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.

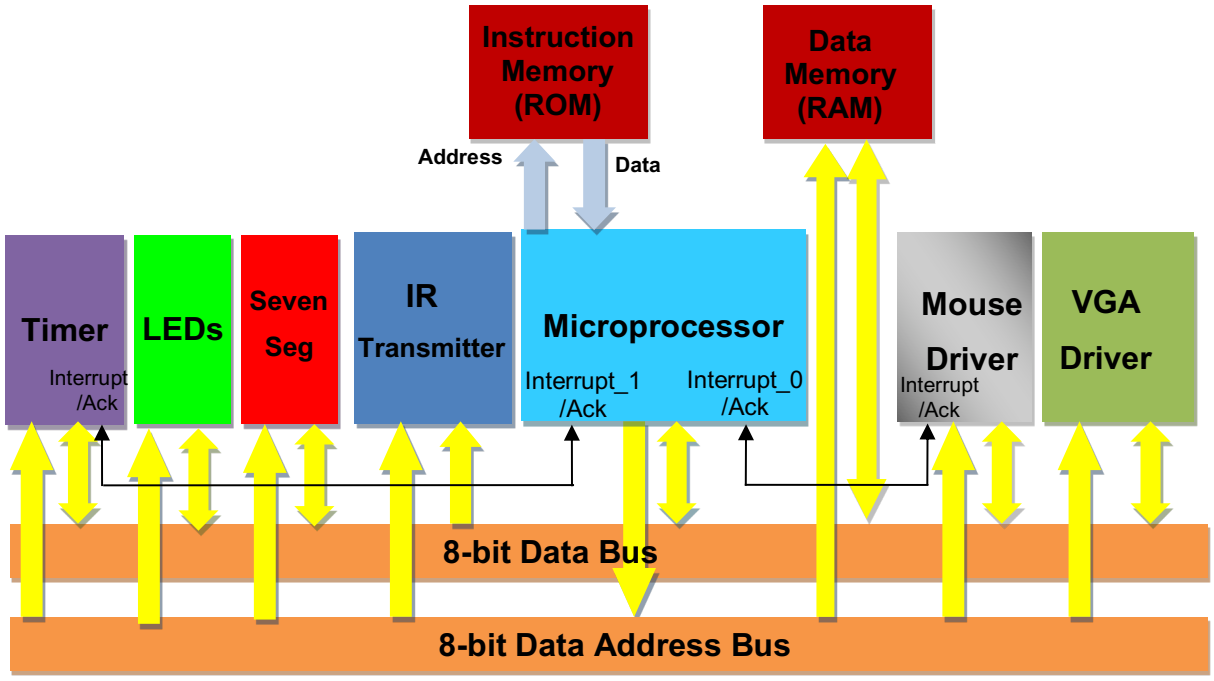

Figure 13. Microprocessor-based system block architecture

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).

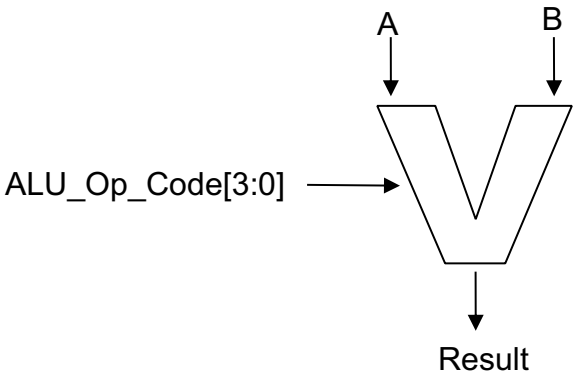

Figure 14. Arithmetic and Logic Unit (ALU)

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.

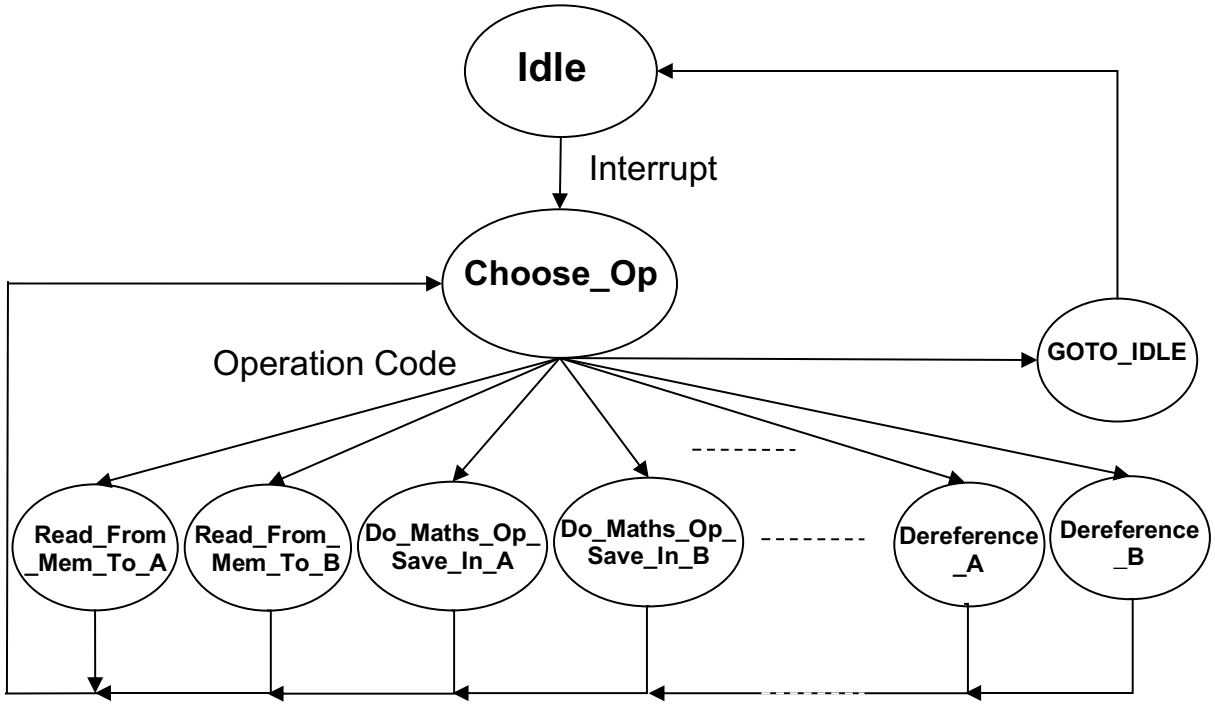

Figure 15. Microprocessor's state machine

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
```