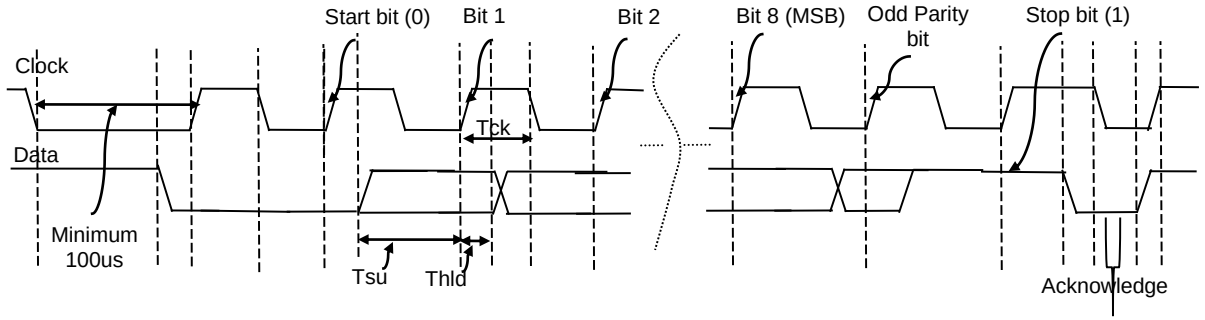


Figure 4. Host to PS/2 Device Signal Timing

Now that we have seen the low level PS/2 protocol, let us look at the high level host-mouse communication. At power-up, a typical host-mouse communication consists of the following steps:

1) The host sends a Reset Command (consisting of byte "FF") to the mouse,
2) The mouse responds with an Acknowledgement byte "FA",
3) The mouse then goes through a self-test process and sends "AA" when this is passed. Then a mouse ID byte "00" is sent to the host, after which the host knows that the mouse is functioning well and ready to transmit data,
4) The host sends byte "F4" to instruct the mouse to "Start Transmitting" its position information,
5) The mouse acknowledges the "Start Transmitting" command by sending byte "FA" back to the host**,
6) After this, the mouse starts transmitting its position information in the form of 3 bytes at a sample rate that can be set by the host (the default is 100Hz)

**Note however that in the Basys3 FPGA board, probably due to the USB to PS/2 conversion, F4 instead of FA is returned, and parity test fails. Hence in state 8 of MasterStateSM module, the acknowledgement code have been changed to F4, and parity check skipped.

Thus, each data transmission from the mouse to the host after initialisation consists of 33 bits, where bits 1 (first bit), 12, and 23 are '0' start bits; bits 10, 21, and 32 are Odd-Parity bits; and bits 11, 22, and 33 are '1' stop bits. The three-byte data fields contain status and movement data as shown in the figure below.

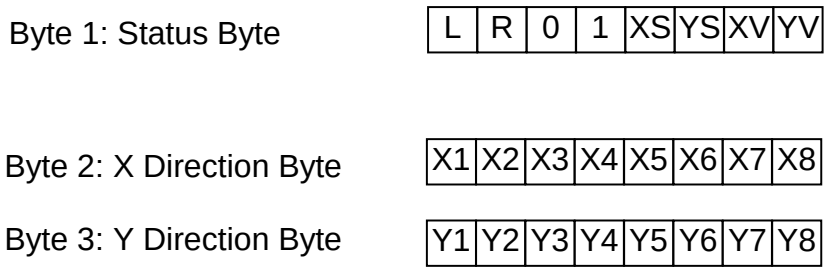


Figure 5. Mouse Data Format

The mouse reports a relative coordinate system whereby a move to the right generates a positive number in the X Direction Byte field, and a move to the left generates a negative number in this field. Similarly, a move upwards generates a positive number in the Y Direction Byte field, and a move downwards generates a negative number. Note that the X and Y Direction Bytes represent the magnitude of the rate of mouse movement, the larger the number the faster the mouse is moving. Bits XS and YS in the Status Byte are the sign bits, whereby a '1' indicates a negative number, whereas XV and YV bits are movement overflow indicators, whereby a '1' means overflow has occurred. The L and R fields in the Status Byte indicate that the left and right button have been pressed, respectively ('1' indicates the button has been pressed).

What you are required to do

You are required to design an FPGA PS/2 mouse interface and implement it on the BASYS 3 board. The clock line is physically connected to pin "C17" of the Artix7 FPGA chip, and the data line is physically connected to pin "B17" of the chip.
Note that in connecting these pins in the XDC file, pull up need to be set true by using the additional comment of the form: ***set_property PULLUP true [get_ports PS2_CLK]*** after the PS/2 clock and PS/2 Data ports.

The FPGA mouse interface can be built from three modules: a "Transmitter" module, a "Receiver" module and a "State Machine" module to control the FPGA-Mouse communication, as shown in Figure 7.

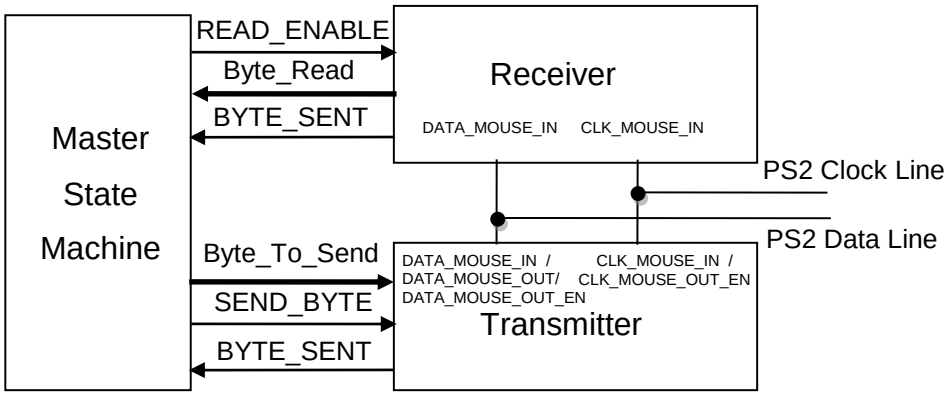


Figure 6. Mouse Interface: Simplified Block Diagram

The following shows Verilog code fragments for the "Receiver" module:

```
module MouseReceiver(
        //Standard Inputs
        input                           RESET,
        input                           CLK,
        //Mouse IO - CLK
        input                           CLK_MOUSE_IN,
        //Mouse IO - DATA
        input                           DATA_MOUSE_IN,
        //Control
        input                           READ_ENABLE,
        output          [7:0]           BYTE_READ,
        output          [1:0]           BYTE_ERROR_CODE,
        output                          BYTE_READY
  );

        //////////////////////////////////////////////////////////////
        // Clk Mouse delayed to detect clock edges
        reg ClkMouseInDly;
        always@(posedge CLK)
                ClkMouseInDly <= CLK_MOUSE_IN;

        //////////////////////////////////////////////////////////////
        //A simple state machine to handle the incoming 11-bit codewords
        reg     [2:0]   Curr_State, Next_State;
        reg     [7:0]   Curr_MSCodeShiftReg,   Next_MSCodeShiftReg;
        reg     [3:0]   Curr_BitCounter,  Next_BitCounter;
        reg             Curr_ByteReceived, Next_ByteReceived;
        reg     [1:0]   Curr_MSCodeStatus, Next_MSCodeStatus;
        reg     [15:0]  Curr_TimeoutCounter, Next_TimeoutCounter;


        //Sequential
        always@(posedge CLK) begin
                if(RESET) begin
                        Curr_State              <= 3'b000;
                        Curr_MSCodeShiftReg     <= 8'h00;
                        Curr_BitCounter         <= 0;
                        Curr_ByteReceived       <= 1'b0;
                        Curr_MSCodeStatus       <= 2'b00;
                        Curr_TimeoutCounter     <= 0;
                end else begin
                        Curr_State              <= Next_State;
                        Curr_MSCodeShiftReg     <= Next_MSCodeShiftReg;
                        Curr_BitCounter         <= Next_BitCounter;
```

```
                    Curr_ByteReceived           <= Next_ByteReceived;
                    Curr_MSCodeStatus           <= Next_MSCodeStatus;
                    Curr_TimeoutCounter         <= Next_TimeoutCounter;
                end
        end

        //Combinatorial
        always@* begin
                //defaults to make the State Machine more readable
                Next_State                  = Curr_State;
                Next_MSCodeShiftReg         = Curr_MSCodeShiftReg;
                Next_BitCounter             = Curr_BitCounter;
                Next_ByteReceived           = 1'b0;
                Next_MSCodeStatus           = Curr_MSCodeStatus;
                Next_TimeoutCounter         = Curr_TimeoutCounter + 1'b1;

                //The states
                case (Curr_State)
                3'b000:  begin
                    //Falling edge of Mouse clock and MouseData is low i.e. start bit
                    if(READ_ENABLE & ClkMouseInDly & ~CLK_MOUSE_IN & ~DATA_MOUSE_IN) begin
                        Next_State              = 3'b001;
                        Next_MSCodeStatus       = 2'b00;
                    end
                    Next_BitCounter             = 0;
                   end
                // Read successive byte bits from the mouse here
                3'b001:  begin
                   if(Curr_TimeoutCounter == 50000)  // 1ms timeout
                        Next_State              = 3'b000;
                   else if(Curr_BitCounter == 8) begin  // if last bit go to parity bit check
                        Next_State              = 3'b010;
                        Next_BitCounter         = 0;
                   end else if(ClkMouseInDly & ~CLK_MOUSE_IN) begin  //Shift Byte bits in
                        Next_MSCodeShiftReg[6:0]    = Curr_MSCodeShiftReg[7:1];
                        Next_MSCodeShiftReg[7]      = DATA_MOUSE_IN;
                        Next_BitCounter             = Curr_BitCounter + 1;
                        Next_TimeoutCounter         = 0;
                    end
                end
                //Check Parity Bit
                3'b010:  begin
                //Falling edge of Mouse clock and MouseData is odd parity
                     if(Curr_TimeoutCounter == 50000)
                        Next_State      = 3'b000;
                     else if(ClkMouseInDly & ~CLK_MOUSE_IN) begin
                        if (DATA_MOUSE_IN != ~^Curr_MSCodeShiftReg[7:0]) // Parity bit error
                                Next_MSCodeStatus[0] = 1'b1;
                        Next_BitCounter         = 0;
                        Next_State              = 3'b011;
                        Next_TimeoutCounter     = 0;
                        end
                    end

/*
Fill in the code for State 3'b011 to detect the Stop bit and set MSCodeStatus [1] accordingly, the final State 3’b100, as well as default State
*/
        ……..
        FILL IN THIS AREA
        …….

/*


                endcase
        end
```

```
	assign BYTE_READY		= Curr_ByteReceived;
	assign BYTE_READ		= Curr_MSCodeShiftReg;
	assign BYTE_ERROR_CODE	= Curr_MSCodeStatus;

endmodule
```

The following shows Verilog code fragments for the “Transmitter” module:

```
module MouseTransmitter(
	//Standard Inputs
	input			RESET,
	input			CLK,
	//Mouse IO - CLK
	input			CLK_MOUSE_IN,
	output			CLK_MOUSE_OUT_EN,  // Allows for the control of the Clock line
	//Mouse IO - DATA
	input			DATA_MOUSE_IN,
	output			DATA_MOUSE_OUT,
	output			DATA_MOUSE_OUT_EN,
	//Control
	input			SEND_BYTE,
	input		[7:0]	BYTE_TO_SEND,
	output			BYTE_SENT
  );

	//////////////////////////////////////////////////////////
	// Clk Mouse delayed to detect clock edges
	reg ClkMouseInDly;
	always@(posedge CLK)
		ClkMouseInDly <= CLK_MOUSE_IN;


	//////////////////////////////////////////////////////////
	//Now a state machine to control the flow of write data
	reg [3:0]	Curr_State,		Next_State;
	reg		Curr_MouseClkOutWE,	Next_MouseClkOutWE;
	reg		Curr_MouseDataOut,	Next_MouseDataOut;
	reg		Curr_MouseDataOutWE,	Next_MouseDataOutWE;
	reg [15:0]	Curr_SendCounter,	Next_SendCounter;
	reg		Curr_ByteSent,		Next_ByteSent;
	reg [7:0]	Curr_ByteToSend,	Next_ByteToSend;

	//Sequential
	always@(posedge CLK) begin
		if(RESET) begin
			Curr_State		<= 4'h0;
			Curr_MouseClkOutWE	<= 1'b0;
			Curr_MouseDataOut	<= 1'b0;
			Curr_MouseDataOutWE	<= 1'b0;
			Curr_SendCounter	<= 0;
			Curr_ByteSent		<= 1'b0;
			Curr_ByteToSend		<= 0;
		end else begin
			Curr_State		<= Next_State;
			Curr_MouseClkOutWE	<= Next_MouseClkOutWE;
			Curr_MouseDataOut	<= Next_MouseDataOut;
			Curr_MouseDataOutWE	<= Next_MouseDataOutWE;
			Curr_SendCounter	<= Next_SendCounter;
			Curr_ByteSent		<= Next_ByteSent;
			Curr_ByteToSend		<= Next_ByteToSend;
		end
	end

	//Combinatorial
```

```
always@* begin
        //default values
        Next_State              = Curr_State;
        Next_MouseClkOutWE      = 1'b0;
        Next_MouseDataOut       = 1'b0;
        Next_MouseDataOutWE     = Curr_MouseDataOutWE;
        Next_SendCounter        = Curr_SendCounter;
        Next_ByteSent           = 1'b0;
        Next_ByteToSend         = Curr_ByteToSend;

        case(Curr_State)
                //IDLE
                4'h0 :  begin
                  if(SEND_BYTE) begin
                        Next_State          = 4'h1;
                        Next_ByteToSend     = BYTE_TO_SEND;
                  end
                  Next_MouseDataOutWE       = 1'b0;
                end

                //Bring Clock line low for at least 100 microsecs i.e. 5000 clock cycles @ 50MHz
                4'h1 :  begin
                  if(Curr_SendCounter == 6000) begin
                        Next_State          = 4'h2;
                        Next_SendCounter    = 0;
                  end else
                        Next_SendCounter    = Curr_SendCounter + 1'b1;

                   Next_MouseClkOutWE = 1'b1;
                end

                //Bring the Data Line Low and release the Clock line
                4'h2 :  begin
                        Next_State          = 4'h3;
                        Next_MouseDataOutWE = 1'b1;
                end

                //Start Sending
                4'h3 :  begin   // change data at falling edge of clock, start bit = 0
                        if(ClkMouseInDly & ~CLK_MOUSE_IN)
                             Next_State     = 4'h4;
                        end

                //Send Bits 0 to 7 - We need to send the byte
                4'h4 :  begin   // change data at falling edge of clock
                           if(ClkMouseInDly & ~CLK_MOUSE_IN) begin
                                if(Curr_SendCounter == 7) begin
                                        Next_State          = 4'h5;
                                        Next_SendCounter    = 0;
                                end else
                                        Next_SendCounter    = Curr_SendCounter + 1'b1;
                           end

                          Next_MouseDataOut     = Curr_ByteToSend[Curr_SendCounter];
                        end

                //Send the parity bit
                4'h5 :  begin  // change data at falling edge of clock
                          if(ClkMouseInDly & ~CLK_MOUSE_IN)
                                Next_State              = 4'h6;

                          Next_MouseDataOut             = ~^Curr_ByteToSend[7:0];
                         end

                //Release Data line
                4'h6 :  begin
                          Next_State                    = 4'h7;
```

```
                                    Next_MouseDataOutWE                 = 1'b0;
                                end
/*
Wait for Device to bring Data line low, then wait for Device to bring Clock line low, and finally wait for Device to release both Data and Clock.

*/
        ……..
        FILL IN THIS AREA
        …….




                endcase
        end

        ////////////////////////////////////////////////////////////
        //Assign OUTPUTs
        //Mouse IO - CLK
        assign  CLK_MOUSE_OUT_EN            = Curr_MouseClkOutWE;

        //Mouse IO - DATA
        assign  DATA_MOUSE_OUT              = Curr_MouseDataOut;
        assign  DATA_MOUSE_OUT_EN           = Curr_MouseDataOutWE;

        //Control
        assign  BYTE_SENT                   = Curr_ByteSent;

endmodule
```

The following shows Verilog code fragments for the Master State Machine module.

```
module MouseMasterSM(
                input                           CLK,
                input                           RESET,
                //Transmitter Control
                output                          SEND_BYTE,
                output          [7:0]           BYTE_TO_SEND,
                input                           BYTE_SENT,
                //Receiver Control
                output                          READ_ENABLE,
                input           [7:0]           BYTE_READ,
                input           [1:0]           BYTE_ERROR_CODE,
                input                           BYTE_READY,
                //Data Registers
                output          [7:0]           MOUSE_DX,
                output          [7:0]           MOUSE_DY,
                output          [7:0]           MOUSE_STATUS,
                output                          SEND_INTERRUPT
  );


        ////////////////////////////////////////////////////////////
        //      Main state machine - There is a setup sequence
        //
        // 1) Send FF -- Reset command,
        // 2) Read FA -- Mouse Acknowledge,
        // 2) Read AA -- Self-Test Pass
        // 3) Read 00 -- Mouse ID
        // 4) Send F4 -- Start transmitting command,
        // 5) Read FA -- Mouse Acknowledge,
        //
        // If at any time this chain is broken, the SM will restart from
        // the beginning.  Once it has finished the set-up sequence, the read enable flag
```

```
// is raised.
// The host is then ready to read mouse information 3 bytes at a time:
// S1) Wait for first read, When it arrives, save it to Status. Goto S2.
// S2) Wait for second read, When it arrives, save it to DX. Goto S3.
// S3) Wait for third read, When it arrives, save it to DY. Goto S1.
//                  Send interrupt.

//State Control
reg     [3:0]   Curr_State,     Next_State;
reg     [23:0]  Curr_Counter,   Next_Counter;

//Transmitter Control
reg             Curr_SendByte,  Next_SendByte;
reg     [7:0]   Curr_ByteToSend, Next_ByteToSend;

//Receiver Control
reg             Curr_ReadEnable, Next_ReadEnable;

//Data Registers
reg     [7:0]   Curr_Status, Next_Status;
reg     [7:0]   Curr_Dx, Next_Dx;
reg     [7:0]   Curr_Dy, Next_Dy;
reg             Curr_SendInterrupt, Next_SendInterrupt;

//Sequential
always@(posedge CLK) begin
        if(RESET) begin
                        Curr_State              <= 4'h0;
                        Curr_Counter            <= 0;
                        Curr_SendByte           <= 1'b0;
                        Curr_ByteToSend         <= 8'h00;
                        Curr_ReadEnable         <= 1'b0;
                        Curr_Status             <= 8'h00;
                        Curr_Dx                 <= 8'h00;
                        Curr_Dy                 <= 8'h00;
                        Curr_SendInterrupt      <= 1'b0;
        end else begin
                        Curr_State              <= Next_State;
                        Curr_Counter            <= Next_Counter;
                        Curr_SendByte           <= Next_SendByte;
                        Curr_ByteToSend         <= Next_ByteToSend;
                        Curr_ReadEnable         <= Next_ReadEnable;
                        Curr_Status             <= Next_Status;
                        Curr_Dx                 <= Next_Dx;
                        Curr_Dy                 <= Next_Dy;
                        Curr_SendInterrupt      <= Next_SendInterrupt;
        end
end

//Combinatorial
always@* begin
        Next_State              = Curr_State;
        Next_Counter            = Curr_Counter;
        Next_SendByte           = 1'b0;
        Next_ByteToSend         = Curr_ByteToSend;
        Next_ReadEnable         = 1'b0;
        Next_Status             = Curr_Status;
        Next_Dx                 = Curr_Dx;
        Next_Dy                 = Curr_Dy;
        Next_SendInterrupt      = 1'b0;

        case(Curr_State)
                //Initialise State - Wait here for 10ms before trying to initialise the mouse.
                4'h0:   begin
                          if(Curr_Counter == 5000000) begin // 1/100th sec at 50MHz clock
                            Next_State    = 4'h1;
                            Next_Counter = 0;
```

```
            end else
              Next_Counter = Curr_Counter + 1'b1;
            end
//Start initialisation by sending FF
4'h1:     begin
            Next_State              = 4'h2;
            Next_SendByte           = 1'b1;
            Next_ByteToSend         = 8'hFF;
            end
//Wait for confirmation of the byte being sent
4'h2: begin
            if(BYTE_SENT)
              Next_State            = 4'h3;
        end
//Wait for confirmation of a byte being received
//If the byte is FA goto next state, else re-initialise.
4'h3: begin
            if(BYTE_READY) begin
              if((BYTE_READ == 8'hFA) & (BYTE_ERROR_CODE == 2'b00))
            Next_State              = 4'h4;
          else
                Next_State          = 4'h0;
        end

            Next_ReadEnable         = 1'b1;
       end
//Wait for self-test pass confirmation
//If the byte received is AA goto next state, else re-initialise
4'h4: begin
            if(BYTE_READY) begin
              if((BYTE_READ == 8'hAA) & (BYTE_ERROR_CODE == 2'b00))
            Next_State              = 4'h5;
          else
                Next_State          = 4'h0;
        end

            Next_ReadEnable         = 1'b1;
      end
//Wait for confirmation of a byte being received
//If the byte is 00 goto next state (MOUSE ID) else re-initialise
4'h5: begin
            if(BYTE_READY) begin
              if((BYTE_READ == 8'h00) & (BYTE_ERROR_CODE == 2'b00))
            Next_State              = 4'h6;
          else
                Next_State          = 4'h0;
        end

            Next_ReadEnable         = 1'b1;
       end
//Send F4 - to start mouse transmit
4'h6:     begin
            Next_State              = 4'h7;
            Next_SendByte           = 1'b1;
            Next_ByteToSend         = 8'hF4;
            end
//Wait for confirmation of the byte being sent
4'h7: if(BYTE_SENT)       Next_State        = 4'h8;
//Wait for confirmation of a byte being received
//If the byte is F4 goto next state, else re-initialise
4'h8: begin
            if(BYTE_READY) begin
              if(BYTE_READ == 8'hF4)
            Next_State              = 4'h9;
          else
                Next_State          = 4'h0;
        end
```

```
                    Next_ReadEnable         = 1'b1;
                end

            ////////////////////////////////////////////////////
            //At this point the SM has initialised the mouse.
            //Now we are constantly reading.  If at any time
            //there is an error, we will re-initialise
            //the mouse - just in case.
            ////////////////////////////////////////////////////

            //Wait for the confirmation of a byte being received.
            //This byte will be the first of three, the status byte.
            //If a byte arrives, but is corrupted, then we re-initialise
            4'h9:   begin
                      if(BYTE_READY & (BYTE_ERROR_CODE == 2'b00)) begin
                          Next_State        = 4'hA;
                          Next_Status       = BYTE_READ;
                      end else
                          Next_State        = 4'h0;

                          Next_Counter      = 0;
                          Next_ReadEnable   = 1'b1;
                    end
            //Wait for confirmation of a byte being received
            //This byte will be the second of three, the Dx byte.
            4'hA: begin

                      ……..
                        FILL IN THIS AREA
                    …….

                    end
            //Wait for confirmation of a byte being received
            //This byte will be the third of three, the Dy byte.
            4'hB: begin

                      ……..
                        FILL IN THIS AREA
                    …….

                    end
            //Send Interrupt State
            4'hC: begin
                    Next_State              = 4'h9;
                    Next_SendInterrupt      = 1'b1;
                end
            //Default State
            default:  begin
                    Next_State              = 4'h0;
                    Next_Counter            = 0;
                    Next_SendByte           = 1'b0;
                    Next_ByteToSend         = 8'hFF;
                    Next_ReadEnable         = 1'b0;
                    Next_Status             = 8'h00;
                    Next_Dx                 = 8'h00;
                    Next_Dy                 = 8'h00;
                    Next_SendInterrupt      = 1'b0;
                end
        endcase
end

////////////////////////////////////////////////
//Tie the SM signals to the IO

//Transmitter
assign SEND_BYTE                = Curr_SendByte;
```

```
        assign BYTE_TO_SEND         = Curr_ByteToSend;

        //Receiver
        assign READ_ENABLE          = Curr_ReadEnable;

        //Output Mouse Data
        assign MOUSE_DX             = Curr_Dx;
        assign MOUSE_DY             = Curr_Dy;
        assign MOUSE_STATUS         = Curr_Status;
        assign SEND_INTERRUPT       = Curr_SendInterrupt;

endmodule
```

Finally, the above three blocks should be connected in a "Mouse Transceiver" module as suggested in the code fragments below:

```
Module MouseTransceiver(
        //Standard Inputs
        input                                   RESET,
        input                                   CLK,
        //IO - Mouse side
        inout                                   CLK_MOUSE,
        inout                                   DATA_MOUSE,
        // Mouse data information
        output          [3:0]               MouseStatus,
        output          [7:0]               MouseX,
        output          [7:0]               MouseY
  );

        // X, Y Limits of Mouse Position e.g. VGA Screen with 160 x 120 resolution
       parameter [7:0]  MouseLimitX            = 160;
        parameter [7:0] MouseLimitY             = 120;


        //////////////////////////////////////////////////////////////////
        //TriState Signals
        //Clk
        reg ClkMouseIn;
        wire ClkMouseOutEnTrans;

        //Data
        wire DataMouseIn;
        wire DataMouseOutTrans;
        wire DataMouseOutEnTrans;

        //Clk Output - can be driven by host or device
        assign CLK_MOUSE      = ClkMouseOutEnTrans ? 1'b0 : 1'bz;

        //Clk Input
        assign DataMouseIn      = DATA_MOUSE;

        //Clk Output - can be driven by host or device
        assign DATA_MOUSE     = DataMouseOutEnTrans ? DataMouseOutTrans : 1'bz;

        //////////////////////////////////////////////////////////////////
        //This section filters the incoming Mouse clock to make sure that
        //it is stable before data is latched by either transmitter
        //or receiver modules
        reg [7:0]MouseClkFilter;
        always@(posedge CLK) begin

                if(RESET)
                        ClkMouseIn              <= 1'b0;
                else begin
                        //A simple shift register
                        MouseClkFilter[7:1]     <= MouseClkFilter[6:0];
```

```
                MouseClkFilter[0]       <= CLK_MOUSE;

                //falling edge
                if(ClkMouseIn & (MouseClkFilter == 8'h00))
                        ClkMouseIn      <= 1'b0;
                //rising edge
                else if(~ClkMouseIn & (MouseClkFilter == 8'hFF))
                        ClkMouseIn      <= 1'b1;
        end
end

//////////////////////////////////////////////////////
//Instantiate the Transmitter module
wire                    SendByteToMouse;
wire                    ByteSentToMouse;
wire    [7:0]           ByteToSendToMouse;
MouseTransmitter T(
                        //Standard Inputs
                        .RESET (RESET),
                        .CLK(CLK),
                        //Mouse IO - CLK
                        .CLK_MOUSE_IN(ClkMouseIn),
                        .CLK_MOUSE_OUT_EN(ClkMouseOutEnTrans),
                        //Mouse IO - DATA
                        .DATA_MOUSE_IN(DataMouseIn),
                        .DATA_MOUSE_OUT(DataMouseOutTrans),
                        .DATA_MOUSE_OUT_EN(DataMouseOutEnTrans),
                        //Control
                        .SEND_BYTE(SendByteToMouse),
                        .BYTE_TO_SEND(ByteToSendToMouse),
                        .BYTE_SENT(ByteSentToMouse)
                        );

//////////////////////////////////////////////////////
//Instantiate the Receiver module
wire                    ReadEnable;
wire    [7:0]           ByteRead;
wire    [1:0]           ByteErrorCode;
wire                    ByteReady;
MouseReceiver R(
                        //Standard Inputs
                        .RESET(RESET),
                        .CLK(CLK),
                        //Mouse IO - CLK
                        .CLK_MOUSE_IN(ClkMouseIn),
                        //Mouse IO - DATA
                        .DATA_MOUSE_IN(DataMouseIn),
                        //Control
                        .READ_ENABLE (ReadEnable),
                        .BYTE_READ(ByteRead),
                        .BYTE_ERROR_CODE(ByteErrorCode),
                        .BYTE_READY(ByteReady)
                        );

//////////////////////////////////////////////////////
//Instantiate the Master State Machine module
wire    [7:0]   MouseStatusRaw;
wire    [7:0]   MouseDxRaw;
wire    [7:0]   MouseDyRaw;
wire            SendInterrupt;
MouseMasterSM MSM(
                        //Standard Inputs
                        .RESET(RESET),
                        .CLK(CLK),
                        //Transmitter Interface
                        .SEND_BYTE(SendByteToMouse),
                        .BYTE_TO_SEND(ByteToSendToMouse),
```

```
                                .BYTE_SENT(ByteSentToMouse),
                                //Receiver Interface
                                .READ_ENABLE (ReadEnable),
                                .BYTE_READ(ByteRead),
                                .BYTE_ERROR_CODE(ByteErrorCode),
                                .BYTE_READY(ByteReady),
                                //Data Registers
                                .MOUSE_STATUS(MouseStatusRaw),
                                .MOUSE_DX(MouseDxRaw),
                                .MOUSE_DY(MouseDyRaw),
                                .SEND_INTERRUPT(SendInterrupt)
                                );


        //Pre-processing - handling of overflow and signs.
        //More importantly, this keeps tabs on the actual X/Y
        //location of the mouse.
        wire    signed  [8:0]   MouseDx;
        wire    signed  [8:0]   MouseDy;
        wire    signed  [8:0]   MouseNewX;
        wire    signed  [8:0]   MouseNewY;

        //DX and DY are modified to take account of overflow and direction
        assign MouseDx  = (MouseStatusRaw[6]) ? (MouseStatusRaw[4] ? {MouseStatusRaw[4],8'h00} :
{MouseStatusRaw[4],8'hFF} ) : {MouseStatusRaw[4],MouseDxRaw[7:0]};

        // assign the proper expression to MouseDy
                ……..
                   FILL IN THIS AREA
                …….

        //  calculate new mouse position
        assign MouseNewX = {1'b0,MouseX} + MouseDx;
        assign MouseNewY = {1'b0,MouseY} + MouseDy;

        always@(posedge CLK) begin
                if(RESET) begin
                        MouseStatus                     <= 0;
                        MouseX                          <= MouseLimitX/2;
                        MouseY                          <= MouseLimitY/2;
                end else if (SendInterrupt) begin
                        //Status is stripped of all unnecessary info
                        MouseStatus                     <= MouseStatusRaw[3:0];

                        //X is modified based on DX with limits on max and min
                        if(MouseNewX < 0)
                                MouseX                  <= 0;
                        else if(MouseNewX > (MouseLimitX-1))
                                MouseX                  <= MouseLimitX-1;
                        else
                                MouseX                  <= MouseNewX[7:0];

                        //Y is modified based on DY with limits on max and min
                                ……..
                                    FILL IN THIS AREA
                                …….

                end
        end

endmodule
```

Note how we dealt with bidirectional ports (inout's) needed here as the Data line and the Clock line are both written to and read from. In general, to make use of an inout port "Port_Inout_Example", you need to create an internal wire ("Port_Inout_Example_In") with

the value of the “Port_Inout_Example” port assigned to it. This can be used internally as an input to user logic when the port is in input mode. To write data to the port, we create an enable signal “Port_Inout_Example_Enable” which when set causes the output port to take the value of an internal signal “Port_Inout_Example_Out” from user logic. Otherwise, the inout port is set to high impedance. The following show Verilog code snippets to the above effect.

```
wire  Port_Inout_Example_In;

assign Port_Inout_Example_In = Port_Inout_Example;

// Use Port_Inout_Example_In as an input to your logic
….
….
// Internal signal Port_Inout_Example_Out can be assigned to the inout port if an enable
// signal  (Port_Inout_Example_Enable) is set

assign Port_Inout_Example = ( Port_Inout_Example_Enable ? Port_Inout_Example_Out : 1’bZ);
```

Assessment

You will need to complete the mouse interface code based on the above, synthesise and verify your code with simple test benches, generate an FPGA bitstream and test on the BASYS 3 board. The team member in charge of this peripheral development will be tested in Week 6. This individual assessment will account for 25% of your overall lab mark. During the test, you will need to demonstrate a functioning mouse interface on the BASYS 3 board through plugging a USB mouse to the board, and showing the various mouse register values (Status Byte, X Direction Byte, and Y Direction Byte) displayed on the LEDs and seven segment displays of the BASYS 3 board as the mouse is moved around. You will be supplied with the Verilog description of a seven segment display interface (SevenSeg). A specification of the latter can also be found in the Digital System Laboratory 3/A material. During the assessment, you will need to be able to take the assessor through your code answering any possible query about your design choices, coding style, etc. Your code will have been uploaded to Learn prior to the start of the laboratory session.

2. IR Transmitter Driver

The BASYS 3 board provides four 12-pin peripheral module connectors. Each is organised in two rows of 6-pins, with each row providing a power pin (Vdd @3.3V), a ground pin (GND), and four unique FPGA signals (see BASYS 3 reference manual, page 17). (Note that one of the connectors is designed for JXADC. Use any of the other three for this lab). Several 6-pin (or 12-pin) module boards that can attach to this connector are available from Digilent (see www.digilentinc.com for more information). For the purpose of this lab, we have developed our own 6-pin IR Transmitter module, which can be attached to any of the three peripheral module connectors. The following gives the circuit diagram of the IR Transmitter module.

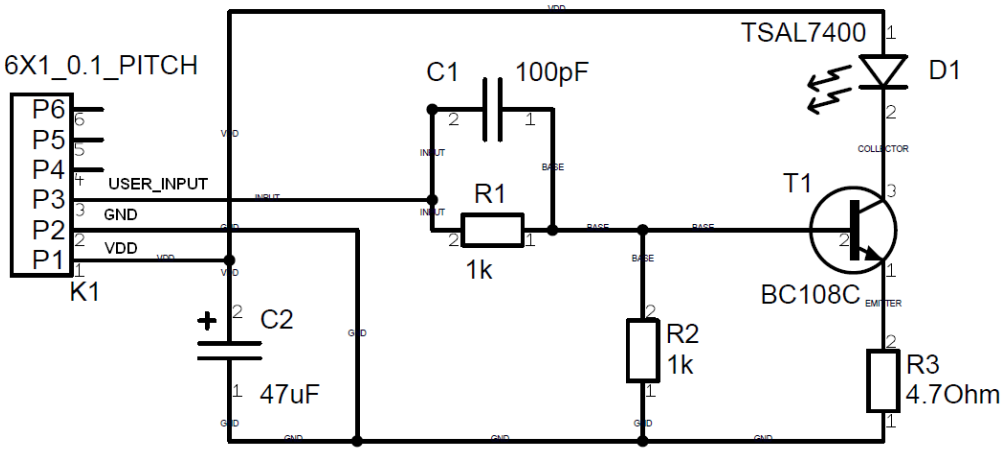


Figure 7. IR Transmitter circuit

2.1 What you are required to do

Your task is to generate the proper pulse codes to control the cars remotely using the Spartan-3 FPGA chip with the above IR Transmitter peripheral module. The following will present the pulse codes needed to control each car type: RED, BLUE, GREEN, YELLOW. In general, however, the pulse code for any car consists of the following generic packet, generated at 10Hz frequency.

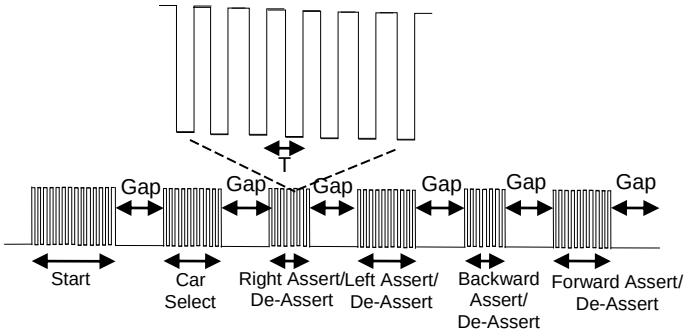


Figure 8. Car Pulse Code: Generic Packet

The following describes the packet details for each car type. These might vary from one car batch to another, so it is always wise to check the packet details using a logic analyser.

Blue-coded Cars:

In the case of blue-coded cars, the pulse frequency F = 1/T (see Figure 8 above) is equal to 36KHz. The following gives the number of pulses in each packet region as labelled in Figure 8:

- Start = 191
- Gap = 25
- CarSelect = 47 (NB. The number of pulses in this region identifies the car type i.e. RED-coded here)
- Right: If the right direction is to be asserted i.e. the car is to move to the right, the number of pulses here is set to 47, otherwise it is 22
- Left: If the left direction is to be asserted i.e. the car is to move to the left, the number of pulses here is set to 47, otherwise it is 22
- Backward: If the Backward direction is to be asserted i.e. the car is to move backwards, the number of pulses here is set to 47, otherwise it is 22
- Forward: If the Forward direction is to be asserted i.e. the car is to move forwards, the number of pulses here is set to 47, otherwise it is 22

Yellow-coded Cars:

In the case of yellow-coded cars, the pulse frequency F = 1/T (see figure above) is equal to 40KHz. The following gives the number of pulses in each packet region as labelled in Figure 8:

- Start = 88
- Gap = 40
- CarSelect = 22
- Right: If the right direction is to be asserted i.e. the car is to move to the right, the number of pulses here is set to 44, otherwise it is 22
- Left: If the left direction is to be asserted i.e. the car is to move to the left, the number of pulses here is set to 44, otherwise it is 22
- Backward: If the Backward direction is to be asserted i.e. the car is to move backwards, the number of pulses here is set to 44, otherwise it is 22
- Forward: If the Forward direction is to be asserted i.e. the car is to move forwards, the number of pulses here is set to 44, otherwise it is 22

Green-coded Cars:

In the case of green-coded cars, the pulse frequency F = 1/T (see figure above) is equal to 37.5 KHz. The following gives the number of pulses in each packet region as labelled in Figure 8:

- Start = 88
- Gap = 40
- CarSelect = 44
- Right: If the right direction is to be asserted i.e. the car is to move to the right, the number of pulses here is set to 44, otherwise it is 22
- Left: If the left direction is to be asserted i.e. the car is to move to the left, the number of pulses here is set to 44, otherwise it is 22
- Backward: If the Backward direction is to be asserted i.e. the car is to move backwards, the number of pulses here is set to 44, otherwise it is 22

- Forward: If the Forward direction is to be asserted i.e. the car is to move forwards, the number of pulses here is set to 44, otherwise it is 22

Red-coded Cars:

In the case of red-coded cars, the pulse frequency F = 1/T (see figure above) is equal to 36 KHz. The following gives the number of pulses in each packet region as labelled in Figure 8:

- Start = 192
- Gap = 24
- CarSelect = 24
- Right: If the right direction is to be asserted i.e. the car is to move to the right, the number of pulses here is set to 48, otherwise it is 24
- Left: If the left direction is to be asserted i.e. the car is to move to the left, the number of pulses here is set to 48, otherwise it is 24
- Backward: If the Backward direction is to be asserted i.e. the car is to move backwards, the number of pulses here is set to 48, otherwise it is 24
- Forward: If the Forward direction is to be asserted i.e. the car is to move forwards, the number of pulses here is set to 48, otherwise it is 24

Generating the necessary pulse codes for all of these variations requires a generic state machine with the following parameters:

```
parameter StartBurstSize        e.g. = 192;
parameter CarSelectBurstSize    e.g. = 24;
parameter GapSize               e.g. = 24;
parameter AsserBurstSize        e.g. = 48;
parameter DeAssertBurstSize     e.g. = 24;
```

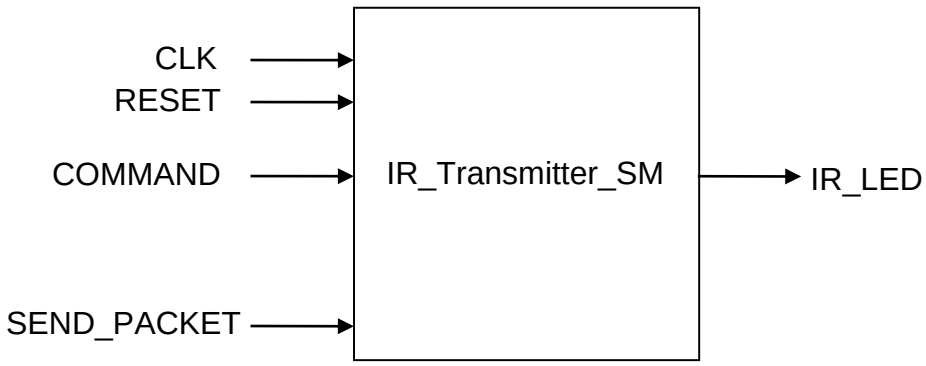


Figure 9. IR Transmitter State Machine

The state machine is illustrated in Figure 9 above where COMMAND consists of four bits: bit 0 to assert or de-assert the right direction, bit 1 to assert or de-assert the left direction, bit 2 to assert or de-assert the backward direction, and bit 3 to assert or de-assert the forward direction. Note that the state machine can be enabled by setting SEND_PACKET to '1' for one clock cycle, which will result in the generation of one single packet of those shown Figure 8. This has to be done ten times per second for the car to respond to the command in question. A 10Hz counter is thus needed as illustrated in Figure 10 below.

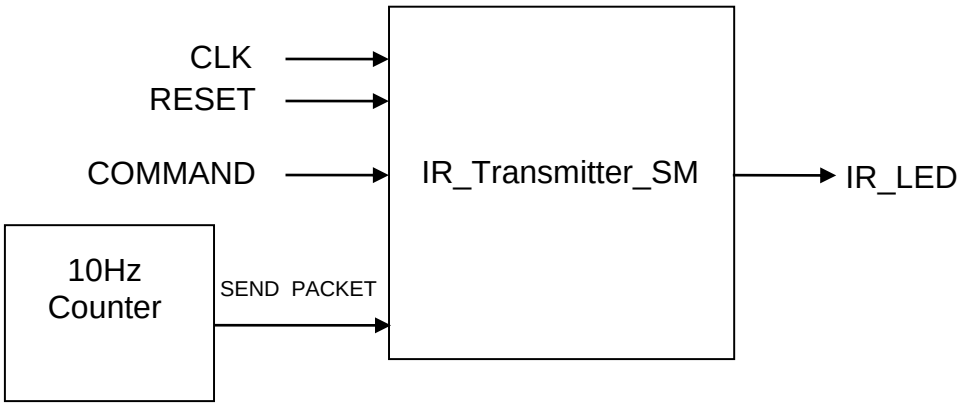


Figure 10. IR Transmitter Interface

The following shows code fragments for the IR Transmitter State Machine:

```
module IRTransmitterSM(
        //Standard Signals
        input           RESET,
        input           CLK,
        // Bus Interface Signals
        input [3:0]     COMMAND,
    input               SEND_PACKET,
        // IF LED signal
        output          IR_LED
  );

/*
Generate the pulse signal here from the main clock running at 50MHz to generate the right frequency for the car in question e.g. 40KHz for BLUE coded cars
*/
        ……..
        FILL IN THIS AREA
        …….

/*
Simple state machine to generate the states of the packet i.e. Start, Gaps, Right Assert or De-Assert, Left Assert or De-Assert, Backward Assert or De-Assert, and Forward Assert or De-Assert
*/

        ……..
        FILL IN THIS AREA
        …….


// Finally, tie the pulse generator with the packet state to generate IR_LED

        ……..
        FILL IN THIS AREA
        …….


endmodule
```

Assessment

You will need to complete your code based on the above, synthesise and verify your code with simple test benches, generate an FPGA bitstream and test on the BASYS 3 board.

For undergrauate students, the team member in charge of this peripheral development will be tested on Week 6 and this individual assessment will account for 25% of your overall lab mark. **For postgrauate students,** the group will be tested in Week 5 and this group assessment will account for 10% of your overall lab mark.

During the test, you will need to demonstrate a functioning car control interface on the BASYS 3 board through pressing buttons on the BASYS 3 board, which cause the remote car to move forward, backward, to the right and to the left. You will need to take the assessor through your code answering any possible query about your design choices, coding style, etc. Your code will have been uploaded to Learn prior to the start of the laboratory session.

3. VGA Interface

In order to display video onto a monitor screen, we will make use of the VGA port available on the BASYS 3 board. The latter uses 14 signals (carried by 14 FPGA pins, see Figure on page 11 of the BASYS 3 Reference Manual) with 12-bit colour and two synchronisation signals (HS – Horizontal Sync, and VS – Vertical Sync). HS and VS allow the monitor to align the incoming colour signals such that individual pixels on the screen are coloured correctly. Colours are derived by a combination of the three primary colours: red, blue and green. Four bits are used for each colour (given 16 possible levels of each colour).

However, it is possible to use only 8 of the 12-bit colours, especially for compatibility with an 8 bit bus processor architecture. In this configuration, three bits are used for Red and Green (giving 8 possible levels of each colour) and two bits are used for Blue (giving four possible levels only, as the human eye is less sensitive to blue than it is to red and green). You could either chose to implement an 8-bit colour (connecting only the 3 LSBs of Red and Green, and 2 LSBs of Blue), or implement the full 12 bits colour but you will have to send the colour information twice on an 8-bit bus. In this description, we target the 8-bit configuration.

You are required to develop an FPGA-based VGA driver which generates the necessary VGA signals (8 bit colour information and 2 synchronisation signals) to a VGA monitor using a frame buffer that holds frame pixel values. Pages 11- 14 of the manual give details of the VGA standard, providing much of the specification for this module - read it carefully. The following will take you through a suggested design of the VGA driver.

3.1 What you are required to do

VGA provides a resolution of 640 x 480 pixels (horizontal x vertical). A VGA driver could be built from two modules as shown in the following figure:

- "Frame_Buffer": A dual ported memory module which can be written to by an outside module e.g. a microprocessor, and read from by the "VGA_Sig_Gen" module (see Figure 11 below) for VGA display
- "VGA_Sig_Gen": VGA signal generator module, which reads consecutive pixel colours in raster pattern, from the "Frame_Buffer" module, and outputs the VGA signals through the proper FPGA pins

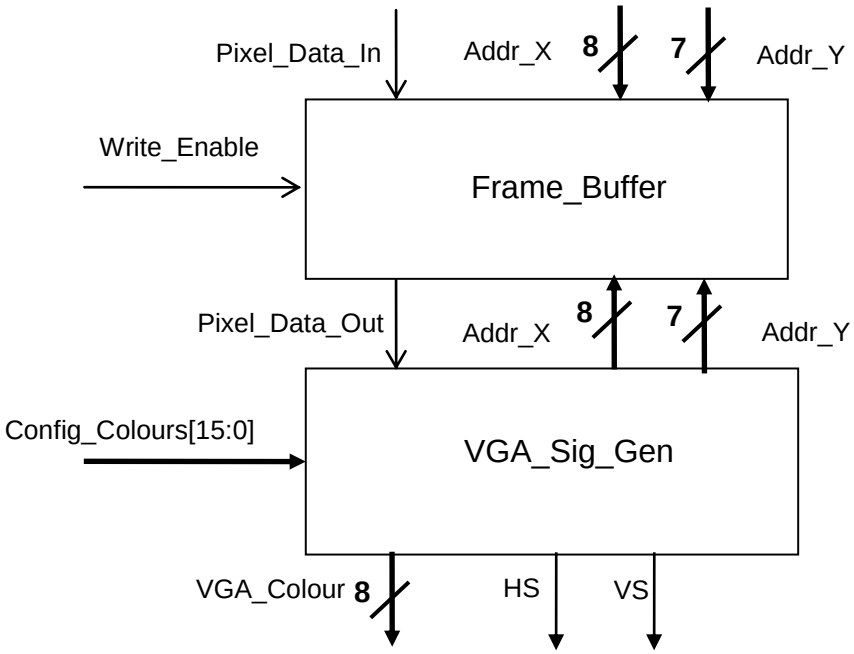


Figure 11. VGA Interface

Unfortunately, the FPGA chip on the BASYS 3 board does not have enough memory resources to store a full VGA frame with 8-bit pixels (i.e. 640 x 480 x 8 bits memory). In fact, few FPGA chips on the market can hold complete frames on-chip. Instead, off-chip memory (e.g. SDRAM) is often used to implement the frame buffer. The BASYS 3 board, however, does not have such external memory resource, which means that FPGA logic must be used to hold frame buffer data. To solve this problem, we can use a reduced colour resolution as well as a reduced pixel resolution to minimise memory storage requirements. In our suggested design, we will reduce the screen resolution by 4 in each direction i.e. each 4x4 pixel region on the screen would be represented by one single data element on the frame buffer, hence the above wordlength of the X and Y addresses in Figure 11 (i.e. 8 and 7 bits respectively instead of 10 and 9 bits for the 640 x 480 VGA resolution). We also reduce the colour resolution from 256 possible colours to just two colours (foreground and background) hence 1 bit data is sufficient for the colour information. A dual ported 256 x 128 1-bit memory should then be large enough to hold the whole frame buffer, which could be easily stored on the Spartan-3 chip. Note that in the proposed code below, the foreground and background colours can be set by an external module e.g. a microprocessor, through input Config_Colour[15:0] (see Figure 11 above) which represents 2 x 8 bits of data for two possible 8-bit colours.

The following shows possible Verilog code for the Frame_Buffer module:

```
module Frame_Buffer(
        /// Port A - Read/Write
        input                           A_CLK,
        input                           [14:0]    A_ADDR,  // 8 + 7 bits = 15 bits hence [14:0]
        input                           A_DATA_IN,  // Pixel Data In
        output   reg                    A_DATA_OUT,
        input                           A_WE,         // Write Enable
        //Port B - Read Only
        input                           B_CLK,
        input                           [14:0]    B_ADDR,   // Pixel Data Out
```

```
        output   reg                         B_DATA
   );

        // A 256 x 128 1-bit memory to hold frame data
        //The LSBs of the address correspond to the X axis, and the MSBs to the Y axis
        reg [0:0] Mem [2**15-1:0];

       // Port A  - Read/Write e.g. to be used by microprocessor
        always@(posedge A_CLK) begin
                if(A_WE)
                        Mem[A_ADDR] <= A_DATA_IN;

                A_DATA_OUT    <= Mem[A_ADDR];
        end

       //  Port B  - Read Only e.g. to be read from the VGA signal generator module for display
        always@(posedge B_CLK)
                B_DATA          <= Mem[B_ADDR];

endmodule
```

The following shows Verilog code fragments for the VGA_Sig_Gen module:

```
module VGA_Sig_Gen(
        input                           CLK,
        //Colour Configuration Interface
        input           [15:0]          CONFIG_COLOURS,
        // Frame Buffer (Dual Port memory) Interface
        output                          DPR_CLK,
        output  [14:0]                  VGA_ADDR,
        input                           VGA_DATA,
        //VGA Port Interface
        output  reg                     VGA_HS,
        output  reg                     VGA_VS,
        output  [7:0]                   VGA_COLOUR
   );


        //Halve the clock to 25MHz to drive the VGA display
        reg VGA_CLK;
        always@(posedge CLK) begin
            if(RESET)
                VGA_CLK <= 0;
            else
                VGA_CLK <= ~VGA_CLK;
         end
/*
Define VGA signal parameters e.g. Horizontal and Vertical display time, pulse widths, front and back porch widths etc.
*/

       // Use the following signal parameters
        parameter HTs           = 800;    // Total Horizontal Sync Pulse  Time
        parameter HTpw          = 96;     // Horizontal Pulse Width Time
        parameter HTDisp        = 640;    // Horizontal Display Time
        parameter Hbp           = 48;     // Horizontal Back Porch Time
        parameter Hfp           = 16;     // Horizontal Front Porch Time

        parameter VTs           = 521;    // Total Vertical Sync Pulse  Time
        parameter VTpw          = 2;      // Vertical Pulse Width Time
        parameter VTDisp        = 480;    // Vertical Display Time
        parameter Vbp           = 29;     // Vertical Back Porch Time
        parameter Vfp           = 10;     // Vertical Front Porch Time

        ……..
        FILL IN THIS AREA
```

```
        ……..


        // Define Horizontal and Vertical Counters to generate the VGA signals

        reg [9:0]       HCounter;

        reg [9:0]       VCounter;

/*
Create a process that assigns the proper horizontal and vertical counter values for raster scan of the display.
*/

        ……..
        FILL IN THIS AREA
        ……..


/*
Need to create the address of the next pixel. Concatenate and tie the look ahead address to the frame buffer address.
*/

        assign DPR_CLK      = VGA_CLK;
        assign VGA_ADDR     = {VCounter[8:2], HCounter[9:2]};



/*
Create a process that generates the horizontal and vertical synchronisation signals, as well as the pixel colour information, using HCounter and VCounter. Do not forget to use CONFIG_COLOURS input to display the right foreground and background colours.
*/
        ……..
        FILL IN THIS AREA
        ……..


/*
Finally, tie the output of the frame buffer to the colour output VGA_COLOUR.
*/
        ……..
        FILL IN THIS AREA
        ……..

endmodule
```

Note that the VGA standard was designed for CRT monitors, and hence it is important to be aware of the timing of the original system, even if we are using a digital display. It should also be noted that in addition to the HS and VS timing definitions within the reference manual, the 8-bit colour signal needs to be set to 8'h00 (zero) whilst not during the display time.

You will need to complete your code based on the above, synthesise and verify your code with simple test benches, generate an FPGA bitstream and test on the BASYS 3 board. Test the above driver block by packaging it within a top level module that generates frame buffer and colour configuration data, and outputs the VGA signals to the proper FPGA pins. Display the following chequered image on the VGA display, changing colours every one second.

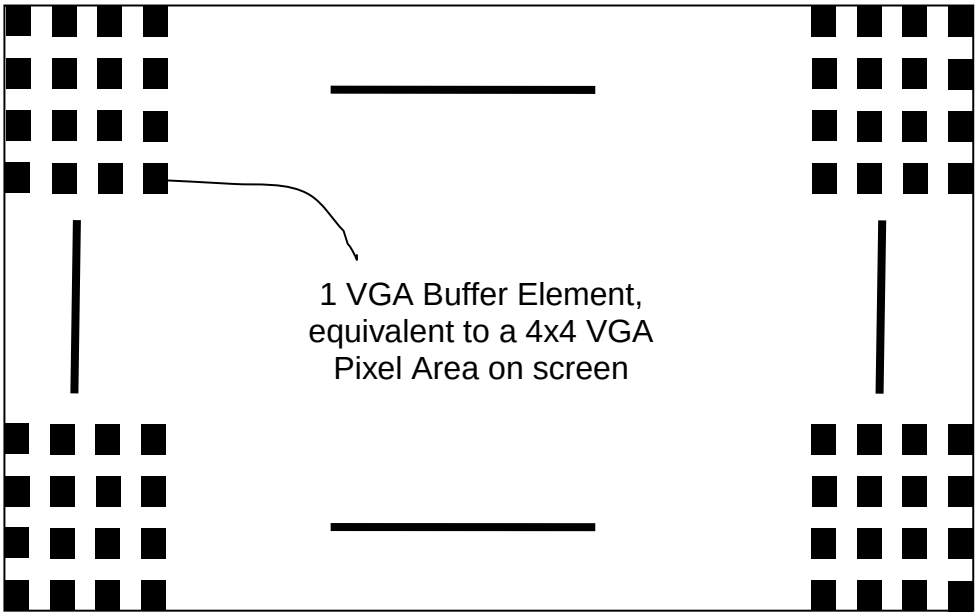


Figure 12. Chequered image to be displayed on the VGA screen

Assessment

The team member in charge of this peripheral development will be tested on Week 6. This individual assessment will account for 25% of your overall lab mark. During the test, you will need to demonstrate a functioning VGA interface on the BASYS 3 board through plugging a VGA monitor to the BASYS 3 board, and showing the above chequered image, changing colours every one second. You will need to take the assessor through your code answering any possible query about your design choices, coding style, etc. Your code will have been uploaded to Learn prior to the start of the laboratory session.

4. Microprocessor-based System for Remote Car Control

The following figure gives a block diagram of the overall FPGA-based system for remote car control. At its heart is a microprocessor which will be the master of an 8-bit bus to which various peripherals (slaves) will be connected.

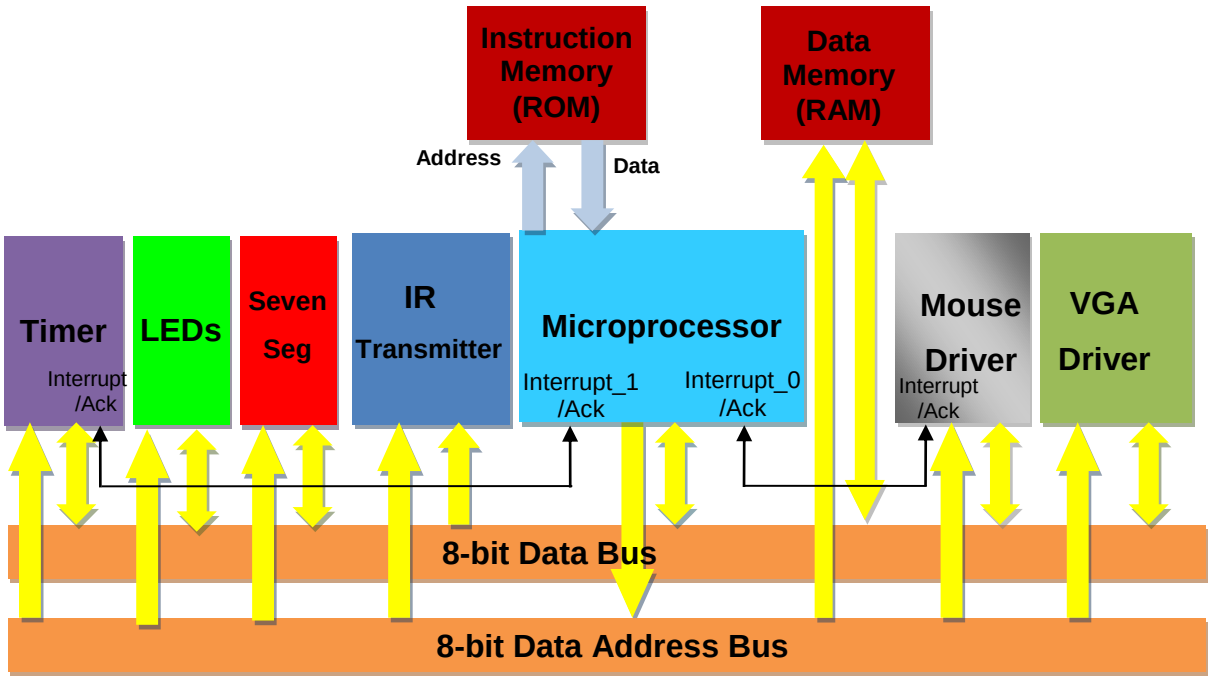


Figure 13. Microprocessor-based system block architecture

The following presents the memory mapping of the peripherals on the microprocessor's data address bus.

Peripheral	Base Address	High Address
IR Transmitter	0x90	0x90
Mouse	0xA0	0xA2
VGA	0xB0	0xB2
LEDs	0xC0	0xC0
SevenSeg	0xD0	0xD1
Timer	0xF0	0xF3

Table 1. Peripheral Memory Address Mapping

Note that the proposed microprocessor has a Harvard architecture which means that the instruction memory and data memory are distinct with separate address and data busses for each of them. Parallel accesses to both instructions and program data can thus be made, resulting in higher performance compared to Von-Neumann processors where instructions and data are stored on the same memory.

In our architecture, the data memory consists of 128 bytes and is mapped to the microprocessor's address space as follow:

Data Memory	Base Address	High Address
	0x00	0x7F

Table 2. Data Memory Address Mapping

Note that in advanced microprocessor systems, data memory is given its own high speed local bus, with a separate peripheral bus for relatively slower devices. In our case, data memory shares the same bus as the peripherals since high performance is not a major concern for us in this lab.

The following presents a suggested Verilog design of the data memory:

```
module RAM(
        //standard signals
        input                                   CLK,
        //BUS signals
        inout           [7:0]                   BUS_DATA,
        input           [7:0]                   BUS_ADDR,
        input                                   BUS_WE
  );

        parameter RAMBaseAddr       = 0;
        parameter RAMAddrWidth      = 7; // 128 x 8-bits memory

        //Tristate
        wire    [7:0]           BufferedBusData;
        reg     [7:0]           Out;
        reg                     RAMBusWE;

        //Only place data on the bus if the processor is NOT writing, and it is addressing this memory
        assign BUS_DATA         = (RAMBusWE) ? Out : 8'hZZ;
        assign BufferedBusData = BUS_DATA;

        //Memory
        reg [7:0] Mem [2**RAMAddrWidth-1:0];

        // Initialise the memory for data preloading, initialising variables, and declaring constants
        initial   $readmemh("Complete_Demo_RAM.txt", Mem);

        //single port ram
        always@(posedge CLK) begin
                // Brute-force RAM address decoding. Think of a simpler way...
                if((BUS_ADDR >= RAMBaseAddr) & (BUS_ADDR < RAMBaseAddr + 128)) begin
                        if(BUS_WE) begin
                                Mem[BUS_ADDR[6:0]] <= BufferedBusData;
                                RAMBusWE <= 1'b0;
                        end else
                                RAMBusWE <= 1'b1;
                end else
                        RAMBusWE <= 1'b0;

                Out     <= Mem[BUS_ADDR[6:0]];
        end

endmodule
```

The instruction memory has its own data and address bus, hence no decoding is necessary. In fact, this is a point to point connection. The following presents a suggested Verilog design of the instruction memory, which consists of 256 bytes and is read-only (ROM).

```
module ROM(
        //standard signals
        input                           CLK,
        //BUS signals
        output  reg     [7:0]           DATA,
        input           [7:0]           ADDR
  );

        parameter RAMAddrWidth          = 8;

        //Memory
        reg [7:0] ROM [2**RAMAddrWidth-1:0];

        // Load program
        initial    $readmemh("Complete_Demo_ROM.txt", ROM);

        //single port ram
        always@(posedge CLK)
                DATA <= ROM[ADDR];


endmodule
```

4.1 **Interrupts**

In our system architecture presented in Figure 13 above, interrupts come from two possible sources: 1) the mouse, when it is moved or clicked, and 2) the timer, which outputs an interrupt signal every one second. In general, an interrupt is an asynchronous signal from hardware indicating the need for attention, or a synchronous event in software indicating the need for a change in execution. Multiple hardware interrupts can be ORed or serviced through an interrupt controller which acts as another bus peripheral whose task is to service interrupts before they are sent to the microprocessor. This includes dealing with priorities for instance. In this lab, interrupts will be served on a "first-come first-served" basis. If two interrupts arrive at exactly the same time, the mouse interrupt will be dealt with first.

When an interrupt request is received by the microprocessor, the current program execution is suspended after the execution of the current instruction. The context information is then saved so that execution can return to the current program after interrupt servicing. In the case of our microprocessor, the context consists in the address of the next instruction to be executed by the interrupted process. After context saving, execution is transferred to an interrupt handler to service the interrupt. The start (or base) address of the interrupt handler's service routine is usually predefined, for each interrupt line, in a particular memory address. In our case, the memory address where the base address of the mouse interrupt handler's service routine is stored is "FF", whereas that of the timer is "FE". This means that whenever a mouse interrupt is received, for instance, the microprocessor fetches the base address of the mouse interrupt handler's service routine to be executed from address "FF". In other words, the content of address "FF" will be the address of the first instruction to be executed by the interrupt handler's service routine.

The following gives you a possible Verilog code to implement the Timer peripheral. Code for the seven segment display peripheral has already been given to you as part of the mouse interface module (you would need to wrap your existing peripheral codes with bus interfaces though), while the LEDs peripheral is straightforward to implement.

```
module Timer(
        //standard signals
        input                                   CLK,
        input                                   RESET,
        //BUS signals
        inout           [7:0]                   BUS_DATA,
        input           [7:0]                   BUS_ADDR,
        input                                   BUS_WE,
        output                                  BUS_INTERRUPT_RAISE,
        input                                   BUS_INTERRUPT_ACK
  );

        parameter [7:0] TimerBaseAddr = 8'hF0;  // Timer Base Address in the Memory Map
        parameter InitialInterruptRate = 100;  // Default interrupt rate leading to 1 interrupt every 100 ms
        parameter InitialInterruptEnable = 1'b1;  // By default the Interrupt is Enabled

        //////////////////////
        //BaseAddr + 0 -> reports current timer value
        //BaseAddr + 1 -> Address of a timer interrupt interval register, 100 ms by default
        //BaseAddr + 2 -> Resets the timer, restart counting from zero
        //BaseAddr + 3 -> Address of an interrupt Enable register, allows the microprocessor to disable
         //                  the timer

        //This module will raise an interrupt flag when the designated time is up.  It will
        //automatically set the time of the next interrupt to the time of the last interrupt plus
        //a configurable value (in milliseconds).

        //Interrupt Rate Configuration - The Rate is initialised to 100 by the parameter above, but can
        //also be set by the processor by writing to mem address BaseAddr + 1;
        reg [7:0] InterruptRate;
        always@(posedge CLK) begin
                if(RESET)
                        InterruptRate <= InitialInterruptRate;
                else    if((BUS_ADDR == TimerBaseAddr + 8'h01) & BUS_WE)
                        InterruptRate <= BUS_DATA;
        end

        //Interrupt Enable Configuration - If this is not set to 1, no interrupts will be
        //created.
        reg InterruptEnable;
        always@(posedge CLK) begin
                if(RESET)
                        InterruptEnable <= InitialInterruptEnable;
                else    if((BUS_ADDR == TimerBaseAddr + 8'h03) & BUS_WE)
                        InterruptEnable <= BUS_DATA[0];
        end


        //First we must lower the clock speed from 50MHz to 1 KHz (1ms period)
        reg [31:0] DownCounter;
        always@(posedge CLK) begin
                if(RESET)
                        DownCounter <= 0;
                else begin
                        if(DownCounter == 32'd49999)
                                DownCounter <= 0;
                        else
                                DownCounter <= DownCounter + 1'b1;
                end
        end


        //Now we can record the last time an interrupt was sent, and add a value to it to determine if it is
        // time to raise the interrupt.

        // But first, let us generate the 1ms counter (Timer)
```

```
	reg [31:0] Timer;
	always@(posedge CLK) begin
		if(RESET | (BUS_ADDR == TimerBaseAddr + 8'h02))
			Timer <= 0;
		else begin
			if((DownCounter == 0))
				Timer <= Timer + 1'b1;
			else
				Timer <= Timer;
		end
	end

	//Interrupt generation
	reg TargetReached;
	reg [31:0] LastTime;
	always@(posedge CLK) begin
		if(RESET) begin
			TargetReached		<= 1'b0;
			LastTime		<= 0;
		end else if((LastTime + InterruptRate) == Timer) begin
			if(InterruptEnable)
				TargetReached	<= 1'b1;
			LastTime	<= Timer;
		end else
			TargetReached		<= 1'b0;

	end

	//Broadcast the Interrupt
	reg Interrupt;
	always@(posedge CLK) begin
		if(RESET)
			Interrupt <= 1'b0;
		else if(TargetReached)
			Interrupt <= 1'b1;
		else if(BUS_INTERRUPT_ACK)
			Interrupt <= 1'b0;
	end

	assign BUS_INTERRUPT_RAISE = Interrupt;

	//Tristate output for interrupt timer output value
	reg TransmitTimerValue;
	always@(posedge CLK) begin
		if(BUS_ADDR == TimerBaseAddr)
			TransmitTimerValue	<= 1'b1;
		else
			TransmitTimerValue	<= 1'b0;
	end

	assign BUS_DATA = (TransmitTimerValue) ? Timer[7:0] : 8'hZZ;

endmodule
```

4.2 **Microprocessor architecture**

The microprocessor that you are required to design is an 8-bit processor with a load-store architecture whereby two internal registers (A and B) are used to hold operands and operation results. Central to the processor is an Arithmetic and Logic Unit (ALU) which can perform the following operations:

- Add two operands A, B
- Subtract two operands A, B

- Multiply two operands A, B
- Left shift operand A (by one bit)
- Right shift operand A (by one bit)
- Increment operand A
- Increment operand B
- Decrement operand A
- Decrement operand B
- Check if two operands A, B are equal (output 0x01 if that's the case, otherwise 0x00)
- Check if A>B (output 0x01 if that's the case, otherwise 0x00)
- Check if A<B (output 0x01 if that's the case, otherwise 0x00)

An operation code of four bits dictates which of these operations is performed (see Figure 14 below).

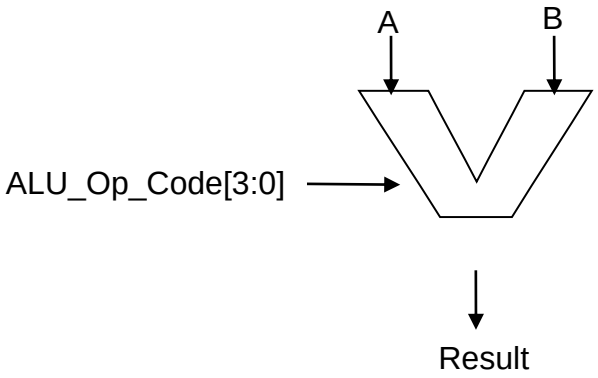


Figure 14. Arithmetic and Logic Unit (ALU)

A possible Verilog code to implement the above ALU is given to you below.

```
module ALU(
        //standard signals
        input                                   CLK,
        input                                   RESET,
        //I/O
        input           [7:0]                   IN_A,
        input           [7:0]                   IN_B,
        input           [3:0]                   ALU_Op_Code,
        output  [7:0]                           OUT_RESULT
        );

        reg [7:0] Out;
        //Arithmetic Computation
        always@(posedge CLK) begin
                if(RESET)
                        Out             <= 0;
                else begin
                        case (ALU_Op_Code)
                                //Maths Operations
                                //Add A + B
                                4'h0:           Out <= IN_A + IN_B;
                                //Subtract A - B
                                4'h1:           Out <= IN_A - IN_B;
                                //Multiply A * B
                                4'h2:           Out <= IN_A * IN_B;
                                //Shift Left A << 1
```

```
                        4'h3:           Out <= IN_A << 1;
                        //Shift Right A >> 1
                        4'h4:           Out <= IN_A >> 1;
                        //Increment A+1
                        4'h5:           Out <= IN_A + 1'b1;
                        //Increment B+1
                        4'h6:           Out <= IN_B + 1'b1;
                        //Decrement A-1
                        4'h7:           Out <= IN_A - 1'b1;
                        //Decrement B+1
                        4'h8:           Out <= IN_B - 1'b1;
                        // In/Equality Operations
                        //A == B
                        4'h9:           Out <= (IN_A == IN_B) ? 8'h01 : 8'h00;
                        //A > B
                        4'hA:           Out <= (IN_A > IN_B) ? 8'h01 : 8'h00;
                        //A < B
                        4'hB:           Out <= (IN_A < IN_B) ? 8'h01 : 8'h00;
                        //Default A
                        default: Out <= IN_A;
                    endcase
                end
        end

        assign OUT_RESULT = Out;

endmodule
```

The microprocessor has 6 types of instructions overall:

1. Read from memory to Register A or B
2. Write to memory from Register A or B
3. Do an ALU operation and save the result in register A or B
4. Jump operations: *conditional jump* depending on whether register A's content is equal, greater than or less than register B's content, or *unconditional jump* to a particular address.
5. Function call and return
6. Dereference Register A or B

Different instructions have different word lengths, but they are always issued in consecutive bytes. The following table presents the instruction set architecture in more detail.

Instruction	**Function**	**Instruction Structure**
A <- [Mem]	Read value from memory address Mem and store in register A	2 bytes: x x x x | 0 0 0 0 Mem
B <- [Mem]	Read value from memory address Mem and store in register B	2 bytes: x x x x | 0 0 0 1 Mem
[Mem] <- A	Write value of register A to memory address Mem	2 bytes: x x x x | 0 0 1 0 Mem
[Mem] <- B	Write value of register B to memory address Mem	2 bytes: x x x x | 0 0 1 1

		Mem
A <- ALU_OP(A,B)	Do Math ALU operation of type ALU_Op_Code on register value(s) and store result in register A	1 byte ALU_Op_Code | 0 1 0 0
B <- ALU_OP(A,B)	Do Math ALU operation of type ALU_Op_Code on register value(s) and store result in register B	1 byte ALU_Op_Code | 0 1 0 1
BREQ ADDR	Branch to address ADDR i.e. load program counter with ADDR if register A's content is equal to Register B's	2 bytes: 1 0 0 1 | 0 1 1 0 Mem
BGTQ ADDR	Branch to address ADDR i.e. load program counter with ADDR if register A's content is greater than Register B's	2 bytes: 1 0 1 0 | 0 1 1 0 Mem
BLTQ ADDR	Branch to address ADDR i.e. load program counter with ADDR if register A's content is less than Register B's	2 bytes: 1 0 1 1 | 0 1 1 0 Mem
GOTO ADDR	Branch to address ADDR i.e. load program counter with ADDR	2 bytes: x x x x | 0 1 1 1 ADDR
GOTO_IDLE	Go to Idle State and wait for Interrupts	1 byte x x x x | 1 0 0 0
FUNCTION_CALL ADDR	Branch to memory address ADDR. Save the next program address to execute from after returning from the function (program context)	2 bytes: x x x x | 1 0 0 1 ADDR
RETURN	Returns from a function call i.e. loads program context to the program counter for next instruction execution	1 byte x x x x | 1 0 1 0
Dereference A	Read memory address given by the value of register A and set the result as the new register A value A <- [A]	1 byte x x x x | 1 0 1 1
Dereference B	Read memory address given by the value of register B and set the result as the new register B value B <- [B]	1 byte x x x x | 1 1 0 0

Table 3. Microprocessor's Instruction Set

Such a small instruction set with direct hardware support for its implementation is referred to as a RISC architecture (RISC = Reduced Instruction Set Computing), as opposed to CISC architecture (CISC = Complex Instruction Set Computing) whereby the instruction set is large and complex with further microcode translation needed for complex instruction execution. RISC architectures are dominant nowadays as they are simpler and faster in hardware.

The proposed microprocessor's behaviour is essentially a state machine, with one sequential pipeline of states for each operation as shown in the following figure.

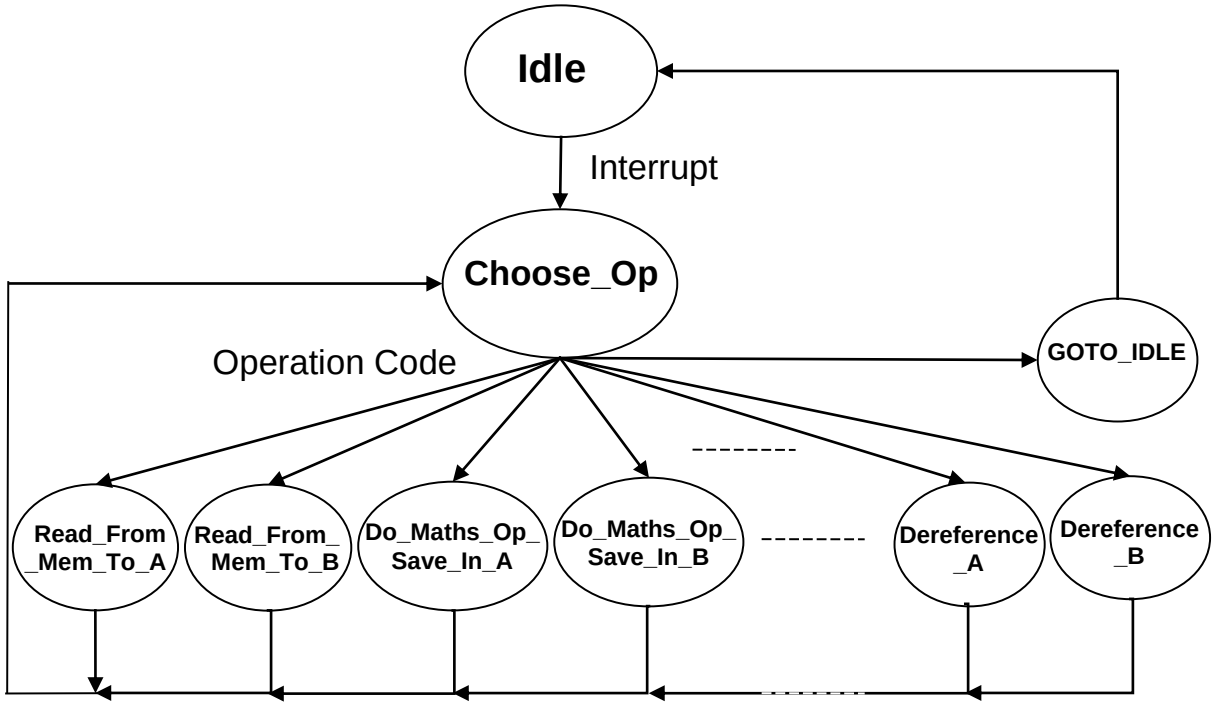


Figure 15. Microprocessor's state machine

Based on the above, the following shows suggested Verilog code fragments for the microprocessor's behaviour.

```
module Processor(
        //Standard Signals
        input                           CLK,
        input                           RESET,
        //BUS Signals
        inout           [7:0]           BUS_DATA,
        output          [7:0]           BUS_ADDR,
        output                          BUS_WE,
        // ROM signals
        output          [7:0]           ROM_ADDRESS,
        input           [7:0]           ROM_DATA,
        // INTERRUPT signals
        input           [1:0]           BUS_INTERRUPTS_RAISE,
        output   [1:0]                  BUS_INTERRUPTS_ACK
  );

        //The main data bus is treated as tristate, so we need a mechanism to handle this.
        //Tristate signals that interface with the main state machine
        wire [7:0]      BusDataIn;
        reg [7:0]       CurrBusDataOut,     NextBusDataOut;
        reg             CurrBusDataOutWE, NextBusDataOutWE;

        //Tristate Mechanism
        assign BusDataIn        = BUS_DATA;
        assign BUS_DATA         = CurrBusDataOutWE ? CurrBusDataOut : 8'hZZ;
        assign BUS_WE           = CurrBusDataOutWE;

        //Address of the bus
        reg [7:0] CurrBusAddr, NextBusAddr;
        assign BUS_ADDR = CurrBusAddr;

        //The processor has two internal registers to hold data between operations, and a third to hold
```

```
//the current program context when using function calls.
reg [7:0] CurrRegA, NextRegA;
reg [7:0] CurrRegB, NextRegB;
reg       CurrRegSelect, NextRegSelect;
reg [7:0] CurrProgContext, NextProgContext;

//Dedicated Interrupt output lines - one for each interrupt line
reg [1:0] CurrInterruptAck, NextInterruptAck;
assign BUS_INTERRUPTS_ACK = CurrInterruptAck;

//Instantiate program memory here
//There is a program counter which points to the current operation.  The program counter
//has an offset that is used to reference information that is part of the current operation
reg [7:0] CurrProgCounter, NextProgCounter;
reg [1:0] CurrProgCounterOffset, NextProgCounterOffset;
wire [7:0] ProgMemoryOut;
wire [7:0] ActualAddress;
assign ActualAddress = CurrProgCounter + CurrProgCounterOffset;
// ROM signals
assign ROM_ADDRESS  = ActualAddress;
assign ProgMemoryOut   = ROM_DATA;

//Instantiate the ALU
//The processor has an integrated ALU that can do several different operations
wire [7:0] AluOut;
ALU ALU0(
        //standard signals
        .CLK(CLK),
        .RESET(RESET),
        //I/O
        .IN_A(CurrRegA),
        .IN_B(CurrRegB),
        .IN_OPP_TYPE(ProgMemoryOut[7:4]),
        .OUT_RESULT(AluOut)
);

//The microprocessor is essentially a state machine, with one sequential pipeline
//of states for each operation.
//The current list of operations is:
// 0: Read from memory to A
// 1: Read from memory to B
// 2: Write to memory from A
// 3: Write to memory from B
// 4: Do maths with the ALU, and save result in reg A
// 5: Do maths with the ALU, and save result in reg B
// 6: if A (== or < or > B) GoTo ADDR
// 7: Goto ADDR
// 8: Go to IDLE
// 9: End thread, goto idle state and wait for interrupt.
// 10: Function call
// 11: Return from function call
// 12: Dereference A
//13: Dereference B

parameter [7:0]   //Program thread selection
IDLE                            = 8'hF0, //Waits here until an interrupt wakes up the processor.
GET_THREAD_START_ADDR_0  = 8'hF1, //Wait.
GET_THREAD_START_ADDR_1  = 8'hF2, //Apply the new address to the program counter.
GET_THREAD_START_ADDR_2  = 8'hF3, //Wait. Goto ChooseOp.

//Operation selection
//Depending on the value of ProgMemOut, goto one of the instruction start states.
CHOOSE_OPP                      = 8'h00,

//Data Flow
READ_FROM_MEM_TO_A        = 8'h10, //Wait to find what address to read, save reg select.
READ_FROM_MEM_TO_B        = 8'h11, //Wait to find what address to read, save reg select.
```

```
        READ_FROM_MEM_0             = 8'h12, //Set BUS_ADDR to designated address.
        READ_FROM_MEM_1             = 8'h13, //wait - Increments program counter by 2. Reset offset.
        READ_FROM_MEM_2             = 8'h14, //Writes memory output to chosen register, end op.
        WRITE_TO_MEM_FROM_A         = 8'h20, //Reads Op+1 to find what address to Write to.
        WRITE_TO_MEM_FROM_B         = 8'h21, //Reads Op+1 to find what address to Write to.
        WRITE_TO_MEM_0              = 8'h22, //wait - Increments program counter by 2. Reset offset.

        //Data Manipulation
        DO_MATHS_OPP_SAVE_IN_A      = 8'h30, //The result of maths op. is available, save it to Reg A.
        DO_MATHS_OPP_SAVE_IN_B      = 8'h31, //The result of maths op. is available, save it to Reg B.
        DO_MATHS_OPP_0              = 8'h32, //wait for new op address to settle. end op.

/*
Complete the above parameter list for In/Equality, Goto Address, Goto Idle, function start, Return from function, and Dereference operations.
*/
        ……..

        FILL IN THIS AREA

        …….

        //Sequential part of the State Machine.

        reg [7:0] CurrState, NextState;

        always@(posedge CLK) begin
        if(RESET) begin
                CurrState               = 8'h00;
                CurrProgCounter         = 8'h00;
                CurrProgCounterOffset   = 2'h0;
                CurrBusAddr             = 8'hFF; //Initial instruction after reset.
                CurrBusDataOut          = 8'h00;
                CurrBusDataOutWE        = 1'b0;
                CurrRegA                = 8'h00;
                CurrRegB                = 8'h00;
                CurrRegSelect           = 1'b0;
                CurrProgContext         = 8'h00;
                CurrInterruptAck        = 2'b00;
            end else begin
                CurrState               = NextState;
                CurrProgCounter         = NextProgCounter;
                CurrProgCounterOffset   = NextProgCounterOffset;
                CurrBusAddr             = NextBusAddr;
                CurrBusDataOut          = NextBusDataOut;
                CurrBusDataOutWE        = NextBusDataOutWE;
                CurrRegA                = NextRegA;
                CurrRegB                = NextRegB;
                CurrRegSelect           = NextRegSelect;
                CurrProgContext         = NextProgContext;
                CurrInterruptAck        = NextInterruptAck;
            end
        end

        //Combinatorial section – large!
        always@* begin
            //Generic assignment to reduce the complexity of the rest of the S/M
            NextState                   = CurrState;
            NextProgCounter             = CurrProgCounter;
            NextProgCounterOffset       = 2'h0;
            NextBusAddr                 = 8'hFF;
            NextBusDataOut              = CurrBusDataOut;
            NextBusDataOutWE            = 1'b0;
            NextRegA                    = CurrRegA;
            NextRegB                    = CurrRegB;
            NextRegSelect               = CurrRegSelect;
            NextProgContext             = CurrProgContext;
```

```
NextInterruptAck                    = 2'b00;

//Case statement to describe each state
case (CurrState)
        ////////////////////////////////////////////////////////////////////////////
        //Thread states.
        IDLE:    begin
                   if(BUS_INTERRUPTS_RAISE[0]) begin   // Interrupt Request A.
                        NextState               = GET_THREAD_START_ADDR_0;
                        NextProgCounter         = 8'hFF;
                        NextInterruptAck        = 2'b01;
                    end else if(BUS_INTERRUPTS_RAISE[1]) begin //Interrupt Request B.
                        NextState               = GET_THREAD_START_ADDR_0;
                        NextProgCounter         = 8'hFE;
                        NextInterruptAck        = 2'b10;
                   end else begin
                        NextState               = IDLE;
                        NextProgCounter         = 8'hFF; //Nothing has happened.
                        NextInterruptAck        = 2'b00;
                    end
                   end

        //Wait state - for new prog address to arrive.
        GET_THREAD_START_ADDR_0: begin
                        NextState               = GET_THREAD_START_ADDR_1;
                        end

        //Assign the new program counter value
        GET_THREAD_START_ADDR_1:  begin
                        NextState               = GET_THREAD_START_ADDR_2;
                        NextProgCounter         = ProgMemoryOut;
                        end

        //Wait for the new program counter value to settle
        GET_THREAD_START_ADDR_2:
                        NextState               = CHOOSE_OPP;



        ////////////////////////////////////////////////////////////////////////////
        //CHOOSE_OPP - Another case statement to choose which operation to perform
        CHOOSE_OPP:  begin
                       case (ProgMemoryOut[3:0])
                            4'h0: NextState     = READ_FROM_MEM_TO_A;
                            4'h1: NextState     = READ_FROM_MEM_TO_B;
                            4'h2: NextState     = WRITE_TO_MEM_FROM_A;
                            4'h3: NextState     = WRITE_TO_MEM_FROM_B;
                            4'h4: NextState     = DO_MATHS_OPP_SAVE_IN_A;
                            4'h5:NextState      = DO_MATHS_OPP_SAVE_IN_B;
                            4'h6:NextState      = IF_A_EQUALITY_B_GOTO;
                            4'h7: NextState     = GOTO;
                            4'h8:NextState      = IDLE;
                            4'h9:NextState      = FUNCTION_START;
                            4'hA:NextState      = RETURN;
                            4'hB:NextState      = DE_REFERENCE_A;
                            4'hC:NextState      = DE_REFERENCE_B;
                            default:
                                NextState       = CurrState;
                        endcase

                         NextProgCounterOffset                  = 2'h1;
                        end

        ////////////////////////////////////////////////////////////////////////////
        //READ_FROM_MEM_TO_A : here starts the memory read operational pipeline.
        //Wait state - to give time for the mem address to be read. Reg select is set to 0
        READ_FROM_MEM_TO_A:begin
```

```
                        NextState       = READ_FROM_MEM_0;
                        NextRegSelect   = 1'b0;
                        end

//READ_FROM_MEM_TO_B : here starts the memory read operational pipeline.
//Wait state - to give time for the mem address to be read. Reg select is set to 1
READ_FROM_MEM_TO_B:begin
                        NextState       = READ_FROM_MEM_0;
                        NextRegSelect   = 1'b1;
                        end

//The address will be valid during this state, so set the BUS_ADDR to this value.
READ_FROM_MEM_0:    begin
                        NextState       = READ_FROM_MEM_1;
                        NextBusAddr     = ProgMemoryOut;
                        end

//Wait state - to give time for the mem data to be read
//Increment the program counter here.  This must be done 2 clock cycles ahead
//so that it presents the right data when required.
READ_FROM_MEM_1:    begin
                        NextState         = READ_FROM_MEM_2;
                        NextProgCounter = CurrProgCounter + 2;
                        end

//The data will now have arrived from memory.  Write it to the proper register.
READ_FROM_MEM_2:    begin
                        NextState       = CHOOSE_OPP;
                        if(!CurrRegSelect)
                                NextRegA        = BusDataIn;
                        else
                                NextRegB        = BusDataIn;
                        end

////////////////////////////////////////////////////////////////////////////////
//WRITE_TO_MEM_FROM_A : here starts the memory write operational pipeline.
//Wait state - to find the address of where we are writing
//Increment the program counter here. This must be done 2 clock cycles ahead
 //so that it presents the right data when required.
WRITE_TO_MEM_FROM_A:begin
                        NextState               = WRITE_TO_MEM_0;
                        NextRegSelect           = 1'b0;
                        NextProgCounter         = CurrProgCounter + 2;
                        end

//WRITE_TO_MEM_FROM_B : here starts the memory write operational pipeline.
//Wait state - to find the address of where we are writing
//Increment the program counter here. This must be done 2 clock cycles ahead
// so that it presents the right data when required.
WRITE_TO_MEM_FROM_B:begin
                        NextState               = WRITE_TO_MEM_0;
                        NextRegSelect           = 1'b1;
                        NextProgCounter         = CurrProgCounter + 2;
                        end


//The address will be valid during this state, so set the BUS_ADDR to this value,
//and write the value to the memory location.
WRITE_TO_MEM_0:     begin
                        NextState               = CHOOSE_OPP;
                        NextBusAddr             = ProgMemoryOut;
                        if(!NextRegSelect)
                          NextBusDataOut        = CurrRegA;
                        else
                          NextBusDataOut        = CurrRegB;

                       NextBusDataOutWE         = 1'b1;
```

```
                                        end
                    ////////////////////////////////////////////////////////////////////////////////
                    //DO_MATHS_OPP_SAVE_IN_A : here starts the DoMaths operational pipeline.
                    //Reg A and Reg B must already be set to the desired values.  The MSBs of the
                    // Operation type determines the maths operation type.  At this stage the result is
                    // ready to be collected from the ALU.
                    DO_MATHS_OPP_SAVE_IN_A:   begin
                                               NextState       = DO_MATHS_OPP_0;
                                               NextRegA        = AluOut;
                                               NextProgCounter = CurrProgCounter + 1;
                                              end

                    //DO_MATHS_OPP_SAVE_IN_B : here starts the DoMaths operational pipeline
                    //when the result will go into reg B.
                    DO_MATHS_OPP_SAVE_IN_B:   begin
                                               NextState       = DO_MATHS_OPP_0;
                                               NextRegB        = AluOut;
                                               NextProgCounter = CurrProgCounter + 1;
                                              end

                    //Wait state for new prog address to settle.
                    DO_MATHS_OPP_0:           NextState       = CHOOSE_OPP;

/*
Complete the above case statement for In/Equality, Goto Address, Goto Idle, function start, Return from
function, and Dereference operations.
*/
        ……..

        FILL IN THIS AREA

        …….

            endcase
        end

endmodule
```

4.2 What you are required to do - Assessment

Before you can design your microprocessor and write software for it, you will need to wrap up you existing peripheral codes (i.e. for Mouse, VGA and IR Transmitter) with a bus interface to connect them to the microprocessor bus. You will need to simulate and verify these new peripheral codes with simple test benches. Make sure they synthesise, before you can test on the BASYS 3 board. You are then required to code the above microprocessor behaviour in Verilog based on the specification and codes provided. Once your system is fully captured in Verilog, simulated with simple test benches and tested on the BASYS 3 board, you need to develop a demo software application for each microprocessor and peripheral combination (you will be tested individually on this task) and then go on to develop the full demo software application as outlined at the start of this document (you will be tested as a group on this task).

Individual Demo Software Application

The following will specify the individual demo software for each microprocessor and peripheral combination.

Microprocessor + Mouse Demo Application

This task should be performed by the team member who developed the mouse peripheral interface. Here, you will need to complete a software demo application running on the microprocessor that the whole team developed, using only the mouse peripheral among the three peripherals developed by the individual team members i.e. do not use the VGA or the IR Transmitter interfaces here. The team member in charge of this task will be tested in Week 9. This individual assessment will account for 35% of your overall lab mark. During the test, you will need to demonstrate your program on the BASYS 3 board and explain your code.

The application in question here consists in re-implementing the same original mouse demo which you developed earlier i.e. the demo which showed the mouse registers content on the LEDs and seven segment displays, but this time purely in software using the instruction set of the microprocessor developed by the whole team.

Microprocessor + IR Transmitter Demo Application

For undergrauate students, this task should be performed by the team member who developed the IR Transmitter peripheral interface. **For postgraduate students**, this task should be performed by all groups.

Here, you will need to complete a software demo application running on the microprocessor that the whole team developed, using only the IT Transmitter peripheral among the three peripherals developed by the individual team members i.e. do not use the VGA or the Mouse here. The team member in charge of this task will be tested in Week 9. This individual assessment will account for 35% of your overall lab mark. During the test, you will need to demonstrate your program on the BASYS 3 board and explain your code.

The application in question here consists in generating the following consecutive actions on the remote car:

1. Apply a move forward command for one second
2. Apply a move backward command for one second
3. Apply a move forward left command for one second
4. Apply a move backward right command for one second
5. Apply a move forward right command for one second
6. Apply a move backward left command for one second

Hints:

- Store constant values e.g. 0, 1, 2, 3, 4 etc. in predefined locations in the data memory, and load them to registers when needed
- Use the dereference instruction to step through a set of instructions using a pointer to the current instruction which can be incremented to step through several instructions

Microprocessor + VGA Interface Demo Application

This task should be performed by the team member who developed the VGA peripheral interface. Here, you will need to complete a software demo application running on the

microprocessor that the whole team developed, using only the VGA peripheral among the three peripherals developed by the individual team members i.e. do not use the Mouse or the IR Transmitter here. The team member in charge of this task will be tested in Week 9. This individual assessment will account for 35% of your overall lab mark. During the test, you will need to demonstrate your program on the BASYS 3 board and explain your code.

The application in question here consists in re-implementing the original VGA demo which showed the chequered image with changing colours on the VGA screen, but this time purely in software using the instruction set of the microprocessor developed by the whole team.

Complete Demo Software Application

This task should be performed by all team members as a group. Here, you will develop the complete software demo application described at the beginning of this document (see page 2) using all peripherals. You need to implement the complete system described in Figure 13 and program the microprocessor to implement the complete demo application. The team as a whole will be tested on this task in Week 11.

For undergraduate students, this group assessment will account for the remaining 40% of your overall lab mark. **For postgraduate students**, this group assessment will account for the remaining 30% of your overall lab mark.

During the test, you will need to demonstrate your program running on the BASYS 3 board and explain your code, especially the software code running on the microprocessor. There will be an element of peer assessment in this final group based assignment where individuals will be asked to evaluate the individual performance of colleagues within the group. More details of this process will be given nearer the time of assessment. Your code will have been uploaded to Learn prior to the start of the final laboratory session.