OPAE
Laboratory Exercise 1

Designing and Implementing an Intel AFU

This is an introductory exercise about heterogeneous computing using Intel technology. Figure 1 depicts a computer that includes both an Intel® processor and a *Field-Programmable Gate Array* (FPGA). The FPGA is a programmable logic device, which can be configured to implement whatever hardware circuit is needed for a particular application. In the computer, the main circuit board that houses the processor is connected to the FPGA board via a *PCI express* (PCIe) port. In this introductory exercise we will show how to utilize the system in Figure 1 as a *heterogeneous* computer in which both the processor and FPGA are used together to implement a computation.

For the development of heterogeneous computing applications Intel provides a set of tools called the *Intel Acceleration Stack*. It includes features for the development of both hardware and software components, as well as various mechanisms for connecting these components together. To gain access to the *Intel Acceleration Stack* we will make use of a cloud-based computing service called the Intel FPGA DevCloud®. The DevCloud offers server-class computers that contain both a high-end Intel processor and an Intel Arria® 10 FPGA.

In various parts of this exercise you will need to execute commands within the DevCloud computing environment. This means that you are expected to obtain a *user account* on this cloud service and to be familiar with its use. Instructions for obtaining access to the Intel FPGA DevCloud, as well as an introduction to this computing environment, can be found in the tutorial *Introduction to the Intel FPGA DevCloud*. This tutorial is provided by the Intel FPGA Academic Program and is available for download along with this laboratory exercise.

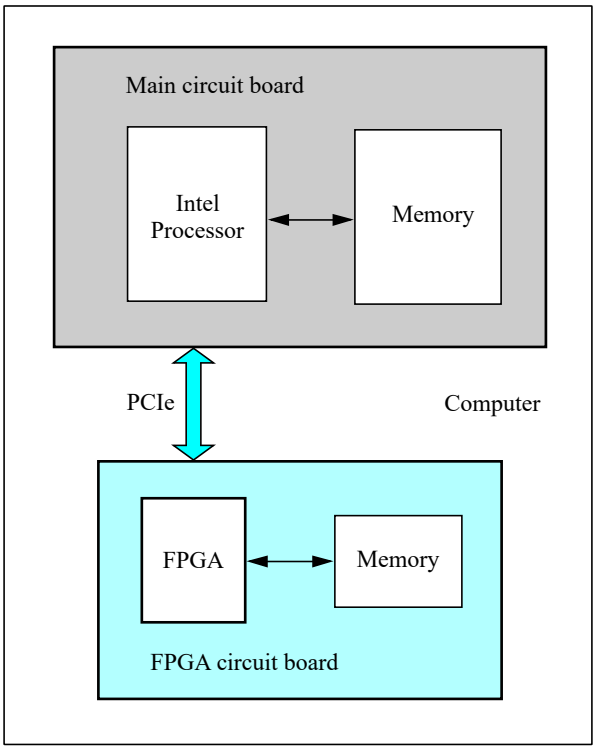

Figure 1: A heterogeneous computer.

In a heterogeneous computing application, Intel refers to the part of the system implemented in an FPGA as an *Accelerator Functional Unit* (AFU). In this exercise you will design an AFU that provides a hardware component which is used to solve part of a computation. To complete the design of this AFU you will need to have a good grasp of the Verilog hardware description language (including some System Verilog extensions), which is used in the design of the AFU hardware, and the C programming language, which is used to write software programs for the processor that make use of the AFU. A good working knowledge of Linux is also desirable, as this is the operating system deployed on the Intel FPGA DevCloud.

This exercise is organized into the following stages:

1. Design a hardware component that will form the main part of an AFU. At this stage the component will be just a "normal" hardware *module* specified in Verilog code. This module should be developed using a computer of your own choice (the DevCloud is not needed at this stage). We expect that you will compile and test your Verilog code on your own computer using a simulation tool with a Verilog *testbench*.

2. Augment the hardware component described above to create an AFU. This step involves adding specific ports and hardware registers that are required in the Verilog code for an AFU. The AFU's registers will be memory-mapped to a specific part of the processor's address space. To compile the AFU we will make use of the hardware development tools that are provided on the DevCloud.

3. Design software applications that utilize the AFU to perform computations along with a processor. The software code will be compiled using the software development tools on the DevCloud.

Part I

Figure 2 provides a high-level block diagram of an AFU connected to a processor via a *PCI Express* (PCIe) port. The part of the system implemented in the FPGA, highlighted in a blue color, is called the *FPGA Interface Manager* (FIM). The FIM comprises two main components: the *FPGA Interface Unit* (FIU) and the *AFU*. The purpose of the FIU is to act as a bridge between the PCIe interface and the AFU. As indicated in the figure the FIU is connected to the AFU via a *Protocol Port*. This port can implement a number of communication protocols based on the AFU designer's preference. For this exercise we will use the *Core Cache Interface* protocol® (CCI-P), which is discussed in Part II.

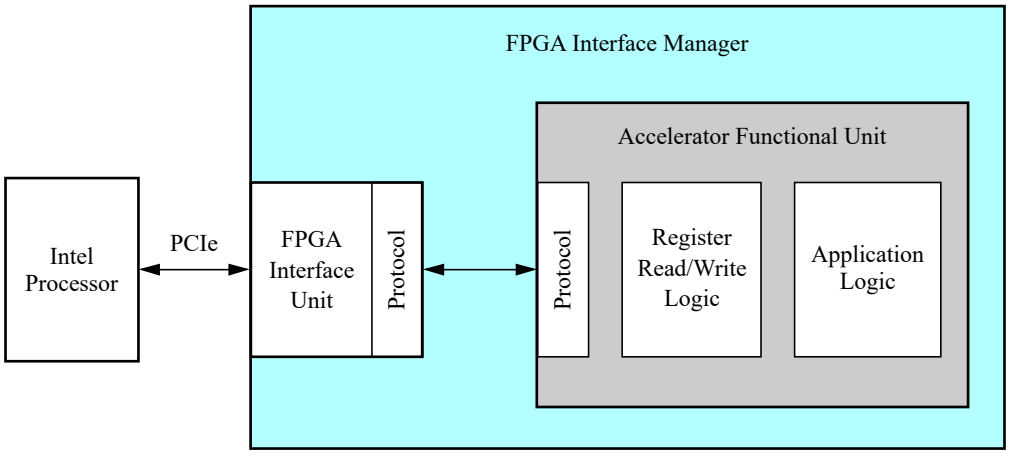

Figure 2: A block diagram.

The AFU in Figure 2 includes logic circuitry that facilitates access via the protocol (CCI-P) port to some *registers*. This circuitry provides address decoding that allows the processor to read/write AFU registers using memory-mapped I/O. Some of these registers have dedicated purposes that are required to be compatible with the Intel Acceleration Stack. Other registers are part of the *Application Logic* in the AFU, which is the part of the AFU that is used to perform computations along with the processor.

The purpose of the AFU in this exercise is to provide a hardware module that generates *random* integers. The processor can configure the AFU so that it produces integers within a desired range, or to affect the sequence of values that are generated. Whenever it is required for a computation, the processor can read a random integer value from the AFU.

In this part of the exercise we will design only the Application Logic component of the AFU. The rest of the AFU circuitry depicted in Figure 2 will be designed in Part II. To generate random integer the Application Logic component uses a *linear feedback shift register* (LSFR).

Figure 3 depicts an LFSR that implements an n-bit register called Q. If the *load* input $L = 1$ then on an active clock edge Q is loaded with the *seed* value S. But if $L = 0$ then the value of Q depends on the *enable* input E. If $E = 0$ then Q cannot change. But if $E = 1$ then Q behaves as a *shift register*. The value that is shifted into each flip-flop is dependent on the *polynomial* P.

For each bit-position, i, in Figure 3 let the data input of the flip-flop, Q_i, be called D_i. When the LFSR is acting as a shift register, if $P_i = 0$ then $D_i = Q_{i+1}$. But if $P_i = 1$, then $D_i = Q_{i+1} \oplus Q_0$. This arrangement of exclusive-OR gates in the LFSR allows it to produce a sequence of n-bit values that can be used as "random" integers. Note that at position $n - 1$ the value of $D_i = 0$ if $P_{n-1} = 0$, and $D_i = Q_0$ if $P_{n-1} = 1$.

In addition to the register Q shown in Figure 3 the Application Logic module includes an $n-$bit register for storing the polynomial P, and a 2-bit *control* register, C. On reset $C = C_1C_0 = 00$ and the LFSR is in *stopped* mode. Software running on the processor can write to the control register to use the LFSR in two different *modes*.

Setting the control register bit $C_1 = 1$ enables the *continuous* mode of operation. While it is in this mode the LFSR will generate a new "random" integer for each of its active clock edges. The LFSR can be stopped by setting $C = 00$. Setting $C_1 = 0$ and $C_0 = 1$ puts the LFSR into *step* mode. In this mode the LFSR will be enabled for only *one* of its clock cycles, so that it will generate exactly one new pseudo-random integer. The LFSR can be placed back into the stopped mode by setting $C = 00$.

A software application may choose to put the LFSR into either continuous or step mode depending on the requirements of the computation being performed. To support these modes of operation the Application Logic module has to be able to control the enable signal on the LFSR. This can be done by using a finite state machine (FSM) controller, such as the one illustrated by the state diagram in Figure 4.

The control bits C_1C_0 are the inputs to the FSM, which produces an output z. This output is intended to be connected directly to the enable input of the LFSR.

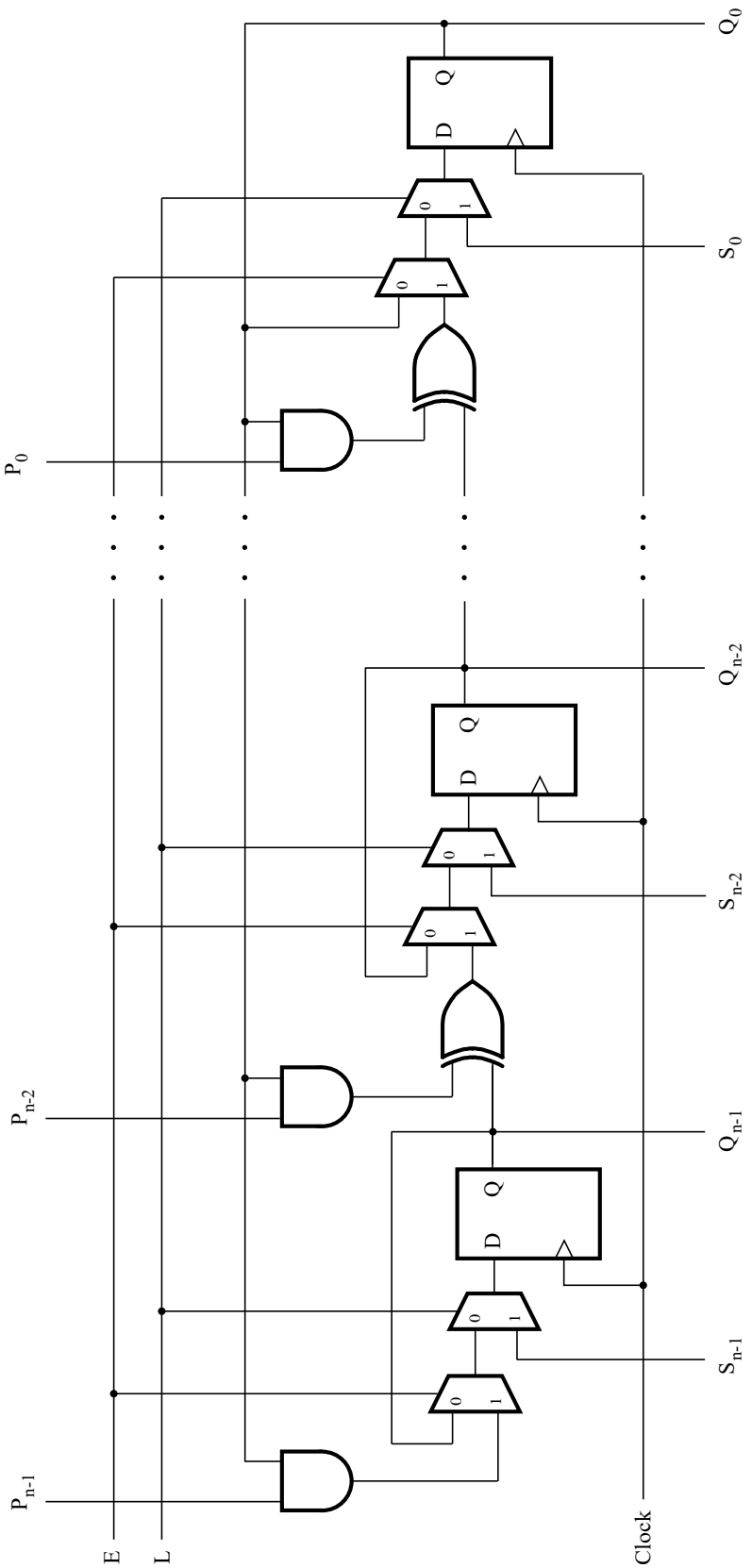

Figure 3: A configurable linear feedback shift register.

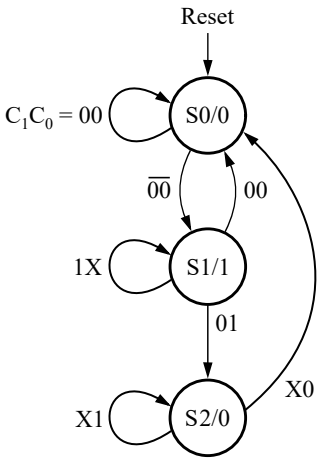

Figure 4: A finite state machine controller.

The FSM in Figure 4 starts in the state S0 and produces the output $z = 0$, which is indicated in the state diagram as S0/0. As long as $C_1C_0 = 00$ the FSM remains in S0. Consider now the case where C_1 changes to 1. On the next active clock edge the FSM transitions to state S1, where it produces $z = 1$ to enable the LFSR. The FSM remains in this state as long as $C_1 = 1$, as indicated by the arrow labeled 1X (the value 1X represents both of the cases $C_1C_0 = 10$ and $C_1C_0 = 11$). This scenario puts the LFSR in the *continuous* operating mode, where it will generate a new random integer for each clock cycle. When C changes back to 00 the FSM transitions back to the starting state S0. Next, consider the case when $C_1 = 0$ but C_0 changes to 1. On the next active clock edge the FSM changes to state S1, setting $z = 1$, but it remains in this state for only one clock cycle. Following the arrow labeled $C_1C_0 = 01$ the subsequent clock edge causes the FSM to move to state S2, where $z = 0$. As long as $C_0 = 1$ the FSM will remain in S2, and will return to S0 when C_0 changes to 0. Since this scenario causes the LFSR to generate exactly one new random integer, it implements the *step* operating mode.

A diagram of the Application Logic circuit is shown in Figure 5. It includes a two-bit address input A, an n-bit data input D, and a write input W that represent the way this circuit will (in Part II) be included in an AFU and connected to a processor. The address decoding, which is done using NOR gates, assigns the address $A_1A_0 = 00$ to the polynomial register, $A_1A_0 = 01$ to the LFSR register, and $A_1A_0 = 10$ to the control register. The AND gates in the circuit ensure that each register can be loaded with the data D when its address is selected and $W = 1$. When data is written to the LFSR it is loaded into the seed input S; the LFSR enable input E is controlled by the z output of the FSM.

Perform the following steps to complete the design of the Application Logic module with Verilog code:

1. You should design your Verilog code on a "home" computer of your choosing. No tools on the DevCloud are needed for this part of the exercise. To compile and simulate your Verilog code we assume that you have access to a Verilog simulator, such as the *ModelSim* simulator. Although any modern Verilog simulator can be used, the specific version that we refer to in the discussion below is called *ModelSim-Intel FPGA Starter Edition 2020.1*. If needed, you can download and install a *free* version of this simulator from Intel's website (any recent version of the software can be used). An introduction to this simulator can be found in the tutorial *Using the ModelSim-Intel FPGA Simulator with Verilog Testbenches*. This tutorial is available on the Internet from the Intel FPGA Academic Program.

2. Make a new folder on your computer for this part of the exercise. Create a Verilog source-code file named *application.sv* (the *sv* filename extension enables the use of System Verilog extensions), and write the code for the circuit in Figure 5. A template for this Verilog code is shown in Figure 6.

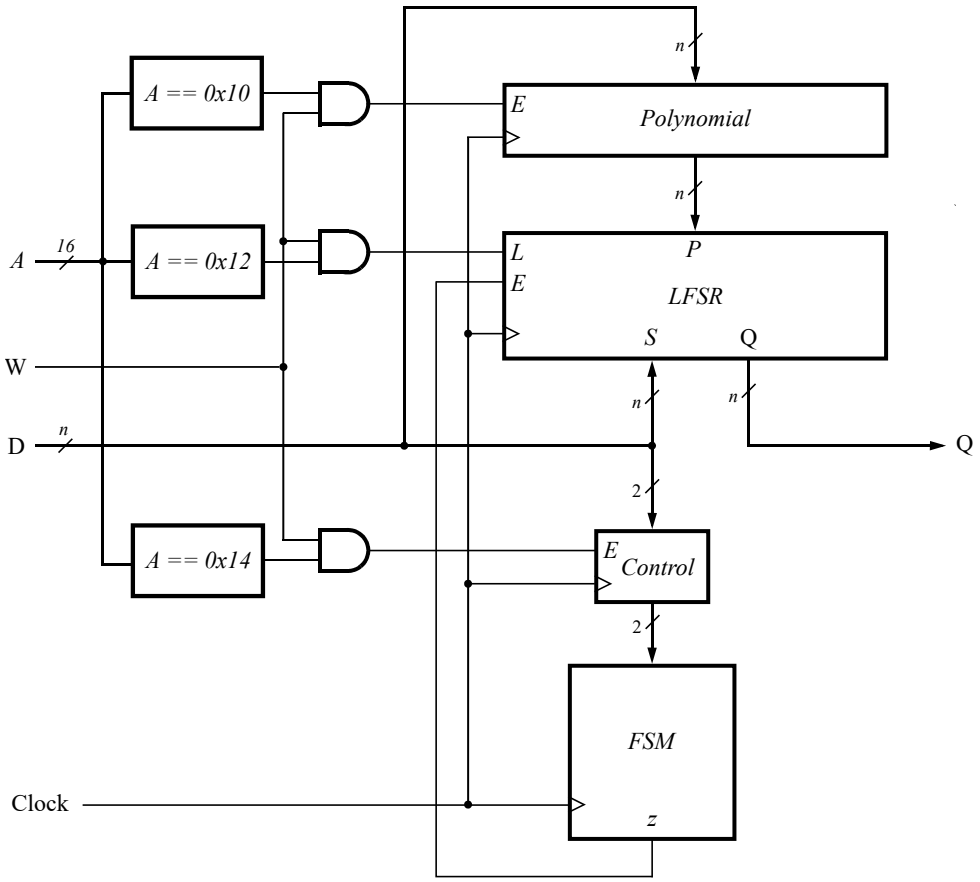

Figure 5: The application logic circuit.

```
module application (reset, clock, W, A, D, Q); // A configurable LFSR.
    parameter n = 8;
    input reset, clock, W;
    input [15:0] A; // POLY_REG: 0x0010, LFSR_REG: 0x0012, CTRL_REG: 0x0014

    input [n-1:0] D;                          // input data
    output logic [n-1:0] Q;                   // LFSR output register
    logic [1:0] Ctrl;                         // control register
    logic [n-1:0] Poly;                       // polynomial register
    logic [n-1:0] Mask, Next;                 // LFSR intermediate values
    enum logic [1:0] {S0, S1, S2 } y, Y;      // FSM, state and next state
    logic z;                                  // FSM output

    always_ff @(posedge clock)                // polynomial register
        if (reset)                            // synchronous reset
            Poly <= '0;
        else if (W && A == 16'h0010)          // set the polynomial
            Poly <= D;

    // define the LFSR
    // define the control register
    // define the finite state machine
endmodule
```

Figure 6: A template for the Application Logic Verilog code.

3. Simulate your code to ensure that it works correctly. Example results produced by using *ModelSim* for a correctly-designed circuit are given in Figures 7 and 8. For convenience of reference, the simulation is based on a clock waveform with a 10 ns period. After resetting the circuit (each register has an active-high synchronous reset capability), the simulation loads initial values into the three registers in the circuit. On the clock edge at 15 ns in simulation time the polynomial register is loaded with the value 221. In the subsequent clock cycle a seed value of 1 is loaded into the LFSR register. Then, the clock edge at 35 ns sets the control register to $C_1C_0 = 10$, causing the FSM to enter the *continuous* operating mode in state S1 at 45 ns. Note that Q $= P = 221$ after the clock edge at 55 ns, which is the result of setting the seed to the value 1. Over the next few clock cycles, as shown in Figure 7, the LFSR produces a sequence of "random" integers. At 135 ns the simulation loads the control register with $C_1C_0 = 00$, which causes the FSM to change back to state S0.

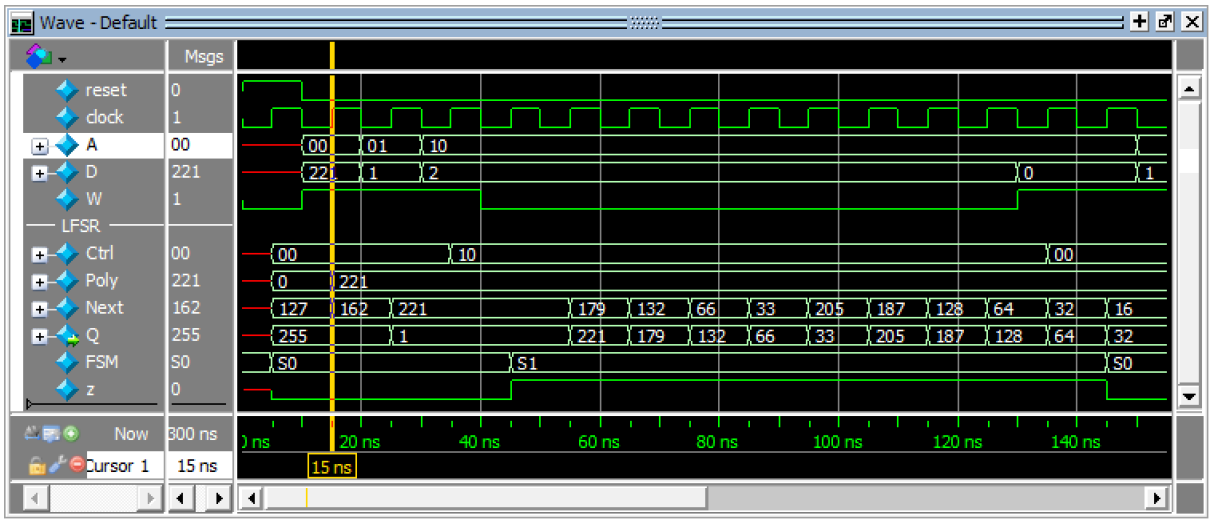

Figure 7: Using the LFSR in continuous mode.

Figure 8 continues the simulation results from Figure 7. At 155 ns the seed value in the LFSR is reinitialized to 1. Then, in the next clock cycle the control register is set to $C_1C_0 = 01$, causing the FSM to enter *step* mode by transitioning through state S1 to S2. The control register is set back to $C_1C_0 = 00$ at 185 ns, so that the FSM moves back to state S0 at 195 ns. Finally, the step mode is used to generate a few more "random" values as shown in the remainder of the simulation. The sequence of integers generated in Figures 7 and 8 are identical, because they are derived from the same seed and polynomial. Due to this behavior, an LFSR is said to generate integers that are *pseudo* random.

The simulation results from Figures 7 and 8 are based on the Verilog *testbench* shown in Figure 9. This testbench and the template code in Figure 6 are provided for your use along with this laboratory exercise.

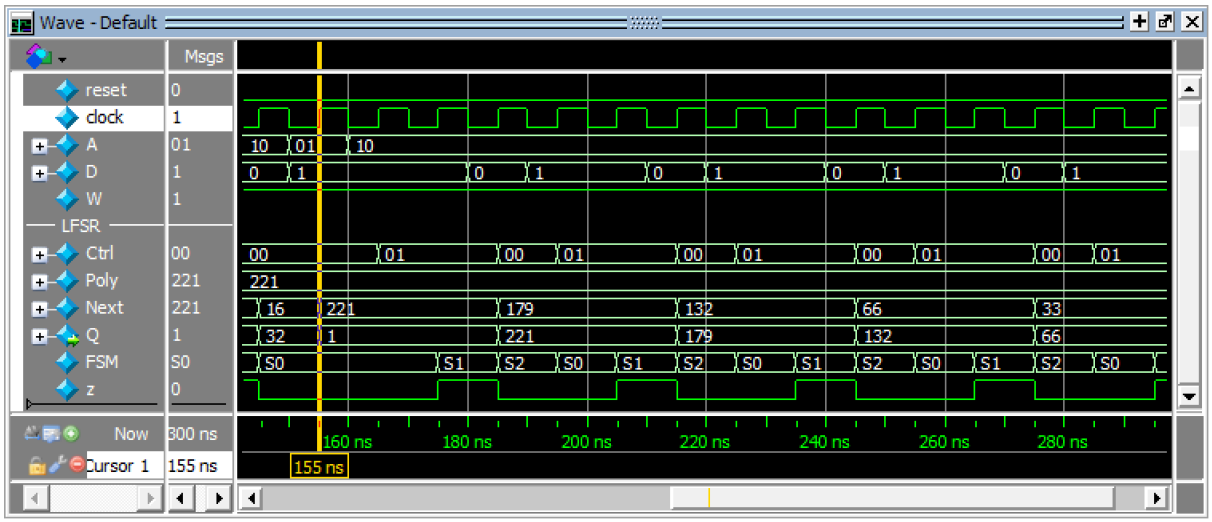

Figure 8: Using the LFSR in step mode.

```
`timescale 1ns / 1ps

module testbench ( );
    reg reset, clock, W;                // declare design under test (DUT)
        inputs
    reg [15:0] A;                       // address
    reg [7:0] D;                        // data
    wire [7:0] Q;                       // declare DUT outputs

    application U1 (reset, clock, W, A, D, Q); // instantiate the DUT

    // define a 100 MHz clock waveform
    always
        #5 clock <= ~clock;

    // assign inputs at various times
    initial
    begin
        clock <= 1'b0;
        reset <= 1'b1;
        W <= 1'b0; A <= 2'bXX;
        #10  reset <= 1'b0;             // initialize the polynomial
             A <= 16'h0010; D <= 221; W <= 1'b1;
        #10  A <= 16'h0012; D <= 1'b1;  // initialize the seed
        #10  A <= 16'h0014; D <= 2'b10; // Set CTRL to continuous mode
        #10  W <= 1'b0;
        #90  D <= 0; W <= 1'b1;         // set CTRL to stopped mode
        #20  A <= 16'h0012; D <= 1;     // re-initialize the seed
        #10  A <= 16'h0014; D <= 8'b01; // set CTRL to step mode
        #20  D <= 8'b00;    // stop
        #10  D <= 8'b01;    // step
        #20  D <= 8'b00;    // stop
        ...
    end // initial
endmodule
```

Figure 9: The simulation testbench.

Part II

In this part of the exercise we will augment the application logic circuit from Part I to create an AFU that can be connected to a processor. This step will require the design of some additional Verilog code, as well as the use of a number of hardware development tools that are provided on the DevCloud. As indicated in Figure 2 an AFU has a *protocol* port and some logic for reading and writing registers based on this protocol. For this exercise we will use the *Core Cache Interface* Protocol (CCI-P) that is provided by Intel. This protocol is part of the FIU interface in Figure 2 and defines a signaling convention for connecting the processor and AFU.

Figure 10 provides a template for your AFU Verilog module. In Part a of the figure, Lines 1 and 2 *include* the Verilog *header* files *platform_if.vh* and *afu_json_info.vh*. The first of these files defines some CCI-P data types that are used in the code, and the second one specifies some key information about your AFU. We will discuss the contents of *afu_json_info.vh* shortly.

Your AFU module is required to have the name *afu*, as given in the figure. Depending on its functionality, an AFU may have different ports. In our case, in addition to *clock* and *reset* inputs, we require an input, called *rx*, for receiving CCI-P data, and an output, named *tx*, for transmitting data. These *rx* and *tx* ports have special data types, shown in Lines 7 and 8, that are defined via the header file *platform_if.vh*.

```
1  `include "platform_if.vh"
2  `include "afu_json_info.vh"
3
4  module afu (clock, reset, rx, tx);
5      input  clock;                    // CCI-P clock
6      input  reset;                    // CCI-P reset
7      input  t_if_ccip_Rx rx;          // receive channel
8      output t_if_ccip_Tx tx;          // transmit channel
9
10     parameter n = 32;
11     logic [n-1:0] Q, Poly;           // LFSR and polynomial registers
12     logic [1:0] Ctrl;                // control register
13     logic [15:0] A;                  // Address
14     logic [n-1:0] D;                 // Data
15     logic W;                         // Write signal
16     ... declare other signals
17
18     t_ccip_c0_ReqMmioHdr mmioHdr;    // channel c0 header
19     assign mmioHdr = t_ccip_c0_ReqMmioHdr'(rx.c0.hdr);
20
21     assign A = mmioHdr.address;      // rename address signal
22     assign D = rx.c0.data;           // rename data signal
23     assign W = rx.c0.mmioWrValid;    // rename write signal
24
25     always_ff @(posedge clock)       // the polynomial register
26         if (reset)
27             Poly <= '0;
28         else if (W && A == 16'h0010)
29             Poly <= D;  // set the polynomial
30
31     ... define the LFSR
32     ... define the control register
33     ... define the finite state machine
```

Figure 10: A template for the AFU Verilog code (Part a).

```
34      logic [127:0] afu_id = `AFU_ACCEL_UUID; // from afu_json_info.vh
35
36      always_ff @(posedge clock) begin // respond to memory-mapped I/O reads
37          if (reset) begin
38              tx.c1.hdr <= '0;
39              tx.c1.valid <= '0;
40              tx.c0.hdr <= '0;
41              tx.c0.valid <= '0;
42              tx.c2.hdr <= '0;
43              tx.c2.mmioRdValid <= '0;
44          end
45          else begin
46              // clear read response flag in case there was a response last cycle
47              tx.c2.mmioRdValid <= 0;
48
49              // serve MMIO read requests
50              if (rx.c0.mmioRdValid == 1'b1) begin
51                  // copy TID, which host needs to map response to request
52                  tx.c2.hdr.tid <= mmioHdr.tid;
53                  // post response
54                  tx.c2.mmioRdValid <= 1;
55
56                  case (A)
57                      // AFU header
58                      16'h0000: tx.c2.data <= {
59                          4'b0001, // Feature type = AFU
60                          8'b0,    // reserved
61                          4'b0,    // afu minor revision = 0
62                          7'b0,    // reserved
63                          1'b1,    // end of DFH list = 1
64                          24'b0,   // next DFH offset = 0
65                          4'b0,    // afu major revision = 0
66                          12'b0    // feature ID = 0
67                      };
68                      16'h0002: tx.c2.data <= afu_id[63:0];   // AFU_ID_L
69                      16'h0004: tx.c2.data <= afu_id[127:64]; // AFU_ID_H
70                      16'h0006: tx.c2.data <= 64'h0;          // DFH_RSVD0
71                      16'h0008: tx.c2.data <= 64'h0;          // DFH_RSVD1
72
73                      // application logic registers
74                      16'h0010: tx.c2.data <= 64'(Poly);
75                      16'h0012: tx.c2.data <= 64'(Q);
76                      16'h0014: tx.c2.data <= 64'(Ctrl);
77
78                      default:  tx.c2.data <= 64'h0;
79                  endcase
80              end
81          end
82      end
83  endmodule
```

Figure 10. A template for the AFU Verilog code (Part *b*).

In Lines 10 to 15 the code sets the parameter $n = 32$ and declares a number of signals. The signal Q represents the LFSR in Figure 5. For the Verilog code in Figure 6 the signal Q served as the output of the module, but for the AFU the processor reads the contents of registers via the *tx* output port.

Lines 18 and 19 declare a special type of CCI-P signal called *mmioHdr*. This signal provides the *address* of the

register currently being accessed by the processor, which is assigned to the 16-bit signal A in Line 21. The address is implemented in the CCI-P as an *offset* from a *base* address that is used for memory-mapped I/O. Each address A is aligned to a double-word (32-bit) boundary in the processor's address space. Lines 14 and 15 assign the processor n-bit *data* signal and *write* control signal to the variables D and W, respectively. These signals are valid whenever the processor is performing a *write* to a register in the AFU.

Using the polynomial register as an example, the *always* block in Line 25 shows how registers can be defined in the AFU. Note that this code is very similar to that from Figure 6.

Part b of Figure 10 gives mandatory code that must be present in an AFU to support read operations from the processor at specific addresses. These addresses are decoded in the AFU by using the *case* statement in Line 56. As shown, each address responds by placing data onto the *tx* output of the AFU. For address $A = 0$ the AFU responds with a particular 64-bit data pattern that is required for any AFU. Addresses $A = 2$ and $A = 4$ respond with the low and high 64-bits of the *afu_id* signal. This signal represents a globally-unique identifier for the AFU. It is declared in Line 34 and initialized with the symbolic constant `AFU_ACCEL_UUID`, which is defined in the file *afu_json_info.vh*. Finally, addresses $A = 6$ and $A = 8$ respond with 0 for our AFU. More details about the mandatory addresses that have to be supported in an AFU can be found by searching online for documentation related to CCI-P.

Lines 74 to 76 allow the processor to read the contents of the polynomial register, LFSR, and control register at the addresses $A = (10)_{16}$, $A = (12)_{16}$, and $A = (14)_{16}$, respectively. As mentioned previously, each address A represents a double-word (32 bit) processor address. This means that the addresses decoded in our case statement are aligned to a quad-word (64 bit) boundary, because the least-significant address bit $A_0 = 0$ for each address. The CCI-P specification requires this alignment for all registers in an AFU.

Perform the following steps to complete the design of the AFU:

1. Make a new folder on your home computer called `lfsr_afu`. Then, in this folder create two subfolders named `hw` and `sw`. The `sw` folder will be used in Parts 3, 4, and 5 of this exercise. Now, in the `hw` folder make a subfolder called `rtl`. You should now have created the folders illustrated in Figure 11. This arrangement of folders is required when using the compilation tools that are provided on the DevCloud.

2. In your `rtl` subfolder, make a new Verilog source-code file called *afu.sv*. Enter the code from Figure 10 into this file and, as indicated in Part a of the figure, fill in the code that is needed to define the LFSR, control register, and finite state machine. Note that the code shown in Figure 10 is provided for you in the file *afu.sv* that is part of the *design_files* material included along with this exercise.

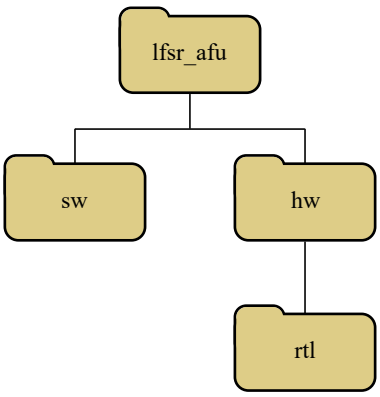

Figure 11: The arrangement of folders for an AFU.

3. To compile the Verilog code for your AFU, the development tools on the DevCloud require several files in addition to *afu.sv*. The names of the required files have to be listed in a *plain-text* file named *filelist.txt* in the `rtl` folder. The contents of this file for our AFU is shown in Figure 12. The first file listed is *lfsr_afu.json*, which gives some important information about the AFU in *JavaScript Object Notation* (JSON) format. As shown in Figure 13, this JSON file specifies the type of CCI-P port used for the AFU, which is called *ccip_std_afu*. This interface consists of *clock*, *reset*, receive (*rx*) and transmit (*tx*) ports, as given in Figure 10. The JSON file also specifies the AFU name, *lfsr_afu*, and its *Universally Unique Identifier* (UUID). The UUID shown in the figure is just a placeholder; we will generate a new UUID for the AFU by using the *uuidgen* command that is available on the DevCloud. The Verilog source-code files listed in *filelist.txt* specify the AFU and its CCI-P interface. The files *ccip_interface_reg.sv* and *ccip_std_afu.sv* have to be present in the AFU's `rtl` folder.

   The files *filelist.txt*, *lfsr_afu.json*, *ccip_interface_reg.sv*, and *ccip_std_afu.sv* are provided for you as part of the *design_files* that are included with this exercise. Copy these files into your `rtl` folder, to create the structure of files illustrated in Figure 14.

```
lfsr_afu.json

afu.sv
ccip_interface_reg.sv
ccip_st_afu.sv
```

Figure 12: The contents of *filelist.txt*.

```
{
    "version": 1,
    "afu-image": {
        "power": 0,
        "afu-top-interface": {
            "class": "ccip_std_afu"
        },
        "accelerator-clusters": [
            {
                "name": "lfsr_afu",
                "total-contexts": 1,
                "accelerator-type-uuid": "850adcc2-6ceb-4b22-9722-d43375b61c66"
            }
        ]
    }
}
```

Figure 13: The JSON file.

4. For the remainder of this exercise we assume that you are able to login to the DevCloud and configure the environment variables and settings that are needed for AFU development. To perform this step it is important to be familiar with the material presented in the tutorial *Using the Intel FPGA DevCloud for AFU Development*. First, copy the folders and files shown in Figure 14 onto the DevCloud. Then, using the Linux command line interface on the DevCloud execute the command `uuidgen` to generate a new UUID for the LFSR AFU. Enter this new UUID into the file *lfsr_afu.json*, replacing the placeholder given in Figure 13.

5. On the DevCloud, set your working directory to the `lfsr_afu` folder, and then run the command:

   ```
   afu_synth_setup -s hw/rtl/filelist.txt build_synth
   ```

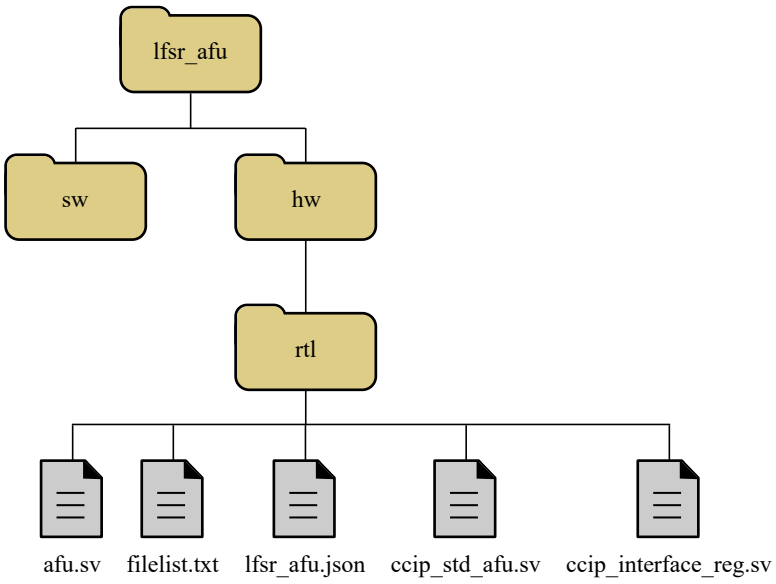

Figure 14: The files needed for an AFU.

Now, change your working directory to the newly-created `build_synth` folder and execute the command `run.sh`. This command uses the information in the *lfsr_afu.json* file to generate the Verilog header file *afu_json_info.vh*, which is included in the code shown in Figure 10. The `run.sh` command then executes the Intel Quartus® Prime software to compile the AFU into a circuit that can be implemented in the target FPGA device.

The Quartus Prime software begins by executing its synthesis tools that compile your Verilog source code. If any syntax errors are reported (they are shown in red), then fix these errors and compile again. Note that it is "normal" to receive a significant number of warning messages from the Quartus Prime software when compiling an AFU; while you should still monitor these messages, those that refer to code that is not part of your *afu.sv* file can usually be ignored.

After successful compilation of your AFU code, the Quartus Prime software generates an FPGA programming bit-stream file called *lfsr_afu.gbs*. In Intel® FPGA literature, this type of file is known as a *Green Bitstream* and represents a *partial-reconfiguation* file for the target FPGA. You can download this *gbs* file into the FPGA device on the DevCloud, where it joins the main bit-stream that is already present in the FPGA. Intel refers to the main bit-stream as the *Blue Bitstream*. To download your AFU into the FPGA execute the command:

```
fpgasupdate lfsr_afu.gbs.
```

Note that if you are using version 1.2.1 of the Arria 10 Development Stack on the DevCloud, then you have to execute two commands to program the FPGA. First, execute:

```
PACSign PR -t UPDATE -H openssl_manager -i lfsr_afu.gbs -o lfsr_afu_unsigned.gbs
```

Type `y` (*yes*) in answer to the queries that are issued by this command. Then, execute:

```
fpgasupdate lfsr_afu_unsigned.gbs.
```

Now that the AFU has been downloaded into the target FPGA device, we can develop software programs which run on the processor and make use of the AFU.

Part III

For the development of software for AFUs Intel provides a collection of open-source utilities known as the *Open Programmable Acceleration Engine* (OPAE). An example of a C program that uses the OPAE infrastructure to access the AFU developed in this exercise is given in Figure 15. This code includes the OPAE header file *mmio.h*, which is required to perform memory-mapped I/O.

```
1  #include <stdio.h>
2  #include <opae/mmio.h>
3
4  // Application Logic register addresses (offsets)
5  #define POLY_REG    0X10 << 2
6  #define LFSR_REG    0X12 << 2
7  #define CTRL_REG    0X14 << 2
8
9  int open_AFU (fpga_handle *);
10 void close_AFU (fpga_handle);
11
12 int main(int argc, char *argv[])
13 {
14     fpga_handle handle = NULL;
15     if (open_AFU (&handle) < 0)
16         return -1;
17
18     uint32_t data;
19     (void) fpgaWriteMMIO32 (handle, 0, POLY_REG, 221);   // set polynomial
20     (void) fpgaReadMMIO32 (handle, 0, POLY_REG, &data);  // set seed
21     printf ("Polynomial set to: %d\n", data);
22
23     (void) fpgaWriteMMIO32 (handle, 0, LFSR_REG, 0x1);
24     (void) fpgaReadMMIO32 (handle, 0, LFSR_REG, &data);
25     printf ("Seed set to: %d\n", data);
26
27     bool found[256] = { false };
28     bool stop = false;
29     int length = 0;
30     while (!stop) {
31         if (found[data]) stop = true;
32         else {
33             ++length;
34             found[data] = true;
35
36             // get a new random integer from the LFSR //
37             (void) fpgaWriteMMIO32 (handle, 0, CTRL_REG, 0x1);  // step
38             (void) fpgaWriteMMIO32 (handle, 0, CTRL_REG, 0x0);  // stop
39             (void) fpgaReadMMIO32 (handle, 0, LFSR_REG, &data);
40             printf ("LFSR: %d\n", data);
41         }
42     }
43     printf("\nLength of random sequence: %d\n", length);
44
45     close_AFU (handle);
46     return 0;
47 }
```

Figure 15: Using the AFU in a C program.

Lines 5 to 7 in Figure 15 define the addresses that software has to use to access the polynomial, LFSR, and control registers in the ALU. These addresses are the same as the ones given in Figure 10, except that they are shifted left by two bit positions. This bit-shifting is done to convert the double-word (32-bit aligned) addresses used in the Verilog code to byte addresses that are issued by the processor.

In Lines 9 and 10 prototypes are given for the functions *open_AFU* and *close_AFU*. The first of these functions sets up a communication mechanism between the software program and the AFU, via a Linux device driver. The second function terminates this connection. The *open_AFU* function calls several OPAE library utilities to check if the AFU is available and working properly. If so, the *handle* variable, declared with the OPAE type *fpga_handle* in Line 14, is set up as a *pointer* to the AFU. The *open_AFU* function uses this pointer to "print" to the Linux Terminal the contents of the mandatory register addresses in the AFU, which are specified in Figure 10. Appendix A shows the source code for *open_AFU*, in Figure 18. If it is able to communicate successfully with the AFU then the function returns 0, otherwise it returns -1. Figure 19 in Appendix A displays the code for the function *print_AFU_regs*, which is called by *open_AFU*, and the code for *close_AFU* is given in Figure 20.

The remainder of the code in Figure 15 uses memory-mapped I/O via the *handle* variable to access the registers in the LFSR AFU. The *fpgaReadMMIO32* function allows the software to read the contents of an AFU register, whereas *fpgaWriteMMIO32* allows a new value to be written to a register. The purpose of the software code is to use the *while* loop in Line 30 to cause the LFSR to reproduce the sequence of random integers that is illustrated in the simulation results in Figures 7 and 8. The while loop exits when the LFSR produces a value that is a duplicate of a previous one, marking the end of the sequence.

To complete this part of the exercise perform the following steps:

1. A file named *part3.c* that contains the code from Figure 15, and a file named *manage_afu.c*, which holds the C code for *open_AFU* and *close_AFU*, are included in the *design_files* that accompany this exercise. Copy these files into the `sw` folder for the AFU on the DevCloud.

2. To compile the code in *part3.c* and *manage_AFU.c* you have to use a special *Makefile* that employs the OPAE infrastructure on the DevCloud. Copy this *Makefile*, which is also included in the *design_files* for this exercise, into your `sw` folder.

3. In the `sw` directory on the DevCloud run the command `make part3`. The results are displayed in Figure 16. As shown in the figure, one of the programs executed by `make` is *afu_json_mgr*. This program reads the file *lfsr_afu.json* shown in Figure 13 to find the UUID for the AFU, and then produces the C header file *afu_json_info.h*. This header file is used by *manage_afu.c*. To execute your program type `./part3`, as illustrated in Figure 16.

Part IV

As shown in Figure 16 the program from Part III produces a sequence of 15 random values based on the polynomial $P = 221$ and starting with the seed value $S = 1$. In general, an LFSR may produce sequences of varying lengths depending on the polynomial and seed combination. For an n-bit LFSR a polynomial that results in a maximal sequence length, which is $2^n - 1$ different values, is known as a *primitive* polynomial. For this part of the exercise you are to write a C program that employs your LFSR AFU to find all eight-bit primitive polynomials. The smallest polynomial to consider is $P = (80)_{16} = 128$ and the largest is $P = (\text{FF})_{16} = 255$.

Perform the following:

1. Write your C code in a file *part4.c*, making use of the functions *open_AFU* and *close_AFU* as shown in Figure 15. For each polynomial of interest set the seed value to $S = 1$ and determine the resulting sequence length. Control the LFSR using step mode so that you can generate one random value at a time. You should display the value of each primitive polynomial on the Linux Terminal window. Also display the sequence of random values produced by each of these polynomials, starting and ending with the seed value $S = 1$.

2. To compile your code on the DevCloud, put *part4.c* into your `sw` folder and execute `make part4`.

```
userid@s005-n005: /lfsr_afu/sw$ ls
Makefile  manage_afu.c  manage_afu.o  part3.c
userid@s005-n005: /lfsr_afu/sw$ make part3
afu_json_mgr json-info --afu-json=../hw/rtl/lfsr_afu.json --c-hdr=afu_json_info.h
Writing afu_json_info.h
gcc -fstack-protector -fPIE -fPIC -O2 -D_FORTIFY_SOURCE=2 -Wformat
    -Wformat-security -Werror -g -O2 -std=c99 -Wall -Wno-unknown-pragmas -c -o
    part3.o part3.c
gcc -fstack-protector -fPIE -fPIC -O2 -D_FORTIFY_SOURCE=2 -Wformat
    -Wformat-security -Werror -g -O2 -std=c99 -Wall -Wno-unknown-pragmas -c -o
    manage_afu.o manage_afu.c
gcc -fstack-protector -fPIE -fPIC -O2 -D_FORTIFY_SOURCE=2 -Wformat
    -Wformat-security -Werror -g -O2 -std=c99 -Wall -Wno-unknown-pragmas -o part3
    part3.o manage_afu.o -z noexecstack -z relro -z now -pie -luuid -lpthread
    -lopae-c
userid@s005-n005: /lfsr_afu/sw$ ls
afu_json_info.h  Makefile  manage_afu.c  manage_afu.o  part3  part3.c  part3.o
userid@s005-n005: /lfsr_afu/sw$ ./part3
Opening lsfr_afu
AFU DFH REG  = 0x1000010000000000
AFU ID HI    = 0x8031be250cf440ee
AFU ID LO    = 0xa411dbaf7e894df5
AFU NEXT     = 0x0000000000000000
AFU RESERVED = 0x0000000000000000
Polynomial set to: 221
Seed set to: 1
LFSR: 221
LFSR: 179
LFSR: 132
LFSR: 66
LFSR: 33
LFSR: 205
LFSR: 187
LFSR: 128
LFSR: 64
LFSR: 32
LFSR: 16
LFSR: 8
LFSR: 4
LFSR: 2
LFSR: 1

Length of random sequence: 15
userid@s005-n005: /lfsr_afu/sw$
```

Figure 16: Compiling and executing *part3.c*.

Part V

In this part of the exercise you will utilize the LFSR AFU to help create an animation. Although the Linux Terminal normally displays ASCII text, we can use the Terminal's *escape commands* to make simple drawings, often called *ASCII graphics*. Animations can be created on the Terminal by using commands to clear the screen, move the Terminal cursor to specific locations, show/hide the cursor, change the color of characters, and so on. An example of a program that uses ASCII graphics is given in Figure 17. It first *includes* the *stdio.h* library and then defines constants, described later, for text colors which can be used in the Terminal window.

```
1  /* This program draws a few characters on the screen. */
2  #include <stdio.h>
3  #define YELLOW 33
4  #define CYAN 36
5  #define WHITE 37
6
7  void plot_pixel(int, int, char, char);
8
9  int main(void) {
10     int i;
11     printf ("\e[2J");            // clear the screen
12     printf ("\e[?25l");          // hide the cursor
13
14     plot_pixel (1, 1, CYAN, 'X');
15     plot_pixel (12, 12, CYAN, 'X');
16     for (i = 2; i < 12; ++i)
17         plot_pixel (6, i + 12, YELLOW, '*');
18
19     (void) getchar ();           // wait for user to press return
20     printf ("\e[2J");            // clear the screen
21     printf ("\e[%2dm", WHITE);   // reset foreground color
22     printf ("\e[%d;%dH", 1, 1);  // move cursor to upper left
23     printf ("\e[?25h");          // show the cursor
24     fflush (stdout);
25 }
26
27 void plot_pixel(int x, int y, char color, char c) {
28     printf ("\e[%2dm\e[%d;%dH%c", color, y, x, c);
29     fflush (stdout);
30 }
```

Figure 17: An example of code that uses ASCII graphics.

A command is sent to the Terminal in line 11 by using *printf*. All Terminal window commands begin with the ASCII `ESC` (escape) character, which is specified in the *printf* string using the syntax \e. The command in line 11, which is `[2J`, instructs the Terminal to clear the screen. Another command, `[?25l`, given in line 12, causes the Terminal to *hide* the cursor so that it is not visible to the user. Next, the function `plot_pixel` is called to draw some characters at specific locations on the screen. Coordinate $(1, 1)$ is at the top-left corner of the screen. The calls to `plot_pixel` in lines 14 to 17 draw cyan-colored `X` characters at coordinates $(1, 1)$ and $(12, 12)$, and a vertical yellow line of ten $\star$ characters along the sixth column.

The `plot_pixel` function, shown in lines 27 to 30 uses two commands to draw a character. The first command is `[ccm`, where cc is called an *attribute*. The attribute can be used to set the color of text characters, by using different values of cc. Examples of color attributes are $cc = 31$ (red), 32 (green), 33 (yellow), 34 (blue), 35 (magenta), 36 (cyan), and 37 (white). The second command in `plot_pixel` is `[yy;xxH`, where yy and xx specify a row and column on the screen, respectively. This command moves the Terminal cursor to coordinate (xx, yy). In line 19 the program waits, using the `getchar` function, for the user to press a key. Finally, commands are sent to the Terminal to *clear* the screen, *set* the color to white, *set* the cursor to coordinates $(1, 1)$, and *show* the cursor.

The use of ASCII graphics described above became popular around the year 1980 when they were available in computer video terminals called the *VT100*, manufactured by Digital Equipment Corporation. The Linux Terminal provides ASCII graphics by *emulating* the capabilites of the VT100 video terminal. A listing of some escape commands is given in Appendix B. More information about ASCII graphics commands can be found by searching on the Internet for a topic such as "VT100 graphics".

Perform the following:

1. Write C code using ASCII graphics to create an animation that "prints" random letters of the alphabet at random screen coordinates. The LFSR AFU should be used in continuous mode to generate random integers over a 32-bit range. You can set the polynomial to $P = (b4bcd35c)_{16} = 3032273756$, which is a 32-bit primitive polynomial. Your code should use random values from the LFSR to choose the letter to display, its color, and the (x, y) screen coordinates. As an example, if the LFSR output is called *D*, then to randomly choose a letter c from one of the five letters in the string "intel", where 0 selects 'i', 1 select 'n', and so on, you can use the expression $c = D \% 5$. The range of x and y coordinates produced by your program should cover the whole Terminal window. Save your code in a file called *part5.c*.

2. On the DevCloud compile your program by executing `make part5`. When you execute the program by typing `./part5` the effect should be that every location in the Terminal window gets modified almost continuously. This result illustrates that the sequence of integers produced by the LFSR are sufficiently random for our purposes. To see an example of a properly-working solution you can watch a *YouTube* video at `https://youtu.be/JSbhbNVzNAY`. In this video the animation selects a random letter from the string "intel" and prints it at a random location on the screen, using a random color. The program is terminated by the user pressing ^C. The video also shows a second animation in which ASCII graphics are used to draw "lines" on the screen from one randomly-selected coordinate to another. Lines that are mostly "horizontal" are drawn in blue, "vertical" lines in green, and diagonal lines in yellow or red. This program first draws a few lines slowly so that the user can see what is happening, and then draws them as quickly as possible until the program is terminated.

Appendix A

```
#include <stdio.h>
#include <uuid/uuid.h>
#include <opae/enum.h>
#include <opae/access.h>
#include <opae/mmio.h>
#include <opae/properties.h>
#include <opae/utils.h>
#include "afu_json_info.h"

// mandatory AFU register addresses (offsets)
#define AFU_DFH_REG   0x0
#define AFU_ID_LO     0x8
#define AFU_ID_HI     0x10
#define AFU_NEXT      0x18
#define AFU_RESERVED  0x20

int print_AFU_regs (fpga_handle);
fpga_properties filter = NULL;
fpga_token token = NULL;

int open_AFU (fpga_handle *handle) {
    fpga_guid guid;
    uint32_t num_matches;
    fpga_result res = FPGA_OK;

    char *AFU_NAME = AFU_ACCEL_NAME;     // from json file
    char *UUID = AFU_ACCEL_UUID;         // from json file
    printf ("Opening %s\n", AFU_NAME);
    if (uuid_parse(AFU_ACCEL_UUID, guid) < 0) {
        fprintf(stderr, "Error parsing guid '%s'\n", UUID);
        return -1;
    }
    if ((res = fpgaGetProperties (NULL, &filter)) != FPGA_OK) { // Look for AFU
        fprintf(stderr, "Error creating properties object: %s\n", fpgaErrStr
            (res));
        return -1;
    }
    if ((res = fpgaPropertiesSetObjectType (filter,FPGA_ACCELERATOR)) != FPGA_OK) {
        fprintf(stderr, "Error setting object type: %s\n", fpgaErrStr (res));
        (void) fpgaDestroyProperties (&filter);
        return -1;
    }
    if ((res = fpgaPropertiesSetGUID (filter, guid)) != FPGA_OK) {
        fprintf(stderr, "Error setting GUID: %s\n", fpgaErrStr (res));
        (void) fpgaDestroyProperties (&filter);
        return -1;
    }
    if ((res = fpgaEnumerate (&filter, 1, &token, 1, &num_matches)) != FPGA_OK) {
        fprintf(stderr, "Error enumerating AFUs: %s\n", fpgaErrStr (res));
        (void) fpgaDestroyProperties (&filter);
        return -1;
    }
    if (num_matches < 1) {
        fprintf(stderr, "Error: AFU not found!\n");
        (void) fpgaDestroyProperties (&filter);
        return -1;
    }
```

Figure 18: The *open_AFU* function (Part a).

```
    /* Open AFU and map MMIO */
    if ((res = fpgaOpen (token, handle, 0)) != FPGA_OK) {
        fprintf(stderr, "Error opening AFU: %s\n", fpgaErrStr (res));
        (void) fpgaDestroyToken (&token);
        (void) fpgaDestroyProperties (&filter);
        return -1;
    }
    if ((res = fpgaMapMMIO (*handle, 0, NULL)) != FPGA_OK) {
        fprintf(stderr, "Error mapping MMIO space: %s\n", fpgaErrStr (res));
        (void) fpgaClose (*handle);
        (void) fpgaDestroyToken (&token);
        (void) fpgaDestroyProperties (&filter);
        return -1;
    }
    /* Reset AFU */
    if ((res = fpgaReset (*handle)) != FPGA_OK) {
        fprintf(stderr, "Error resetting AFU: %s\n", fpgaErrStr (res));
        (void) fpgaUnmapMMIO (*handle, 0);
        (void) fpgaClose (*handle);
        (void) fpgaDestroyToken (&token);
        (void) fpgaDestroyProperties (&filter);
        return -1;
    }
    return print_AFU_regs (*handle);
}
```

Figure 18. The *open_AFU* function (Part *b*).

```
// Displays the contents of mandatory AFU registers
int print_AFU_regs (fpga_handle handle) {
    uint64_t data = 0;
    bool fail = false;
    fpga_result res = FPGA_OK;

    if ((res = fpgaReadMMIO64 (handle, 0, AFU_DFH_REG, &data)) != FPGA_OK) {
        fprintf (stderr, "Error reading from MMIO: %s\n", fpgaErrStr (res));
        fail = true;
    }
    else
        printf("AFU DFH REG  = 0x%016lx\n", data);

    if ((res = fpgaReadMMIO64 (handle, 0, AFU_ID_HI, &data)) != FPGA_OK) {
        fprintf (stderr, "Error reading from MMIO: %s\n", fpgaErrStr (res));
        fail = true;
    }
    else
        printf("AFU ID HI    = 0x%016lx\n", data);
    if ((res = fpgaReadMMIO64 (handle, 0, AFU_ID_LO, &data)) != FPGA_OK) {
        fprintf (stderr, "Error reading from MMIO: %s\n", fpgaErrStr (res));
        fail = true;
    }
    else
        printf("AFU ID LO    = 0x%016lx\n", data);
```

Figure 19: The *print_AFU_regs* function (Part *a*).

```
    if ((res = fpgaReadMMIO64 (handle, 0, AFU_NEXT, &data)) != FPGA_OK) {
        fprintf (stderr, "Error reading from MMIO: %s\n", fpgaErrStr (res));
        fail = true;
    }
    else
        printf("AFU NEXT     = 0x%016lx\n", data);

    if ((res = fpgaReadMMIO64 (handle, 0, AFU_RESERVED, &data)) != FPGA_OK) {
        fprintf (stderr, "Error reading from MMIO: %s\n", fpgaErrStr (res));
        fail = true;
    }
    else
        printf("AFU RESERVED = 0x%016lx\n", data);

    if (fail) return -1;
    else return 0;
}
```

Figure 19. The *print_AFU_regs* function (Part *b*).

```
void close_AFU (fpga_handle handle) {
    /* Unmap MMIO space */
    (void) fpgaUnmapMMIO (handle, 0);
    /* Release accelerator */
    (void) fpgaClose (handle);
    /* Destroy token */
    (void) fpgaDestroyToken (&token);
    /* Destroy properties object */
    (void) fpgaDestroyProperties (&filter);
}
```

Figure 20: The *close_AFU* function.

Appendix B

Command	Result
\e7	save cursor position and attributes
\e8	restore cursor position and attributes
\e[H	move the cursor to the home position
\e[?25l	hide the cursor
\e[?25h	show the cursor
\e[2J	clear window
\e[ccm	set foreground color to cc[1]
\e[yy;xxH	set cursor location to row yy, column xx

ASCII escape commands.

[1]

[1] Terminal window colors: $cc = 31$ (red), 32 (green), 33 (yellow), 34 (blue), 35 (magenta), 36 (cyan), and 37 (white)