Laboratory Exercise 8

An Enhanced Processor

In Laboratory Exercise 7 we described a simple processor. In Part I of that exercise the processor itself was designed, and in Part II the processor was connected to an external counter and a memory unit. This exercise describes subsequent parts of the processor design. The numbering of figures and tables in this exercise are continued from those in Parts I and II of the preceding lab exercise.

In this exercise we will extend the capability of the processor so that the external counter is no longer needed, and so that the processor can perform read and write operations using memory or other devices. A schematic of the enhanced processor is given in Figure 12. In this figure registers $r0$ to $r6$ are the same as in Figure 1 of Lab 7, but register $r7$ has been changed to a counter. This counter is used to provide the addresses in the memory from which the processor's instructions are read; in the preceding lab exercise, a counter external to the processor was used for this purpose. We will refer to $r7$ as the processor's *program counter* (*pc*), because this terminology is common for real processors available in the industry. When the processor is reset, *pc* is set to address 0. At the start of each instruction (in time step T_0) the value of *pc* is used as an address to read an instruction from the memory. The instruction returned from the memory is stored into the *IR* register and the *pc* is automatically incremented to point to the next instruction.

The processor's control unit increments *pc* by using the *pc_incr* signal, which is just an enable on this counter. It is also possible to load an arbitrary address into *pc* by having the processor execute an instruction in which the destination register is specified as *pc*. In this case the control unit uses pc_{in} to perform a *load* of the counter. Thus, the processor can execute instructions at any address in the memory, as opposed to only being able to execute instructions that are stored at successive addresses. The current contents of *pc*, which always has the address of the *next* instruction to be executed, can be copied into another register if needed by using a *mv* instruction.

The enhanced processor will have four new instructions, which are listed in Table 3. The *ld* (load) instruction *reads* data into register *rX* from the external memory address specified in register *rY*. Thus, the syntax [*rY*] means that the contents of register *rY* are used as an *external address*. The *st* (store) instruction *writes* the data contained in register *rX* into the memory address found in *rY*. The *and* instruction is similar to the *add* and *sub* instructions that were introduced in Lab 7. This instruction extends the adder/subtracter unit in the processor into an *arithmetic logic unit*. Besides performing addition and subtraction, it has the ability to generate a bit-wise logical AND (&) of the destination register *rX* with the second operand *Op2*. As discussed in Lab 7, the operand *Op2* can be either another register *rY*, or immediate data *#D*.

The *b{cond}* instruction in Table 3 is used to cause a processor *branch*, which means to change the program counter (*pc*) to the address of a specific instruction. The *cond* part of the branch instruction is optional and represents a *condition*. The instruction loads the address *Label* into *pc* only if the specified condition evaluates to true. An example of a condition is *eq*, which stands for *equal* (to zero). The instruction `beq Label` will load the address *Label* into *pc* if the last result produced by the arithmetic logic unit, which is stored in register G, was 0. The 3-bit register F shown in Figure 12 is required for the *b{cond}* instruction, which is discussed in more detail in Part V.

Operation	Function performed
ld rX, [*rY*]	*rX* ← [*rY*]
st rX, [*rY*]	[*rY*] ← *rX*
and rX, *Op2*	*rX* ← *rX* & *Op2*
b{cond} Label	if (*cond*), *pc* ← *Label*

Table 3: New instructions in the enhanced processor.

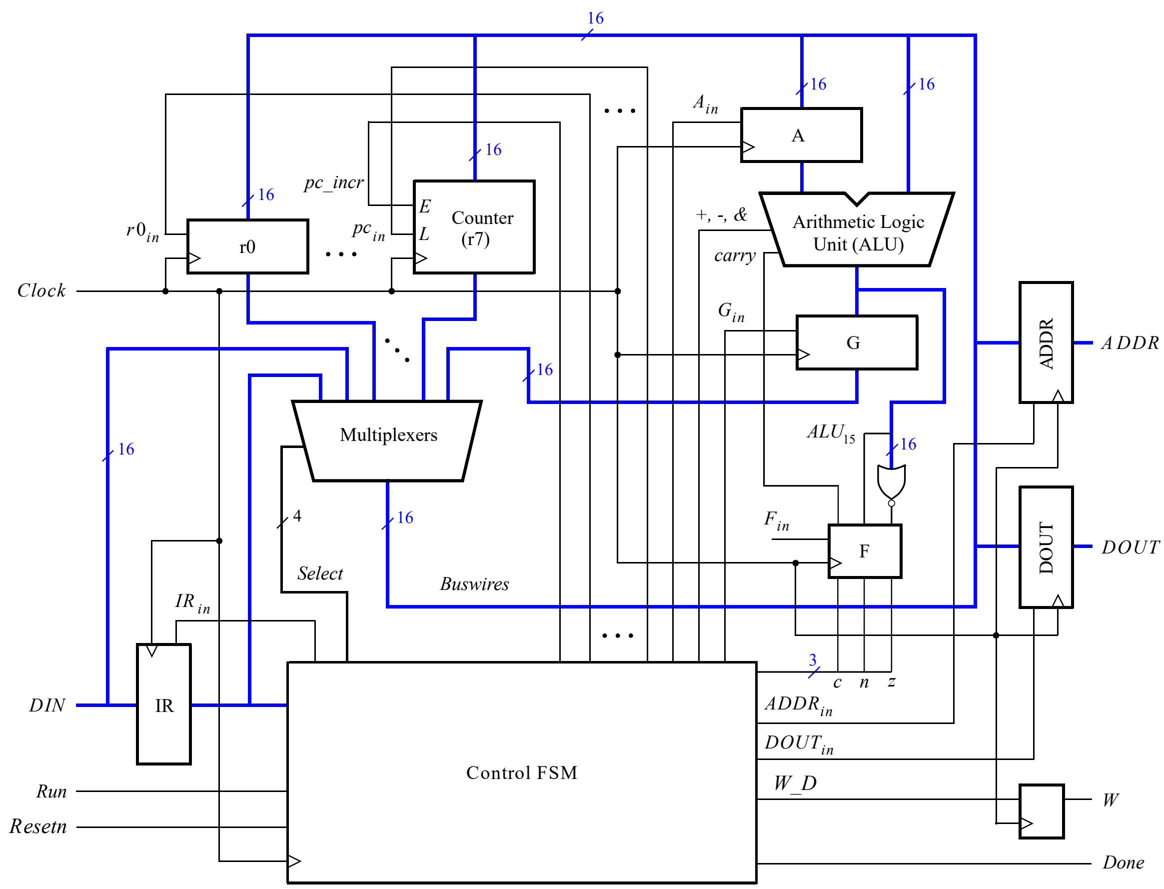

Figure 12: An enhanced version of the processor.

Recall from Lab 7 that instructions are encoded using a 16-bit format. For instructions that specify *Op2* as a register the encoding is `III0XXX000000YYY`, and if *Op2* is an immediate constant the format is `III1XXXDDDDDDDDD`. You should use these same encodings for this exercise. Assume that `III` = 100 for the *ld* instruction, 101 for *st*, and 110 for *and*. The encoding for *b{cond}* is discussed in Part V of this exercise.

Figure 12 shows two registers in the processor that are used for data transfers. The *ADDR* register is used to send addresses to an external device, such as a memory module, and the *DOUT* register is used by the processor to provide data that is to be stored outside of the processor. One use of the *ADDR* register is for reading, or *fetching*, instructions from memory; when the processor wants to fetch an instruction, the contents of *pc* are transferred across the bus and loaded into *ADDR*. This address is provided to the memory.

In addition to fetching instructions, the processor can read data at any address by using the *ADDR* register. Both data and instructions are read into the processor on the *DIN* input port. The processor can write data for storage at an external address by placing this address into the *ADDR* register, placing the data to be stored into the *DOUT* register, and asserting the output of the *W* (*Write*) flip-flop to 1.

Connecting the Processor to External Devices

Figure 13 illustrates how the enhanced processor can be connected to memory and other devices. The memory unit in the figure is 16-bits wide and 256-words deep. A diagram of this memory is given in Figure 14. It supports both read and write operations and therefore has both address and data inputs, as well as a write-enable input. As depicted in Figure 14, the memory has a clock input that is used to store the address, data, and write enable inputs into registers. This type of memory unit is called a *synchronous* static random access memory (SSRAM).

Figure 13 also includes a 9-bit output port (register) that can be used to store data from the processor. In the figure this output port is connected to a set of LEDs, like the ones available on the DE1-SoC board. To allow the processor to select either the memory unit or output port when performing a write operation, the circuit includes *address decoding*, which is done using NOR gates and AND gates. Let the processor's address lines be referred to as ADDR = $A_{15}A_{14}\cdots A_1A_0$. If the upper address lines $A_{15}A_{14}A_{13}A_{12} = 0000$, then the memory unit can be written. Figure 13 shows n *lower* address lines connected from the processor to the memory; since the memory has 256 words, then $n = 8$ and the memory's *address* port is driven by the processor address lines $A_7 \ldots A_0$. For addresses in which $A_{15}A_{14}A_{13}A_{12} = 0001$, the data written by the processor is loaded into the output port connected to *LEDs* in Figure 13.

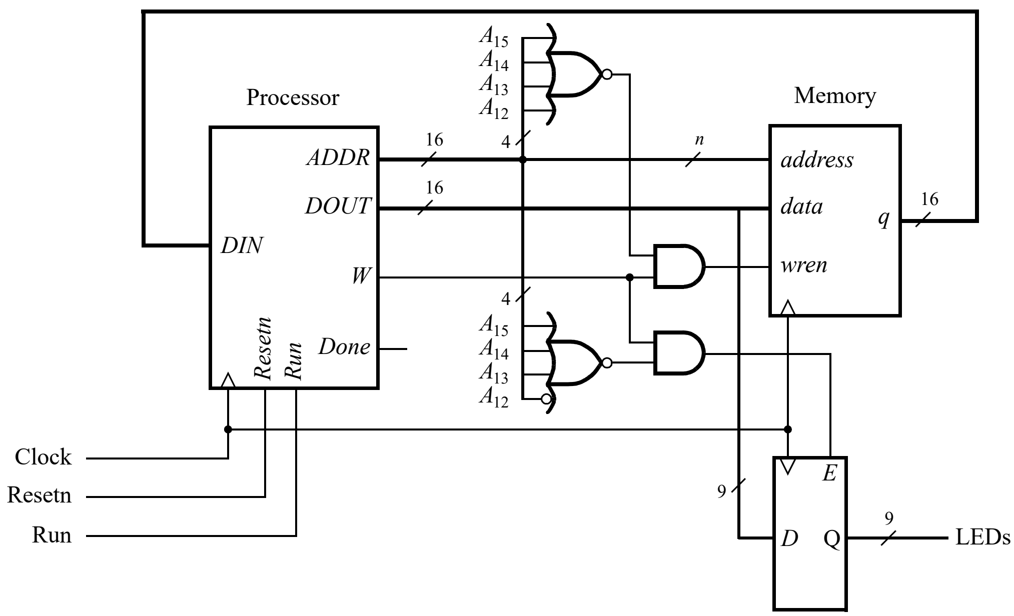

Figure 13: Connecting the enhanced processor to a memory unit and output register.

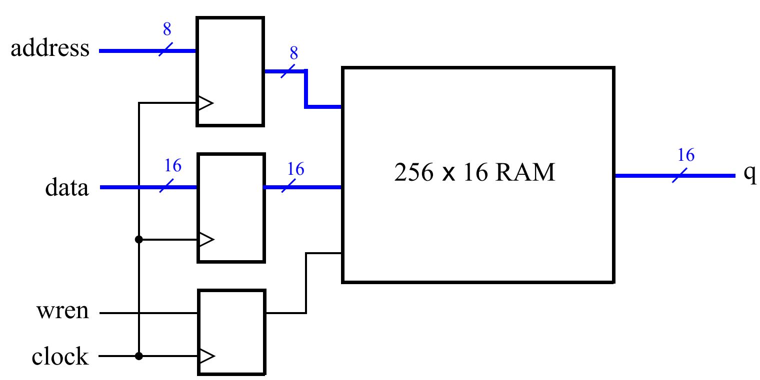

Figure 14: The synchronous SRAM unit.

Part III

Figure 15 gives Verilog code for a top-level file that you can use for this part of the exercise. The input and output ports for this module are chosen so that it can be implemented on a DE1-SoC board. The Verilog code corresponds to the circuit in Figure 13, plus an additional input port that is connected to switches $SW_8 \dots SW_0$. This input port can be read by the processor at addresses in which $A_{15} \dots A_{12} = 0011$. (Switch SW_9 is not a part of the input port, because it is dedicated for use as the processor's *Run* input.) To support reading from both the SW input port and the memory unit, the top-level circuit includes a multiplexer that feeds the processor's *DIN* input. This multiplexer is described by using an `if-else` statement inside the `always` block in Figure 15.

The code in Figure 15 is provided with this exercise, along with a few other source-code files: *flipflop.v*, *inst_mem.v*, *inst_mem.mif*, and (part of) *proc.v*. The *inst_mem.v* source-code file was created by using the Quartus IP Catalog to instantiate a `RAM:1-PORT` memory module. It has a 16-bit wide read/write data port and is 256-words deep, corresponding to Figure 14.

The Verilog code in the *proc.v* file implements register *r7* as a program counter, as discussed above, and includes a number of changes that are needed to support the new *ld*, *st*, *and*, and *b{cond}* instructions. In this part you are to augment this Verilog code to complete the implementation of the *ld* and *st* instructions, as well as the *and* instruction. You do not need to work on the *b{cond}* instruction for this part.

```
module part3 (KEY, SW, CLOCK_50, LEDR);
   input [0:0] KEY;
   input [9:0] SW;
   input CLOCK_50;
   output [9:0] LEDR;

   wire [15:0] DOUT, ADDR;
   wire Done, W;
   reg [15:0] DIN;
   wire inst_mem_cs, SW_cs, LED_reg_cs;
   wire [15:0] inst_mem_q;
   wire [8:0] LED_reg, SW_reg;      // LED[9] and SW[9] are used for Run

   proc U3 (DIN, KEY[0], CLOCK_50, SW[9], DOUT, ADDR, W, Done);

   assign inst_mem_cs = (ADDR[15:12] == 4'h0);
   assign LED_reg_cs = (ADDR[15:12] == 4'h1);
   assign SW_cs = (ADDR[15:12] == 4'h3);
   inst_mem U4 (ADDR[7:0], CLOCK_50, DOUT, inst_mem_cs & W, inst_mem_q);

   always @ (*)                     // input multiplexer
      if (inst_mem_cs == 1'b1)
         DIN = inst_mem_q;
      else if (SW_cs == 1'b1)
         DIN = {7'b0000000, SW_reg};
      else
         DIN = 16'bxxxxxxxxxxxxxxxx;

   regn #(.n(9)) U5 (DOUT[8:0], Resetn, LED_reg_cs & W, CLOCK_50, LED_reg);
   assign LEDR[8:0] = LED_reg;
   assign LEDR[9] = SW[9];

   regn #(.n(9)) U7 (SW[8:0], Resetn, 1'b1, CLOCK_50, SW_reg); // Run = SW[9]
endmodule
```

Figure 15: Verilog code for the top-level file.

Perform the following:

1. Augment the code provided in *proc.v* so that the enhanced processor reads each of its instructions from the external memory, using the program counter to provide the memory address. Also, implement the *ld*, *st*, and *and* instructions. The control FSM requires six time steps for the enhanced processor, as indicated in Table 4. The first three steps are needed to *fetch* an instruction into the processor from memory. In step T_0, the program counter is copied into the *ADDR* register, so that this address will be provided to the memory. This action is accomplished by placing the program counter onto the *Buswires*, and asserting $ADDR_{in}$. Also, *pc_incr* is asserted so that the program counter will be incremented to the address of the next instruction in memory. Since the memory has a synchronous interface, as shown in Figure 14, the processor must use time-step T_1 to wait for the memory to respond. Then, the IR_{in} signal can be asserted in step T_2, so that in step T_3 the *IR* register will hold the machine code of the instruction to be executed.

   We can compare the time steps in the enhanced processor to those of the simple processor from Laboratory Exercise 7. In the enhanced processor, step T_2 serves the same function as step T_0 in the simple processor. Thus, in Table 4 the control signals asserted in steps T_2 to T_5 for the *mv*, *mvt*, *add*, and *sub* instructions are the same as those used in time steps T_0 to T_3 for the simple processor. The *and* instruction uses the same control signals as for *add* and *sub*, with one difference—for *and*, the control signal *ALU_and* is asserted in step T_4, which causes the ALU to perform the logical AND operation.

   The last two lines in Table 4 show the timing needed for *ld* and *st*. In both instructions the contents of register *rY* is transferred in step T_3 to the *ADDR* register. For *ld* the processor uses T_4 to wait for the memory to respond with data, and then step T_5 causes this data to be loaded into register *rX*. For *st*, the data in *rX* to be written to the memory is transferred to register *DOUT* in step T_4, and *W_D* (see Figure 12) is asserted to set the *W* (*write*) signal for the memory.

	T_0	T_1	T_2	T_3	T_4	T_5
mv	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select rY* or *IR*, rX_{in}, *Done*		
mvt	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select IR*, rX_{in}, *Done*		
add	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select rX*, A_{in}	*Select rY* or *IR*, G_{in}	*Select G*, rX_{in}, *Done*
sub	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select rX*, A_{in}	*Select rY* or *IR*, *AddSub*, G_{in}	*Select G*, rX_{in}, *Done*
and	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select rX*, A_{in}	*Select rY* or *IR*, *ALU_and*, G_{in}	*Select G*, rX_{in}, *Done*
ld	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select rY*, $ADDR_{in}$		*Select DIN*, rX_{in}, *Done*
st	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select rY*, $ADDR_{in}$	*Select rX*, $DOUT_{in}$, W_D, *Done*	

Table 4: Control signals asserted in each instruction/time step.

   Test your Verilog code by using the ModelSim simulator. Sample setup files for ModelSim, including a testbench, are provided along with the other files for this exercise. The sample testbench first resets the processor system and then asserts the *Run* switch, SW_9, to 1. A sample program to test your processor is also provided, in a file called *inst_mem.mif*. This file represents the assembly-language program shown in Figure 16, which tests the *ld* and *st* instructions by reading the values of the SW switches and writing these values to the LEDs, in an endless loop. At the beginning of a simulation, ModelSim loads the contents of the file *inst_mem.mif* into the *inst_mem* memory module, so that the program can be executed by the processor.

Examine the signals inside your processor, as well as the external LEDR values, as the program executes within the ModelSim simulation.

An *assembler* software tool, called *sbasm.py*, is provided for use with your processor. The Assembler is written in Python and is included along with the design files for this exercise. To use this Assembler you need to have Python (version 3) installed on your computer. The Assembler includes a README file that explains how to install and use it. The *sbasm.py* Assembler can generate machine code for all of the processor's instructions. The provided file *inst_mem.mif* was created by using *sbasm.py* to *assemble* the program in Figure 16. As the figure indicates, you can define symbolic constants in your code by using the .define *directive*, and you can use labels to refers to lines of code, such as `MAIN`. Comments are specified in the code by using `//`. The assembler ignores anything on a line following `//`.

```
.define LED_ADDRESS 0x10
.define SW_ADDRESS 0x30

// Read SW switches and display on LEDs
         mvt   r3, #LED_ADDRESS  // point to LED port
         mvt   r4, #SW_ADDRESS   // point to SW port
MAIN:    ld    r0, [r4]          // read SW values
         st    r0, [r3]          // light up LEDs
         mv    pc, #MAIN
```

Figure 16: Assembly-language program that uses *ld* and *st* instructions.

An example result produced by using *ModelSim* for a correctly-designed circuit is given in Figure 17. It shows the execution of the first four instructions in Figure 16.

2. Once your ModelSim simulation results look correct, you can then implement your Verilog code on a DE1-SoC board. You are encouraged to make use of the DESim tool, which provides a convenient way of observing the behaviour of programs running on your processor that make use of the lights, switches, and other features of the board. Note that you are not *required* to execute programs on your processor using DESim, because all demonstrations to TAs in your lab period will be done on an actual DE1-SoC board. However, you are encouraged to make use of DESim, as you will probably find that it is a convenient way of debugging issues, especially when you do not have access to a physical board.

   The setup files that are needed to use DESim for this part of the exercise are provided along with its design files. When using DESim, the memory module in your design will be initialized with the contents of the *inst_mem.mif* file, so that the program in the memory can be executed by your processor. Once you start the simulation make sure to reset the circuit by using the `Push Button` that corresponds to KEY_0, and assert the *Run* signal to 1 by setting the `Switch` that corresponds to SW_9. Toggle the values of the `Switches` in the DESim GUI and observe the `LEDs`.

3. In your lab period at the University, use the Quartus Prime software to implement your Verilog code on a DE1-SoC board. A sample Quartus project file, *part3.qpf*, and Quartus settings file, *part3.qsf*, are provided with the exercise. Compile your code using the Quartus software, and download the resulting circuit to the board. Toggle the SW switches and observe the LEDs to test your circuit.

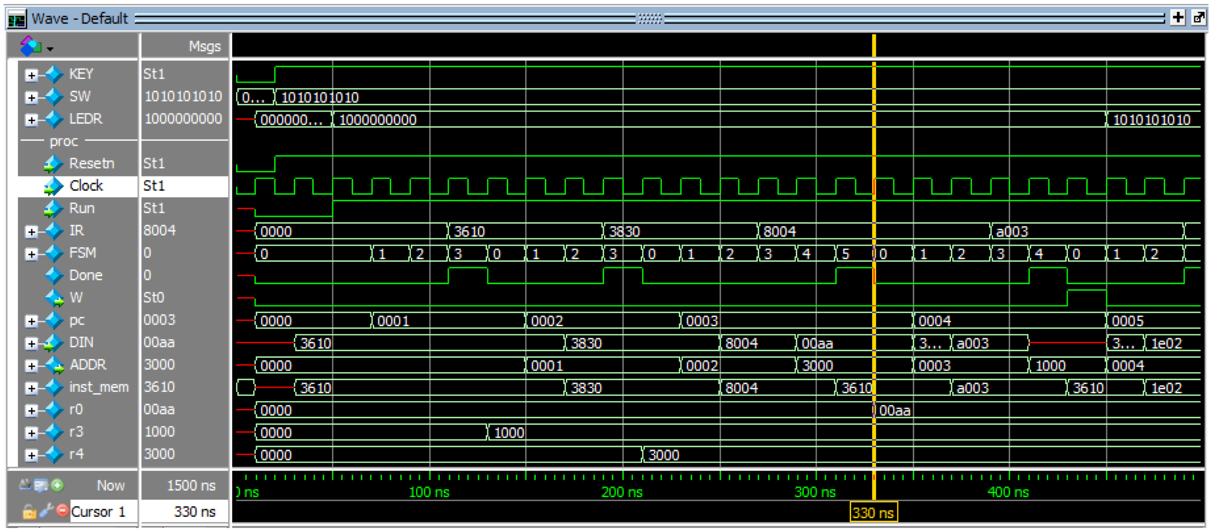

Figure 17: Simulation results for the processor.

Part IV

In this part you are to create a new Verilog module that represents an output port called *seg7*. It will allow your processor to write data to each of the six 7-segment displays on a DE1-SoC board. The *seg7* module will include six write-only seven-bit registers, one for each display. Each register should directly drive the segment lights for one seven-segment display, so that the processor can write characters onto the displays.

Perform the following:

1. A top-level file is provided for this part called *part4.v*. The top-level module has output ports for connecting to each of the 7-segment displays. For each display, segment 0 is on the top of the display, and then segments 1 to 5 are assigned in a clockwise fashion, with segment 6 being in the middle of the display.

   The *part4.v* Verilog code includes address decoding for the new *seg7* module, so that processor addresses in which $A_{15}A_{14}A_{13}A_{12} = 0010$ select this module. The intent is that address `0x2000` should write to the register that controls display *HEX0*, `0x2001` should select the register for *HEX1*, and so on. For example, if your processor writes `0` to address `0x2000`, then the *seg7* module should turn off all of the segment-lights in the *HEX0* display; writing `0x7f` should turn on all of the lights in this display.

2. You are to complete the partially-written Verilog code in the file *seg7.v*, so that it contains the required six registers—one for each 7-segment display.

3. You can compile and test your Verilog code by using the ModelSim setup files that are provided for this part of the exercise. An *inst_mem.mif* file is also provided that corresponds to the assembly-language program shown in Figure 18. This program works as follows: it reads the *SW* switch port and lights up a seven-segment display corresponding to the value read on SW_{2-0}. For example, if $SW_{2-0} = 000$, then the digit 0 is shown on *HEX0*. If $SW_{2-0} = 001$, then the digit 1 is displayed on *HEX1*, and so on, up to the digit 5 which would be shown on *HEX5* if $SW_{2-0} = 101$.

4. Once your ModelSim simulation results look correct, you can then implement your Verilog code on a DE1-SoC board. As we discussed in Part III, above, you are encouraged (by not *required*) to make use of the DESim tool for testing your processor and programs (project set-up files for DESim are included with this exercise). When using your processor, remember to reset the circuit and set $Run = 1$, and then toggle the values of the `Switches` and observe the `Seven-segment Displays`. When compiling your Verilog code with Quartus Prime, you may wish to make use of the sample Quartus project file, *part4.qpf*, and Quartus settings file, *part4.qsf*, that are included along with this exercise.

```
.define HEX_ADDRESS 0x20
.define SW_ADDRESS 0x30

// This program shows the digits 543210 on the HEX displays. Each digit has to
// be selected by using the SW switches.
MAIN:    mvt     r2, #HEX_ADDRESS      // point to HEX port
         mv      r3, #DATA             // used to get 7-segment display pattern

         mvt     r4, #SW_ADDRESS       // point to SW port
         ld      r0, [r4]              // read switches
         and     r0, #0x7              // use only SW2-0
         add     r2, r0                // point to correct HEX display
         add     r3, r0                // point to correct 7-segment pattern

         ld      r0, [r3]              // load the 7-segment pattern
         st      r0, [r2]              // light up HEX display

         mv      pc, #MAIN

DATA:    .word 0b00111111              // '0'
         .word 0b00000110              // '1'
         .word 0b01011011              // '2'
         .word 0b01001111              // '3'
         .word 0b01100110              // '4'
         .word 0b01101101              // '5'
```

Figure 18: Assembly-language program that tests the seven-segment displays.

Part V

In this part you are to enhance your processor so that it implements the *b{cond}* instruction. The *conditions* supported by the processor are called *eq*, *ne*, *cc*, *cs*, *pl*, and *mi*, which means that the variations of the branch instruction are *b*, *beq*, *bne*, *bcc*, and so on. The *b* instruction *always* branches. For example, `b MAIN` loads the address of the label `MAIN` into the program counter. The meanings of the conditional versions are explained below.

The instruction `beq LABEL` means *branch if equal* (to zero). It performs a branch (sets *pc* = `LABEL`) if the most recent result of an instruction executed using the arithmetic logic unit (ALU), which is stored in register G, was 0. Similarly, *bne* means *branch if not equal* (to zero). It performs a branch only if the contents of G are not equal to 0. The instruction *bcc* stands for *branch if carry clear*. It branches if the last add/subtract operation did *not* produce a carry-out. The opposite branch condition, *bcs*, *branch if carry set*, branches if the most recent add/sub generated a carry-out. The conditions *bpl* and *bmi* allow a branch to be taken when the value in register G is a positive or negative (2's complement) value, respectively.

To support the conditional branch instructions, you should create three *condition-code flags*, called z, n, and c in your processor. The z flag should have the value 1 when the ALU generates a result of zero; otherwise z should be 0. The n flag should be 1 when the ALU generates a result that is negative, meaning that the most-significant bit (the *sign* bit) is 1; otherwise n should be set to 0. Finally, the c flag should reflect the carry-out from the ALU; this flag should be 1 when an *add* instruction generates a carry-out, or when a *sub* operation does *not* generate a borrow. Figure 12 indicates how you can implement the flags as the outputs of a three-bit register, named F. The z flag is controlled by a NOR gate that is used to check when the output of the ALU is zero, the n flag is connected to the sign-bit from the ALU's output, and a carry-out from the ALU drives the c flag. These flags are connected to the FSM controller, which should examine the flags in the appropriate clock cycles when executing a *b{cond}* instruction.

The branch instructions are encoded similarly to the *mvt* instruction introduced in Lab Exercise 7, which has the format `III1XXXDDDDDDDDD`, with `III` = 001. The format of *b{cond}* is `III0XXXDDDDDDDDD`, where `III` = 001, `XXX` gives the condition, and `DDDDDDDDD` specifies an *offset*. The condition `XXX` is encoded as *none* (always branch) = 000, *eq* = 001, *ne* = 010, *cc* = 011, *cs* = 100, *pl* = 101, and *mi* = 110.

The *offset* `DDDDDDDDD` is the 2's-complement value needed to reach the target `LABEL` relative to the current contents of the *pc* register. This offset assumes that the *pc* has already been incremented after fetching the *b{cond}* instruction from memory. For example, the instruction `HERE: b HERE` would be encoded as `0010000111111111`, where the offset is the 2's-complement value -1.

Perform the following:

1. Enhance your processor so that it implements the condition-code flags *z*, *c*, and *n* and supports the *b{cond}* instruction. Table 5 indicates the control signal timing that can be used for this instruction. Step T_3 copies the contents of the *pc* into register A, in the ALU. This step also checks whether or not the branch should be *taken*, based upon the *condition*. If the *condition* is not true, indicated in Table 5 using the syntax (!*cond*), then the *Done* signal is asserted to abort the branch instruction. But if the *condition* is satisfied, then the finite state machine continues to step T_4. This step places the branch offset, which is in the instruction register (*IR*), onto the *Buswires* so that the ALU can add it to the value of the *pc* that was previously copied into register A. Finally, step T_5 transfers the computed branch-target address to the *pc*, so that the branch will be taken.

	T_0	T_1	T_2	T_3	T_4	T_5
b{cond}	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select pc*, A_{in}, if (!*cond*) *Done*	*Select IR*, G_{in}	*Select G*, pc_{in}, *Done*

Table 5: Control signals asserted for *b{cond}* in each time step.

To help with testing and debugging of your processor, setup files for ModelSim are provided, including a testbench. It simulates your processor instantiated in the top-level file *part5.v*, which is the same as the one from Part IV. An example *inst_mem.mif* file is also provided, which corresponds to the program in Figure 19. This program is quite short, which makes it suitable for visual inspection of the waveforms produced by a ModelSim simulation. The program uses a sequence of instructions that tests the various conditional branches. If the program reaches the line of code labelled `DEAD`, then at least one instruction has not worked properly.

An example of ModelSim output for a correctly-working processor is given in Figure 20. It shows the processor executing instructions near the start of the code in Figure 19. The instruction that is completed at simulation time 510 ns is `sub r0, #1` (`0x7001`). As shown in the figure, this instruction causes the zero flag, *z*, to become 1. The next instruction loaded into *IR*, at time 570 ns, is `bne 0x1` (`0x25fe`). This instruction does not take the branch, because $z = 1$. Finally, the instruction loaded at 650 ns is `beq 0x5` (`0x2201`), which does take the branch.

2. Once your ModelSim simulation indicates a correctly-functioning processor, implement it on a DE1-SoC board. As we discussed in Parts III and IV, above, you can use the DESim tool as a convenient way of testing and debugging your design (using DESim is optional and is not required) and you can use Quartus Prime to download your circuit to a DE1-SoC board. The required project set-up files for both DESim and Quartus Prime are included along with this exercise. To test your processor, you can use the assembly-language program displayed in Figure 21. It provides code that tests for the correct operation of instructions supported by the enhanced processor. If all of the tests pass, then the program shows the word PASSEd on the `Seven-segment Displays`. It also shows a binary value on the LEDs that represents the number of successful tests performed. If any test fails, then the program shows the word FAILEd on the `Seven-segment Displays` and places on the `LEDs` the address in the memory of the instruction that

caused the failure. Assemble the program, which is provided in a file called *sitbooboosit.s*, by using the *sbasm.py* assembler. Store the output produced by *sbasm.py* in the file *inst_mem.mif*.

If you compile your processor system using the DESim tool, it uses the current contents of the *inst_mem.mif* file to initialize the memory. When simulating your processor make sure to reset it by using the `Push Button` that corresponds to $\texttt{KEY}_0$. Then, set the *Run* signal to 1 by setting the `Switch` that corresponds to $\texttt{SW}_9$. If the *sitbooboosit* program displays FAILEd on the `Seven-segment Displays`, then you can identify the offending instruction by cross-referencing the LED pattern with the corresponding address in the file *inst_mem.mif*.

```
MAIN:  mv     r0, #2
LOOP:  sub    r0, #1          // subtract to test bne
       bne    LOOP
       beq    T1              // r0 == 0, test beq
       mv     pc, #DEAD
T1:    mvt    r0, #0xFF
       add    r0, #0xFF       // r0 = 0xFFFF
       bcc    T2              // carry = 0, test bcc
       mv     pc, #DEAD
T2:    add    r0, #1
       bcs    T3              // carry = 1, test bcs
       mv     pc, #DEAD
T3:    bpl    T4
       mv     pc, #DEAD
T4:    add    r0, #-1
       bmi    T5
       mv     pc, #DEAD
T5:    b      MAIN
DEAD:  mv     pc, #DEAD
```

Figure 19: Assembly-language program that uses various branches.

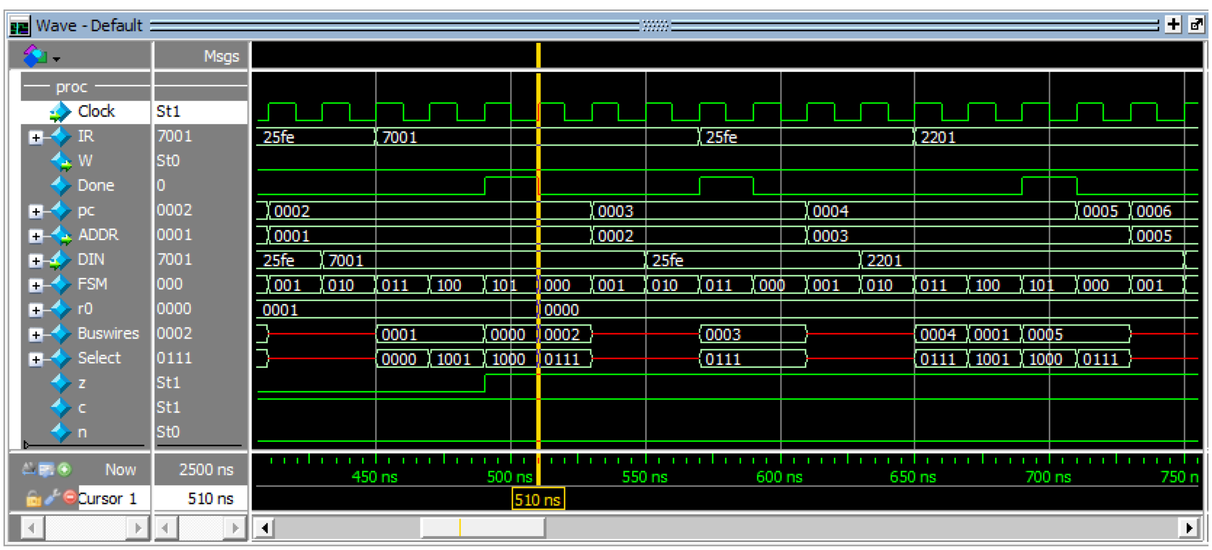

Figure 20: Simulation results for the processor.

```
.define LED_ADDRESS 0x10
.define HEX_ADDRESS 0x20

// shows on HEX displays either PASSEd or FAILEd
            mv      r2, #0      // counts each successful test
            mv      r6, #T1     // save address of next test
            sub     r0, r0      // set the z flag
// test bne and beq
T1:         bne     FAIL        // should not take the branch!
            mv      r6, #C1     // save address of next test
C1:         beq     C2          // should take the branch
            mv      pc, #FAIL   // Argh!

C2:         add     r2, #2      // count the last two successful tests
            mv      r6, #T2     // save address of next test
// test bne and beq
T2:         bne     S1          // should take the branch!
            mv      pc, #FAIL
S1:         mv      r6, #C3     // save address of next test
C3:         beq     FAIL        // should not take the branch
            add     r2, #2      // count the last two successful tests
            mv      r6, #T3     // save address of next test
            mv      r3, #-1     // r3 = 0xFFFF
            add     r3, #1      // set the c flag
// test bcc and bcs
T3:         bcc     FAIL        // should not take the branch!
            mv      r6, #C4     // save address of next test
C4:         bcs     C5          // should take the branch
            mv      pc, #FAIL   // Argh!
C5:         add     r2, #2      // count the last two successful tests
            mv      r6, #T4
            mv      r3, #0
            add     r3, r3      // clear carry flag
// test bcc and bcs
T4:         bcc     S2          // should take the branch!
            mv      pc, #FAIL
S2:         mv      r6, #C6     // save address of next test
C6:         bcs     FAIL        // should not take the branch!
            add     r2, #2      // count the last two successes
            mv      r6, #T5     // save address of next test
            mv      r3, #0
            add     r3, #-1
// test bpl and bmi
T5:         bpl     FAIL        // should not take the branch!
            mv      r6, #C7     // save address of next test
C7:         bmi     C8          // should take the branch
            mv      pc, #FAIL   // Argh!
C8:         add     r2, #2      // count the last two successful tests
            mv      r6, #T6
            mv      r3, #0
            add     r3, r3      // clear negative flag
// test bpl and bmi
T6:         bpl     S3          // should take the branch!
            mv      pc, #FAIL
```

Figure 21: Assembly-language program that tests various instructions. (Part a)

```
S3:         mv      r6, #C9     // save address of next test
C9:         bmi     FAIL        // should not take the branch!
            add     r2, #2      // count the last two successes
// finally, test ld and st from/to memory
            mv      r6, #T7         // save address of next test
            mv      r4, #_LDTEST
            ld      r4, [r4]
            mv      r3, #0x0A5
            sub     r3, r4
T7:         bne     FAIL            // should not take the branch!
            add     r2, #1          // incr success count

            mv      r6, #T8         // save address of next test
            mv      r3, #0x0A5
            mv      r4, #_STTEST
            st      r3, [r4]
            ld      r4, [r4]
            sub     r3, r4
T8:         bne     FAIL            // should not take the branch!
            add     r2, #1          // incr success count

            mv      pc, #PASS

// Loop over the six HEX displays
FAIL:       mvt     r3, #LED_ADDRESS
            st      r6, [r3]        // show failed test address on LEDs
            mv      r5, #_FAIL
            mv      pc, #PRINT
PASS:       mvt     r3, #LED_ADDRESS
            st      r2, [r3]        // show success count on LEDs
            mv      r5, #_PASS

PRINT:      mvt     r4, #HEX_ADDRESS // address of HEX0
            // We would normally use a loop counting down from 6 with
            // conditional branching, but in this testing code we can't
            // assume that branching even works!
            ld      r3, [r5]        // get letter
            st      r3, [r4]        // send to HEX display
            add     r5, #1          // ++increment character pointer
            add     r4, #1          // point to next HEX display
            ld      r3, [r5]        // get letter
            st      r3, [r4]        // send to HEX display
            add     r5, #1          // ++increment character pointer
            add     r4, #1          // point to next HEX display
            ld      r3, [r5]        // get letter
            st      r3, [r4]        // send to HEX display
            add     r5, #1          // ++increment character pointer
            add     r4, #1          // point to next HEX display
```

Figure 21: Assembly-language program that tests various instructions. (Part *b*)

```
            ld      r3, [r5]          // get letter
            st      r3, [r4]          // send to HEX display
            add     r5, #1            // ++increment character pointer
            add     r4, #1            // point to next HEX display
            ld      r3, [r5]          // get letter
            st      r3, [r4]          // send to HEX display
            add     r5, #1            // ++increment character pointer
            add     r4, #1            // point to next HEX display
            ld      r3, [r5]          // get letter
            st      r3, [r4]          // send to HEX display
            add     r5, #1            // ++increment character pointer
            add     r4, #1            // point to next HEX display

HERE:       mv      pc, #HERE

_PASS:      .word   0b0000000001011110     // d
            .word   0b0000000001111001     // E
            .word   0b0000000001101101     // S
            .word   0b0000000001101101     // S
            .word   0b0000000001110111     // A
            .word   0b0000000001110011     // P

_FAIL:      .word   0b0000000001011110     // d
            .word   0b0000000001111001     // E
            .word   0b0000000000111000     // L
            .word   0b0000000000110000     // I
            .word   0b0000000001110111     // A
            .word   0b0000000001110001     // F

_LDTEST: .word   0x0A5
_STTEST: .word   0x05A
```

Figure 21: Assembly-language program that tests various instructions. (Part *c*)

3. When compiling your design using the Quartus Prime software, it is possible to change the *inst_mem.mif* file without completely recompiling your Verilog code for the processor system. You can execute the Quartus command `Processing > Update Memory Initialization File` to include a new *inst_mem.mif* file in your Quartus project. Then, select the Quartus command `Processing > Start > Start Assembler` to produce a new programming *bitstream* for your DE1-SoC board. Finally, use the Quartus Programmer to download the new bitstream onto your board. If the *Run* signal is asserted, your processor should execute the new program.

Part VI

Write an assembly-language program that displays a binary counter on the LED port. Initialize the counter to 0, and then increment the counter by one in an endless loop. You should be able to control the speed at which the counter is incremented by using nested delay loops; the inner loop should have a fixed delay, and the outer loop should be controlled by the SW switch settings. Changing the settings of the SW switches should cause the counter to increment more slowly/quickly on the LEDs.

Assemble your program by using the *sbasm.py* assembler, and then run it on your processor. If you are using the DESim tool, then compiling your Verilog code will initialize the processor's memory with the current contents of the *inst_mem.mif*, as mentioned before. If you are using Quartus Prime to run your processor on the DE1-SoC board, then follow the procedure described previously to update the *MIF* file and then download the new circuit to the board.

Part VII

This part of the exercise is optional, so you do not need to do it. But if you decide to perform this part, and can provide good answers to questions about it from your TA, then it can be worth up to 2 bonus marks. Augment your assembly-language program from Part VI so that counter values are displayed on the seven-segment display port rather than on the LED port. You should display the counter values as decimal numbers from 0 to 65535. The speed of counting should be controllable using the SW switches in the same way as for Part VI. As part of your solution you may want to make use of the code shown in Figure 22. This code provides a subroutine, *DIV10*, that divides the number in register $r0$ by 10, returning the quotient in $r1$ and the remainder in $r0$. Dividing by 10 is a useful operation when performing binary-to-decimal conversion. A skeleton of the required code for this part is shown in Figure 23. Since the enhanced processor does not provide a method for calling and returning from a subroutine, the code in Figures 22 and 23 uses an ad hoc method, in which register $r6$ is used to compute a return address for the *DIV10* subroutine.

As described previously, assemble your code by using the *sbasm.py* assembler tool, and then execute the new program on your processor system.

```
// subroutine DIV10
//      This subroutine divides the number in r0 by 10
//      The algorithm subtracts 10 from r0 until r0 < 10, and keeps count in r1
//      This subroutine also changes r2
//      input: r0
//      returns: quotient Q in r1, remainder R in r0
DIV10:
            mv     r1, #0                // init Q
DLOOP:      mv     r2, #9                // check if r0 is < 10  yet
            sub    r2, r0
            bcs    RETDIV                // if so, then return

INC:        add    r1, #1                // but if not, then increment Q
            sub    r0, #10               // r0 -= 10
            b      DLOOP                 // continue loop
RETDIV:
            add    r6, #1                // adjust the return address
            mv     pc, r6                // return results
```

Figure 22: A subroutine that divides by 10

```
.define HEX_ADDRESS 0x20
.define SW_ADDRESS 0x30

// This program shows a decimal counter on the HEX displays
MAIN:   mv     r6, pc            // return address for subroutine
        mv     pc, #BLANK        // call subroutine to blank the HEX displays
        mv     r0, #0            // initialize counter
LOOP:   mvt    r3, #HEX_ADDRESS // point to HEX port
        ...
        ... use a loop to extract and display each digit
        ...

// Delay loop for controlling the rate at which the HEX displays are updated
        ...
        ... read from SW switches, and use a nested delay loop
        ...
        add    r0, #1            // counter += 1
        bcc    LOO               // continue until counter overflows

        b      MAIN

// subroutine DIV10
        ...
        ... code not shown here
        ...
        add    r6, #1            // adjust the return address
        mv     pc, r6            // return results

// subroutine BLANK
        ...
        ... code not shown here
        ...
        add    r6, #1
        mv     pc, r6            // return from subroutine

DATA:   .word  0b00111111        // '0'
        ....
```

Figure 23: Skeleton code for displaying decimal digits.