Laboratory Exercise 9

A Simple Processor

Figure 1 shows a digital system that contains a number of nine-bit registers, a multiplexer, an adder/subtractor unit, and a control unit (finite state machine). Data is input to this system via the nine-bit *DIN* input. This data can be loaded through the nine-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 of signed numbers is performed by using the multiplexer to first place one nine-bit number onto the bus wires and loading this number into register *A*. Once this is done, a second nine-bit number is placed onto the bus, the adder/subtractor 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.

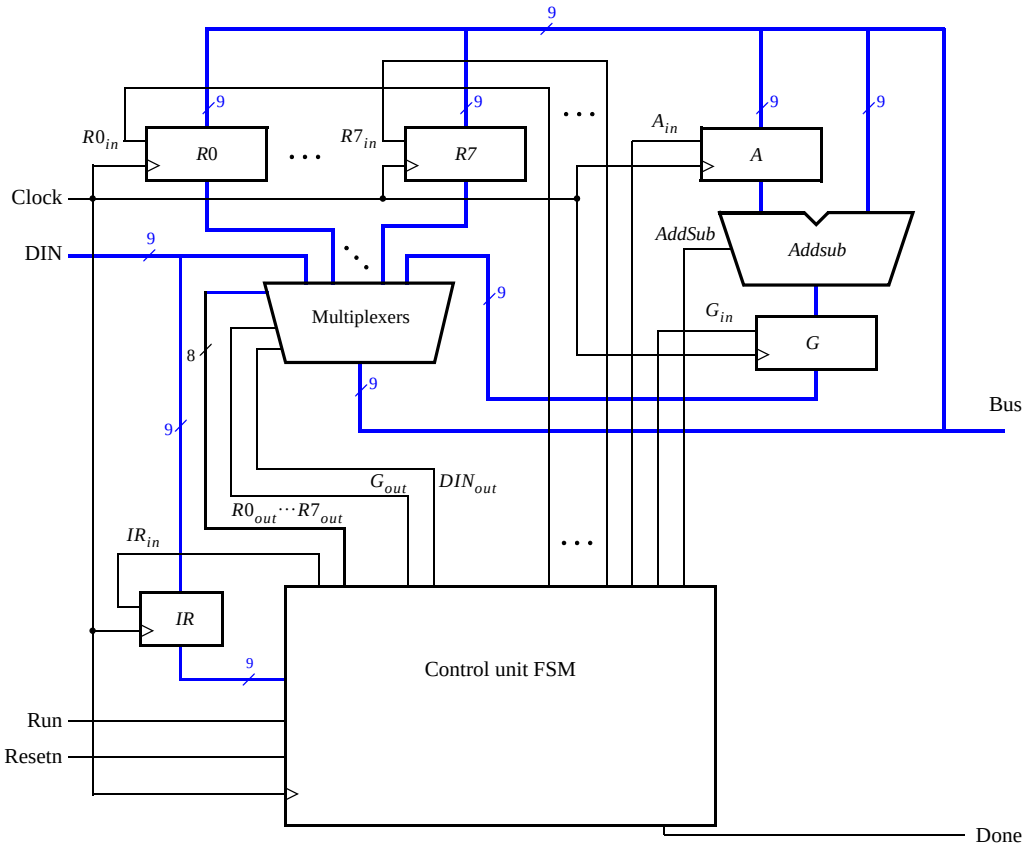

Figure 1: A digital system.

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 on the next active clock edge into register *A*.

A system like the one in Figure 1 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 operands. The meaning of the syntax R$x \leftarrow$ [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 R$x \leftarrow$ D indicates that the nine-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 using the nine-bit format *IIIXXXYYY*, where *III* specifies 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 the exercise. Assume that *III* = 000 for the **mv** instruction, 001 for **movi**, 010 for **add**, and 011 for **sub**. Instructions are loaded from the the external input *DIN*, and stored into the *IR* register, using the connection 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 *DIN* input in the clock cycle 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 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 $(010)_8$ being loaded into *IR* from *DIN* at time 30 ns. This pattern 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* and *IR* in octal, and it shows the contents of other registers in hexadecimal.
4. Now, create another Quartus project which will be used for implementation of the circuit on your Intel FPGA DE-series board. This project should consist of a top-level module that contains the appropriate input and output ports for the DE-series 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 pushbutton KEY_0 for *Resetn* and KEY_1 for *Clock*. Connect the processor bus wires to $LEDR_{8-0}$ and connect the *Done* signal to $LEDR_9$.
5. Add to your project the necessary pin assignments for your board. Compile the circuit and download it into the FPGA chip.
6. Test the functionality of your circuit by toggling the switches and observing the LEDs. Since the processor's clock input is controlled by a pushbutton switch, it is possible 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 2: Skeleton Verilog code for the processor. (Part a)

```
    // 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/subtractor unit

    . . . define the bus

endmodule
```

Figure 2: Skeleton Verilog code for the processor. (Part *b*)

```
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: Subcircuit modules for use in the processor. (Part c)

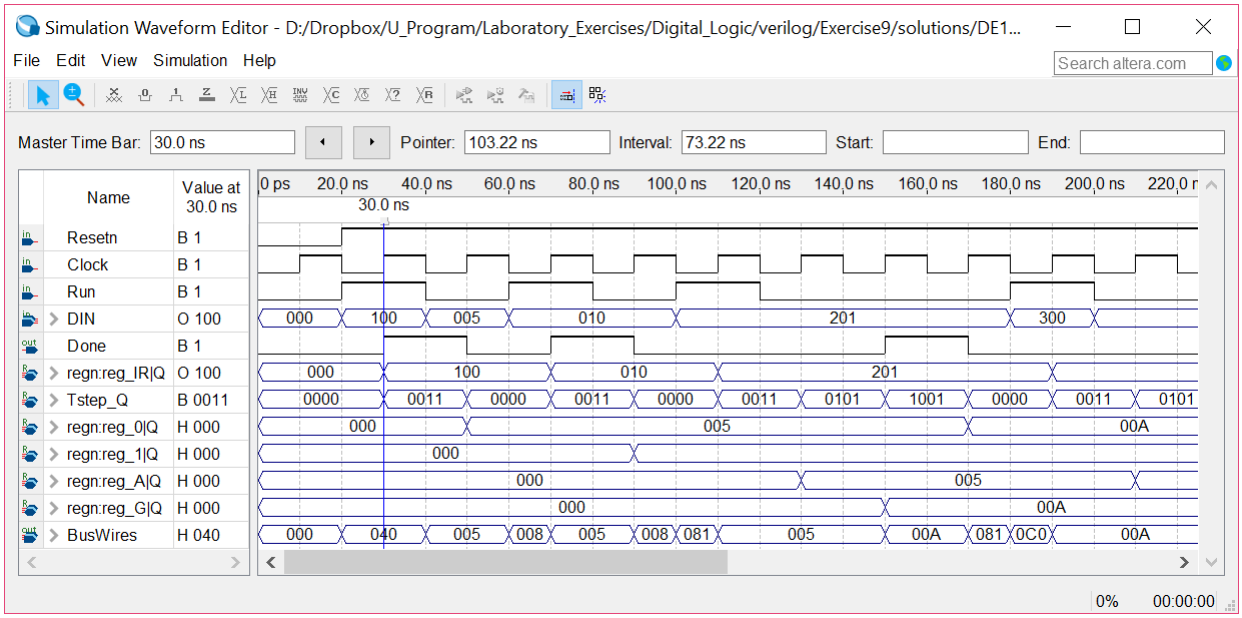

Figure 3: Simulation results for the processor.

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.

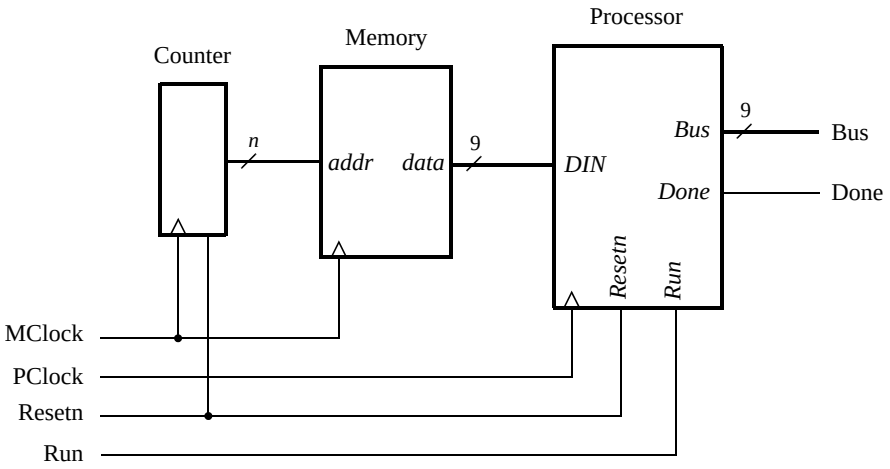

Figure 4: Connecting the processor to a memory module and counter.

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

2. Generate a top-level Verilog file that instantiates the processor, memory module, and counter.

   A diagram of the memory module that we need to create is depicted in Figure 5. Since this memory module has only a read port, and no write port, it is called a *synchronous read-only memory (synchronous ROM)*.

Note that the memory module includes a register for synchronously loading addresses. This register is required due to the design of the memory resources in the Intel FPGA chip. Use the Quartus IP Catalog tool to create the memory module, by clicking on Tools > IP Catalog. In the IP Catalog window choose the *ROM: 1-PORT* module, which is found under the Basic Functions > On Chip Memory category. Select Verilog HDL as the type of output file to create, and give the file the name *inst_mem.v*.

Follow through the provided sequence of dialogs to create a memory that has one nine-bit wide read data port and is 32 words deep. Figures 6 and 7 show the relevant pages and how to properly configure the memory.

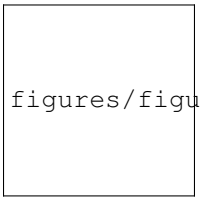

Figure 5: The 32 x 9 ROM with address register.

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 initializing the memory using the contents of a *memory initialization file (MIF)*. The appropriate screen is illustrated in Figure 8. We have specified a file named *inst_mem.mif*, which then has to be created in the folder that contains the Quartus project. An example of a memory initialization file is given in Figure 9. Note that comments (% ... %) are included in this file as a way of documenting the meaning of the provided instructions. Set the contents of your *MIF* file such that it provides enough processor instructions to test your circuit.

3. Make sure your project includes the necessary port names and pin location assignments to implement the circuit on your DE-series board. Use switch SW_9 to drive the processor's *Run* input, use SW_0 for *Resetn*, use KEY_0 for *MClock*, and use KEY_1 for *PClock*. Connect the processor bus wires to $LEDR_{8-0}$ and connect the *Done* signal to $LEDR_9$.

4. Compile your Verilog code and use functional simulation to test the circuit. Ensure that instructions are read properly out of the ROM and executed by the processor. An example of functional simulation using the MIF file from Figure 9 is shown in Figure 10.

5. Once your simulation shows a properly-working circuit, download it into the FPGA chip. Test the functionality of your design on the DE-series board by toggling the switches and observing the LEDs. Since the circuit's clock inputs are controlled by pushbutton switches, it is possible to step through the execution of instructions and observe the behavior of the circuit.

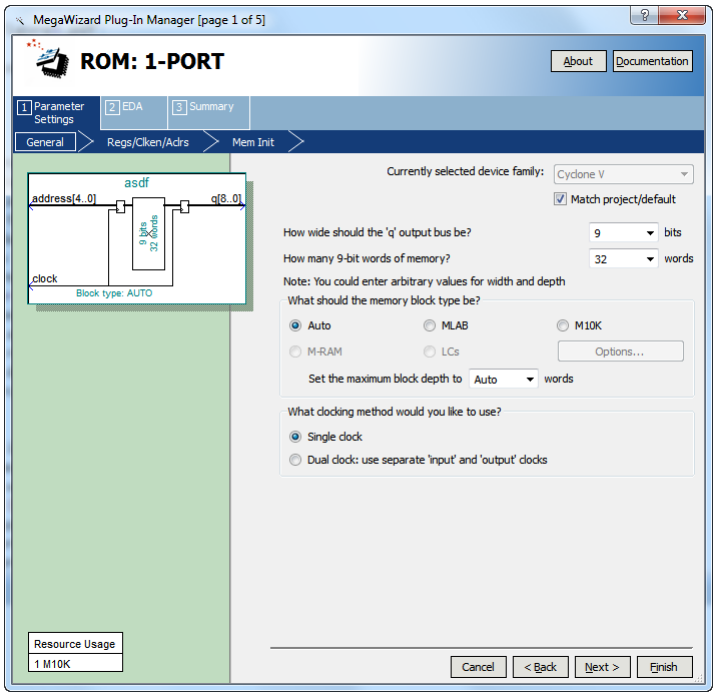

Figure 6: Specifying memory size.

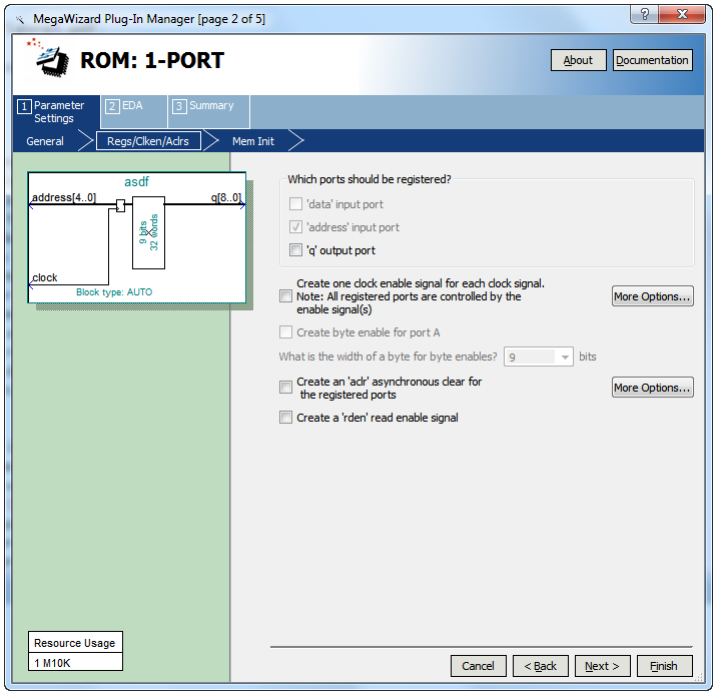

Figure 7: Specifying which memory ports are registered.

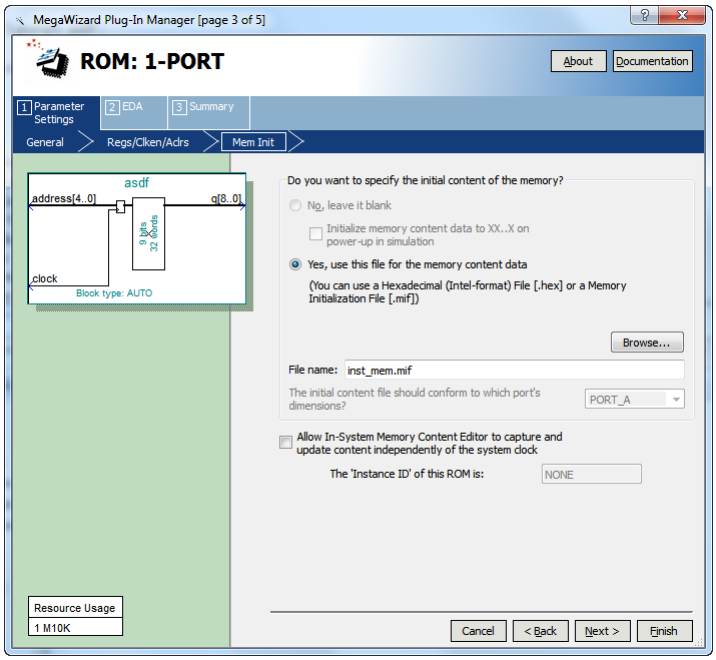

Figure 8: Specifying a memory initialization file (MIF).

```
DEPTH = 32;
WIDTH = 9;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

00 : 001000000;     % mvi r0,#5       %
01 : 000000101;
02 : 000001000;     % mv r1,r0        %
03 : 010000001;     % add r0, r1      %
04 : 011000000;     % sub r0, r0      %
05 : 000000000;
06 : 000000000;
. . . (some lines not shown)
1E : 000000000;
1F : 000000000;

END;
```

Figure 9: An example memory initialization file (MIF).

figures/figure8.png

Figure 10: An example simulation output using the MIF in Figure 9.

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 Laboratory Exercise 10.