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. Note that 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 has the ability to perform read and write operations using memory or other devices. We will add four new instructions to the processor, as displayed in Table 3. The *ld* (load) instruction loads data into register *rX* from the external memory address specified in register *rY*. The *st* (store) instruction stores the data contained in register *rX* into the memory address found in *rY*. Finally, the instructions *mvnz* (move if not zero) performs a *mv* operation only under the condition that the current contents of register G (the adder/subtracter output) is not equal to 0, and the *mvnc* instruction performs a *mv* operation only if the most-recently executed *add* (or *sub*) instruction did not produce a carry-out (or borrow).

Operation	Function performed
ld rX, [*rY*]	*rX* $\leftarrow$ [*rY*]
st rX, [*rY*]	[*rY*] $\leftarrow$ *rX*
mvnz rX, *rY*	if G != 0, *rX* $\leftarrow$ *rY*
mvnc rX, *rY*	if no *carry-out*, *rX* $\leftarrow$ *rY*

Table 3: New instructions performed in the processor.

A schematic of the enhanced processor is given in Figure 12. All general-purpose registers are now 16-bits wide, whereas in Laboratory Exercise 1 they were nine-bits wide. In the figure registers $r0$ to $r6$ are the same as in Figure 1 of Exercise 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 contents of *pc* is used as an address to read an instruction from the memory. The instruction is stored into *IR* and the *pc* is automatically incremented to point to the next instruction (in the case of *mvi* the *pc* provides the address of the immediate data and is then incremented again).

The processor's control unit increments *pc* by using the *incr_pc* signal, which is just an enable on this counter. It is also possible to directly load an address into *pc* ($r7$) by having the processor execute a *mv* or *mvi* instruction in which the destination register is specified as $r7$. In this case the control unit uses the signal $r7_{in}$ to perform a parallel 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 in successive addresses. Similarly, the current contents of *pc*, which always has the address of the *next* instruction to be executed, can be copied into another register by using a *mv* instruction.

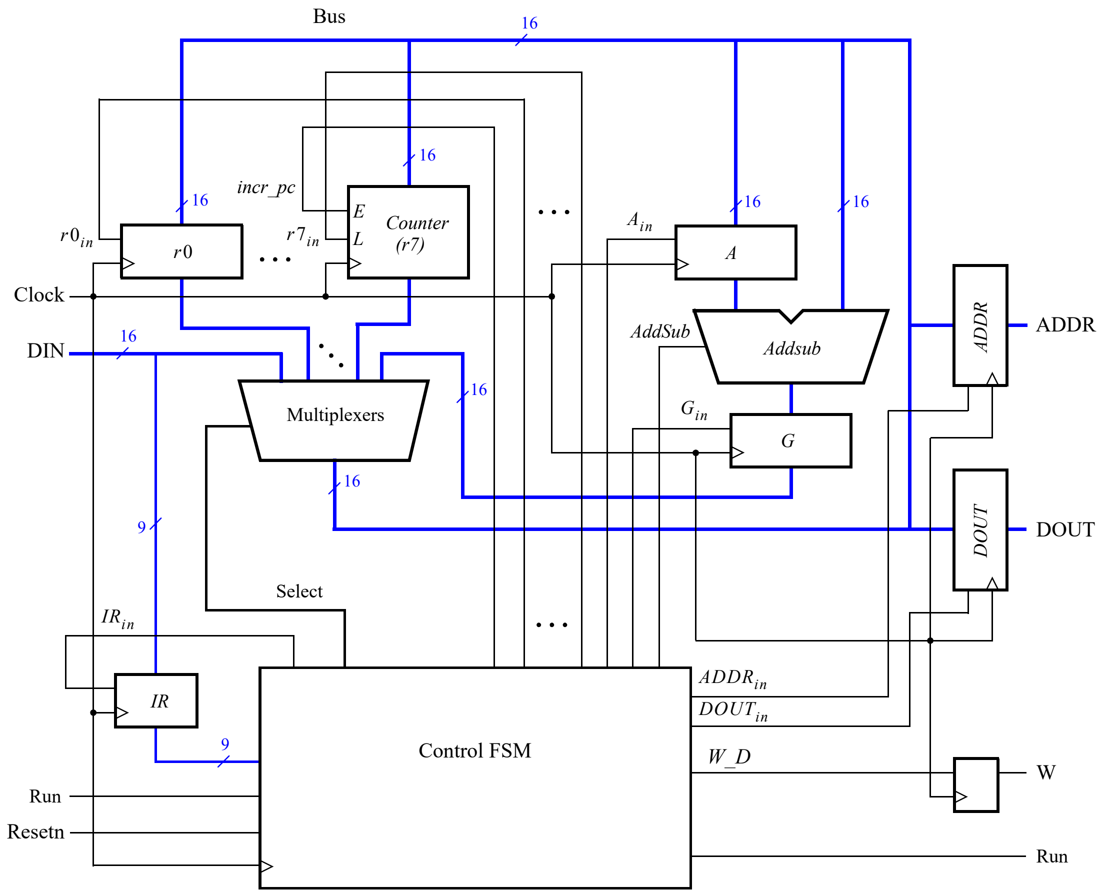

Figure 12: An enhanced version of the processor.

An example of code that uses the *pc* register to implement a loop is shown below, where the text after the // on each line is just a comment. The instruction *mv r5,r7* places into *r5* the address in memory of the instruction *sub r2,r1*. Then, the instruction *mvnz r7,r5* causes the *sub* instruction to be executed repeatedly until *r2* becomes 0. This type of loop could be used in a larger program as a way of creating a delay.

```
mvi    r1, #1
mvi    r2, #10000  // delay value
mv     r5, r7      // save address of next instruction
sub    r2, r1      // decrement delay count
mvnz   r7, r5      // loop until delay count gets to 0
```

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 can be stored outside 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* ($r7$) 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.

Figure 13 illustrates how the enhanced processor can be connected to memory and other devices. The memory unit in the figure supports both read and write operations and therefore has both address and data inputs, as well as a write enable input. The memory also has a clock input, because the address, data, and write enable inputs must be loaded into the memory on an active clock edge. This type of memory unit is usually called a *synchronous static random access memory (synchronous SRAM)*. Figure 13 also includes a 9-bit register that can be used to store data from the processor; this register might be connected to a set of LEDs to allow display of data on your DE-series board. To allow the processor to select either the memory unit or register when performing a write operation, the circuit shows NOR gates that perform *address decoding*: if the upper address lines are $A_{15} \ldots A_{12} = 0$, then the memory module will be written. Figure 13 shows n lower address lines connected to the memory; for this exercise a memory with 256 words is probably sufficient, which implies that $n = 8$ and the memory address port is driven by $A_7 \ldots A_0$. For addresses in which $A_{15} \ldots A_{12} = 1$, the data written by the processor is loaded into the register whose outputs are called *LEDs* in Figure 13.

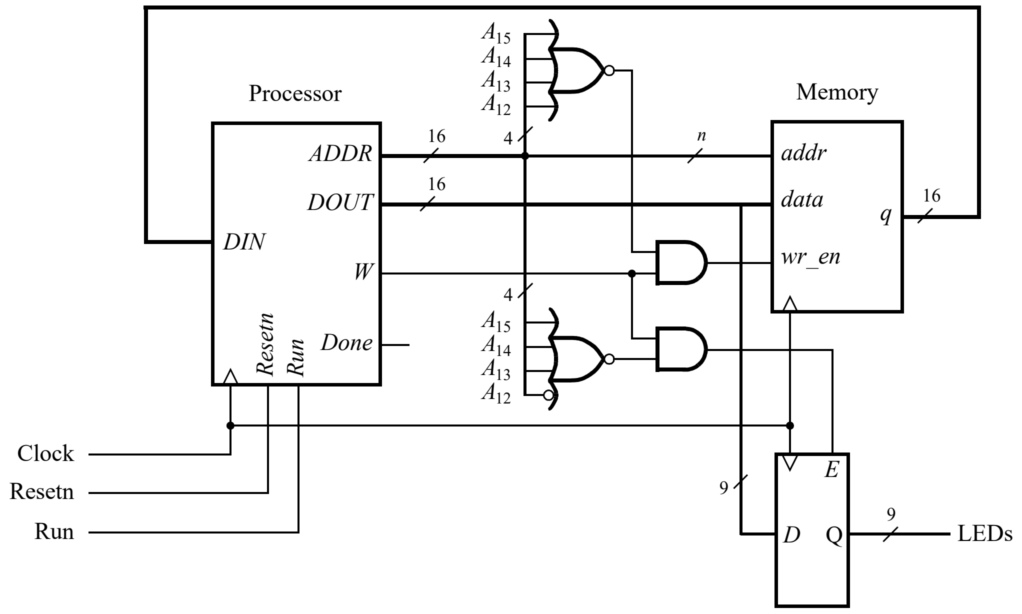

Figure 13: Connecting the enhanced processor to a memory unit and output register.

Part III

Figure 14 gives Verilog code for a top-level file that you can use for this part of the exercise. This file implements the circuit illustrated 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 $A_{15} \ldots A_{12} = 3$. Switch SW_9 is not a part of the switch port, because it is dedicated for use as the processor's *Run* input.

The code in Figure 14 is provided on the course website, along with a few other source-code files: *flipflop.v*, *inst_mem.v*, *inst_mem.mif*, and *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. The memory has one 16-bit wide read/write data port and is 256 words deep.

The *proc.v* file provides partially-completed Verilog code for the enhanced processor. This code 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*, *mvnz*, and *mvnc* instructions. In this part you are to augment the provided Verilog code to complete the implementation of the *ld* and *st* instructions. You do not need to work on the *mvnz* or *mvnc* instructions 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;
   reg [15:0] DIN;
   wire W, Sync, Run;
   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

   // synchronize the Run input
   flipflop U1 (SW[9], KEY[0], CLOCK_50, Sync);
   flipflop U2 (Sync, KEY[0], CLOCK_50, Run);

   proc U3 (DIN, KEY[0], CLOCK_50, Run, 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 @ (*)
   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], LED_reg_cs & W, CLOCK_50, LED_reg);
   assign LEDR[8:0] = LED_reg;
   assign LEDR[9] = Run;

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

Figure 14: Verilog code for the top-level file.

Perform the following:

1. Augment the code in *proc.v* so that the enhanced processor fully implements the *ld* and *st* instructions. Test your Verilog code by using the ModelSim simulator. Include in the project the top-level file from Figure 14 and the other related source-code files. Sample setup files for ModelSim are provided on the course website, along with the other files for this exercise. The memory module *inst_mem* will be initialized with the contents of the file *inst_mem.mif*. This file represents the assembly-language program shown in Figure 15, 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. Make sure that your testbench sets the *Run* switch, SW_9, so that your processor will execute instructions. Observe the signals inside your processor, toggle the other SW switches, and observe the LED signals in the top-level module.

   The file *inst_mem.mif* was created by *assembling* the program in Figure 15 using the Assembler *sbasm.py*. This Assembler is written in Python and is available on the course website. To use this Assembler you have to first install Python (version 2.7) on your computer. The Assembler includes a README file that explains its use. The *sbasm.py* assembler allows you to define symbolic constants using a .define *directive*, and it supports the use of labels for referring to lines of code, such as MAIN in the figure.

```
.define LEDR_ADDRESS 0x1000
.define SW_ADDRESS 0x3000

// Read SW switches and display on LEDs
         mvi    r3, #LEDR_ADDRESS // point to LED port
         mvi    r4, #SW_ADDRESS   // point to SW port
MAIN:    ld     r0, [r4]          // read SW values
         st     r0, [r3]          // light up LEDs
         mvi    r7, #MAIN
```

Figure 15: Assembly-language program that uses *ld* and *st* instructions.

2. Once your simulation results are correct, create a new Quartus project for your Verilog code. Compile your code using the Quartus Prime software, and download the resulting circuit into the DE1-SoC board. Toggle the SW switches and observe the LEDs to test your circuit.

Part IV

In this part you are to create a new Verilog module to be instantiated in your top-level module from Part III. The new module will function as an output port called *seg7_scroll*. It will allow your processor to write data to each of the six 7-segment displays on a DE-series board. Include six write-only seven-bit registers in the *seg7_scroll* module, 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.

Create the necessary address decoding to allow the processor to write to the registers in the *seg7_scroll* module at addresses in which $A_{15} \dots A_{12} = 2$. Address 0x2000 should select the register that controls display *HEX0*, 0x2001 should select the register for *HEX1*, and so on. If your processor writes 0 to address 0x2000, then the *seg7_scroll* 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. You will need to add ports to your top-level design for the six 7-segment displays. Pin assignments for these ports, which are called *HEX0*[6:0], *HEX1*[6:0], . . ., *HEX6*[6:0], are included in the Quartus settings files provided on the course website. Segment 0 is on the top of a seven-segment display, and then segments 1 to 5 are assigned in a clockwise fashion, with segment 6 being in the middle of the display. Thus, the Verilog assignment statement

```
HEX0[6:0] = 7'b01101101;
```

would display the digit 5 on *HEX0*. Perform the following:

1. You may want to first simulate/debug your code with ModelSim. Then, create a Quartus project for this part of the exercise and write the Verilog code for the *seg7_scroll* module. Modify your top-level module to connect the processor to the *seg7_scroll* module, and to connect this module to the HEX display pins.

2. To test your circuit you can use the *inst_mem.mif* file provided for this part of the exercise on the course website, which corresponds to the assembly-language program shown in Figure 16. This program works as follows: it reads the SW switch port and lights up a seven-segment display corresponding to the value read. For example, if the SW switches were set to 0, then the digit 0 would be shown on *HEX0*. If the switches were set to 1, then the digit 1 would be displayed on *HEX1*, and so on, up to the digit 5 which would be shown on *HEX5*.

```
.define LED_ADDRESS 0x1000
.define HEX_ADDRESS 0x2000
.define SW_ADDRESS 0x3000

// This program shows a decimal digit on the HEX displays
         mv     r5, r7             // return address for subroutine
         mvi    r7, #BLANK         // call subroutine to clear HEX displays
MAIN:    mvi    r2, #HEX_ADDRESS   // point to HEX port
         mvi    r3, #DATA          // used to get 7-segment display patterns

         mvi    r4, #SW_ADDRESS    // point to SW port
         ld     r0, [r4]           // read switches
         mvi    r4, #LED_ADDRESS   // point to LED port
         st     r0, [r4]           // light up LEDs
         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

         mvi    r7, #MAIN

// Subroutine BLANK
// This subroutine clears all of the HEX displays
// input: none
// returns: nothing
BLANK:
         mvi    r0, #0             // used for clearing
         mvi    r1, #1             // used to add/sub 1
         mvi    r2, #HEX_ADDRESS   // point to HEX displays
         st     r0, [r2]           // clear HEX0
         add    r2, r1
         st     r0, [r2]           // clear HEX1
         add    r2, r1
         st     r0, [r2]           // clear HEX2
         add    r2, r1
         st     r0, [r2]           // clear HEX3
         add    r2, r1
         st     r0, [r2]           // clear HEX4
         add    r2, r1
         st     r0, [r2]           // clear HEX5

         add    r5, r1
         add    r5, r1
         mv     r7, r5             // return from subroutine

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

Figure 16: 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 *mvnz* and *mvnc* instructions. To do this, you should create two *condition-code flags* in your processor. One flag, z, should be set to 1 when the adder/subtracter unit in the processor generates a result of zero; otherwise z should be 0. The other flag, c, should be set to 1 when the adder/subtracter unit produces a carry-out of 1; otherwise c should be 0. Thus, c should be 1 when an *add* instruction generates a carry-out, or when a *sub* instruction requires a borrow for the most-significant bit. The *mvnz* and *mvnc* instructions should perform a *mv* operation when $z = 0$ and $c = 0$, respectively.

The *mv* instruction should also affect the z flag, based on whether the result produced by the instruction is zero. For example, if register *r0* contains 0, then *mv r1,r0* should set the z flag to 1, because the instruction produces the result *r1* = 0. The *mv* instruction may have an undefined effect on the c flag.

Perform the following:

1. Enhance your processor so that it implements the condition-code flags z and c, and supports the *mvnz* and *mvnc* instructions. Also, modify your processor's *mv* instruction so that it sets, or clears, the z flag appropriately. You may want to use ModelSim to test/debug your code, and then create a new Quartus project to implement this part of the exercise in the DE1-SoC board.
2. To test your processor, you can use the *inst_mem.mif* file that is provided for this part of the exercise on the course website. This file corresponds to the assembly-language program displayed in Figure 17. 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 the binary value `1000` on the LEDs, representing eight successful tests. 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. If a failure occurs, then the offending instruction can be identified by cross-referencing the LED pattern with the addresses in the file *inst_mem.mif*.

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

// This code tests various instructions
         mvi    r0, #0
         mvi    r1, #1
         mvi    r2, #0      // used to count number of successful tests
         mvi    r5, #FAIL

         mvi    r6, #T1     // save address of next instruction
         add    r0, r0      // set the z flag
// test mvnz
T1:      mvnz   r7, r5      // should not take the branch!

         add    r2, r1      // incr success count
         mvi    r4, #S1
         mvi    r6, #T2     // save address of next instruction
// test mv's effect on z flag
         mv     r3, r1      // reset the z flag
         mvnz   r7, r4      // should take the branch!
T2:      mvi    r7, #FAIL

S1:      add    r2, r1      // incr success count
         mvi    r4, #S2
         mvi    r6, #T3     // save address of next instruction
```

Figure 17: Assembly-language program that tests various instructions. (Part a)

```
         add   r3, r1      // reset the z flag
         mvnz  r7, r4      // should take the branch!
T3:      mvi   r7, #FAIL

S2:      add   r2, r1      // incr success count
         mvi   r6, #T4
         mv    r0, r0      // set the z flag
// test mv's effect on z flag
T4:      mvnz  r7, r5      // should not take the branch!

         add   r2, r1      // incr success count
         mvi   r6, #T5     // save address of next instruction
         mvi   r3, #0xffff
         add   r3, r1      // set the c flag
// test mvnc
T5:      mvnc  r7, r5      // should not take the branch!

         add   r2, r1      // incr success count
         mvi   r4, #S3
         mvi   r6, #T6
         add   r3, r0      // clear carry flag
// test mvnc
         mvnc  r7, r4      // should take the branch!
T6:      mvi   r7, #FAIL

S3:      add   r2, r1      // count the success

// finally, test ld and st from/to memory
         mvi   r6, #T7     // save address of next instruction
         mvi   r4, #_LDTEST
         ld    r4, [r4]
         mvi   r3, #0xA5A5
         sub   r3, r4
T7:      mvnz  r7, r5      // should not take the branch!

         add   r2, r1      // incr success count
         mvi   r6, #T8     // save address of next instruction
         mvi   r3, #0xA5A5
         mvi   r4, #_STTEST
         st    r3, [r4]
         ld    r4, [r4]
         sub   r3, r4
T8:      mvnz  r7, r5      // should not take the branch!

         add   r2, r1      // incr success count
         mvi   r7, #PASS

// Show the result on the HEX displays
FAIL:    mvi   r3, #LED_ADDRESS
         st    r6, [r3]    // show address of failed test on LEDs
         mvi   r5, #_FAIL
         mvi   r7, #PRINT
PASS:    mvi   r3, #LED_ADDRESS
         st    r2, [r3]    // show success count on LEDs
         mvi   r5, #_PASS
```

Figure 17: Assembly-language program that tests various instructions. (Part *b*)

```
PRINT:    mvi   r1, #1
          mvi   r4, #HEX_ADDRESS  // address of HEX0
          // We would normally use a loop counting down from 6
          // with mvnz to display the six letters. But in this
          // testing code we can't assume that mvnz even works!
          ld    r3, [r5]     // get letter
          st    r3, [r4]     // send to HEX display
          add   r5, r1       // ++increment character pointer
          add   r4, r1       // point to next HEX display
          ld    r3, [r5]     // get letter
          st    r3, [r4]     // send to HEX display
          add   r5, r1       // ++increment character pointer
          add   r4, r1       // point to next HEX display
          ld    r3, [r5]     // get letter
          st    r3, [r4]     // send to HEX display
          add   r5, r1       // ++increment character pointer
          add   r4, r1       // point to next HEX display
          ld    r3, [r5]     // get letter
          st    r3, [r4]     // send to HEX display
          add   r5, r1       // ++increment character pointer
          add   r4, r1       // point to next HEX display
          ld    r3, [r5]     // get letter
          st    r3, [r4]     // send to HEX display
          add   r5, r1       // ++increment character pointer
          add   r4, r1       // point to next HEX display
          ld    r3, [r5]     // get letter
          st    r3, [r4]     // send to HEX display
          add   r5, r1       // ++increment character pointer
          add   r4, r1       // point to next HEX display

HERE:     mvi   r7, #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 0xA5A5
_STTEST:  .word 0x5A5A
```

Figure 17: Assembly-language program that tests various instructions. (Part *c*)

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, along with the SW switches. If the SW switches are set to their maximum value, `0b111111111`, then the delay loops should cause the counter to increment slowly enough so that each change in the counter can be visually observed on the LEDs. Lowering the value of the SW switches should make the counter increment more quickly up to some maximum.

Assemble your program by using the *sbasm.py* Assembler (or assemble it by hand if you prefer!). Save the output produced by *sbasm.py* in a new file named *inst_mem.mif*. Make sure that the new *MIF* file is stored in the folder that holds your Quartus project for Part V. Then use the Quartus command `Processing > Update Memory Initialization File`, to include your new *inst_mem.mif* file in your Quartus project. Next, select the Quartus command `Processing > Start > Start Assembler` to produce a new programming *bitstream* for your DE-series board. Finally, use the Quartus Programmer to download the new bitstream onto your board. If the *Run* signal is asserted, then your processor will execute your the program.

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 18. This code provides a subroutine that divides the number in register r0 by 10, returning the quotient in r2 and the remainder in r0. Dividing by 10 is a useful operation when performing binary-to-decimal conversion.

```
// subroutine DIV10
//   This subroutine divides the number in r0 by 10
//   The algorithm subtracts 10 from r0 until r0 < 10, and keeps count in r2
//   input: r0
//   returns: quotient Q in r2, remainder R in r0
DIV10:
         mvi   r1, #1
         sub   r6, r1              // save registers that are modified
         st    r3, [r6]
         sub   r6, r1
         st    r4, [r6]            // end of register saving

         mvi   r2, #0              // init Q
         mvi   r3, RETDIV          // for branching
DLOOP:   mvi   r4, #9              // check if r0 is < 10  yet
         sub   r4, r0
         mvnc  r7, r3              // if so, then return

INC:     add   r2, r1              // but if not, then increment Q
         mvi   r4, #10
         sub   r0, r4              // r0 -= 10
         mvi   r7, DLOOP           // continue loop
RETDIV:
         ld    r4, [r6]            // restore saved regs
         add   r6, r1
         ld    r3, [r6]            // restore the return address
         add   r6, r1
         add   r5, r1              // adjust the return address by 2
         add   r5, r1
         mv    r7, r5              // return results
```

Figure 18: A subroutine that divides by 10

As described for Part VI, assemble your program with *sbasm.py*, update your *MIF* file in the Quartus software, generate a new bitstream file by using the Quartus Assembler, and then download the new bitstream onto your DE-series board to run your code.