Laboratory Exercise 9

A Simple Processor

Figure 1 shows a digital system that contains a number of 9-bit registers, a multiplexer, an adder/subtracter unit, and a control unit(finite state machine). Data is input to this system via the 9-bit *DIN* input. This data can be loaded through the 9-bit wide multiplexer into the various registers, such as $R0, \ldots, R7$ and *A*. The multiplexer also allows data to be transferred from one register to another. The multiplexer's output wires are called a *bus* in the figure because this term is often used for wiring that allows data to be transferred from one location in a system to another.

Addition or subtraction is performed by using the multiplexer to first place one 9-bit number onto the bus wires and loading this number into register *A*. Once this is done, a second 9-bit number is placed onto the bus, the adder/subtracter unit performs the required operation, and the result is loaded into register *G*. The data in *G* can then be transferred to one of the other registers as required.

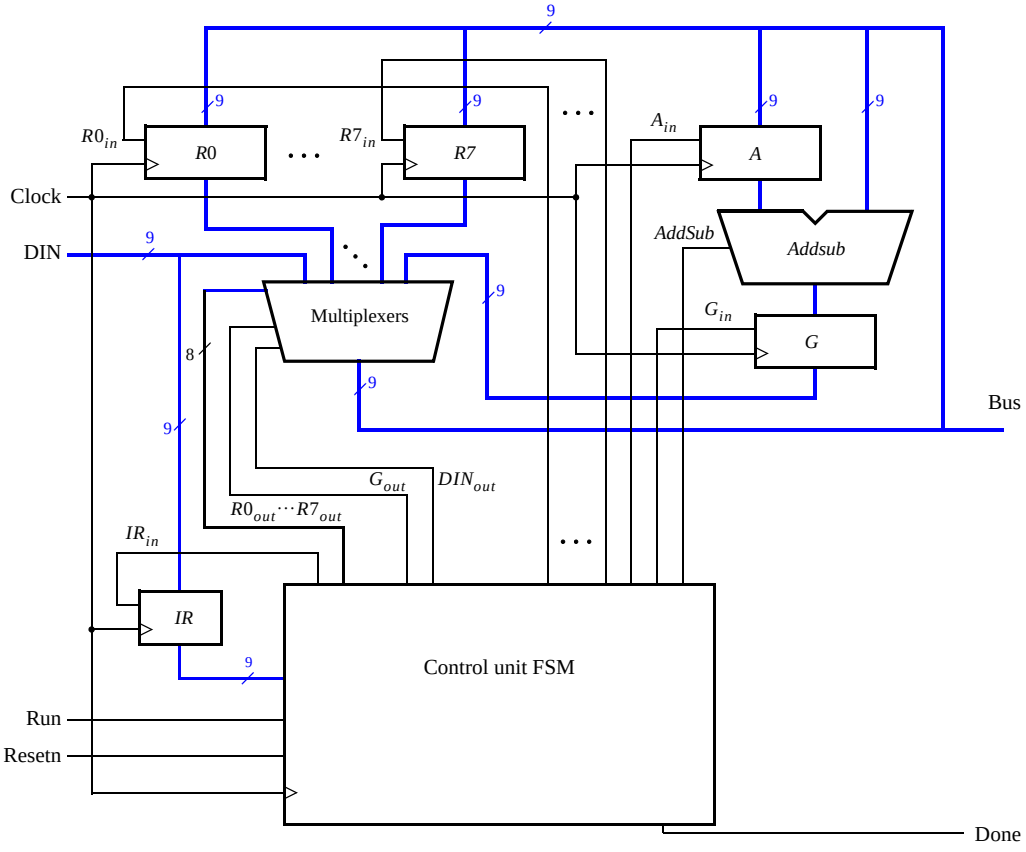

Figure 1: A digital system.

The system can perform different operations in each clock cycle, as governed by the *control unit*. This unit determines when particular data is placed onto the bus wires and it controls which of the registers is to be loaded with this data. For example, if the control unit asserts the signals $R0_{out}$ and A_{in}, then the multiplexer will place the contents of register $R0$ onto the bus and this data will be loaded by the next active clock edge into register *A*.

A system like this is often called a *processor*. It executes operations specified in the form of instructions. Table 1 lists the instructions that the processor has to support for this exercise. The left column shows the name

of an instruction and its operand. The meaning of the syntax RX ← [RY] is that the contents of register RY are loaded into register RX. The **mv** (move) instruction allows data to be copied from one register to another. For the **mvi** (move immediate) instruction the expression RX ← D indicates that the 9-bit constant D is loaded into register RX.

Operation	Function performed
mv Rx,Ry	$Rx \leftarrow [Ry]$
mvi Rx,#D	$Rx \leftarrow D$
add Rx, Ry	$Rx \leftarrow [Rx] + [Ry]$
sub Rx, Ry	$Rx \leftarrow [Rx] - [Ry]$

Table 1. Instructions performed in the processor.

Each instruction can be encoded and stored in the *IR* register using the 9-bit format IIIXXXYYY, where III represents the instruction, XXX gives the RX register, and YYY gives the RY register. Although only two bits are needed to encode our four instructions, we are using three bits because other instructions will be added to the processor in later parts of this exercise. Hence *IR* has to be connected to the nine bits of the *DIN* input, as indicated in Figure 1. For the **mvi** instruction the YYY field has no meaning, and the immediate data #D has to be supplied on the 9-bit *DIN* input after the **mvi** instruction word is stored into *IR*.

Some instructions, such as an addition or subtraction, take more than one clock cycle to complete, because multiple transfers have to be performed across the bus. The finite state machine in the control unit "steps through" such instructions, asserting the control signals needed in successive clock cycles until the instruction has completed. The processor starts executing the instruction on the *DIN* input when the *Run* signal is asserted and the processor asserts the *Done* output when the instruction is finished. Table 2 indicates the control signals that can be asserted in each time step to implement the instructions in Table 1. Note that the only control signal asserted in time step 0 is IR_{in}, so this time step is not shown in the table.

	T_1	T_2	T_3
(**mv**): I_0	RY_{out}, RX_{in}, *Done*		
(**mvi**): I_1	DIN_{out}, RX_{in}, *Done*		
(**add**): I_2	RX_{out}, A_{in}	RY_{out}, G_{in}	G_{out}, RX_{in}, *Done*
(**sub**): I_3	RX_{out}, A_{in}	RY_{out}, G_{in}, *AddSub*	G_{out}, RX_{in}, *Done*

Table 2. Control signals asserted in each instruction/time step.

Part I

Design and implement the processor shown in Figure 1 using Verilog code as follows:

1. Create a new Quartus II project for this exercise.
2. Generate the required Verilog file, include it in your project, and compile the circuit. A suggested skeleton of the Verilog code is shown in parts a and b of Figure 2, and some subcircuit modules that can be used in this code appear in Figure 2c.
3. Use functional simulation to verify that your code is correct. An example of the output produced by a functional simulation for a correctly-designed circuit is given in Figure 3. It shows the value $(040)_{16}$ being loaded into *IR* from *DIN* at time 30 ns. This pattern (the leftmost bits of *DIN* are connected to *IR*) represents the instruction **mvi** R0,#D, where the value $D = 5$ is loaded into $R0$ on the clock edge at 50 ns. The simulation then shows the instruction **mv** R1,R0 at 90 ns, **add** R0,R1 at 110 ns, and **sub** R0,R0 at 190 ns. Note that the simulation output shows *DIN* as a 3-digit hexadecimal number, and it shows the contents of *IR* as a 3-digit octal number.
4. Create a new Quartus II project which will be used for implementation of the circuit on the Altera DE0 board. This project should consist of a top-level module that contains the appropriate input and output ports for the Altera board. Instantiate your processor in this top-level module. Use switches SW_{8-0} to drive the *DIN* input port of the processor and use switch SW_9 to drive the *Run* input. Also, use push button KEY_0 for *Resetn* and KEY_1 for *Clock*. Connect the processor bus wires to $LEDG_{8-0}$ and connect the *Done* signal to $LEDG_9$.
5. Add to your project the necessary pin assignments for the DE0 board. Compile the circuit and download it into the FPGA chip.
6. Test the functionality of your design by toggling the switches and observing the LEDs. Since the processor's clock input is controlled by a push button switch, it is easy to step through the execution of instructions and observe the behavior of the circuit.

```
module  proc (DIN, Resetn, Clock, Run, Done, BusWires);
    input [8:0] DIN;
    input Resetn, Clock, Run;
    output Done;
    output [8:0] BusWires;

    parameter T0 = 2'b00, T1 = 2'b01, T2 = 2'b10, T3 = 2'b11;
    ... declare variables

    assign I = IR[1:3];
    dec3to8 decX (IR[4:6], 1'b1, Xreg);
    dec3to8 decY (IR[7:9], 1'b1, Yreg);
```

Figure 2a. Skeleton Verilog code for the processor.

```
    // Control FSM state table
    always @(Tstep_Q, Run, Done)
    begin
        case (Tstep_Q)
            T0: // data is loaded into IR in this time step
                if (!Run) Tstep_D = T0;
                else Tstep_D = T1;
            T1: . . .
        endcase
    end

    // Control FSM outputs
    always @(Tstep_Q or I or Xreg or Yreg)
    begin
        . . . specify initial values
        case (Tstep_Q)
            T0: // store DIN in IR in time step 0
            begin
                IRin = 1'b1;
            end
            T1: //define signals in time step 1
                case (I)
                    . . .
                endcase
            T2: //define signals in time step 2
                case (I)
                    . . .
                endcase
            T3: //define signals in time step 3
                case (I)
                    . . .
                endcase
        endcase
    end

    // Control FSM flip-flops
    always @(posedge Clock, negedge Resetn)
        if (!Resetn)
        . . .

    regn reg_0 (BusWires, Rin[0], Clock, R0);
    . . . instantiate other registers and the adder/subtracter unit

    . . . define the bus

endmodule
```

Figure 2*b*. Skeleton Verilog code for the processor.

```
module  dec3to8(W, En, Y);
    input [2:0] W;
    input En;
    output [0:7] Y;
    reg [0:7] Y;

    always @(W or En)
    begin
        if (En == 1)
            case (W)
                3'b000: Y = 8'b10000000;
                3'b001: Y = 8'b01000000;
                3'b010: Y = 8'b00100000;
                3'b011: Y = 8'b00010000;
                3'b100: Y = 8'b00001000;
                3'b101: Y = 8'b00000100;
                3'b110: Y = 8'b00000010;
                3'b111: Y = 8'b00000001;
            endcase
        else
            Y = 8'b00000000;
    end
endmodule

module  regn(R, Rin, Clock, Q);
    parameter n = 9;
    input [n-1:0] R;
    input Rin, Clock;
    output [n-1:0] Q;
    reg [n-1:0] Q;

    always @(posedge Clock)
        if (Rin)
            Q <= R;
endmodule
```

Figure 2*c*. Subcircuit modules for use in the processor.

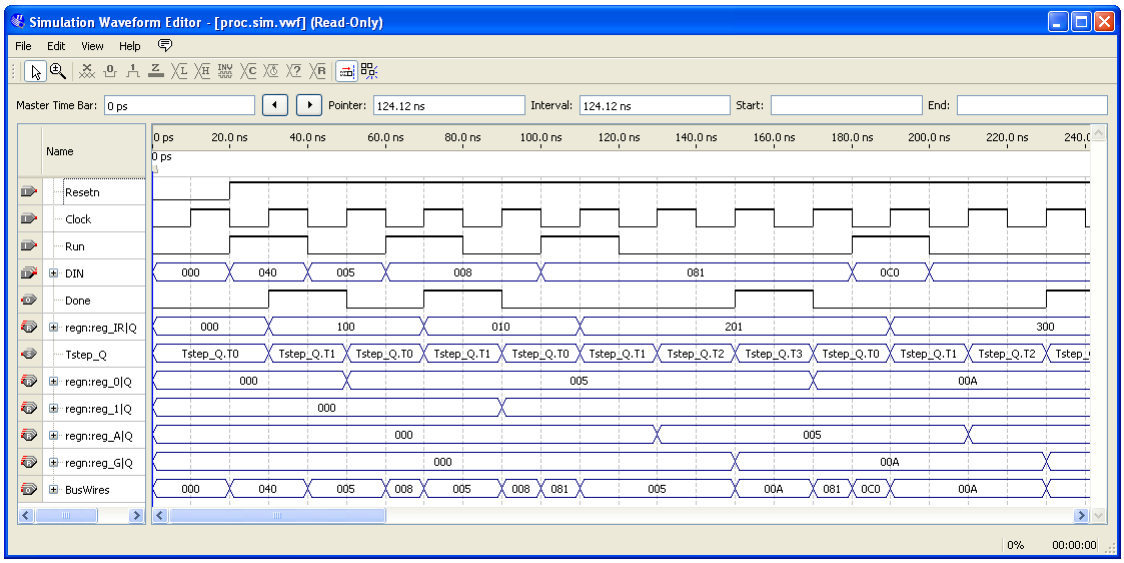

Figure 2: Simulation of the processor.

Part II

In this part you are to design the circuit depicted in Figure 4, in which a memory module and counter are connected to the processor from Part I. The counter is used to read the contents of successive addresses in the memory, and this data is provided to the processor as a stream of instructions. To simplify the design and testing of this circuit we have used separate clock signals, *PClock* and *MClock*, for the processor and memory.

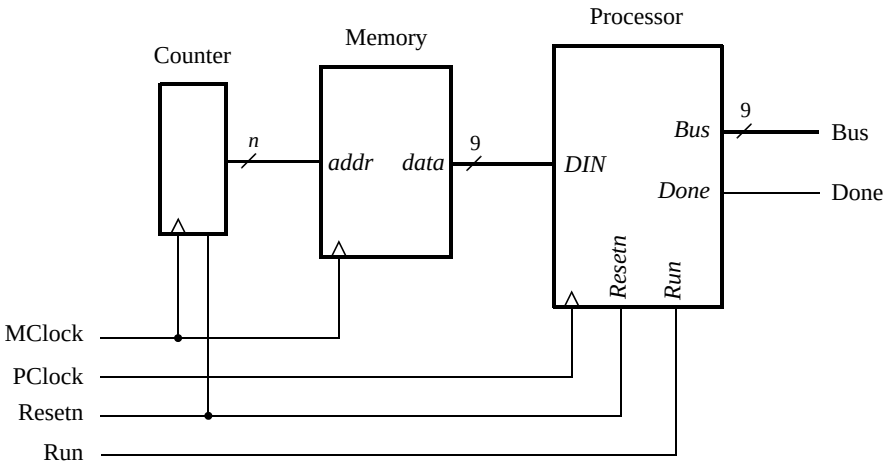

Figure 3: Connecting the processor to a memory and counter.

1. Create a new Quartus II project which will be used to test your circuit.

2. Generate a top-level Verilog file that instantiates the processor, memory, and counter. Use the Quartus II MegaWizard Plug-In Manager tool to create the memory module from the Altera library of parameterized modules (LPMs). The correct LPM is found under the *Memory Compiler* category and is called *ROM: 1-PORT*. Follow the instructions provided by the wizard to create a memory that has one 9-bit wide read

data port and is 32 words deep. Page 3 of the wizard is shown in Figure 5. Advance to the subsequent page and *deselect* the setting called 'q' output port under the category Which ports should be registered?. Since this memory has only a read port, and no write port, it is called a *synchronous read-only memory (synchronous ROM)*. Note that the memory includes a register for synchronously loading addresses. This register is required due to the design of the memory resources on the Cyclone III FPGA; account for the clocking of this address register in your design.

To place processor instructions into the memory, you need to specify *initial values* that should be stored in the memory once your circuit has been programmed into the FPGA chip. This can be done by telling the wizard to initialize the memory using the contents of a *memory initialization file (MIF)*. The appropriate screen of the MegaWizard Plug-In Manager tool is illustrated in Figure 6. We have specified a file named *inst_mem.mif*, which then has to be created in the directory that contains the Quartus II project. Use the Quartus II on-line Help to learn about the format of the *MIF* file and create a file that has enough processor instructions to test your circuit.

3. Use functional simulation to test the circuit. Ensure that data is read properly out of the ROM and executed by the processor.

4. Make sure your project includes the necessary port names and pin location assignments to implement the circuit on the DE0 board. Use switch SW_9 to drive the processor's *Run* input, use KEY_0 for *Resetn*, use KEY_1 for *MClock*, and use KEY_2 for *PClock*. Connect the processor bus wires to $LEDG_{8-0}$ and connect the *Done* signal to $LEDG_9$.

5. Compile the circuit and download it into the FPGA chip.

6. Test the functionality of your design by toggling the switches and observing the LEDs. Since the circuit's clock inputs are controlled by push button switches, it is easy to step through the execution of instructions and observe the behavior of the circuit.

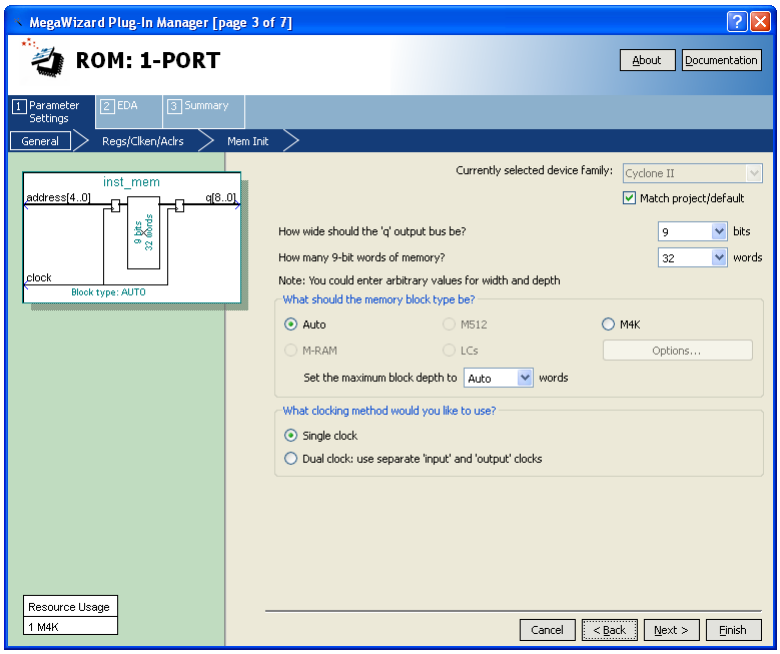

Figure 4: ROM: 1-PORT configuration.

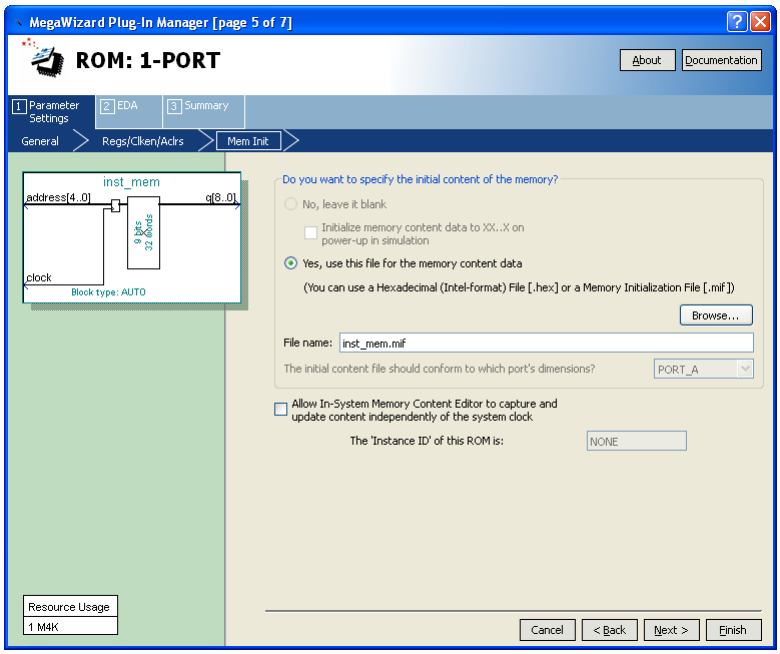

Figure 5: Specifying a memory initialization file (MIF).

Enhanced Processor

It is possible to enhance the capability of the processor so that the counter in Figure 4 is no longer needed, and so that the processor has the ability to perform read and write operations using memory or other devices. These enhancements involve adding new instructions to the processor and the programs that the processor executes are therefore more complex; they are described in a subsequent lab exercise available from Altera.