Laboratory Exercise 1

A Simple Processor

Figure 1 shows a digital system that contains a number of 16-bit registers, a multiplexer, an adder/subtracter, and a control unit (finite state machine). Information is input to this system via the 16-bit *DIN* input, which is loaded into the *IR* register. Data can be transferred through the 16-bit wide multiplexer from one register in the system to another, such as from register *IR* into one of the *general purpose* registers $r0, \ldots, r7$. The multiplexer's output is called *Buswires* in the figure because the term *bus* is often used for wiring that allows data to be transferred from one location in a system to another. The FSM controls the *Select* lines of the multiplexer, which allows any of its inputs to be transferred to any register that is connected to the bus wires.

The system can perform different operations in each clock cycle, as governed by the FSM. It determines when particular data is placed onto the bus wires and controls which of the registers is to be loaded with this data. For example, if the FSM 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*.

Addition or subtraction of signed numbers is performed by using the multiplexer to first place one 16-bit number onto the bus wires, and then loading this number into register *A*. Once this is done, a second 16-bit number is placed onto the bus, the adder/subtracter performs the required operation, and the result is loaded into register *G*. The data in *G* can then be transferred via the multiplexer to one of the other registers, as required.

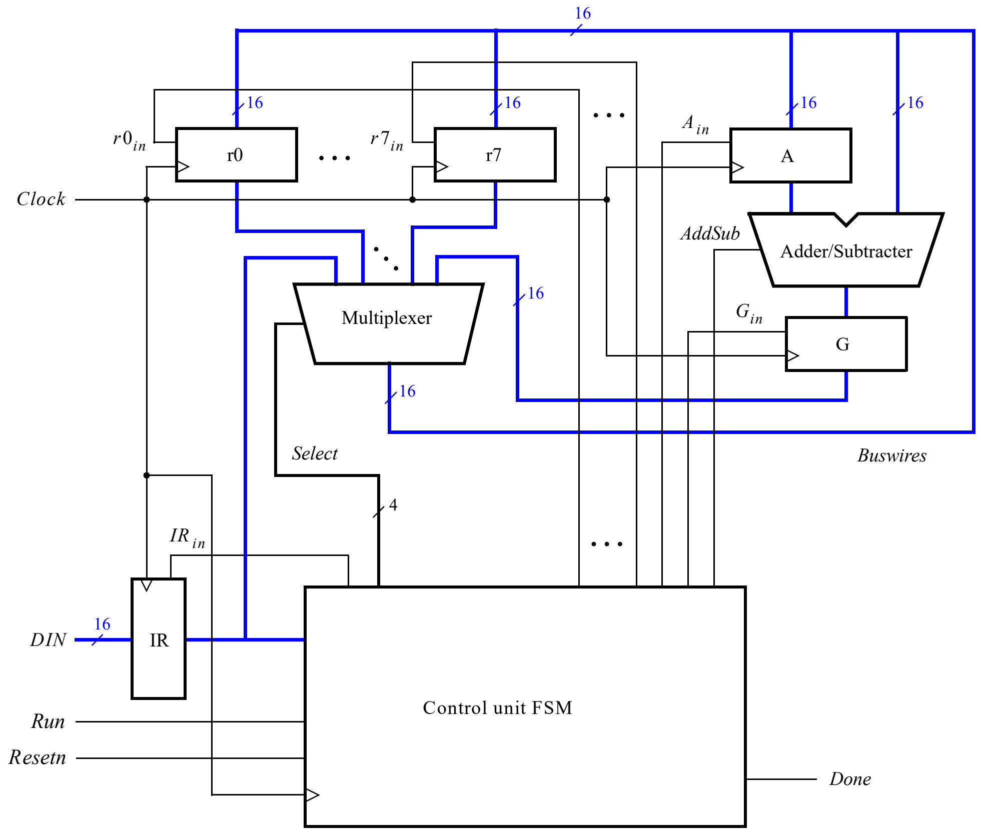


Figure 1: A digital system.

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* ← *Op2* is that the second operand, *Op2*, is loaded into register *rX*. The operand *Op2* can be either a register, *rY*, or *immediate data*, #*D*.

Instruction		Function performed
mv	*rX*, $Op2$	*rX* ← $Op2$
mvt	*rX*, #*D*	$rX_{15-8} \leftarrow D_{15-8}$
add	*rX*, $Op2$	*rX* ← *rX* + $Op2$
sub	*rX*, $Op2$	*rX* ← *rX* − $Op2$

Table 1: Instructions performed in the processor.

Instructions are loaded from the external input *DIN*, and stored into the *IR* register, using the connection indicated in Figure 1. Each instruction is *encoded* using a 16-bit format. If $Op2$ specifies a register, then the instruction encoding is `III0XXX000000YYY`, where `III` specifies the instruction, `XXX` gives the *rX* register, and `YYY` gives the *rY* register. If $Op2$ specifies immediate data #*D*, then the encoding is `III1XXXDDDDDDDDD`, where the field `DDDDDDDDD` represents a nine-bit *signed* (2's complement) value. 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 *mvt*, 010 for *add*, and 011 for *sub*.

The *mv* instruction (*move*) copies the contents of one register into another, using the syntax `mv rX, rY`. It can also be used to initialize a register with immediate data, as in `mv rX, #D`. Since the data *D* is represented inside the encoded instruction using only nine bits, the processor has to *sign-extend* the data, as in $D_8D_8D_8D_8D_8D_8D_8D_{8-0}$, before loading it into register *rX*. The *mvt* instruction (*move top*) is used to initialize the most-significant byte of a register. The `mvt rX, #D` loads the 16-bit value D_{15-8}`00000000` into *rX*. As an example, to load register $r0$ with the value `0xFF00`, you would use the instruction `mvt r0,#0xFF00`. The instruction `add rX,rY` produces the sum *rX* + *rY* and loads the result into *rX*. The instruction `add rX,#D` produces the sum *rX* + *D*, where *D* is sign-extended to 16 bits, and saves the result in *rX*. The *sub* instruction generates either *rX* − *rY*, or *rX* − #*D* and loads the result into *rX*.

Some instructions, such as an *add* or *sub*, take a few clock cycles to complete, because multiple transfers have to be performed across the bus. The finite state machine in the processor "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. The only control signal asserted in time step T_0, for all instructions, is IR_{in}. The meaning of *Select* = *rY* or *IR* in the table is that the multiplexer selects either register *rY* or the immediate data in *IR*, depending on the value of $Op2$. For the *mv* instruction, when *IR* is selected the multiplexer outputs `DDDDDDDDD` sign-extended to 16 bits, and for *mvt* the multiplexer outputs `DDDDDDDD00000000`. Only signals from Figure 1 that have to be asserted in each time step are listed in Table 1; all other signals are not asserted. The meaning of *AddSub* in step T_2 of the *sub* instruction is that this signal is set to 1, and this setting causes the adder/subtracter unit to perform subtraction using 2's-complement arithmetic.

The processor in Figure 1 can perform various tasks by using a sequence of instructions. For example, the sequence below loads the number 28 into register $r0$ and then calculates, in register $r1$, the 2's complement value −28.

```
mv     r0, #28        // original number = 28
mvt    r1, #0xFF00
add    r1, #0x00FF    // r1 = 0xFFFF
sub    r1, r0         // r1 = 1's-complement of r0
add    r1, #1         // r1 = 2's-complement of r0 = -28
```

This sequence of instructions produces the same result as the single instruction `mv r1,#-28`.

	T_0	T_1	T_2	T_3
mv	IR_{in}	*Select* = *rY* or *IR*, rX_{in}, *Done*		
mvt	IR_{in}	*Select* = *IR*, rX_{in}, *Done*		
add	IR_{in}	*Select* = *rX*, A_{in}	*Select* = *rY* or *IR*, G_{in}	*Select* = *G*, rX_{in}, *Done*
sub	IR_{in}	*Select* = *rX*, A_{in}	*Select* = *rY* or *IR*, *AddSub*, G_{in}	*Select* = *G*, rX_{in}, *Done*

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

Part I

Implement the processor shown in Figure 1 using Verilog code, as follows:

1. Make a new folder for this part of the exercise. Part of the Verilog code for the processor is shown in parts a to c of Figure 2, and a more complete version of the code is provided with this exercise, in a file named *proc.v*. You can modify this code to suit your own coding style if desired—the provided code is just a suggested solution. Fill in the missing parts of the Verilog code to complete the design of the processor.

```
module proc(DIN, Resetn, Clock, Run, Done);
    input [15:0] DIN;
    input Resetn, Clock, Run;
    output Done;

    parameter T0 = 2'b00, T1 = 2'b01, T2 = 2'b10, T3 = 2'b11;
    ... declare variables
    assign III = IR[15:13];
    assign IMM = IR[12];
    assign rX = IR[11:9];
    assign rY = IR[2:0];
    dec3to8 decX (rX_in, rX, R_in); // produce r0 - r7 register enables

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

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

```
parameter mv = 3'b000, mvt = 3'b001, add = 3'b010, sub = 3'b011;
// selectors for the BusWires multiplexer
parameter SEL_R0 = 4'b0000, SEL_R1 = 4'b0001, ..., SEL_R7 = 4'b0111,
    SEL_G = 4'b1000, SEL_D = 4'b1001, SEL_D8 = 4'b1010;

// control FSM outputs
always @(*) begin
    rX_in = 1'b0; Done = 1'b0; ... // default values for variables
    case (Tstep_Q)
        T0: // store DIN into IR
            IR_in = 1'b1;
        T1: // define signals in time step T1
            case (III)
                mv: begin
                    if (!Imm) Sel = rY;    // mv rX, rY
                    else Sel = SEL_D;      // mv rX, #D
                    rX_in = 1'b1;          // enable the rX register
                    Done = 1'b1;
                end
                mvt: // mvt rX, #D
                ...
            endcase
        T2: // define signals in time step T2
            case (III)
                ...
            endcase
        T3: // define signals in time step T3
            case (III)
                ...
            endcase
        default: ;
    endcase
end

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

regn reg_0 (BusWires, Resetn, R_in[0], Clock, r0);
regn reg_1 (BusWires, Resetn, R_in[1], Clock, r1);
...
regn reg_7 (BusWires, Resetn, R_in[7], Clock, r7);

... instantiate other registers and the adder/subtracter unit
```

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

```
    // define the internal processor bus
    always @(*)
        case (Sel)
            SEL_R0: BusWires = R0;
            SEL_R1: BusWires = R1;
            ...
            SEL_G: BusWires = G;
            SEL_D: BusWires = ...;     // used for mv, add, ..., with #D
            SEL_D8: BusWires = ...;    // used for mvt
            default: BusWires = 16'bxxxxxxxxxxxxxxxx;
        endcase
endmodule

module dec3to8(E, W, Y);
    input E; // enable
    input [2:0] W;
    output [0:7] Y;
    reg [0:7] Y;

    always @(*)
        if (E == 0)
            Y = 8'b00000000;
        else
            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
endmodule
```

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

2. Set up the required subfolder and files so that your Verilog code can be compiled and simulated using the ModelSim Simulator to verify that your processor works properly. An example result produced by using *ModelSim* for a correctly-designed circuit is given in Figure 3. It shows the value `0x101C` being loaded into *IR* from *DIN* at time 30 ns. This pattern represents the instruction `mv r0,#28`, where the immediate value $D = 28$ (`0x1C`) is loaded into $r0$ on the clock edge at 50 ns. The simulation results then show the instruction `mvt r1,#0xFF00` at 70 ns, `add r0,#0xFF` starting at 110 ns, and `sub r1,r0` starting at 190 ns.

   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 along with this exercise.

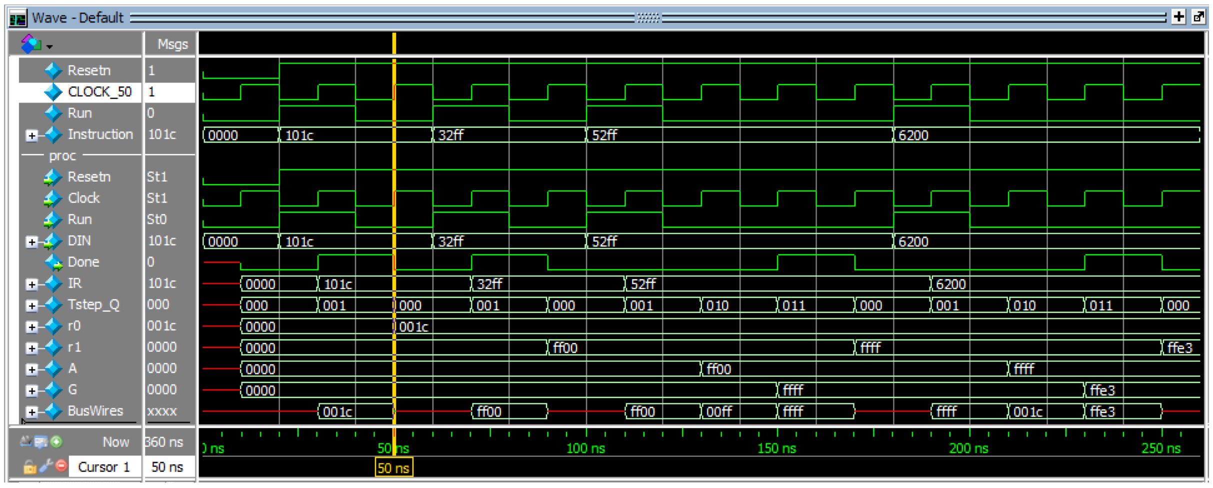


Figure 3: Simulation results for the processor.

Part II

In this part we will design the circuit illustrated 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.

You are to "implement" the circuit in Figure 4 using an Intel FPGA device. There are two possible approaches: you can use an actual FPGA laboratory board, or you can use a simulation-based approach. The former choice can be taken if you have access to an appropriate laboratory board, such as the DE1-SoC board. If not, then the latter approach should be followed by simulating your designed circuit using the *ModelSim* and/or *DESim* software.

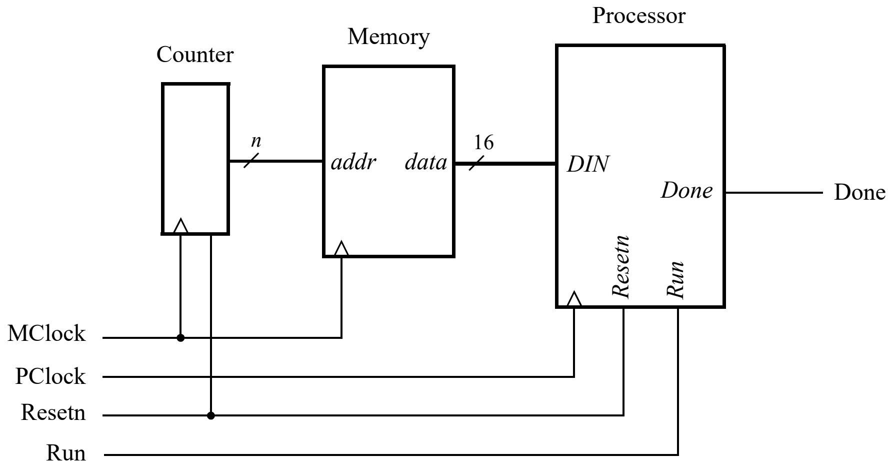


Figure 4: Connecting the processor to a memory module and counter.

Do the following:

1. A sample top-level Verilog file that instantiates the processor, memory module, and counter is shown in Figure 5. This code is provided, along with this exercise, in a file named *part2.v*. It is the top-level Verilog file for this part of the exercise. The code instantiates a memory module called *inst_mem*. A diagram of this memory module is depicted in Figure 6. Since this module has only a read port, and no write port, it

is called a *synchronous read-only memory (synchronous ROM)*. 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.

The synchronous ROM is defined in a Verilog source-code file named *inst_mem.v*, which is provided along with this exercise. You can use the provided file, or you can create one yourself (if you want to see how this is done) by using the Quartus software. The instructions for creating the *inst_mem.v* file using the Quartus software are given below. If you do not wish to perform these steps, and just want to make use of the provided file, then skip to item 3, below.

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

    wire Done, Resetn, PClock, MClock, Run;
    wire [15:0] DIN;
    wire [4:0] pc;

    assign Resetn = SW[0];
    assign MClock = KEY[0];
    assign PClock = KEY[1];
    assign Run = SW[9];
    proc U1 (DIN, Resetn, PClock, Run, Done);
    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.

2. A Quartus project file is provided along with this part of the exercise. Use the Quartus software to open this project, which is called *part2.qpf*. The top-level file in this Quartus project is *part2.v*. 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 dialogue to create a memory that has one 16-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.

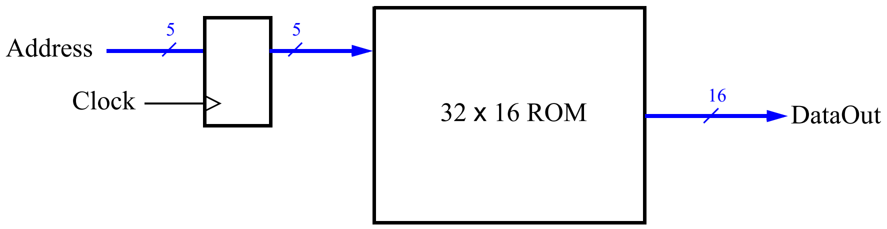
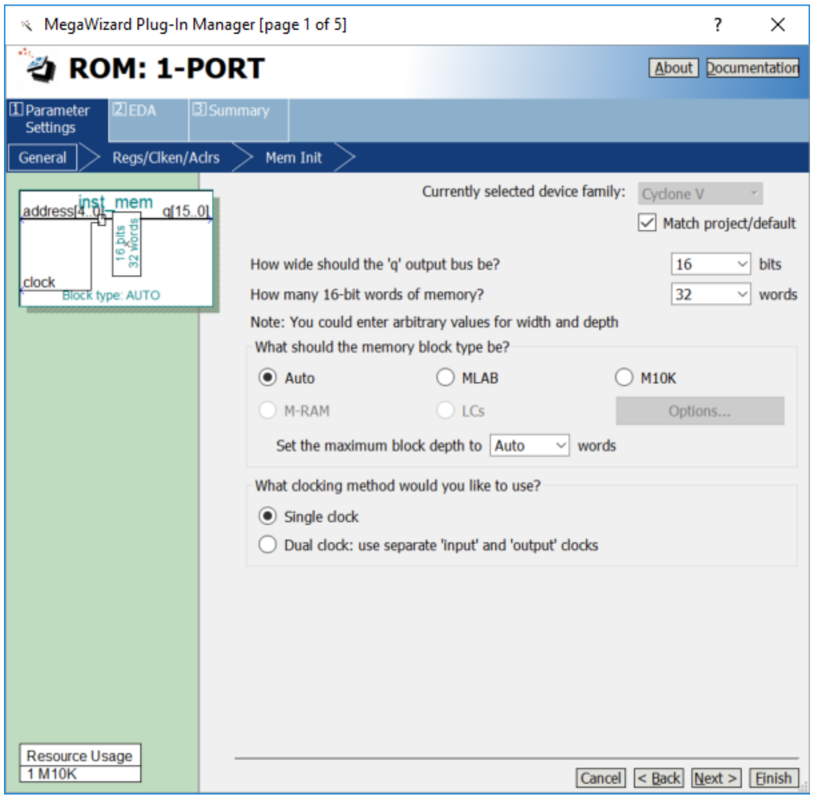


Figure 6: The 32 x 16 ROM with address register.

We have specified a file named *inst_mem.mif*, which then has to be created in the folder that contains the Quartus project. Clicking `Next` two more times will advance to the `Summary` screen, which lists the names of files that will be created for the memory IP. You should select *only* the Verilog file *inst_mem.v*. Make sure that none of the other types of files are selected, and then click `Finish`.



Currently selected device family: Cyclone V

[✓] Match project/default

How wide should the 'q' output bus be? 16 bits

How many 16-bit words of memory? 32 words

Note: You could enter arbitrary values for width and depth

What should the memory block type be?

[● Auto] MLAB M10K
M-RAM LCs

Set the maximum block depth to Auto words

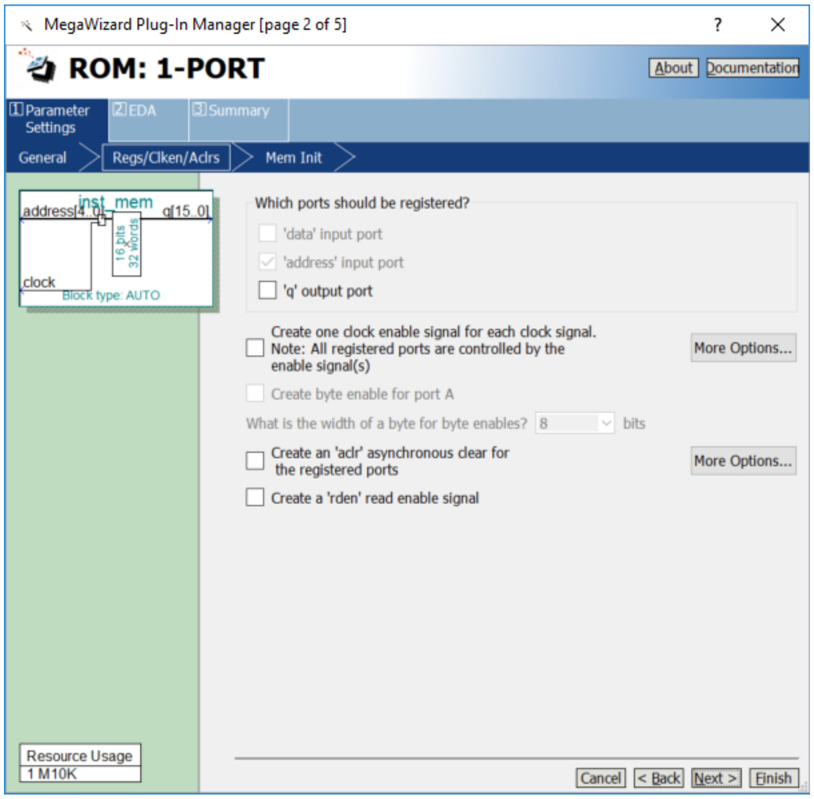
What clocking method would you like to use?

[● Single clock]
Dual clock: use separate 'input' and 'output' clocks

Figure 7: Specifying memory size.

3. As described above, you need to provide a memory initialization file (MIF) called *inst_mem.mif* to specify the contents of the ROM. 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.

4. The Verilog code in Figure 5 includes the appropriate port names for implementation of the design on an FPGA board, like the DE1-SoC. 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 Run signal is displayed on $LEDR_9$ and *Done* is connected to $LEDR_0$.


MegaWizard Plug-In Manager [page 2 of 5]

ROM: 1-PORT

Which ports should be registered?

- [ ] 'data' input port
- [x] 'address' input port
- [ ] 'q' output port

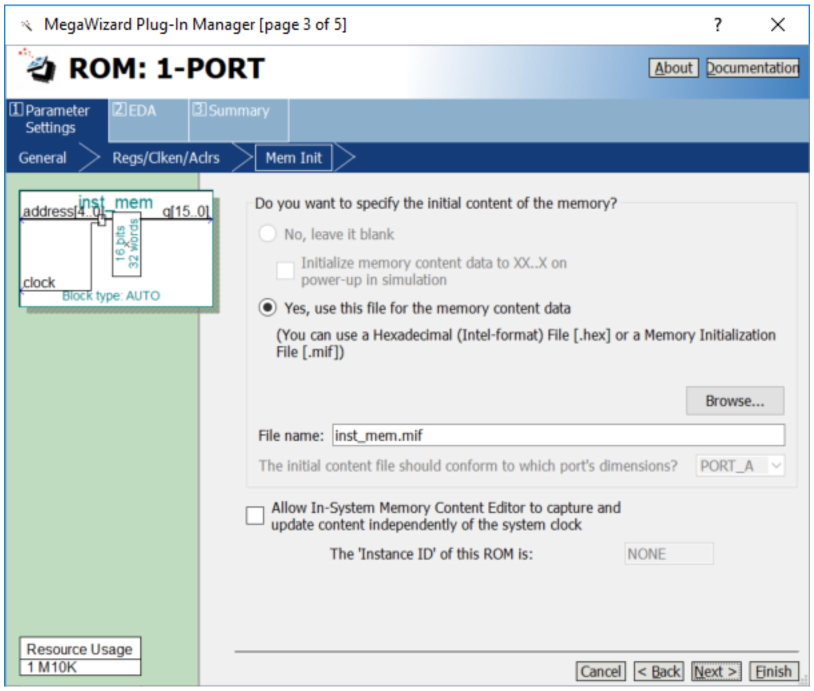
- [ ] Create one clock enable signal for each clock signal. Note: All registered ports are controlled by the enable signal(s)
- [ ] Create byte enable for port A

What is the width of a byte for byte enables? 8 bits

- [ ] Create an 'aclr' asynchronous clear for the registered ports
- [ ] Create a 'rden' read enable signal

Resource Usage: 1 M10K

Figure 8: Specifying which memory ports are registered.


MegaWizard Plug-In Manager [page 3 of 5]

ROM: 1-PORT

Do you want to specify the initial content of the memory?

- No, leave it blank
  - [ ] Initialize memory content data to XX..X on power-up in simulation
- [✓ Yes, use this file for the memory content data]

(You can use a Hexadecimal (Intel-format) File [.hex] or a Memory Initialization File [.mif])

File name: inst_mem.mif

The initial content file should conform to which port's dimensions? PORT_A

- [ ] Allow In-System Memory Content Editor to capture and update content independently of the system clock

The 'Instance ID' of this ROM is: NONE

Resource Usage: 1 M10K

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

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 along with this exercise.

6. Once your simulations show a properly-working circuit, you may wish to "implement" your design in an FPGA board. As mentioned previously, there are two possibilities: using an FPGA laboratory board, or a simulation-based approach.

   To target your design to an FPGA board, you would need to compile the Verilog code with the Quartus software; then, the resulting circuit could be downloaded into your available hardware, such as the DE1-SoC board.

   To use the *DESim* software to "implement" your Verilog code, you would need to create a *DESim project* and then compile and simulate your code within the graphical user interface provided by this tool.

   Regardless of whether you are using a hardware or simulation-based approach, the functionality of your circuit can be demonstrated 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.

```
DEPTH = 32;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN
00 : 0001000000011100;     % mv  r0, #28         %
01 : 0011001011111111;     % mvt r1, #0xFF00     %
02 : 0101001011111111;     % add r1, #0xFF       %
03 : 0110001000000000;     % sub r1, r0          %
04 : 0101001000000001;     % add r1, #1          %
05 : 0000000000000000;
. . . (some lines not shown)
1F : 0000000000000000;
END;
```

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

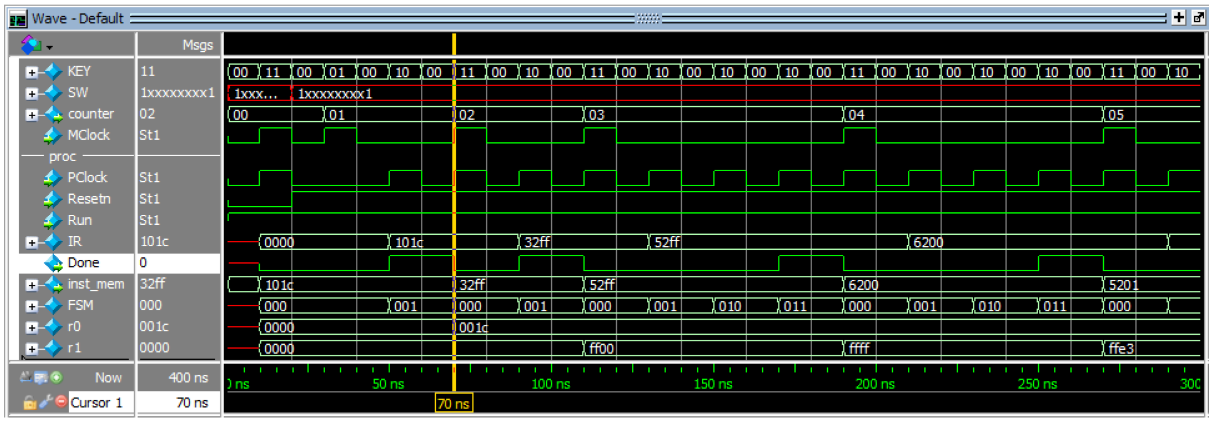


Figure 11: An example simulation output using the MIF in Figure 10.

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.