Laboratory Exercise 9

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.

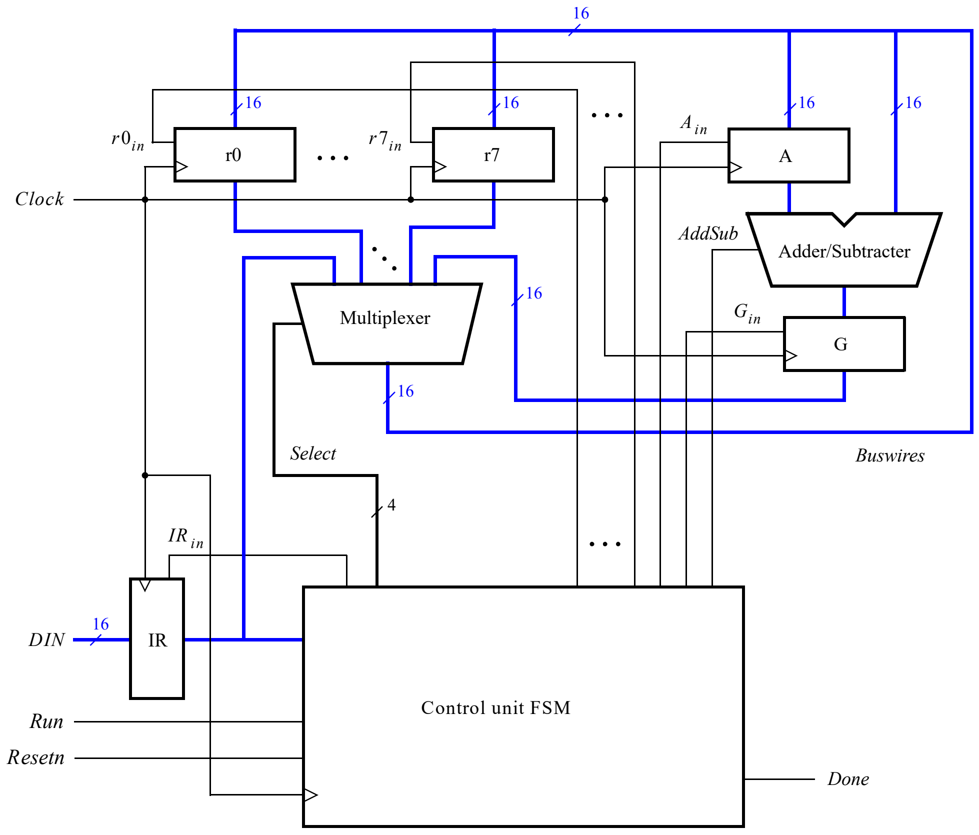


Figure 1: A digital system.

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 $rX \leftarrow Op2$ 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*, *Op*2	$rX \leftarrow Op2$
mvt	*rX*, *#D*	$rX_{15-8} \leftarrow D_{7-0}$
add	*rX*, *Op*2	$rX \leftarrow rX + Op2$
sub	*rX*, *Op*2	$rX \leftarrow rX - Op2$

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 *Op*2 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 *Op*2 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 instruction `mvt rX, #D` loads the 16-bit value $\texttt{D}_{7-0}\texttt{00000000}$ into *rX*. As an example, to load register $r0$ with the value `0xFF00`, you would use the instruction `mvt r0, #0xFF`. 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 and puts onto the bus (*BusWires*) either the contents of register *rY* or the immediate data in *IR*, depending on the value of *Op*2. 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, #0xFF
add    r1, #0xFF       // 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 VHDL code, as follows:

1. Make a new folder for this part of the exercise. Part of the VHDL 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.vhd*. 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 VHDL code to complete the design of the processor.

```
library ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY proc IS
    PORT ( DIN                  : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
           Resetn, Clock, Run   : IN  STD_LOGIC;
           Done                 : BUFFER  STD_LOGIC);
END proc;

ARCHITECTURE Behavior OF proc IS
    ... declare components ...
    TYPE State_type IS (T0, T1, T2, T3);
    SIGNAL Tstep_Q, Tstep_D: State_type;
    ... declare other signals ...
    CONSTANT mv : STD_LOGIC_VECTOR(2 DOWNTO 0) := "000";
    ... encodings for each instruction ...
    CONSTANT SEL_R0 : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0000";
    ... selectors for the bus multiplexer
    CONSTANT SEL_G : STD_LOGIC_VECTOR(3 DOWNTO 0) := "1000";
    CONSTANT SEL_IR8_IR8_0 : STD_LOGIC_VECTOR(3 DOWNTO 0) := "1001";
    CONSTANT SEL_IR7_0_0 : STD_LOGIC_VECTOR(3 DOWNTO 0) := "1010" ;
BEGIN
```

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

```
III <= IR(15 DOWNTO 13);
Imm <= IR(12);
rX <= IR(11 DOWNTO 9);
rY <= IR(2 DOWNTO 0);
decX: dec3to8 PORT MAP (rXin, rX, Rin); -- r0 - r7 register enables

statetable: PROCESS(Tstep_Q, Run, Done)
BEGIN
    CASE Tstep_Q IS
        WHEN T0 =>    -- data is loaded into IR in this time step
            IF Run = '0' THEN Tstep_D <= T0;
            ELSE Tstep_D <= T1;
            END IF;
        WHEN T1 =>   ...
                ...
    END CASE;
END PROCESS;

controlsignals: PROCESS (Tstep_Q, III, Imm, rX, rY)
BEGIN
    rXin <= '0'; Done <= '0'; ... -- default assignments
    CASE Tstep_Q IS
        WHEN T0 => -- store DIN into IR
            IRin <= '1';
        WHEN T1 => -- define signals in time step T1
            CASE III IS
                WHEN mv =>
                    IF Imm = '0' THEN Sel <= '0' & rY;  -- mv rX, rY
                    ELSE Sel <= SEL_IR8_IR8_0;          -- mv rX, #D
                    END IF;
                    rXin <= '1';                        -- enable rX
                    Done <= '1';
                WHEN mvt =>                             -- mvt Rx, #D
                    ...
                ...
            END CASE;
        WHEN T2 => -- define signals in time step T2
            CASE III IS
                ...
            END CASE;
        WHEN T3 => -- define signals in time step T3
            CASE III IS
                ...
            END CASE;
    END CASE;
END PROCESS;

fsmflipflops: PROCESS (Clock, Resetn, Tstep_D)
BEGIN
    IF (Resetn = '0') THEN
        ...
```

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

```
    reg_0:  regn PORT MAP (BusWires, Resetn, Rin(0), Clock, R0);
    ...
    reg_A:  regn PORT MAP (BusWires, Resetn, Ain, Clock, A);
    reg_IR: regn PORT MAP (DIN, Resetn, IRin, Clock, IR);

    alu: PROCESS (AddSub, A, BusWires)
    BEGIN
        IF AddSub = ’0’ THEN
            ...
    END PROCESS;
    reg_G: regn PORT MAP (Sum, Resetn, Gin, Clock, G);

    busmux: PROCESS (Sel, R0, R1, R2, R3, R4, R5, R6, R7, G, IR)
    BEGIN
        CASE Sel IS
            WHEN SEL_R0 => BusWires <= R0;
            ...
            WHEN SEL_G => BusWires <= G;
            WHEN SEL_IR8_IR8_0 => BusWires <= (15 DOWNTO 9 => IR(8)) &
               IR(8 DOWNTO 0);
            WHEN SEL_IR7_0_0 => BusWires <= IR(7 DOWNTO 0) &
               "00000000";
            WHEN OTHERS => BusWires <= (OTHERS => ’0’);
        END CASE;
    END PROCESS;
END Behavior;

...

ENTITY dec3to8 IS
    PORT ( E    : IN    STD_LOGIC;
           W    : IN    STD_LOGIC_VECTOR(2 DOWNTO 0);
           Y    : OUT   STD_LOGIC_VECTOR(0 TO 7));
END dec3to8;

ARCHITECTURE Behavior OF dec3to8 IS
BEGIN
    PROCESS (E, W)
    BEGIN
        IF E = ’0’THEN
            Y <= "00000000";
        ELSE
            CASE W IS
                WHEN "000" => Y <= "10000000";
                ...
                WHEN "111" => Y <= "00000001";
                WHEN OTHERS => Y <= "00000000";
            END CASE;
        END IF;
    END PROCESS;
END Behavior;
```

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

2. Set up the required subfolder and files so that your VHDL code can be compiled and simulated using the Questa or ModelSim Simulator to verify that your processor works properly. An example simulation result 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,#0xFF` 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. A sample VHDL testbench file, *testbench.vht*, execution script, *testbench.tcl*, and waveform file, *wave.do* are provided along with this exercise.

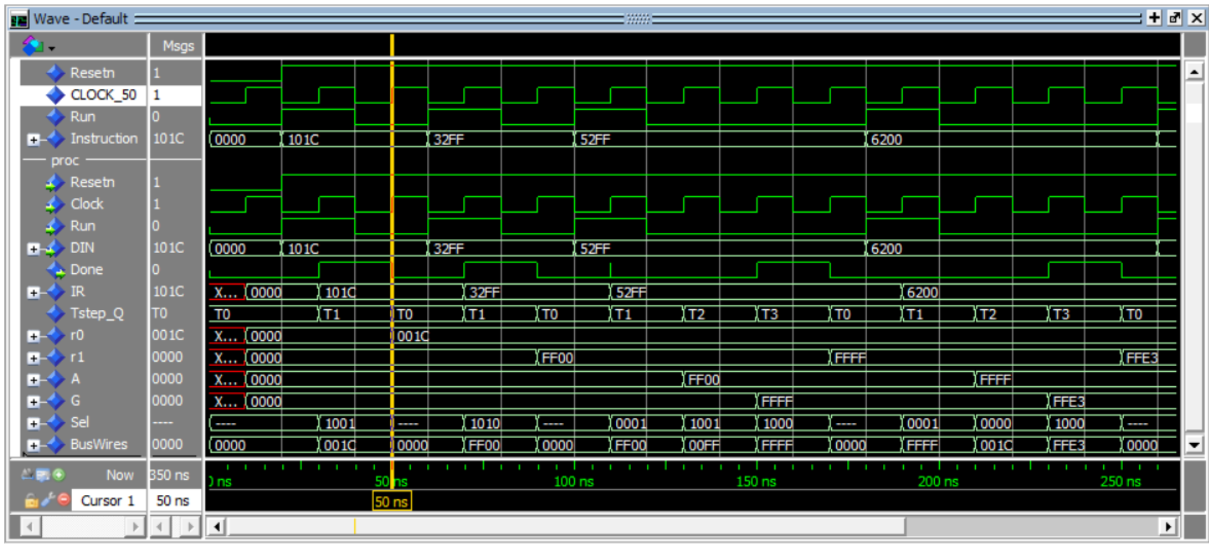


Figure 3: Simulation results for the processor.

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 in a suitable FPGA (for example, the DE1-SoC, DE10-Standard, or DE10-Lite) board. As part of the process for designing and debugging the circuit, you must use the Questa or ModelSim simulator to test your VHDL code's functionality. Also, you can use the DESim tool as a way of observing how your circuit will behave on an FPGA board, even if you do not have access to a physical board (for example, at home). You are encouraged to make use of the DESim tool to help in the design and debug of your VHDL code. This tool is available from `FPGAcademy.org`. To make it easy to use the DESim tool, the design files for this part of the exercise include all required DESim setup files. Perform the following steps:

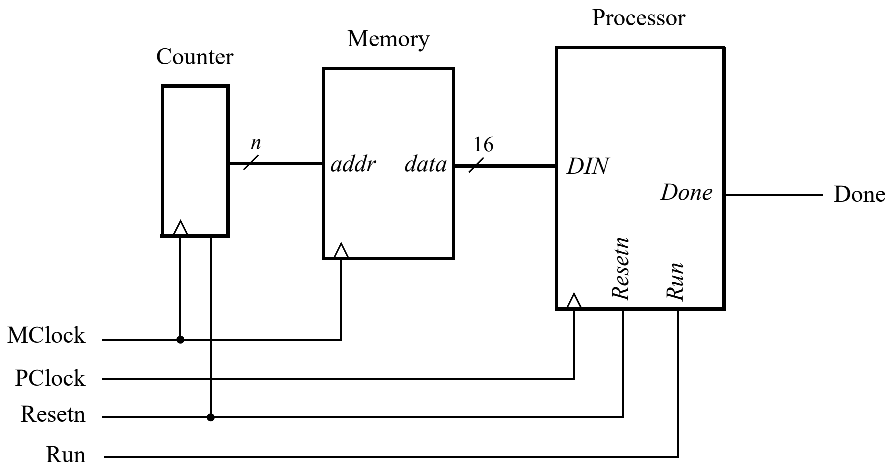


Figure 4: Connecting the processor to a memory module and counter.

1. A sample top-level VHDL 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.vhd*. 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 VHDL source-code file named *inst_mem.vhd*, 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 Prime software. The instructions for creating the *inst_mem.vhd* 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.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY part2 IS
    PORT ( KEY  : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
           SW   : IN  STD_LOGIC_VECTOR(9 DOWNTO 0);
           LEDR : OUT STD_LOGIC_VECTOR(9 DOWNTO 0));
END part2;

ARCHITECTURE Behavior OF part2 IS
   COMPONENT proc
      PORT ( DIN                  : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
             Resetn, Clock, Run : IN STD_LOGIC;
             Done                 : BUFFER STD_LOGIC);
   END COMPONENT;
   COMPONENT inst_mem
      PORT ( address : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
             clock   : IN STD_LOGIC ;
             q       : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
   END COMPONENT;
   COMPONENT count5
      PORT ( Resetn, Clock : IN STD_LOGIC;
             Q             : OUT STD_LOGIC_VECTOR(4 DOWNTO 0));
   END COMPONENT;

   SIGNAL Resetn, PClock, MClock, Run, Done : STD_LOGIC;
   SIGNAL DIN : STD_LOGIC_VECTOR(15 DOWNTO 0);
   SIGNAL pc : STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN
   Resetn <= SW(0);
   MClock <= KEY(0);
   PClock <= KEY(1);
   Run <= SW(9);
   U1: proc PORT MAP (DIN, Resetn, PClock, Run, Done);
   LEDR(0) <= Done;
   LEDR(9) <= Run;

   U2: inst_mem PORT MAP (pc, MClock, DIN);
   U3: count5 PORT MAP (Resetn, MClock, pc);
END Behavior;
```

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

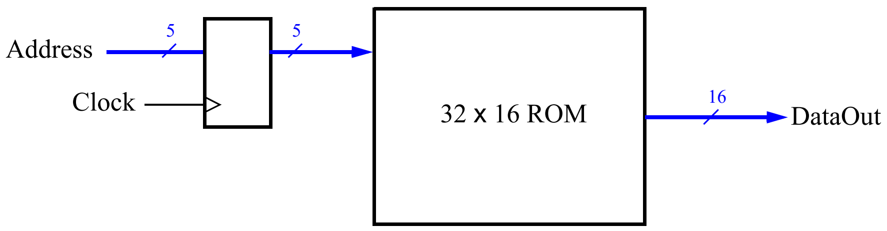
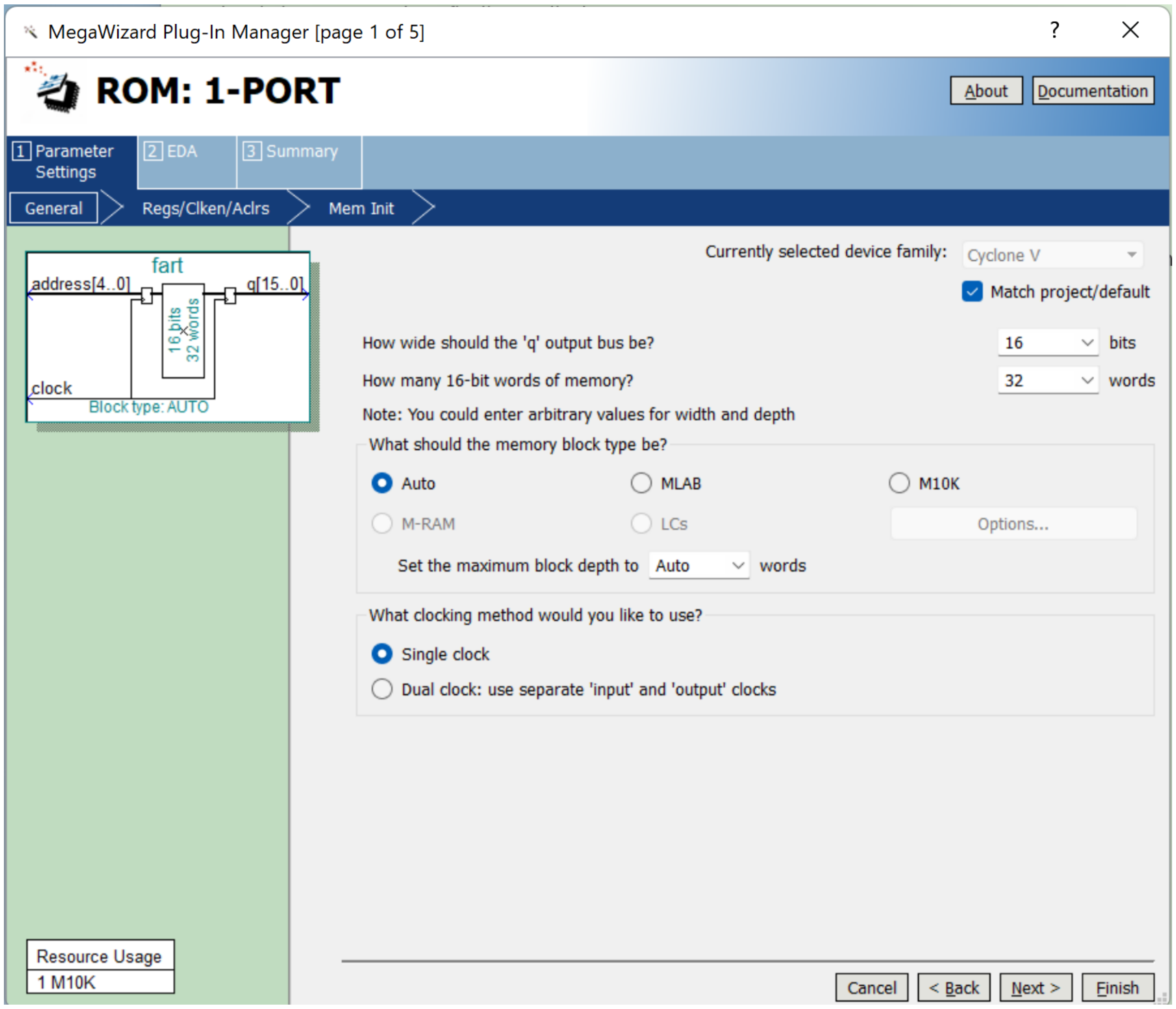


Figure 6: The 32 x 16 ROM with address register.

2. A Quartus Prime 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.vhd*. 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 VHDL as the type of output file to create, and give the file the name *inst_mem.vhd*.

   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. In Figure 8, under `Which ports should be registered?`, ensure that the `'q' output port` is *not* selected. To place processor instructions into the memory, you need to specify *initial values* for the memory. 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. 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 VHDL file *inst_mem.vhd*. 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 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 Options...

Set the maximum block depth to Auto words

What clocking method would you like to use?

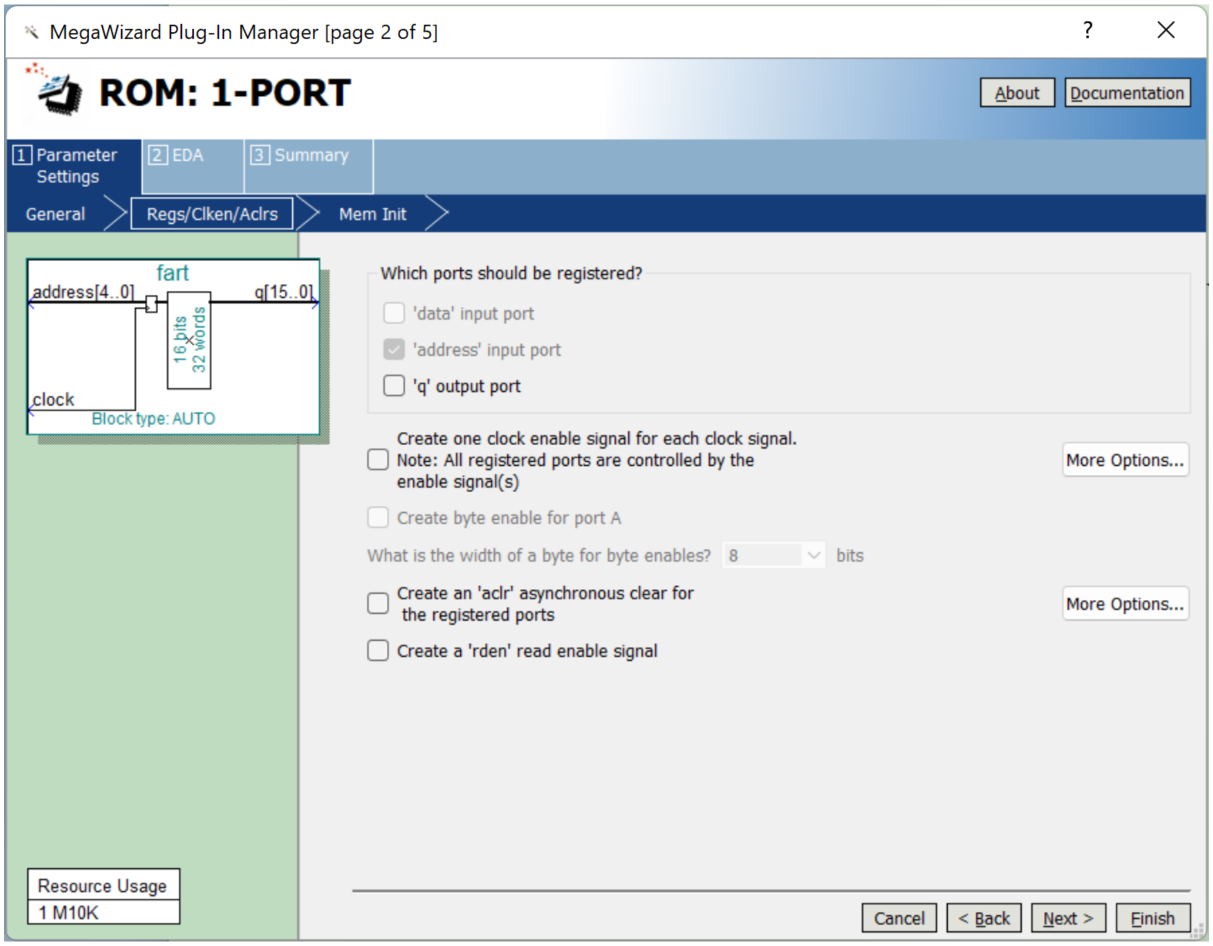
[✓ Single clock]

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 VHDL 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 $LEDR_0$.



Which ports should be registered?

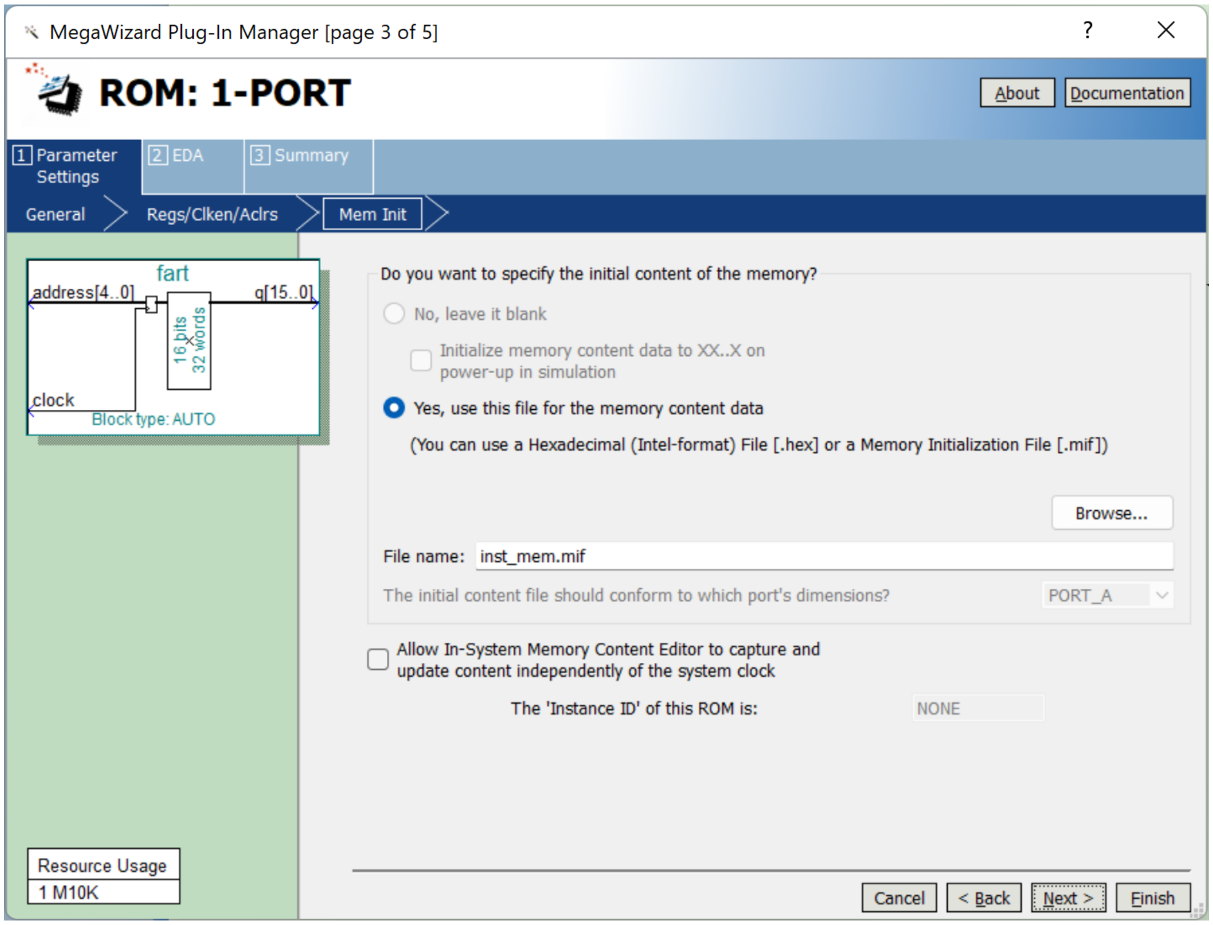
- [ ] 'data' input port
- [x] 'address' input port
- [ ] 'q' output port

- [ ] 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

Figure 8: Specifying which memory ports are registered.



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

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

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

6. Once your simulations show a properly-working circuit, you should implement your design on a DE1-SoC (or similar) board. Use the Quartus project files that are provided along with this exercise to compile your VHDL code with the Quartus Prime software, and then download the resulting circuit to the board.

   Demonstrate the functionality of your circuit by toggling the switches on the board 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, #0xFF      %
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).

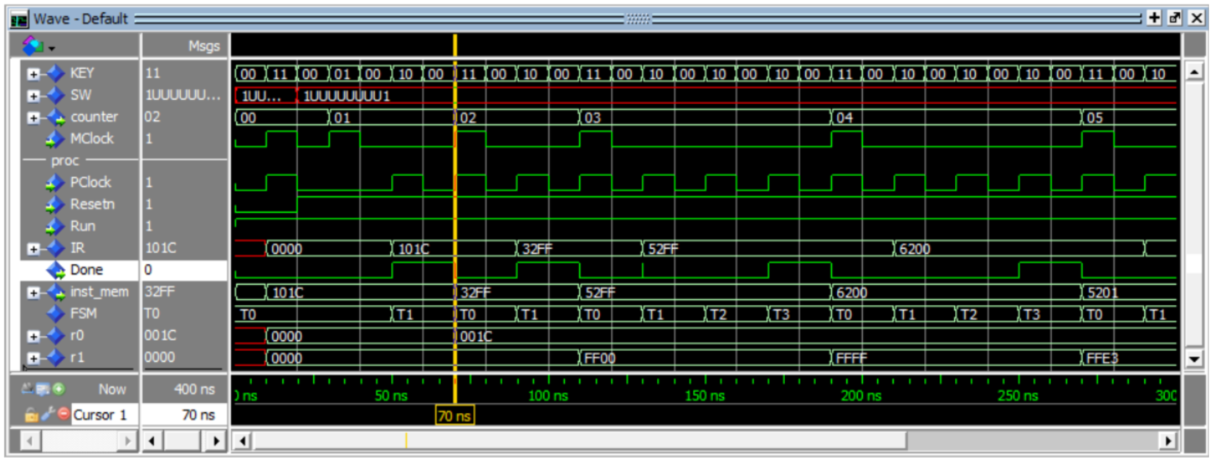


Figure 11: An example simulation output using the MIF in Figure 10.

Enhanced Processor

You can enhance your 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.