Laboratory Exercise 1

A Simple Processor

Figure 1 shows a digital system that contains a number of nine-bit registers, a multiplexer, an adder/subtracter 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 FSM controls the *Select* lines of the multiplexer, which 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/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.

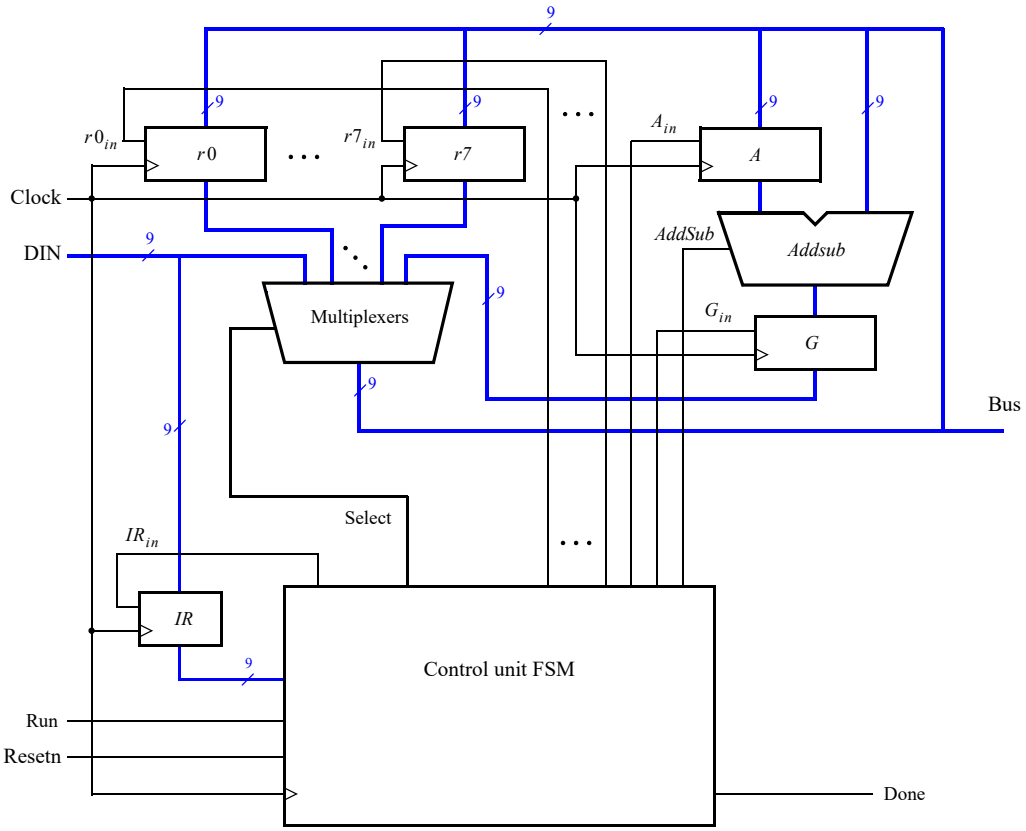

Figure 1: A digital system.

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 selects $r0$ as the output of the bus multiplexer and also asserts A_{in}, then the contents of register *r0* 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 this processor supports. The left column shows the name of an instruction and its operands. 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 nine-bit constant *D* is loaded into register *rX*.

Instruction		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 later. Assume that *III* = 000 for the *mv* instruction, 001 for *mvi*, 010 for *add*, and 011 for *sub*. Instructions are loaded from 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 is stored into *IR*.

Some instructions, such as an addition or subtraction, take a few clock cycles 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 from Figure 1 that have to be asserted in each time step to implement the instructions in Table 1. Note that the only control signal asserted in time step T_0, for all instructions, is IR_{in}. In the table, the meaning of rY_{out} is that the multiplexer in Figure 1 places the contents of register *rY* onto the *Bus* wires. Similarly, DIN_{out} means that the multiplexer places the data that is on *DIN* onto the *Bus* wire.

	T_0	T_1	T_2	T_3
mv	IR_{in}	rY_{out}, rX_{in}, *Done*		
mvi	IR_{in}	DIN_{out}, rX_{in}, *Done*		
add	IR_{in}	rX_{out}, A_{in}	rY_{out}, G_{in}	G_{out}, rX_{in}, *Done*
sub	IR_{in}	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

In this part you will implement the processor shown in Figure 1 using Verilog code, as follows:

1. Create a new Quartus® project for this part of the exercise.

2. Generate the required Verilog file, include it in your project, and compile the circuit. Part 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. A more complete version of the Verilog code is provided on the course website. You can modify this code to suit your own coding style, if desired—the provided code is just a suggested solution.

3. Use the ModelSim Simulator to verify that the processor works properly. An example result produced by using *ModelSim* for a correctly-designed circuit is given in Figure 3. It shows the value $(100)_8$ 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 70 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.

   You should perform a thorough simulation of your processor with the ModelSim simulator. A sample Verilog testbench file, *testbench.v*, execution script, *testbench.tcl*, and waveform file, *wave.do* are provided on the course website along with this exercise.

4. Once your simulations are complete, you can download and test the processor on a DE1-SoC board. For this purpose, the course website provides another Quartus project. It contains a top-level module that includes the appropriate input and output ports for a DE-series board. The top-level module instantiates the processor as a subcircuit. Switches SW_{8-0} drive the *DIN* input port of the processor and switch SW_9 provides the *Run* input. Also, pushbutton KEY_0 is used for *Resetn* and KEY_1 for *Clock*. The processor bus wires are connected to $LEDR_{8-0}$ and the *Done* signal is shown on $LEDR_9$.

5. The Quartus project includes the necessary *pin assignments* for the DE-series board. Compile the circuit and download it into the FPGA chip.

6. Test the functionality of the 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

parameter mv = 3'b000, mvi = 3'b001, add = 3'b010, sub = 3'b011;
// control FSM outputs
always @(Tstep_Q or I or Xreg or Yreg)
begin
   Done = 1'b0; Ain = 1'b0; ... // provide default values for variables
   case (Tstep_Q)
      T0: // store DIN in IR
         begin
            IRin = 1'b1;
         end
      T1:   //define signals in time step T1
         case (I)
            mv: // mv rX, rY
            begin
               Rout = Yreg;
               Rin = Xreg;
               Done = 1'b1;
            end
            mvi: // mvi rX, #D
            ...
         endcase
      T2:   //define signals in time step T2
         case (I)
            ...
         endcase
      T3:   //define signals in time step T3
         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 (see next page)
```

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

```
   assign Sel = ...

   always @(*)
   begin
      if (Sel == 10'b1000000000)
         BusWires = R0;
      else if (Sel == 10'b0100000000)
         BusWires = R1;
      else if (Sel == 10'b0010000000)
         BusWires = R2;
      ...
      else if (Sel == 10'b0000000010)
         BusWires = G;
      else BusWires = DIN;
   end
endmodule

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: Skeleton Verilog code for the processor. (Part c)

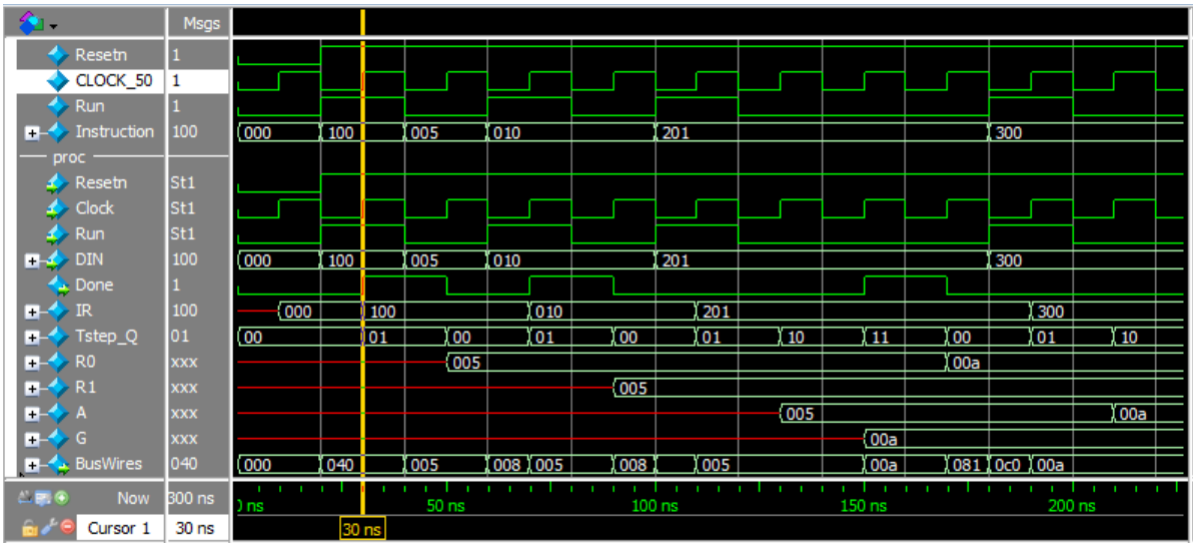

Figure 3: Simulation results for the processor.

Part II

In this part we will implement the circuit depicted in Figure 4, in which a memory module and counter are connected to the processor. The counter is used to read the contents of successive locations 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. Do the following.

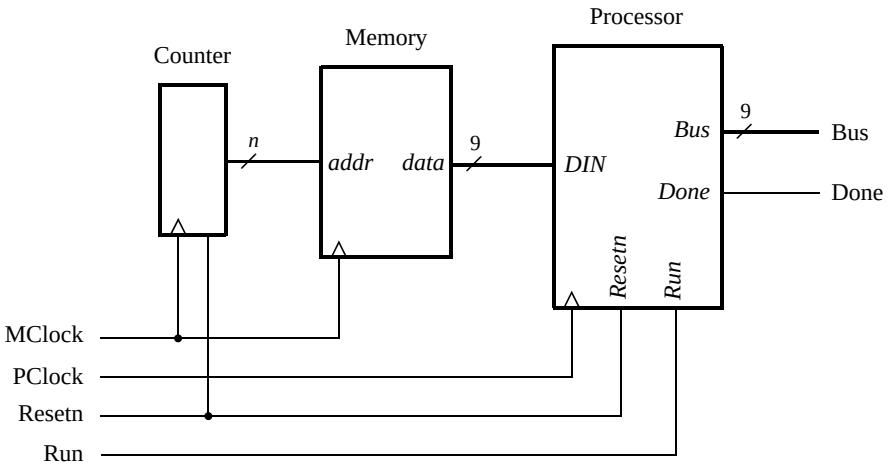

Figure 4: Connecting the processor to a memory module and counter.

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

2. A sample top-level Verilog file that instantiates the processor, memory module, and counter is shown in Figure 5. Type this code into a file, such as *part2*.v. The code instantiates a memory module called *inst_mem*. The procedure that you can follow to create this module is described below.

3. A diagram of the memory module that we need to create is depicted in Figure 6. Since this memory module has only a read port, and no write port, it is called a *synchronous read-only memory (synchronous ROM)*.

```
module part2 (KEY, SW, LEDR);
   input [1:0] KEY;
   input [9:0] SW;
   output [9:0] LEDR;

   wire Done, Resetn, PClock, MClock, Run;
   wire [8:0] DIN, Bus;
   wire [4:0] pc;

   assign Resetn = SW[0];
   assign MClock = KEY[0];
   assign PClock = KEY[1];
   assign Run = SW[9];

   // module proc(DIN, Resetn, Clock, Run, Done, BusWires);
   proc U1 (DIN, Resetn, PClock, Run, Done, Bus);
   assign LEDR[8:0] = Bus;
   assign LEDR[9] = Done;

   inst_mem U2 (pc, MClock, DIN);
   count5 U3 (Resetn, MClock, pc);

endmodule

module count5 (Resetn, Clock, Q);
   input Resetn, Clock;
   output reg [4:0] Q;

   always @ (posedge Clock, negedge Resetn)
      if (Resetn == 0)
         Q <= 5'b00000;
      else
         Q <= Q + 1'b1;
 endmodule
```

Figure 5: Verilog code for the top-level module.

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 Quartus software. 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 7 and 8 show the relevant pages and how to properly configure the memory.

To place processor instructions into the memory, you need to specify *initial values* that should be stored in the memory when your circuit is 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 9. 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 10. 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.

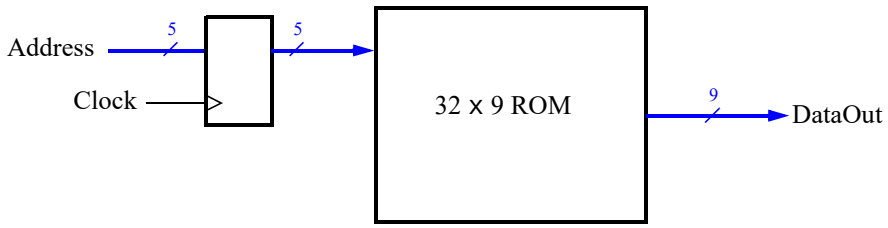

Figure 6: The 32 x 9 ROM with address register.

4. The code in Figure 5, and the Quartus project, includes the necessary port names and pin location assignments to implement the circuit on a DE-series board. The switch SW_9 drives the processor's *Run* input, SW_0 is connected to *Resetn*, KEY_0 to *MClock*, and KEY_1 to *PClock*. The processor bus wires are attached to $LEDR_{8-0}$ and the *Done* signal is connected to $LEDR_9$.

5. Use the ModelSim Simulator to test your Verilog code. Ensure that instructions are read properly out of the ROM and executed by the processor. An example of simulation results produced using ModelSim with the MIF file from Figure 10 is shown in Figure 11. The corresponding ModelSim setup files are provided on the course website along with this exercise.

6. Once your simulations show a properly-working circuit, you can download it into a DE-series board. The functionality of the circuit on the board can be tested 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.

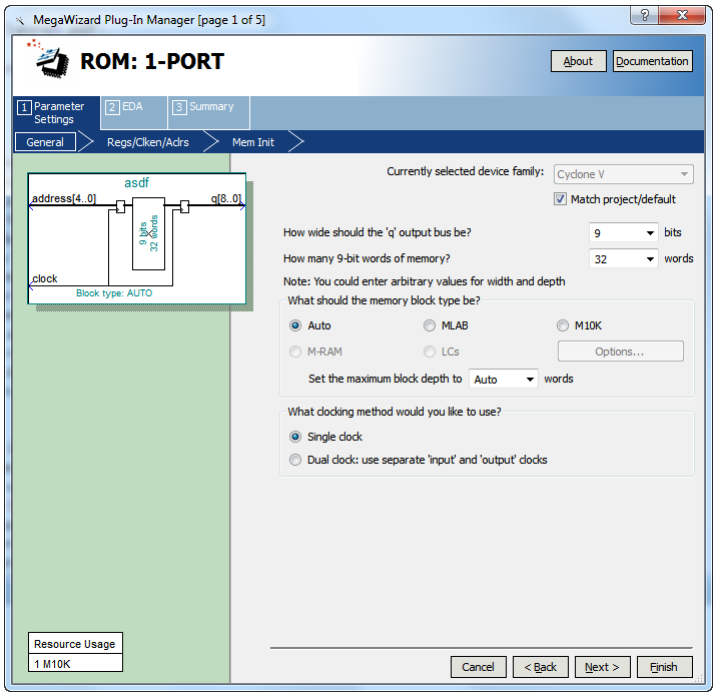

Figure 7: Specifying memory size.

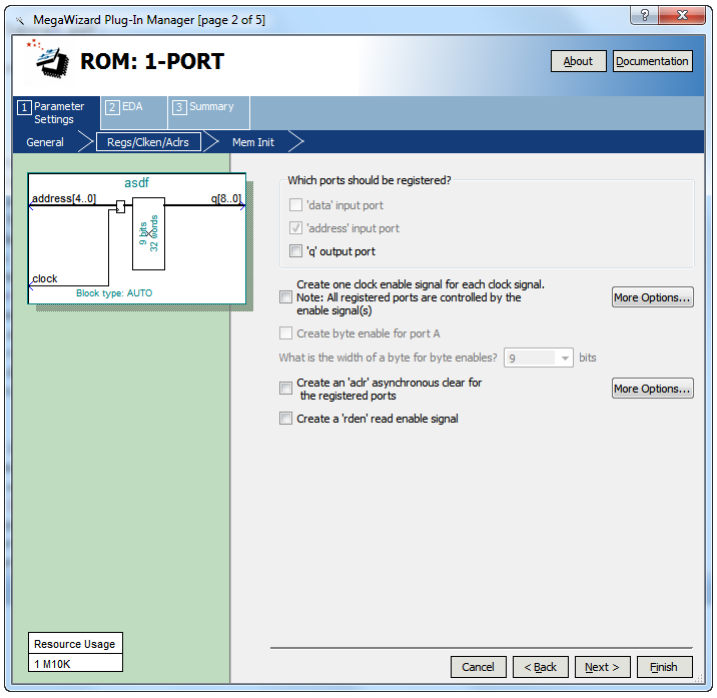

Figure 8: Specifying which memory ports are registered.

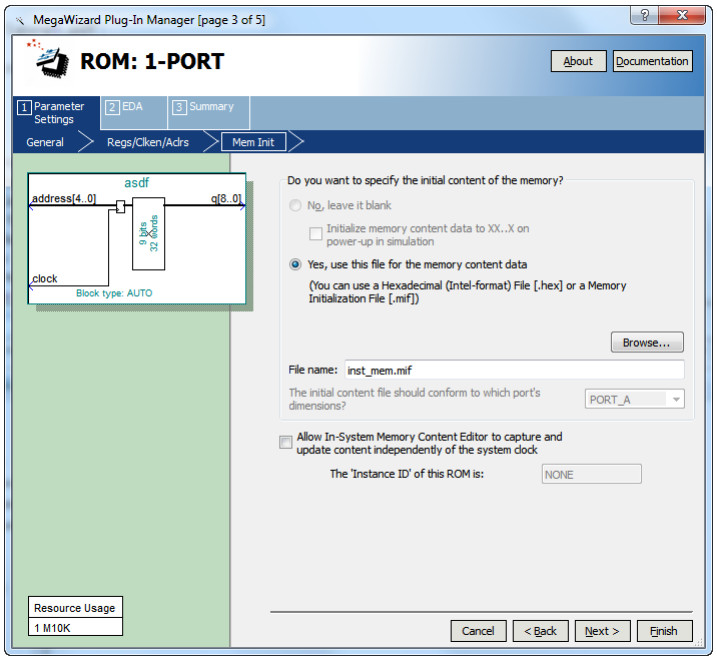

Figure 9: Specifying a memory initialization file (MIF).

```
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;
. . . (some lines not shown)
1F : 000000000;

END;
```

Figure 10: An example memory initialization file (MIF).

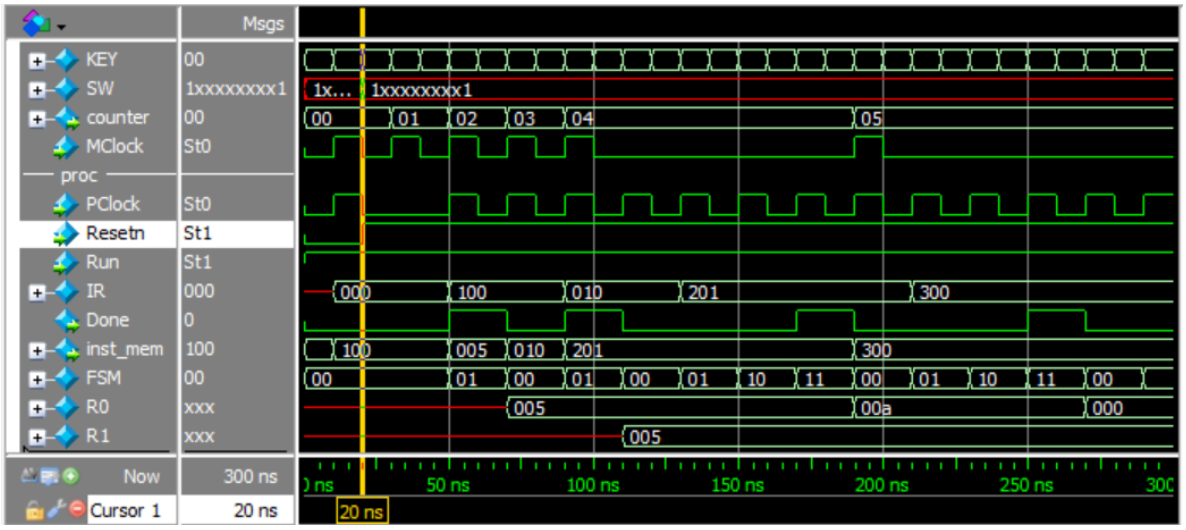

Figure 11: An example simulation output using the MIF in Figure 10.

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, as well as other capabilities—they are discussed in the next lab exercise.