ECSE 323 - Digital System Design
After Action Report - Plentiful Discoveries and the Rules-Dealer Paradigm

Harley Wiltzer (260690006)
Spiros-Daniel Mavroidakos (260689391)

March 30, 2017

Contents

Part I

A Pleasant Preamble

Sometimes in life one may be presented with situations that make him rethink his beliefs and question who, in fact, he really is. It is moments like these that actually define an individual *ad postremum*. Of course, it is up to the individual in question to *realize* that he shall be withdrawing cathexis from the myriad objects of empirical reality around him if enlightenment should be obtained. People that occasionally experience this enlightenment are called patricians. Those who continuously experience this enlightenment are called Engineers.

Consider the infamous game of Blackjack. Some may enjoy this game as a nice way to pass the time, others may be violently obsessed with it. Regardless of who is playing, the game can only be played reliably when players understand and follow the rules and when a credible dealer is present. Naturally, the players are needed for the game to be played, and of course the game is only really *being played* when the rules are followed. The presence of the dealer, however, is far more interesting than what the uninformed reader may believe.

Some say the dealer's job is to deal the cards, but that is a vast and decadent oversimplification. Sure, the dealer *should* deal the cards (at the appropriate times, that is), but the dealer is also responsible for establishing structure and ensuring that the game does not get out of hand. In fact, the rules of the game themselves have their strings pulled by the dealer.
But some dealers do not follow the rules.

It is no longer news that the goal of these laboratory sessions is to build a *crazy eights* game, and not a Blackjack game. Although the rules of the two games are different, both take on the rules-dealer paradigm of gameplay. The layman, and even the patrician, may focus on the rules of the game principally. However, as Engineers, the main focus of this laboratory was to create not just a functional dealer, but a reliable, ethical, and ultimately compliant dealer. As shown later in this report, this was successfully accomplished due to the design of a clever *finite state machine*, a construct that has also been used in the design of underwhelming robots.

In the interest of moral behavior, it would be unethical to display the accomplishments of this laboratory session without giving due credit to the Altera Quartus II and Modelsim software, whose magical functionalities allow such complex designs to be mapped onto an FPGA. Given Altera's great text editing accomodations and incredible timing simulation tools, the development of this system was dream-like.

At this stage, the reader is invited to explore the remainder of this report, which will go through in a (hopefully) simple and organized manner the discoveries made during this laboratory session. Beyond that, this report will discuss *how* these discoveries were made, and how they were reinforced, so the reader may gain intuition on developing state of the art digital systems.

Please enjoy the discoveries and details that follow, and try to learn something from them. Much is to be gained by grasping the concepts of the rules-dealer paradigm, as they apply to more in life than simply digital systems and card games. To conclude this pleasant preamble, the reader is encouraged to, above all, *have fun* with this report, and better yet, to have fun with life. Finally, it is important to remember that while by law a man is guilty for violating the rules, in ethics a man is guilty merely by *considering* such violation. Be careful, be wise, and Baba Booey to all.

Part II

The Rules Circuit

Introduction and Rationale

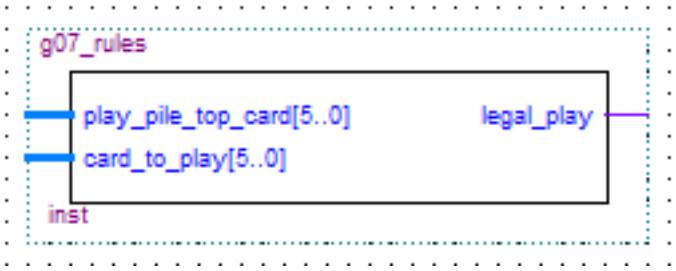
It is important in any game, not just crazy eights, that the rules of the game are *well-defined*. It would only do this game justice to ensure that the crazy eights rules are well defined in circuitry, and that is what will be demostrated in this part of the report.

The circuit itself is relatively simple but nothing short of genius. The circuit was written in VHDL and works as follows. Firstly, it has two input ports, `play_pile_top_card[5..0]` and `card_to_play[5..0]`, corresponding to the card on the top of the pile and the card that the player wishes to use respectively. The output of the circuit named `legal_play` is simply a single bit which is given a value of '1' when the move is in accordance with the rules of this great game. The modulo 13 circuit that was created in the prior lab was used to get the value and the suit of both cards (what foresight!). Once the nature of the cards is known, it is easy to apply the appropriate rules. A process block in the VHDL code checks to see if either of the cards are of a value of eight because if either of them are eight, then the play is always legal by definition. Furthermore, the process block is intelligent enough to also test the remaining two rules of the game as well. If the two cards have the same value or the same suit, the process block will graciously change the bit value of `legal_play` to '1' since it will recognize that the player has not tried to trick the other players.

Though the explanation above is nothing short of thorough, a complete view of the `g07_rules` VHDL design may be seen in the VHDL code below.

Circuit Description

Figure 1: Pin-out diagram of the g07_rules circuit



Above is a pin-out diagram of the g07_rules circuit. A more detailed description of these ports will be given below.

Description of ports

play_pile_top_card[5..0]

The `play_pile_top_card[5..0]` is a 6 input bit vector that represents the card at the top of the play pile.

card_to_play[5..0]

The `card_to_play[5.0]` is a 6 input bit vector that corresponds to the users card that they wish to play.

legal_play

The `legal_play` output bit is active when the g07_rules circuit determines that the input corresponds to a legal play and not active otherwise.

VHDL charicature of the `g07_rules` circuit

The VHDL code shown below was inspired by the tactics described in the rationale section above.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
```

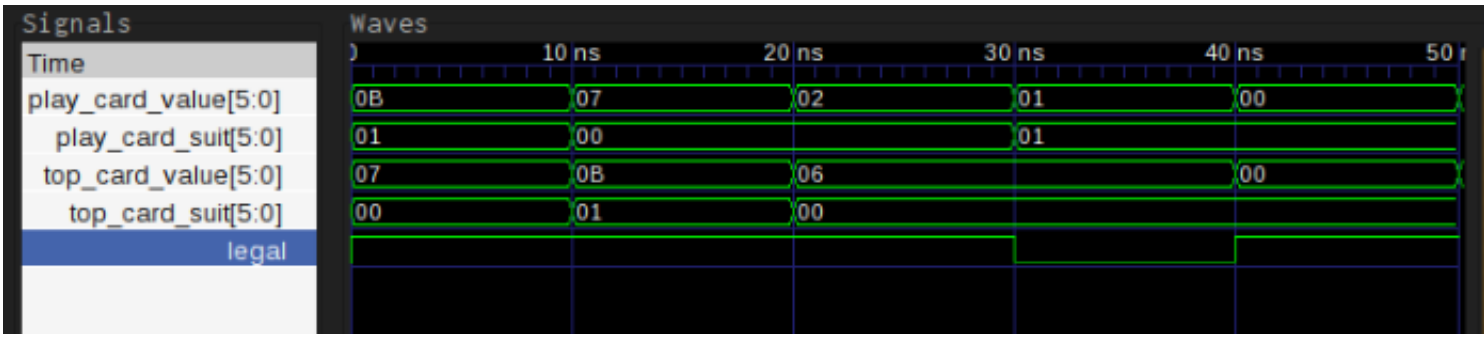
```
4  entity g07_rules is
5          port ( play_pile_top_card: in std_logic_vector(5 downto 0);
6                          card_to_play: in std_logic_vector(5 downto 0);
7                          legal_play: out std_logic);
8  end g07_rules;
9
10 architecture legal of g07_rules is
11         component g07_mod13
12                 port ( x: in std_logic_vector(5 downto 0);
13                                 modulo: out std_logic_vector(5 downto 0);
14                                 floor: out std_logic_vector(5 downto 0) );
15         end component;
16
17         signal pile_value, pile_suit, card_value, card_suit: std_logic_vector(5
               downto 0);
18 begin
19         m1: g07_mod13 port map(x => play_pile_top_card, floor => pile_suit, modulo
               => pile_value);
20         m2: g07_mod13 port map(x => card_to_play, floor => card_suit, modulo =>
               card_value);
21
22         arbitration: process (pile_value,pile_suit,card_value,card_suit)
23         begin
24                 if (pile_value = "000111") then legal_play <= '1';
25                 elsif (card_value = "000111") then legal_play <= '1';
26                 elsif (pile_value = card_value) then legal_play <= '1';
27                 elsif (pile_suit = card_suit) then legal_play <= '1';
28                 else legal_play <= '0';
29                 end if;
30         end process arbitration;
31 end legal;
```

With Regards to the Testing of the Rules Circuit

The rules of the game are very simple, since crazy eights is a game that is played by people of all ages, and people with dementia. Here are the rules just in case the reader forgot: A card of face value of 8 in any suit can be played on any card and can have any card played on it. Furthermore, any card of a particular suit can be played on a card of that same suit, regardless of the cards' values. Finally, a card with a given face value can be played on another card with the same face value, regardless of the suits.

For any testing process to be considered legitimate, the testers must implement the appropriate tests. Therefore, the test that were generated for this tremendous circuit were created to make sure that every rule was tested. It was all done in the spirit of good testing. The test results may be perused below. The returning reader may expect to find the simulation results from a Modelsim simulation wave, however, the Modelsim simulator is simply *too powerful* for such a simple simulation. It would be inefficient to harness so much power for a circuit that doesn't require it. Some people are less fortunate and can make use of the excess energy. So, the simulation was done using the GTKWave VCD viewer, and is more than satisfactory for showing simple waves.

Figure 2: Timing simulation of the rules circuit



The first clock cycle is used to show that any card can be played on top of an eight, since the eight is crazy after all. Note that the design of the test had card values starting at 0 rather than 1, so the value of 7 for the card values actually corresponds to a card value of 8. The second clock cycle was also used to show how crazy the eights can be. It displayed the rule that an eight of any suit can be played over any card and due to the design of the circuit, the test was passed. The third clock cycle was designed to make sure that a given card of a certain suit can be played on any card of the same suit and it was an engineering success. The penultimate clock cycle showed

that a card of a given value and suit cannot be played on another with a different value and suit, and the circuit performed like a dream. The final test gave insight into the behavior when a card with a given face value would be played on a card with the same face value and as stated in the rules, it was legal and did not require any additional action to fix this violation.

The Computer Analysis of the Rules Circuit

The Flow Summary

Quartus II's Flow Summary reveals that the `g07_rules` circuit uses only 46 total combinational functions, which is quite impressive. There is a total of 18752 combinational logic functions available in the Cyclone II DE1, thus making the total usage under 1%. This is a tremendous achievement because so many logic functions are left over to be used in the final design of the great crazy eights card game. The `g07_rules` circuit also uses zero memory bits which is a great benefit overall.

The Timing Analysis

By means of the mighty Quartus II TimeQuest Timing Analyzer, propagation delays for the two inputs to the rules circuit were found. It was seen that the maximum propagation delay of the `card_to_play` input was 19.363 nanoseconds, and the maximum propagation delay of the `play_pile_top_card` input was 19.350 nanoseconds. Therefore, the maximum propagation delay from the input to the output is 19.363 nanoseconds. Given this propagation delay, inputs of the system should not change more than once within a 19.363 nanosecond interval. This is reasonable so long as the inputs of the system do not change more than once in any given interval of 19.363 nanoseconds, which is a very short amount of time.

Part III

The Dealer Circuit

Introduction and Rationale

As foreshadowed in the pleasant preamble above, having a loyal, dependable dealer will be crucial to the reliable functionality of the crazy eights system. Without such a dealer, the *rules* of the game *may not* be followed, causing complete and utter pandemonium, ultimately resulting in a rather atrocious crazy eights game.

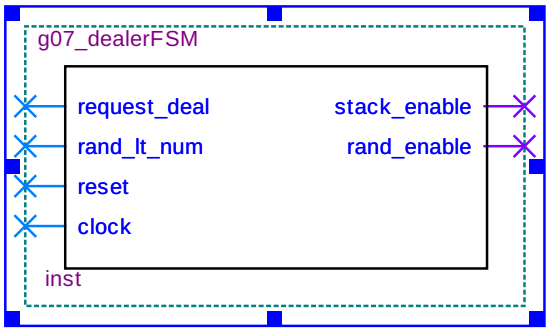
Therefore, it would benefit the system to create a robust *finite state machine*, a machine of a finite number of states. The system was reduced to a machine of 4 states (where $4 < \infty$ holds, proved in this appendix), of which include the state that waits for the previous request to end (denoted by A), the state that waits for the next request to *be* sent (denoted by B), the state that generates random numbers, so as to deal the cards (denoted by C), and finally, the state that activates a stack (or more accurately, a bi-directional Jenga tower) circuit (denoted by D).

All of the cleverness and power that has been teased above was encompassed by a circuit that has since been named the `g07_dealerFSM`. A more detailed insight to the design of the finite state machine and the `g07_dealerFSM` follows below, and is complemented by thoughtful pictorial support for easier understanding.

Given the immense power of the VHDL language, implementing the state machine in code was fairly simple. Firstly, an enumeration of states was created, and a Signal representing a state was defined. For simplicity, it was decided to implement a Moore-type machine, meaning the output depends only on the state of the system, held in the state signal. Then, in a *process* block, a case statement was designed, which governs state transitions based on the inputs of the FSM and the current state of the FSM. Please enjoy the VHDL code provided below for a complete view of the dealer's design.

Circuit Description

Figure 3: Pin-out diagram of the `g07_dealerFSM` circuit



Above is a pin-out diagram of the miraculous `g07_dealerFSM` circuit, showcasing its input and output ports. A more detailed description of these ports will be given below.

Description of ports

request_deal

The `request_deal` input bit is activated when the system requests that the dealer should deal a card.

rand_lt_num

The `rand_lt_num` input bit is controlled by an external circuit that is high when a random number has a value that is less than the BJT's `NUM` output that the dealer is associated with. This allows the dealer to tell whether it should keep generating random numbers (in state C), or move on.

reset

The `reset` input bit causes the finite state machine to be reset to its initial state when `reset` is high. It is asynchronous.

stack_enable

The `stack_enable` output bit is active when a valid random number has been generated in the proper state. It is used to enable the BJT associated with the dealer to pop a card.

rand_enable

The rand_enable output bit is active after request_deal has been asserted anew, and enables a random number generator to generate random numbers until rand_lt_num is high.

A complete VHDL description of the g07_dealerFSM circuit is provided in the following section.

VHDL portrayal of the g07_dealerFSM circuit

The tactics used to develop this VHDL code may be examined in the rationale section above.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity g07_dealerFSM is
6          port (
7                  request_deal: in std_logic;
8                  rand_lt_num: in std_logic;
9                  reset: in std_logic;
10                 clock: in std_logic;
11                 stack_enable: out std_logic;
12                 rand_enable: out std_logic
13         );
14 end g07_dealerFSM;
15
16 architecture machineOfState of g07_dealerFSM is
17         SIGNAL initial_seed: std_logic_vector(31 downto 0) := std_logic_vector(
               to_unsigned(1337,32));
18         SIGNAL random: std_logic_vector(31 downto 0);
19         SIGNAL reg_out: std_logic_vector(5 downto 0);
20         SIGNAL comp_lt: std_logic;
21         SIGNAL stack_num: std_logic_vector(5 downto 0);
22
23         TYPE State_type is (A,B,C,D);
24         SIGNAL state: State_Type;
25 begin
26         rand_enable <= '1' when state = C else '0';
27         stack_enable <= '1' when state = D else '0';
28         machine: process (clock, reset)
29         begin
30                 if(reset = '1') then state <= A;
31                 elsif (clock'event and clock = '1') then
32                         case state is
33                                 WHEN A =>
34                                         if(request_deal = '0') then state <= B;
35                                         else state <= A;
36                                         end if;
37                                 WHEN B =>
38                                         if(request_deal = '0') then state <= B;
39                                         else state <= C;
40                                         end if;
```

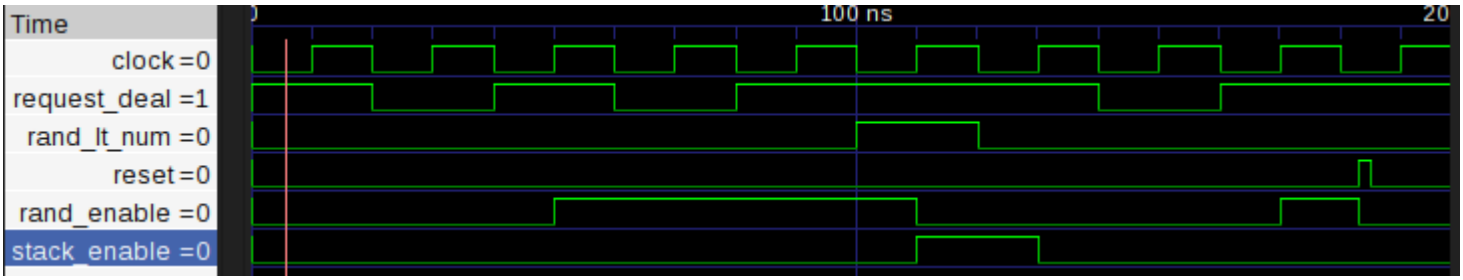
```
41                                  WHEN C =>
42                                          if(rand_lt_num = '0') then state <= C;
43                                          else state <= D;
44                                          end if;
45                                  WHEN D =>
46                                          state <= A;
47                          end CASE;
48                  end if;
49          end process machine;
50  end machineOfState;
```

On the Testing of the Dealer Circuit

It was an important and difficult challenge to resist being seduced by the beauty of the VHDL code that describes the `g07_dealerFSM` circuit. Despite it's beauty, a dealer cannot be trusted without being tested. Think about this following section as an analog to an interview process, if you will. Before putting the `g07_dealerFSM` in commission, it was important to make sure it satisfied all dealing requirements (and boy, did it ever), and to make sure it functioned correctly on the hardware. The demonstrations that follow will prove that the `g07_dealerFSM` is in fact an excellent candidate.

The software circuit simulation: a poor man's analysis

The first order of business in judging the effectiveness of the `g07_dealerFSM` was to test it with the magical Modelsim simulator. Unfortunately, said software was not functioning to the tester's standards. As a consequence, the miraculous GtkWave VCD viewer came to the rescue, and the `g07_dealerFSM` was examined thoroughly according to a carefully-written testbench.
For the less-enthused, the testbench starts the state machine in its default state, and examines each state transition in the trivial order. Then, the machine is brought to state C (where `rand_enable` is high) and reset is activated to ensure that it causes the machine to return to its initial state.
The resulting waveforms can be viewed below.



Of course, the results shown above cannot conclusively confirm the excellence of the `g07_dealerFSM` circuit on their own, because they give no evidence of the dealer functioning on real hardware. Hence, this test was called a *poor man's analysis*. Fortunately, the testers *are not* poor men. In fact, they have access to an Altera DE1 FPGA development board equipped with an Altera Cyclone II FPGA. The tests in the following section will describe how *full confidence* of the `g07_dealerFSM` was obtained.

The hardware circuit simulation: an Engineer's victory

Some tests are easy, leaving the participants in a relaxing state of mind. Yet, some tests are hard, leaving participants in a state of panic. However, occasionally a test may reach a transcendental level of difficulty, such that it may be said that the test *separates the boys from the men*. That, dear reader, is what you will witness in the remainder of this section. Anyone from laymen to patricians can make a software circuit simulation, neglecting the possiblity that something may go wrong on the hardware. While seemingly reasonable to the un-seasoned tester, this train of thought is highly dangerous and irresponsible. It is up to the Engineer to realize that this, in fact, will not suffice. This realization was exhibited, more than anything else, in the results below.

The first order of business was to create a *test bed*, or circuit that bridges the gap between human input and dealer communication. This was done using the not-short-of-incredible Altera Quartus II schematic designer, and may be perused here.

For the less-motivated reader, a brief description of the testbed will be given. Firstly, The `stack_enable` output of the dealer circuit is passed as the `enable` input of the BJT. The `rand_enable` output was passed to the `enable` input of a `g07_register6` circuit, which is a masterfully-simple 6-bit register designed by the same designers of the `g07_dealerFSM`. The design of the `g07_register6` is unfortunately beyond the scope of this report. The register serves the purpose of latching a random number generated by the random number generator module. Then, the value stored by the register is passed through a comparator, which sends a 1 to the `g07_dealerFSM` when the random value is less than the `NUM` output of the BJT. The `request_deal` input comes from the output of the infamous `g07_debounder` debouncer circuit (discussed in a previous laboratory report), so as to remove the all-feared bouncing of the DE1's hardware buttons. The random number generator module consists of a `g07_RANDU` (discussed in a previous laboratory report) which takes as input the multiplexation of a constant seed, and the last generated random output of itself. When the BJT is full, the constant seed is passed to the RANDU, and otherwise it uses its own seed to enliven itself.

Finally, the `NUM` output of the BJT as well as the `VALUE` output corresponding to the address described by the value stored in the `g07_register6` are passed through `mod13` circuits (discussed in previous laboratory reports) to be conveniently displayed on the 7 segment decoders on the DE1 board.

Some say a picture is worth a thousand words, however only an infinite amount of words can be used to describe the joy of the testers upon witnessing the massive success of the `g07_dealerFSM` circuit on the testbed. It would be infeasible to include so many pictures in this report. Instead, a brief overview of what was accomplished will be offered.

There were just two more criteria that needed to be confirmed to ensure the proper functioning of the dealer. Firstly, it needed to be confirmed that the `NUM` output of the BJT decreases by only 1 after each deal (thus implying that the dealer deals one card at a time). This test passed immediately with flying carpets. Furthermore, it had to be ensured that the cards being dealt were random! Although this part did provide some inconsistencies early one (due to a mistake with the comparator connections, mostly), eventually it was seen that random numbers in the range of 0 to `NUM` were seen being popped from the BJT. Victory was gracefully achieved.

Computer Analysis of the Dealer Circuit

If you thought the validation of the g07_dealerFSM circuit was complete, you have successfully been tricked. What blasphemy! Surely, the g07_dealerFSM must be shown to not inhibit the circuitry that it will integrated with. To verify that this will not be an issue, Altera's magical Quartus II Flow Summary and TimeQuest Timing Analyzer will be used extensively. Please continue reading the final sections of this report to determine whether or not the g07_dealerFSM is a *feasible* circuit, given all its unwieldly power.

The Flow Summary

According to Quartus II's Flow Summary, the g07_dealerFSM allegedly uses only 3 combinational logic functions, and 3 dedicated logic registers. Given that there are 18752 logic registers and combinational logic functions available in the FPGA, these numbers make for less than 1% of the resources available. Then, it can be safely concluded that the g07_dealerFSM does not require too much hardware, and is in fact *very* feasible in that regard. Furthermore, the g07_dealerFSM allegedly requires absolutely no memory bits, further convincing the feasibility of the dealer.

The Timing Analysis

Lastly (and you are not being fooled this time), the dealer must not require particularly low clock frequencies for its operation to be successful. It also would be beneficial for it to not impose any restrictions on the envelope of the clock pulse, as that would be rather inconvenient. According to the Timing Analyzer of Quartus II, the g07_dealerFSM allegedly imposes a minimum pulse width of -1.631 nanoseconds. Negative pulse widths are not a problem in the measuring systems in use today. In fact, most may say negative pulse widths do not exist (but it's better not to generalize in a report like this). However, due to the negative minimum pulse width, it can be safely concluded that no testable pulse width should cause a problem for the g07_dealerFSM circuit.

Finally, the Quartus II Timing Analyzer reported a restricted maximum frequency of 380.08MHz. This is extremely impressive, given that the maximum clock frequency that will be supplied to the g07_dealerFSM is the DE1's clock of 50MHz. Therefore, there shall be no timing violations caused by the g07_dealerFSM given its clock. This maximum frequency corresponds to a maximum propagation delay within the circuit of $1/38008000 = 2.63$ns, which is exceedingly small.

Given these results, it is clear that the g07_dealer circuit uses very little hardware and imposes no considerable timing issues. Finally, the Engineer may leave his mark.

Appendix A

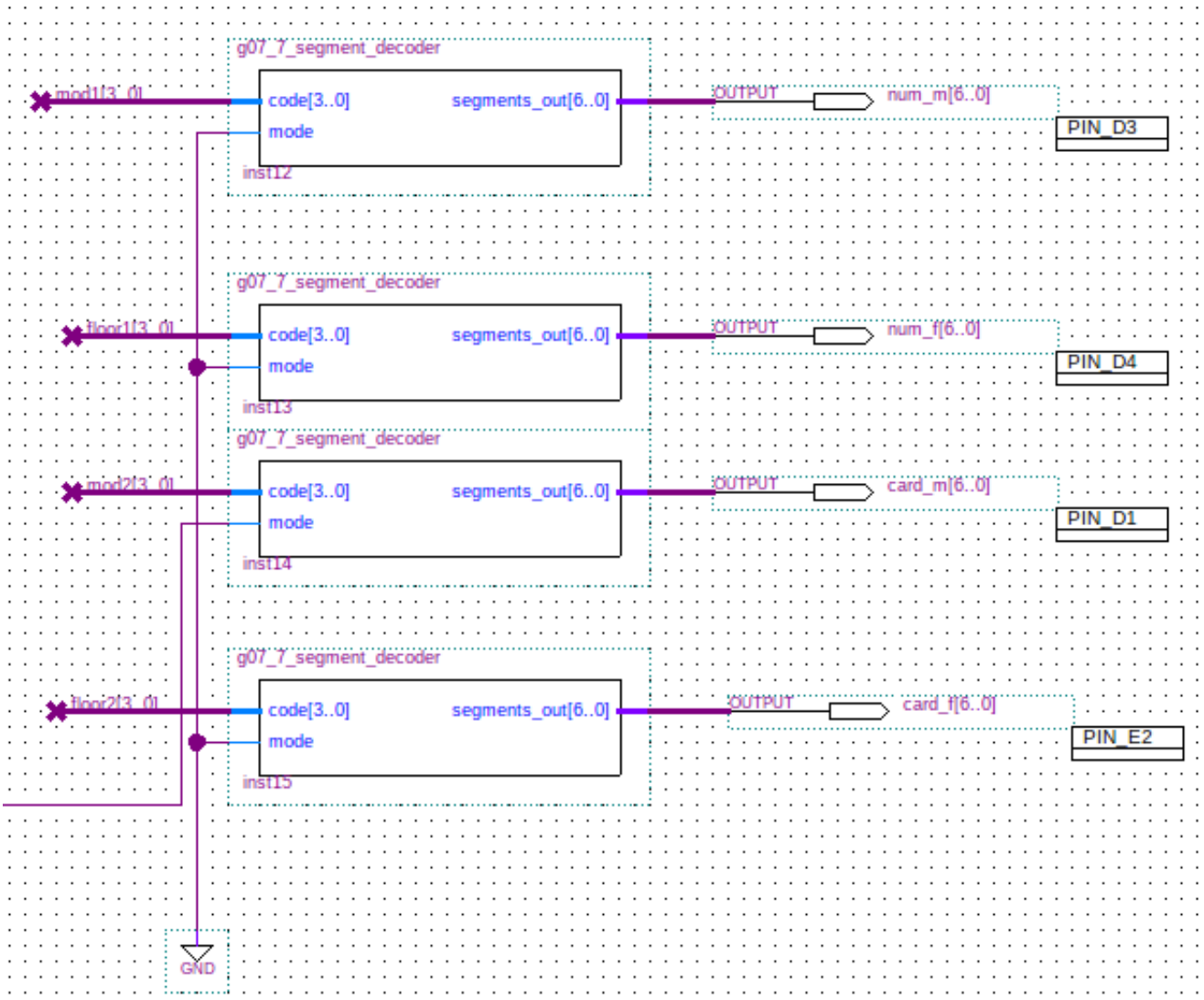
VHDL Testbench for the `g07_dealerFSM`

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity dealerFSM_tst is
6  end;
7
8  architecture test of dealerFSM_tst is
9          component g07_dealerFSM
10                 port (
11                         request_deal: in std_logic;
12                         rand_lt_num: in std_logic;
13                         reset: in std_logic;
14                         clock: in std_logic;
15                         stack_enable: out std_logic;
16                         rand_enable: out std_logic
17                 );
18         end component;
19         SIGNAL request_deal, rand_lt_num, reset, stack_enable, rand_enable:
               std_logic;
20         SIGNAL clock: std_logic := '0';
21         SIGNAL finished: std_logic := '0';
22
23 begin
24         machine: g07_dealerFSM
25         port map (
26                 request_deal => request_deal,
27                 rand_lt_num => rand_lt_num,
28                 reset => reset,
29                 clock => clock,
30                 stack_enable => stack_enable,
31                 rand_enable => rand_enable
32         );
33
34         clock <= '0' when finished = '1' else not clock after 10 ns;
35
36         always: process
37         begin
38                 request_deal <= '1';
39                 rand_lt_num <= '0';
40                 reset <= '0';
```

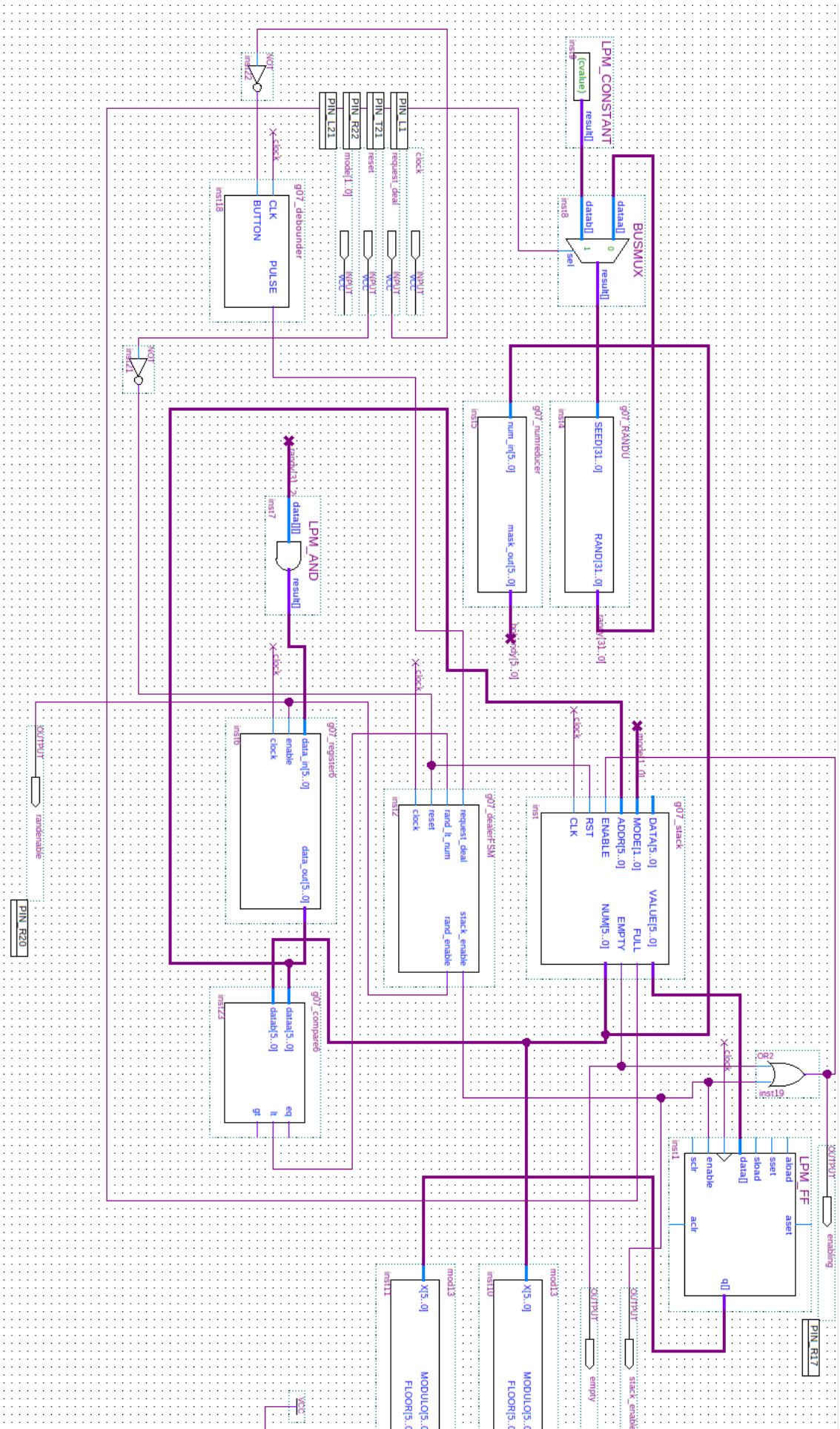
```
41                  WAIT FOR 20 ns;
42                  request_deal <= '0';
43                  WAIT FOR 20 ns;
44                  request_deal <= '1';
45                  WAIT FOR 20 ns;
46                  request_deal <= '0';
47                  WAIT FOR 20 ns;
48                  request_deal <= '1';
49                  WAIT FOR 20 ns;
50                  rand_lt_num <= '1';
51                  WAIT FOR 20 ns;
52                  rand_lt_num <= '0';
53                  WAIT FOR 20 ns;
54                  request_deal <= '0';
55                  WAIT FOR 20 ns;
56                  request_deal <= '1';
57                  WAIT FOR 23 ns;
58                  reset <= '1';
59                  WAIT FOR 2 ns;
60                  reset <= '0';
61                  WAIT FOR 15 ns;
62                  finished <= '1';
63                  WAIT;
64          end process always;
65  end test;
```

Appendix B

Testbed for the `g07_dealerFSM`



Above is the section of the testbed that showws how data is output to the 7 segment displays on the DE1. On the following page, the main body of the testbed may be observed.


LPM_CONSTANT
BUSMUX
LPM_AND
LPM_FF
g07_debounder
g07_numreducer
g07_RANDU
g07_register6
g07_deasertFSM
g07_stack
g07_compare6
PIN_L21
PIN_R22
PIN_T21
PIN_L1
PIN_R20
PIN_R17
CLK
BUTTON
PULSE
num_in[5..0]
mask_out[5..0]
SEED[31..0]
RAND[31..0]
data[][]
resu[]
datab[]
dataa[]
sel
clock
enable
data_in[5..0]
data_out[5..0]
reset
rand_lt_num
request_deal
stack_enable
rand_enable
RST
ADDR[5..0]
MODE[1..0]
DATA[5..0]
VALUE[5..0]
FULL
EMPTY
NUM[5..0]
datab[5..0]
dataa[5..0]
gt
lt
eq
OR2
inst19
scir
aclr
sload
sset
aload
aset
q[]
MODULO[5..0]
FLOOR[5..0]
empty
enabling
randenable
OUTPUT
VCC

Appendix C

Proving the Finite-ness of the State Machine

For the precise reader, stating that the desgined state machine is actually finite may not be concrete enough to be taken seriously. Given that the amount of states in the machine is 4, it can be shown that the machine is in fact finite.

Proof: $4 < \infty$
Follow a proof by contradiction. Assume $4 \geq \infty$. Then,

$$\sum_{k=0}^{\infty} x^k \leq \infty, \ |x| < 1$$
$$10 \sum_{k=0}^{\infty} x^k \leq \infty$$
$$10 \sum_{k=0}^{\infty} x^k \leq 4$$

Choose $x = \frac{1}{2}$. It can be shown combinatorially that $\frac{1}{2} < 1$ as follows: suppose two individuals are sharing a cake. By definition, the amount of cake each individual can eat is $\frac{1}{2}$ of the cake. Since energy can neither be created or destroyed, the amount of cake cannot be increased when two people share it. Thus, $\frac{1}{2} < 1$.

Corollary 1: $\frac{1}{2} < 1$.
Binding x to $\frac{1}{2}$, the following can be discovered:

$$10 \sum_{n=0}^{\infty} \frac{1}{2}^k \leq 4$$
$$10 \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots \right) \leq 4$$
$$10 \left(1 + \aleph\right) \leq 4, \ \aleph > 0$$
$$10 + 10\aleph \leq 4 \rightarrow 10\aleph \leq -6$$

Clearly, this is a contradiction since the definition of $\aleph$ guarantees that it is positive.
Therefore, by contradiction, $4 < \infty$.