Project 1: Register-Transfer Level (RTL) Design

(A Simple Datapath and Control Unit)

Instructor: Yifeng Zhu*

Full Score: 100

Due: September 28 (Monday)

Submit your report and project files through http://www.eece.maine.edu/hw/

Objectives

1. Learn the basic concepts involved in RTL design
2. Become familiar with some commonly used RTL components
3. Reinforce the skills in using of the graphical (schematic-capture) design tool to wire up a relatively complex design
4. Reinforce the skills of Verilog or VHDL programming

1. Introduction

Register transfer level (RTL) design is the design of a digital electronic circuit in terms of data flow between registers, which store information between clock cycles in a digital circuit. It is widely used to implement algorithms at the hardware level. The primitives or components in RTL include **registers**, **data-steering logic** (multiplexers and demultiplexers), **buses**, and **functional blocks**. A typical statement consists of a *register transfer* whereby the contents of one or more source registers are combined through a functional operator and the result is copied into a destination register. Once the registers are associated with variables, the semantics of such an RTL statement is identical to that of an assignment statement in a programming language. Indeed, RTL algorithms resemble programming languages in many respects and this fact accounts for their popular use. They also differ from sequential programming languages in one essential way. In hardware, multiple operations can take place simultaneously by spatially replicating hardware components, therefore RTL languages allow for concurrent execution of multiple RTL statements.

The RTL is very suitable for describing processor designs like MIPS. It is equally good for expressing designs of special-purpose processors that implement specific algorithms (e.g. for data compression or data encryption). In this project, you will design a special purpose processor that generates a number sequence, called *Ulam sequence*, for a given initial input.

In an **Ulam sequence** $\{u_i\}$ for any positive non-zero integer u_0 as the initial value, the element u_{i+1} of the sequence is computed from its previous element u_i, recursively, according to the following rules:

1. If u_i is 1, then stop
2. If u_i is even, then $u_{i+1} = u_i / 2$
3. If u_i is odd, then $u_{i+1} = 3 \cdot u_i + 1$
4. Continue with this process until reaching 1.

The algorithm for generating the Ulam sequence is shown in Figure 1.

Mathematician Stanislav Ulam proposed that any positive integer would reduce to 1 if the above rules were repeated a sufficient number of times. For example, if the original number were 13, the algorithm would generate the following sequence of numbers: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. This Ulam sequence has 10 numbers including the original number and the 1. *In this project, to*

* The project borrows parts from CSE230 by Dr. Sharad Seth at UNL.

avoid having to deal with large data widths, we will assume that all the arithmetic operations in the above rules are carried out in module 2^n, where n is the word width of unsigned numbers.

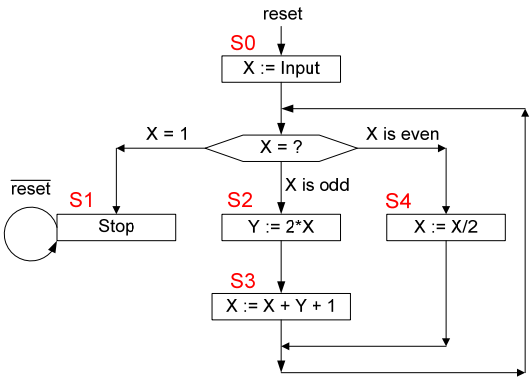


Figure 1. The flowchart of our Ulam sequence processor.

In this project, you will build a special-purpose processor to generate the ULam sequence. This processor takes an initial integer value as input and then computes the successive numbers in the sequence until the stopping condition is reached. The design of the processor can be logically divided into two units: ***datapath*** and ***control***, same as MIPS and other commercial processors, but on a much smaller scale. Thus, this project serves as a basic practice for the forthcoming team design project on designing a general-purpose processor.

To ease your design task, a pre-designed datapath will be provided to you. Your main task then is to design an appropriate control unit, integrate it with the given datapath, and verify that the overall design is functional by running tests on it (in simulation and on the university board).

2. Datapath Design

The datapath for this special-purpose processor consists of an ALU and two registers, X and Y, as shown in Figure 1. Under the control of two function-select lines <FS1, FS0>, it implements four operations encoded as follows:

- If <FS1, FS0> = 00, then Z = X;
- If <FS1, FS0> = 01, then Z = X+Y+1;
- If <FS1, FS0> = 10, then Z = X/2;
- If <FS1, FS0> = 11, then Z = 2*X.

In the datapath shown in Figure 2, X and Y are two 8bit *active-low* registers (Active low of an electronics signal is the signal value having a valid effect when its voltage is low.). You can use the registers 74377 or 74377b. (Octal DFF). You can also use 2x8MUX from that library, which is an 8-bit, 2-to-1 multiplexer. When the SEL input is set to 0, the input[7..0] is sent to the output. When the SEL input is set to 1, the Z[7..0] is sent to the output. WEN is the signal of write enable. Register Y is always enabled so that the write operations are always allowed. However, you need to use WEN to control the write operations in Register X. The *One Detector* is an 8bit NOR gate. If X is 00000001, the output of *One detector* is true; Otherwise, the output is false. ALU performs four different arithmetic operations under the control of the function-selection signals FS1 and FS0.

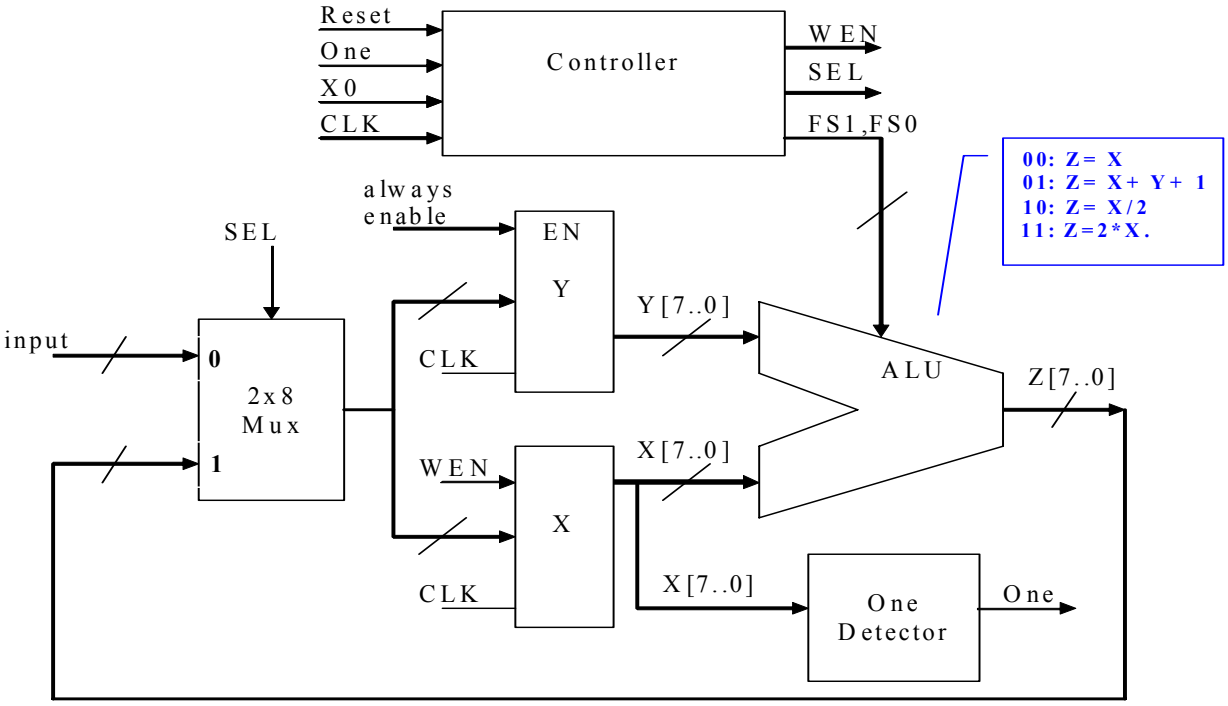


Figure 2. The datapath of our Ulam sequence processor

The running process can be briefly described as follows.

- If X is even, it's straightforward. Only one cycle is needed to finish the computation job of X/2, and the results are saved back to the register X[7..0].
- If X is odd and not equal to 1, it takes two cycles to calculate 3*X + 1 (first cycle Y=2*X, second cycle X+Y+1), and we don't want to put the result of first cycle back to Y[7..0] (otherwise, the results would be 2*X + 2*X + 1). Therefore we need to use WEN to control write operations on the X register. Note how WEN works to avoid writing the result of Z[7..0] back to X[7..0] after the first cycle.

The ALU and One Detector, shown in the Figure 2, will be provided. In this project, you will design the controller, integrate them together and run tests to verify your design.

3. Design the Control Unit as a FSM

Sequential logic circuits usually can be implemented in a fixed number of states, which can be modeled as a *Finite State Machine* (FSM). A FSM consists of states, transitions, and actions. A *state* stores information about the past and it reflects the input changes from the system start to the present moment. A *transition* indicates a state change and is described by a condition that would need to be fulfilled to enable the transition. An *action* is a description of an activity that is to be performed at a given moment.

The generic steps of using FSM to design logic circuits are as follows: 1) Understand the problem; 2) Draw state transition diagram; 3) Perform state minimization; 4) Encode states and build state transition table; 5) Build the truth table. 6) Simplify the logic equations and wire the circuit together; 7) Implement the circuits and verify the design.

The following provides some basic analysis of this special-purpose processor.

3.1 Draw the state transition diagram

There are five possible states in the controller, as shown in the flowchart (Figure 1). Figure 3 presents the state-transition graph that indicates how the states change in response to actions performed.

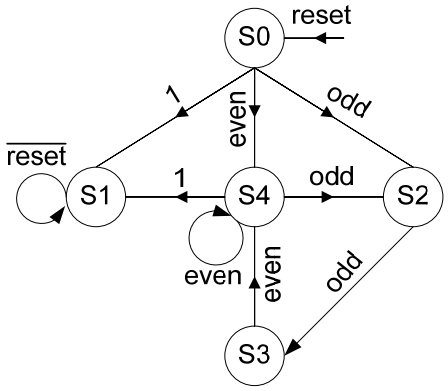


Figure 3. The state transition graph represented through Finite State Machine (FSM) Note that in State S3, X must be an even integer and thus the machine will never transit from State S3 to State S2 or State S1.

3.2 Encode the states

The interactions between the datapath and the FSM controller are shown in Figure 4. The control unit takes the statuses of X as inputs. Driven by the clock signal, it outputs WEN, SEL, FS1 and FS0 to synchronously control the datapath. In addition, the FSM control unit is able to remember its current state and correctly change its current state according to the state transition graph shown in Figure 3. Since there are a total of five states, at least three bits are required to encode the states. You could use D flip-flops to store the encodings of states. The D stands for "data"; these flip-flops can be thought of as a basic memory cell.

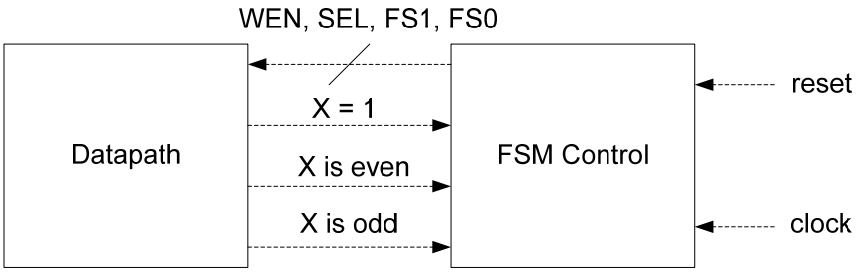


Figure 4. The schematic diagram of the processor.

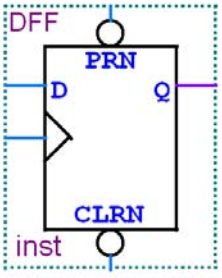


Figure 5. D flip flop provided in Quartus II

Figure 5 shows the D flip-flop in Quartus II. The ">" input is the clock input. The flip-flop triggers on the rising clock edge. The PRN input is used to "preset" the flip-flop. If PRN = 0, the output Q is set to 1 on the following rising clock edge. The CLRN is used to "clear" the flip-flop. If CLRN = 0, the output Q is set to 0 on the following rising clock edge. The following shows the truth table of a D flip-flop.

Inputs			Next State
PRN	CLRN	D	Q+
0	1	X	1
1	0	X	0
1	1	0	0
1	1	1	1

3.3 Build the state transition table

Table 1 shows a concise state transition table, where the inputs composed of the current FSM state, reset, X, and X0, and the outputs include the next states and the control signals <WEN, SEL, FS1, FS0>. For example, for the table entry marked below, it means that if the current state is S0 and the reset and X are 0 and 1, respectively, then the next state is S1, and WEN, SEL, FS1, FS0 are -, 1, 0, 0, respectively. ("-" means "*don't care.*")

Current state	reset=0 and X=1	reset=0 and X!=1 and X0=1 (odd)	reset=0 and X!=1 and X0=0 (even)	reset=1
S0	**S1 / -100**	S2 / 1111	S4 / 0110	S0 / 00--
S1	S1 / -100	- / ----	- / ----	S0 / 00--
S2	- / ----	**S3 / 0101**	- / ----	S0 / 00--
S3	- / ----	- / ----	S4 / 0110	S0 / 00--
S4	S1 / -100	S2 / 1111	S4 / 0110	S0 / 00--

Table 1. State transition table. The outputs are "state / <WEN, SEL, FS1, FS0>".

Note that in State S2 (Y = 2*X), Y is updated. But X is not because WEN = 0. Thus the X0 bit is still one.

Please verify the state transition table and make sure that you fully understand it. Depending on your design, your state transition table might be different with the one given above. In this design, we make the following assumptions.

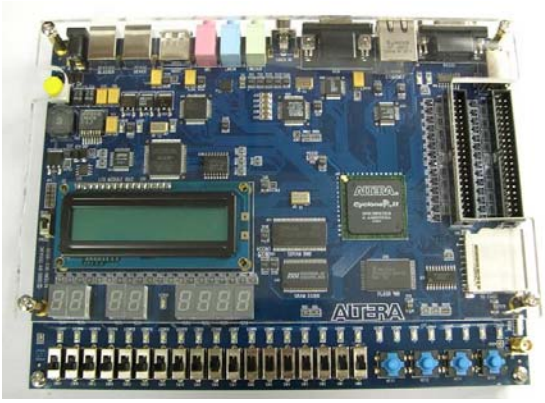
1. When WEN = 0, the write operations are allowed. Otherwise, write operations cannot be performed.
2. When SEL = 1, the output of MUX is Z[7..0]. Otherwise, the output of MUX is the input integer.

Then with this transition table, you could build a compact truth table, where the inputs are the current state, reset, X, and X0, and the outputs are the control signals <WEN, SEL, FS1, FS0> as well as next states.

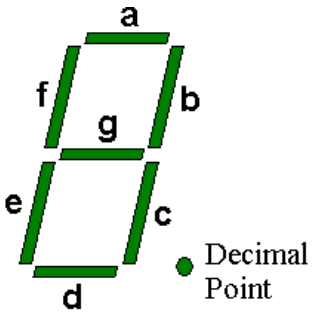
We have provided some basic information and you are required to complete the design of this special-purpose processor. In the Appendix, please find a short tutorial to a tool for Boolean function simplification, called *espresso*.

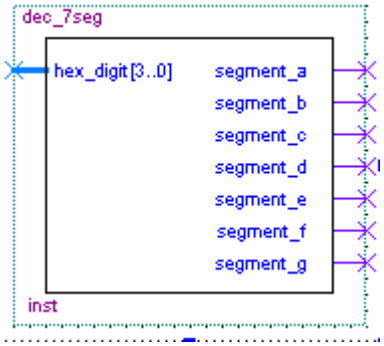
Part 2: Show your Results on LCD and LED

Download your code into the DE2 board and dynamically show the Ulam sequence numbers on the LCD screen. You might need to follow some tutorials of DE2 on our project website or altera.com. You can find LCD information (HITACHI HD44780 Dot Matrix LCD) and the DE2 PIN assignment documents on the course project website.

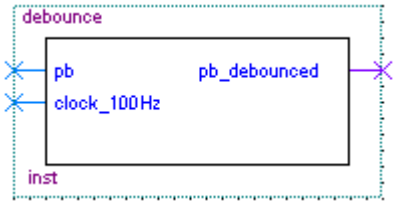


Each digit should stay on the LCD for a sufficient amount of time so that we can read it. For example, you could set the clock rate to 0.5Hz. Then every 2 seconds, an Ulam number is shown on the LCD. The numbers should be dynamically shown on the screen, i.e., you cannot just show all numbers together at the first step. Similarly your results should also be shown on the LED simultaneously. A sample Clock, LED LCD driver will be provided. The following shows some diagram of some drivers.

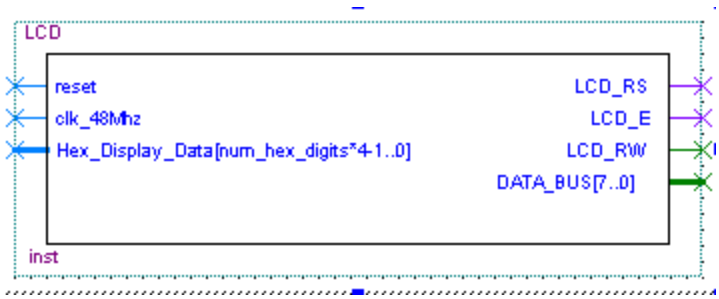




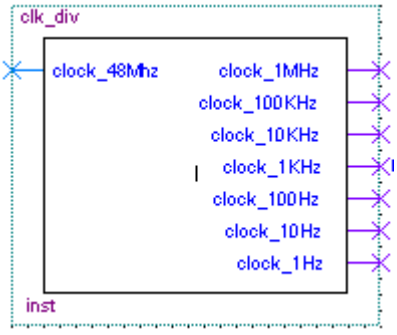
LED Controller



Push bottom Debounce



LCD controller



Clock Divider

The LCD_Display is used to display static ASCII characters and changing hex values from hardware on 2 line LCD display panel. Number_Hex_Digits is used to set the size of the Hex_Display_Data input. Each hex digit displayed requires a 4-bit signal. The reset signal should be active high whenever you want update the LCD display. **Make sure that the pins LCD_ON and LCD_BLON are set as active high.**

These two pins are not set by the controller. You can find more information about the LCD controller in the course project website. LCD_E is actually LCD_EN.

The debounce circuit is used to filter mechanical contact bounce in the DE2's push buttons. A shift register is used to filter out the switch contact bounce. The shift register takes several time spaced samples of the push button input and changes the output only after several sequential samples are the same value. Clock is a clock signal of approximately 100Hz that is used for the internal *50ms* switch debounce filter design. The pb_debounced is the output. The output will remain Low until the pushbutton is released.

Appendix: A Short Tutorial to *Espresso*

Espresso is a package for constructing or simplifying Boolean Functions from binary Truth Tables. It comes as a part of the SIS package and is available as open source at UC Berkeley CAD page. The following gives two examples to show how to use espresso. You can download a copy of espresso from our course web site.

1. Example One

Input file named ex1.txt:

```
# example input file named ex1.txt
.i 4
.o 1
.ilb A B C D
.ob F
0 0 0 0   0
0 0 0 1   0
0 0 1 0   0
0 0 1 1   0
0 1 0 0   0
0 1 0 1   0
0 1 1 0   1
0 1 1 1   1
1 0 0 0   1
1 0 0 1   1
1 0 1 0   0
1 0 1 1   0
1 1 0 0   0
1 1 0 1   0
1 1 1 0   0
1 1 1 1   0
.e
```

COMMENTS ABOUT THE INPUT FILE

- Comments are allowed within the input by placing a pound sign (#)
- .i 4 (Input variables = 4)
- .o 1 (Output variables = 1)
- .ilb A B C D (Input variable names)
- .ob F (Output variable name)
- 0 0 0 0 0 (Truth table row, 4 inputs and 1 output)
- .e (Signifies the end of the input file.)

Commands to run: **espresso ex1.txt > ex1.out.txt**

The outputs are saved into a file named *ex1.out.txt.*

Outputs

```
# example file
1
.i 4
.o 1
.ilb A B C D
.ob F
.p 2
100- 1
011- 1
.e
```

Interpretation:
.p 2 (Indicates that there are two terms in the output expression)
100- 1 (This term is $A \cdot \overline{B} \cdot \overline{C}$)
011- 1 (This term is $\overline{A} \cdot B \cdot C$)

Thus logic expression is $F = A \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot C$. In the output lines, 1 is the variable, 0 is the inverse, and - means the variable is "Don't care".

2. Example Two
Specifying the truth table entries only where the function is 1 is sufficient to define the entire truth table. The following is another example named *ex2.txt*. This is a multiple-output problem and shows that the input can be in the form of cubes.

NOTE:
- A don't care at the output is specified as a "-".
- 1 in an output column indicates that cube is included in the cover of the function.
- 0 indicates the cube is not included in the cover of that function, but it does not mean the function is necessarily 0 at the minterms covered by the cube.

Input

```
# example 2

.i 3
.o 3
.ilb A B C
.ob F G H
000 100
1-- 101
1-1 0-1
010 010
.e
```

Output

```
# example 2
.i 3
.o 3
.ilb A B C
.ob F G H
.p 3
010 010
-00 100
1-- 101
.e
```

Thus, we have

$$F = \overline{B} \cdot \overline{C} + A$$
$$G = \overline{A} \cdot B \cdot \overline{C}$$
$$H = A$$

Detailed information of the algorithm used in Expresso can be found in the following paper.
Brayton, R. et. al., "*A Comparison of Logic Minimization Strategies using EXPRESSO: An APL Program Package for Partitioned Logic Minimization*," Proceedings of the IEEE International Conference on Circuits and Computers, 1982.

You can download the software from the project website.

Report of Project 1

Design of a Special-purpose Processor
September 28 (Monday)
100 points

Name: _____ Last 4 digits of SSN: _____

Part 1: ULam Processor Design (85 points)

1. Complete the following compact truth table

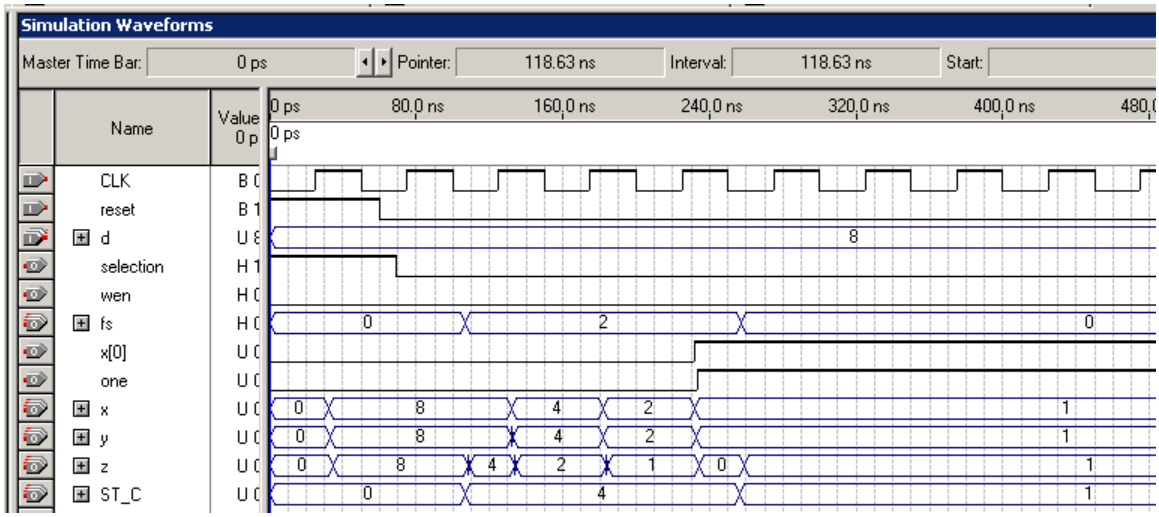
		Inputs				Outputs				
		Current state	reset	ONE	X0	Next state	WEN	Selection	FS1	FS0
1	Reset	-	1	-	-					
2	S0	000	0	0	0					
3		000	0	0	1					
4		000	0	1	0					
5		000	0	1	1					
6	S1	001	0	0	0					
7		001	0	0	1					
8		001	0	1	0					
9		001	0	1	1					
10	S2	010	0	0	0					
11		010	0	0	1					
12		010	0	1	0					
13		010	0	1	1					
14	S3	011	0	0	0					
15		011	0	0	1					
16		011	0	1	0					
17		011	0	1	1					
18	S4	100	0	0	0					
19		100	0	0	1					
20		100	0	1	0					
21		100	0	1	1					
22		1-1	-	-	-					
23		11-	-	-	-					

2. **Use espresso to simplify the logic. Please write your input file and output file below.**

3. Design your controller. Please show the schematic figure of your controller.

4. Re-implement the controller by using VHDL or Verilog.

5. **Test the design with at least the following inputs: 7, 32, 13, and 1, and give your simulation results. Please copy your screen outputs and paste them here, as shown in the following example.**



An example simulation result when the input integer is 8. (d is the input integer, ST_C is the current state in the controller, and the others are shown in Figure 2.)

Part 2: Implementation on DE2 boards (Totally 15 points)
Please demo your design on DE2 board (10 points for LCD, 5 points for 7-segment Display).