Laboratory Exercise 2

An Enhanced Processor

In Laboratory Exercise 1 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 the figure registers $r0$ to $r6$ are the same as in Figure 1 of Lab 1, 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 $r7_{in}$ 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 have 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 1. 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 1, 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 *b{cond}* instruction is discussed in more detail in Part V of this exercise.

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

Table 3: New instructions in the enhanced processor.

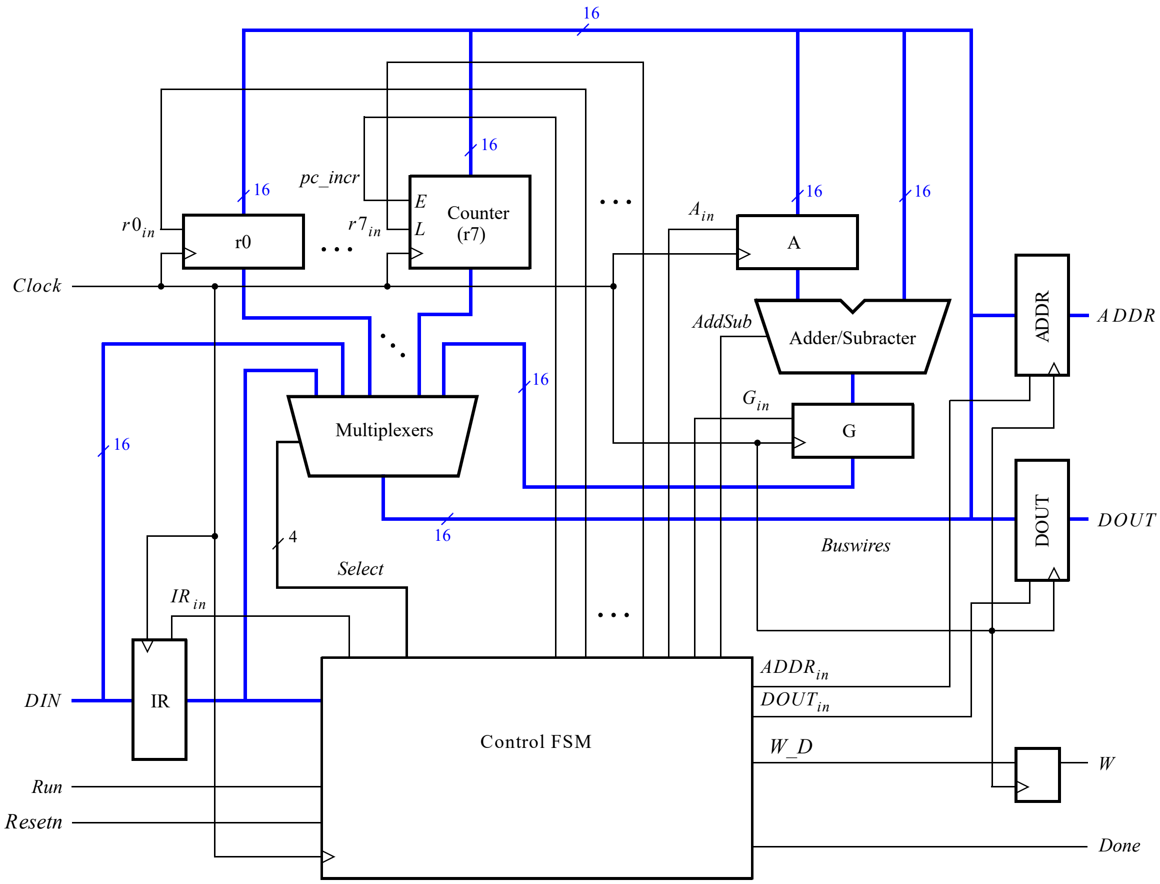

Figure 12: An enhanced version of the processor.

Recall from Lab 1 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 its *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 $A_7 \dots A_0$. 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.

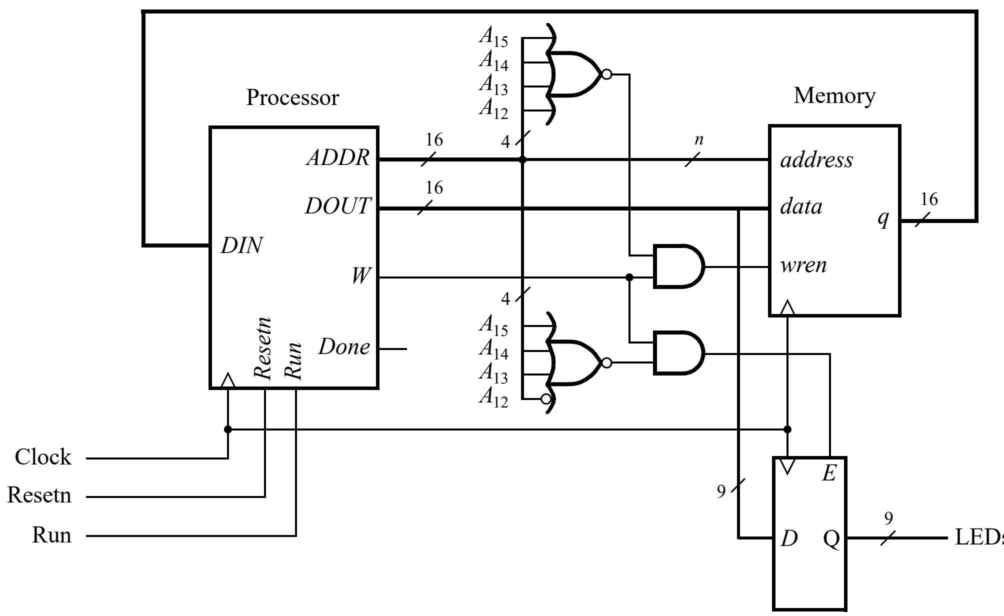

Figure 13: Connecting the enhanced processor to a memory unit and output register.

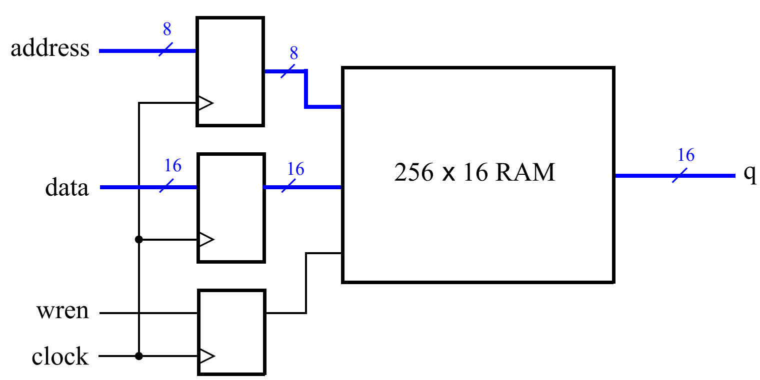

Figure 14: The synchronous SRAM unit.

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 $SW_8 \ldots SW_0$. This input port can be read by the processor at addresses in which $A_{15} \ldots A_{12} = 0011$. (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. Extend the code in *proc.v* so that the enhanced processor fully implements the *ld*, *st*, and *and* instructions. 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 have to first install Python (version 3) 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 0x1000
.define SW_ADDRESS 0x3000

// 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 should "implement" your Verilog code using the *DESim* tool. The required *DESim* project files are provided with the design files that accompany this exercise. When using the *DESim* tool, 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 $\texttt{KEY}_0$, and assert the *Run* signal to 1 by setting the `Switch` that corresponds to $\texttt{SW}_9$. Toggle the values of the `Switches` in the *DESim* GUI and observe the `LEDs`.

3. Optionally, if you have access to a DE1-SoC board, then you could also compile your design using the Quartus Prime software and then download the resulting circuit into the board. A sample Quartus project file, *part3.qpf*, and Quartus settings file, *part3.qsf*, are provided along with this exercise.

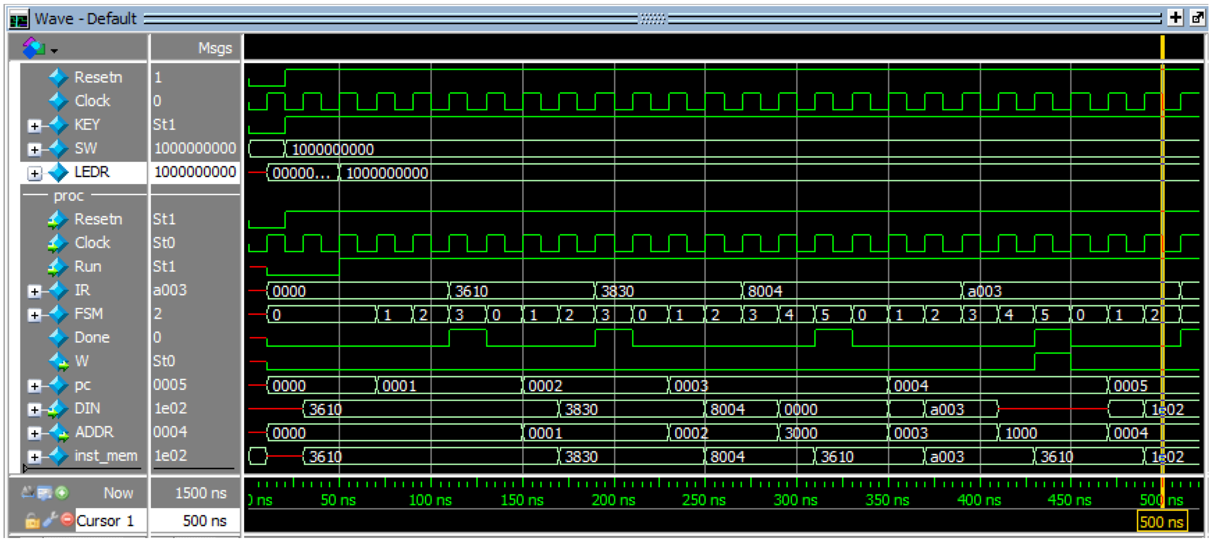

Figure 17: Simulation results for the processor.

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, "implement" your Verilog code using the *DESim* tool. The required *DESim* project files are provided along with this exercise. After compiling your code and starting the simulation, reset the circuit and set $Run = 1$, and then toggle the values of the `Switches` in the *DESim* GUI and observe the `Seven-segment Displays`. Optionally, if you have access to a DE1-SoC board, then you could also compile your design using the Quartus Prime software and then download the resulting circuit into the board. A sample Quartus project file, *part4.qpf*, and Quartus settings file, *part4.qsf*, are provided along with this exercise.

```
.define HEX_ADDRESS 0x2000
.define SW_ADDRESS 0x3000

// This program shows the digits 543210 on the HEX displays. Each digit has to
// be selected by using the SW switches.
          mv     r6, pc             // return address for subroutine
          mv     pc, #BLANK         // call subroutine to blank the HEX displays
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

// subroutine BLANK
//    This subroutine clears all of the HEX displays
// input: none
// returns: nothing
// changes: r0 and r1. Register r6 provides the return address
BLANK:    mv     r0, #0             // used for clearing
          mvt    r1, #HEX_ADDRESS   // point to HEX displays
          st     r0, [r1]           // clear HEX0
          add    r1, #1
          st     r0, [r1]           // clear HEX1
          add    r1, #1
          st     r0, [r1]           // clear HEX2
          add    r1, #1
          st     r0, [r1]           // clear HEX3
          add    r1, #1
          st     r0, [r1]           // clear HEX4
          add    r1, #1
          st     r0, [r1]           // clear HEX5

          add    r6, #1
          mv     pc, r6             // return from subroutine

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*, *c*, and *n* 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 *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. Finally, 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. Your FSM controller should examine these flags in the appropriate clock cycles when executing the *b{cond}* instructions.

The branch instructions are encoded similarly to the *mvt* instruction introduced in Lab Exercise 1, 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. 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 test 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 end of the code in Figure 19. The instruction that is completed at simulation time 1550 ns is `add r0, #-1` (`0x51ff`). As shown in the figure, this instruction causes the negative flag, *n*, to become 1. The next instruction loaded into *IR*, at time 1610 ns, is `bmi 0x11` (`0x2c01`). Finally, the instruction loaded at 1730 ns is `b 0` (`0x21ee`).

2. Once your ModelSim simulation indicates a correctly-functioning processor, implement it using the *DESim* tool. The required *DESim* project files are provided 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*.

```
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, #0xFF00
       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.

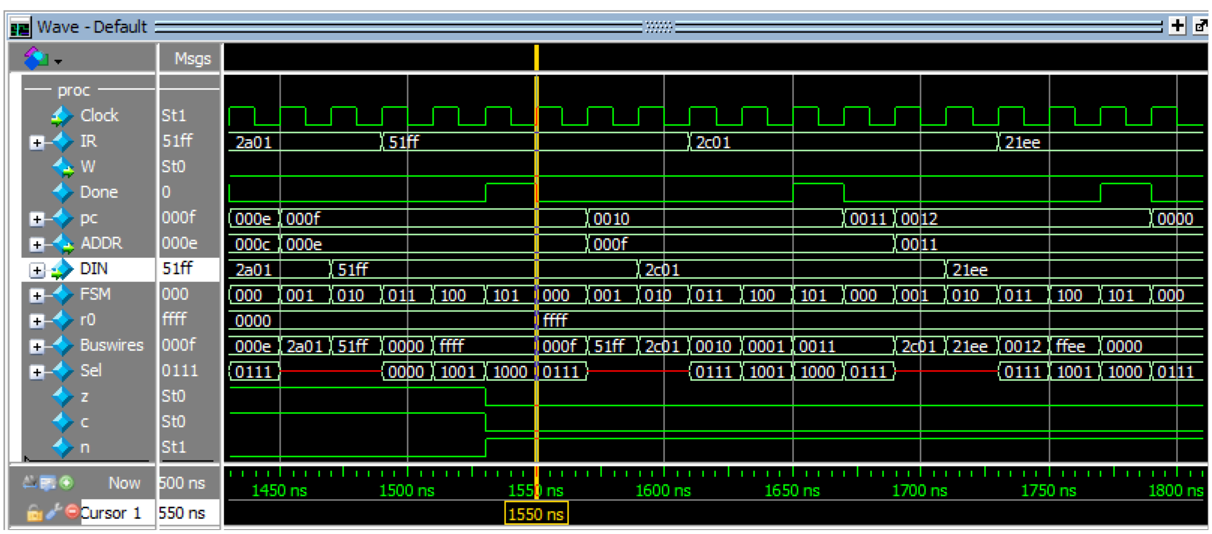

Figure 20: Simulation results for the processor.

When you compile your *DESim* project it will use the latest 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 $\mathtt{KEY}_0$. Then, set the *Run* signal to 1 by setting the `Switch` that corresponds to $\mathtt{SW}_9$. 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*.

```
.define LED_ADDRESS 0x1000
.define HEX_ADDRESS 0x2000

        mv    r2, #0        // used to count number of successful tests

        mv    r6, #T1       // save address of next test
        sub   r0, r0        // set the z flag
T1:     bne   FAIL          // test bne; should not take the branch!
        mv    r6, #C1       // save address of next test
C1:     beq   C2            // test beq; 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
T2:     bne   S1            // test bne; should take the branch!
        mv    pc, #FAIL
S1:     mv    r6, #C3       // save address of next test
C3:     beq   FAIL          // test beq; 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
T3:     bcc   FAIL          // test bcc; should not take the branch!
        mv    r6, #C4       // save address of next test
C4:     bcs   C5            // test bcs; 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
T4:     bcc   S2            // test bcc; should take the branch!
        mv    pc, #FAIL
S2:     mv    r6, #C6       // save address of next test
C6:     bcs   FAIL          // test bcs; 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
T5:     bpl   FAIL          // test bpl; should not take the branch!
        mv    r6, #C7       // save address of next test
C7:     bmi   C8            // test bmi; 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
T6:     bpl   S3            // test bpl; should take the branch!
        mv    pc, #FAIL
S3:     mv    r6, #C9       // save address of next test
C9:     bmi   FAIL          // test bpl; should not take the branch!
        add   r2, #2        // count the last two successes
```

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

```
// finally, test ld and st from/to memory
        mv    r6, #T7        // save address of next test
        mv    r4, #_LDTEST
T7:     ld    r4, [r4]
        mv    r3, #0x0A5
        sub   r3, r4
        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
T8:     st    r3, [r4]
        ld    r4, [r4]
        sub   r3, r4
        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 address of failed test 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 mvne
            to display the
        // six letters. But in this testing code we can't assume that
            mvne 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
        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
```

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

```
            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. As mentioned in previous parts of this exercise, if you have access to a DE1-SoC board, then you could compile your design using the Quartus Prime software and then download the resulting circuit into the board. This step is optional, and is not required. A sample Quartus project file, *part5.qpf*, and Quartus settings file, *part5.qsf*, are provided with this exercise.

   When running your processor on the DE1-SoC board 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.

You can assemble your program by using the *sbasm.py* assembler, and then run it on your processor using the *DESim* tool. Save the output produced by *sbasm.py* in the *inst_mem.mif* file, and then recompile your *DESim* project. When you run the simulation again remember to reset the circuit and set the processor's *Run* signal to 1.

If you are running your processor on the DE1-SoC board, then follow the procedure described previously to update the *MIF* file and then download the new circuit into the board.

Part VII

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 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. The `DIV10` subroutine assumes that $r5$ is set up to be used as a *stack pointer*. Register $r2$ is saved on the stack at the beginning of the subroutine, and then restored before returning. This is done so that $r2$ is not unnecessarily changed by the subroutine. A skeleton of the required code for this part is shown in Figure 23.

```
// 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 assumes that r5 can be used as a stack pointer, and r6
//        has the subroutine's return address
//   input: r0
//   returns: quotient Q in r1, remainder R in r0
DIV10:
         sub    r5, #1            // save registers that are modified
         st     r2, [r5]          // save on the stack

         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:
         ld     r2, [r5]          // restore from the stack
         add    r5, #1
         add    r6, #1            // adjust the return address
         mv     pc, r6            // return results
```

Figure 22: A subroutine that divides by 10

As described previously, assemble your code with *sbasm.py* and execute the new program on your processor, using the *DESim* tool and (optionally) the Quartus software and DE1-SoC board.

```
.define HEX_ADDRESS 0x2000
.define SW_ADDRESS 0x3000
.define STACK 255                  // bottom of memory

// This program shows a decimal counter on the HEX displays
         mv     r5, #STACK         // stack pointer
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    r1, #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    LOOP               // 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.