EE 552
PROJECT GAMEPAL
Final Report

Submission Information
April 7, 1998
Dr. Duncan Elliott

GamePal Development Team
Mark Fedorak
Ron Smith
Vera Casteel
Kris Pucci

Table Of Contents

Abstract 03

Overview 04

IC Information and Details 05

Design Details and Documentation
- Entity GAMEPAL 07
- Entity KEYPAD 08
- Entity VIDEO 11
- Entity DICE 12
- Entity MINTIME, MIN3TIME, and MIN5TIM 14

Testing Procedures 17

Appendix A 18
- GAMEPAL Code Listing 19
- VIDEO Code Listing 25
- KEYPAD Code Listing 32
- DICE Code Listing 37
- MINTIME Code Listing 41
- MIN3TIME Code Listing 42
- MIN5TIME Code Listing 43
- MAINMENU.MIF 44
- DICE.MIF 45
- TIMER.MIF 46
- CHARBANK.MIF 47

Appendix B 55
- TIMER Code Listing 56
- COUNTDWN Code Listing 57
- COUNTER2 Code Listing 60
- COUNTER3 Code Listing 64
- KEYBTST Code Listing 68
- CHIP (VIDEO TESTER) Code Listing 69

Appendix C 71
- DICE Simulation 72
- COUNTDWN (non-working countdown without timer component) 75
- TIMER (component for non-working countdown timers) 78
- COUNTER2 (first non-working countdown with timer component) 82
- COUNTER3 (second non-working countdown with timer component) 86
- MINTIME 90
- MIN3TIME 93
- MIN5TIME 95

Abstract

The following report outlines the development of the GamePal by the GamePal development team.

You should read this report if you are interested in finding out the technical details of the GamePal. The overview and IC Information sections contain details on what the GamePal is and what is required in order to run it. The design description outlines how each individual component works, and how we obtained the solution we did. The testing procedures section outlines how we went about testing the individual components to ensure that they were working correctly. And finally the VHDL code, and simulations are located in the Appendix.

A softcopy is available in PDF format on Nyquist in:
/users/prof/elliott/www552projects/98w/GamePal

Overview

The GamePal was designed to be a "handy" device used for games that require certain functions in order to play them (i.e. board games). In our design, we include a 1 minute, 3 minute, and a 5 minute timer as well as a dice rolling function that allows you to choose the number of dice to be rolled with however many sides the user desires (up to 6).

The way it works is that the user programs the chip and is greeted with a main menu that lists all of the choices and the number that must be pressed in order to instantiate that function. The user types the number they want and the screen changes to display a screen containing the current function.

If the user choose one of the timers then they would see a timer counting down on the screen. This can be restarted by pressing reset and choosing timer again. If the user ever gets stuck, pressing reset will always move them back to the main menu. Another function the user can access is dice rolling. What the user does is press the key to enter the dice screen. The user then chooses the number of dice (1 to 6) followed by enter and then chooses the number of sides (2 to 6) followed by enter. The user must then press the roll key in order to display the first set of dice. Every time the user presses roll the screen will be updated with a new set of numbers corresponding to a dice. If a zero is displayed it indicates that the dice is not to be updated. If the user wants to choose different dice parameters they must press reset to go to the main screen and re-enter the dice function and enter the paramaters they want.

There is no graceful exit or power off. The user must unplug the GamePal, or reprogram the chip in order to turn it off. The keypad has some keys that are unused and the keypad must be powered in order for it to work correctly.

Our original design had many more functions, however, we dropped most of them because we believed that the display portion would take up much more space then it ended up taking. Also the functions that were included all required a lot more work than initally anticipated. For example, it tool over 2 months to come up with a video display process that was acceptable for this particular implementation.

Special thanks and recognition should be given to Jim Hamblen or Georgia Tech School of Electrical and Computer Engineering. The display engine and character set were supplied by him on the following web site.

http://www.ee.gatech.edu/users/hamblen/ALTERA/altera.htm

All the other code and ideas were generated by the development team. We solved many interesting problems and learned a lot in the process. Overall a very worthwhile endeavor.

IC Information and Details

IMPLEMENTATION DETAILS

To see full descriptions of all features implemented please see the design details section. The features that we implemented are:

- Up to 6 dice with up to 6 sides
- One, Three, and Five minute countdown timers
- VGA screens read from ROM banks
- Keypad with 12 keys

The Altera UP1 Education Development Board was used to implement the GamePal. The individual components on the board that are utilized are:

- Altera Flex EPF10K20RC240-4
- Flex Side LED (MSB and LSB)
- 8 Pins on Flex Expansion Array B
- VGA Port

CHIP INFORMATION

The FLEX 10K20 device is what the VHDL code included in the appendix is executed on. The following table has all of the major information regarding the use of this board.

Logic Cells Utilized	699 or 60% of the total 1152 LC's available		
Logic Cell Breakdown (approximate)	GamePal Module		113
	Dice Module		206
	KeyPad Module		110
	Timers		66
	Video		204
Memory Bits Utilized	8704 or 70% of the total 12288 bits available		
Memory Usage Breakdown	Character ROM	4096 Bits	
	Main Menu Screen	1536 Bits	
	Dice Screen	1536 Bits	
	Timers Screen	1536 Bits	
Flip Flops Utilized	192		
Total Compile Time	00:09:27.00		
Output Pins	25		
Input Pins	6		

PIN DETAILS

Name	Type	Number	Description
Blue	Output	238	Used for VGA. Displays Blue Color.
Green	Output	237	Used for VGA. Displays Green Color.
Red	Output	236	Used for VGA. Displays Red Color.
Horiz_sync	Output	240	Used for VGA. Synchronizes horizontal.
Vert_sync	Output	239	Used for VGA. Synchronizes vertical.
Clock	Input	91	Global clock for the GAMEPAL.
pinA	Output	120	Connects keypad pin A.
pinB	Output	118	Connects keypad pin B.
pinC	Input	116	Connects keypad pin C.
pinD	Input	114	Connects keypad pin D.
pinE	Input	119	Connects keypad pin E.
pinF	Input	117	Connects keypad pin F.
pinG	Output	115	Connects Keypad pin G.
pinH	Output	113	Connects keypad pin H.
LSB_a	Output	6	Segment A of the LSB on LED Display.
LSB_b	Output	7	Segment B of the LSB on LED Display.
LSB_c	Output	8	Segment C of the LSB on LED Display.
LSB_d	Output	9	Segment D of the LSB on LED Display.
LSB_e	Output	11	Segment E of the LSB on LED Display.
LSB_f	Output	12	Segment F of the LSB on LED Display.
LSB_g	Output	13	Segment G of the LSB on LED Display.
MSB_dp	Output	25	Decimal Point of the MSB on LED Display.
MSB_a	Output	17	Segment A of the MSB on LED Display.
MSB_b	Output	18	Segment B of the MSB on LED Display.
MSB_c	Output	19	Segment C of the MSB on LED Display.
MSB_d	Output	20	Segment D of the MSB on LED Display.
MSB_e	Output	21	Segment E of the MSB on LED Display.
MSB_f	Output	23	Segment F of the MSB on LED Display.
MSB_g	Output	24	Segment G of the MSB on LED Display.
MSB_dp	Output	25	Decimal Point of the MSB on LED Display.

Design Details and Documentation

The GamePal was designed by splitting up the large design into several smaller components that each group member could work on. This was done using the principles of top-down design. Therefore there are several distinct entities each interconnected and described below.

ENTITY: GAMEPAL

The main entity that controls all functions and input/output is that of the entity GAMEPAL. This entity manages external input and output, and acts as an interconnect for the rest of the chip. The individual entities (VIDEO, KEYPAD, DICE, MINTIME, MIN3TIME, MIN5TIME) are declared as components and their ports are mapped to signals that are either used as internal data paths, external data paths, or control the actions of the GamePal. The signal Current_Function controls whether the main menu, dice, or timers are being used. The vector signals pass data from the dice and timers to the video display.

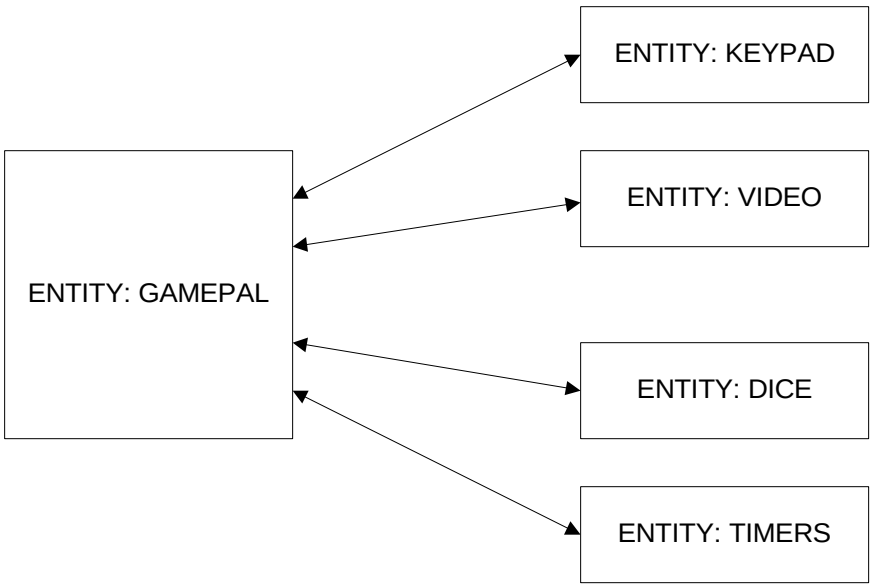

Figure 4.1

A state machine is responsible for switching modes and enabling each component. The state machine goes to a null state after the particular function (dice, timers) has been executed. This is due to the fact that you do not want to be switching states all the time. When the user presses the reset key (obtained from the keypad component) the state is reset to the first state and the user can select another function. To see what the state machine looks like please see Figure 4.3 located on the next page.

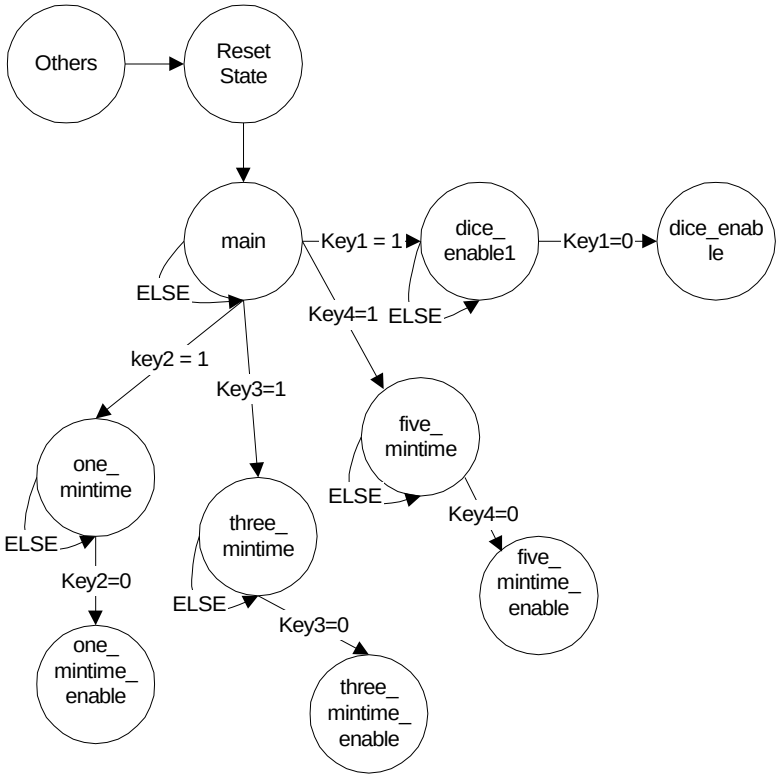

Figure 4.2

ENTITY: KEYPAD

In order to get input into our GamePal, we needed some form of keypad. We decided on a 16 key device to facilitate the numbers one through nine as well as some other function keys. The keypad was much more difficult to develop than we had initially anticipated, and the decoder logic took several attempts before developing a routine that gave satisfactory output. There are some shortcomings with our particular implementation which are described below, but they do not have any impact on the performance of our circuit, so the design was not changed. This decoder is inappropriate for some applications, and should be fully understood before attempting to export the design to another implementation.

The keypad we used was a Series 83-BB1-002 obtained from the electronic storehouse at the University. It is a very simple pad that must be powered and grounded at different pins at different times by the controller (GAMEPAL entity) in order to give the desired output. Table 4.1 is a truth table describing the function of the switches.

We decided to make the switches active low, by using pins A, B, G and H as inputs to the keypad and taking pins E, F, D and C as outputs. The way this works is that the input pins are all driven high except for one. The outputs are held high by a pull up resistor between the output pin and Vcc. When a switch is pressed, the corresponding input pin is driven low, causing a current to flow through the resistor, and the voltage on the output pin will be dropped signaling to the decoding logic that the key has been

Button	Pin E	F	D	C	A	B	G	H
1	X	-	-	-	X	-	-	-
2	-	X	-	-	X	-	-	-
3	-	-	X	-	X	-	-	-
4	-	-	-	X	X	-	-	-
5	X	-	-	-	-	X	-	-
6	-	X	-	-	-	X	-	-
7	-	-	X	-	-	X	-	-
8	-	-	-	X	-	X	-	-
9	X	-	-	-	-	-	X	-
10	-	X	-	-	-	-	X	-
11	-	-	X	-	-	-	X	-
12	-	-	-	X	-	-	X	-
13	X	-	-	-	-	-	-	X
14	-	X	-	-	-	-	-	X
15	-	-	X	-	-	-	-	X
16	-	-	-	X	-	-	-	X

Table 4.1

pressed. If the correct input pin is not driven low, there will be no change in the voltage, and no key press will be detected. (See Figure 4.3)

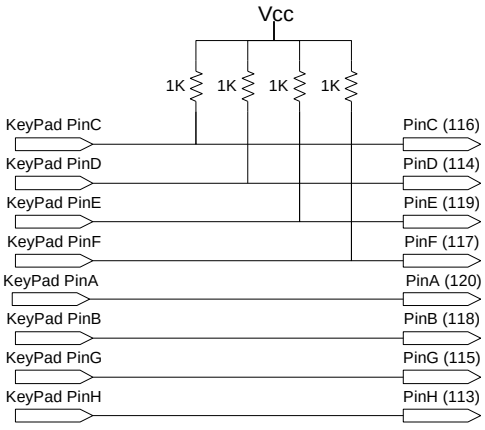

Figure 4.3

One of the problems we may have been encountering in the early versions of the keypad decoder is the capacitance of the prototype board we were using. The input pins were cycled through with one logic 0 at a frequency of 25 MHz, and the capacitance of the board may have been the cause of the apparent cross-talk between the different keys.

In our final, working decoder several extra states were added to avoid this cross-talk and to debounce the key presses so that no single input would be interpreted by the rest of the chip as multiple inputs. First of all we included a stepped down clock that cycles through at about 1 kHz. This is slow enough to get reliable outputs, yet fast enough that the inputs have no visible delay to the user. The first state resets all the keys

to zero, and they cannot be changed except by the "set" states. Next the state machine cycles through states which enable one row at a time on the keypad. When a key press is detected, the state machine is advanced to the corresponding "debounce" state. This causes the enable cycling to cease and hold the current row enabled. If the key is still unstable the state machine returns to reset_state and continues searching for a keypress. When the input is stable the state machine advances to "set", which assigns the appropriate key to one and all others to zero. The state machine remains in this state until the key is released. Because of the slower clock, switches likely will not give multiple inputs on one keypress, and the "debounce state" filters out any signals that could still be unstable. (See Figure 4.4)

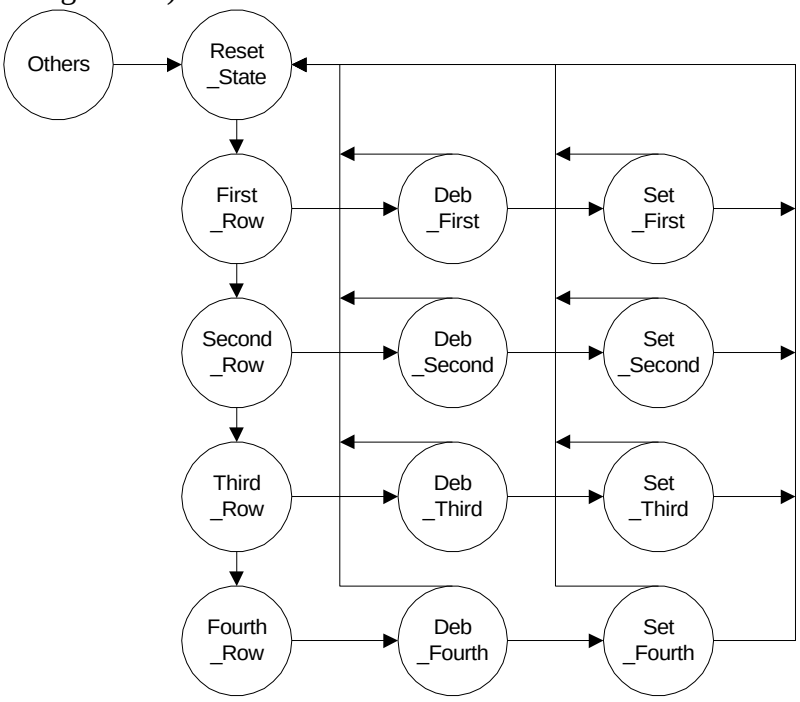

Figure 4.4

Careful inspection of the code reveals some shortcomings; however, these shortcomings have no effect on this implementation. If two or more keys on the same row are held down at the same time, both will be output from the decoder, but only the first one pressed will be properly debounced. Pushing a key on an alternate row however will not produce any results. This will not impact our GamePal design since it only accepts one input at a time from the keypad and waits until that key has been released before registering another one.

Upon completion the keypad was extensively tested by connecting the outputs to the LED display on the prototype board and observing the output of when different combinations of buttons were pressed. All results were acceptable with what was expected according to the explanation above. The pull up resistors used were 1 kΩ each. With a 5 volt source as on the prototype board, each one can draw only 5mA of current. The entire configuration can only draw a maximum of 20mA, but will rarely do so. The voltage on the power source was measured with and without this current drawn and had

no appreciable loss in voltage, so the switches will not adversely affect the power supply and thus the rest of the circuit.

ENTITY: VIDEO

We first decided to use a LCD display, which would display the output of the chosen function. Since a LCD is somewhat limited for what it can display, we switched to using a VGA monitor for displaying our output. VGA offers more flexibility for our project, because we are able to display more characters on the screen than we could on an LCD display, as well as change the colors of the text, etc.

The other reason is that more reference material was available on using the VGA monitor (*http://www.ee.gatech.edu/users/hamblen/ALTERA/altera.htm*) and the monitor itself was readily available for us to use in the lab.

The GAMEPAL is able to switch between full screens of text that are stored in ROM. It switches between these screens depending on the function that the user chooses to perform. If for instance the 3-minute countdown timer were chosen from the main function menu, the screen would change to the timer screen and begin to countdown from 3 minutes. The reset button on the keypad will always take you back to the main menu screen.

When we chose to use the VGA monitor, we first had to find out how to display characters on the monitor itself. The Altera UP1 board contains a VGA port, which allows a possible resolution of 640x480 pixels using five output signals (red, green, blue, horizontal sync, and vertical sync).

The screen refresh process begins in the top left corner and “paints” 1 pixel at a time from left to right. At the end of the first column, the row increments and the column address is reset to the first column. Each row is painted until all pixels have been displayed .

Our implementation supports a maximum of 32x8 characters using a character set where each letter is an 8x8-unit block. Each block unit consists of 16x16 pixels. ROM and LC (logic cell) limited the complexity of the screens. We put all the static screen data in a ROM file which is initialized using a memory initialization file (.mif) file (see Appendix A).

On examination of the mif file you will see something similar to the following:
XXXXXXXX : YY YY YY YY…
XX is the address of the characters to written to screen(8 bits required) and YY is the address of the actual individual characters in octal. The first 3 bits of the address are used to indicate which row the data is to be displayed on. This is where the 8 rows comes from. The last 5 bits correspond to 32 columns across. The octal characters are read in from the character repository in ROM.

The aforementioned character set characters are also stored in ROM (see the charbank.mif file in Appendix A) and are called in from the above file to display the appropriate text on the monitor. Only 6 bits are used above, because the letters, as mentioned above, are 8 X 8 pixels. As you can see for example the letter "A" is equivalent to 01 in octal.

The character ROM would show the letter "A" as the following addresses.

The Letter "A"
010 : 000**11**000 ;
011 : 00**111**100 ;
012 : 0**11**00**11**0 ;
013 : 0**1111**10 ;
014 : 0**11**00**11**0 ;
015 : 0**11**00**11**0 ;
016 : 0**11**00**11**0 ;
017 : 00000000 ;

We also are able to change the color of the background and text as long as the color is a combination of red, blue and green. To create the ROM banks we make use of use of Altera's library of paramartized modules. The video process creates 3 ROM banks for data screens, and one bank to store the character set as mentioned above.

The top level component of our project, interfaces the code for the dice and timer functions with the video_display process and allows them to be displayed on the monitor. Special format characters were imbedded in the mif file. As the data in the ROM banks is read in those special characters are flagged and the appropriate data from the timers and dice components is put in place of that flag character, and then displayed on the screen.

ENTITY: DICE

Our original design for the dice called for a total of 1 to 6 dice, chosen by the user, to be displayed with random values with one keypress. These dice would have an optional number of sides from 2 to 20. Eventually to simplify the design, we changed the maximum number of sides to 6. Once the number of dice and sides have been chosen by the user, these values will stay in memory, and the dice can be continuously rolled. If new values of sides or dice are required, the user can simply reset the system and choose appropriate new values. VHDL was used to describe the behaviour of the Dice entity, and it was simulated and shown to be working correctly with the simulation capabilities of MaxPlus2.

To begin, a simple explanation of the operation of the dice entity will be given. Dice was designed to work as a separate entity receiving signals indicating what keys had been pressed, and responding appropriately. We needed to have up to six random

numbers to be generated simultaneously each with a value between 1 and 6. As this is intended for use as a game, it is necessary that it be sufficiently random that no digits will have a higher likelihood of appearing than any other. After considering various possibilities, we came to the conclusion that we would have a six counters count through all the different possible combinations of dice outputs that were selected by the user, and that these values would be sampled only when the roll key was pressed. This makes the number is completely random, because there is no way to tell when the key press will be registered. The maximum number of dice is 6 with a maximum possible number of sides of 6. This gives 6^6 or 46656 possible values which with the 25 MHz clock on the Altera board will cycle through many times per second, which will ensure that all 6 dice will be sufficiently random. To illustrate the function of the counters see Table 4.2 below which displays 6 dice with values up to 3.

First a state machine was created to accept the users inputs and begin counting out the dice. The state machine is rather simple. It first takes a value for the number of dice desired, then upon pressing and releasing enter it takes a value for the number of dice. Acknowledge signals are used to ensure that the state machine does not advance unless proper values have been input for both. These acknowledge signals are initialized and set

Counter1	123	123	123	123	123	123	123	123	123	123	123
Counter2	111	222	333	111	222	333	111	222	333	111	222
Counter3	111	111	111	222	222	222	333	333	333	111	111
Counter4	111	111	111	111	111	111	111	111	111	222	222
Counter5	111	111	111	111	111	111	111	111	111	111	111
Counter6	111	111	111	111	111	111	111	111	111	111	111

Table 4.2

in the processes Choose_Num and Number_OF_Sides. A reset at any point will clear all the values and new values can be entered. At this point the counter becomes active, and will continue to cycle through possible values until reset is pressed (see the state diagram in Figure 4.5). The counters were set up to cascade so that each one would only increment if the one before it has just completed its cycle. A large set of nested if statements were used to implement this. As a note, this process proved quite difficult to write. The first attempts used a large set of nested if statements similar to the ones now implemented, but were rejected by the compiler even after numerous revisions. We attempted many other algorithms to rectify this problem, but none of them gave functionally correct results. All the values would be cycled through, but not for correct amounts of time, or sometimes erroneous values would be given which was unacceptable. After many attempts at making this work it was evident that it wouldn't and another try was made at the large nested if statement. For some reason still unknown to us, it compiled correctly this time, and then gave a functionally correct simulation. The Altera compiler seems quite picky about the placement of the rising_edge keyword, and this may have been one of the reasons why it would not accept our earlier code.

One last process assigns the dice values to signals, which are then output from the dice entity. When the rising edge of a key press on the roll key is detected, each dice is assigned the value of its corresponding counter. Other unused dice are assigned to a value of zero.

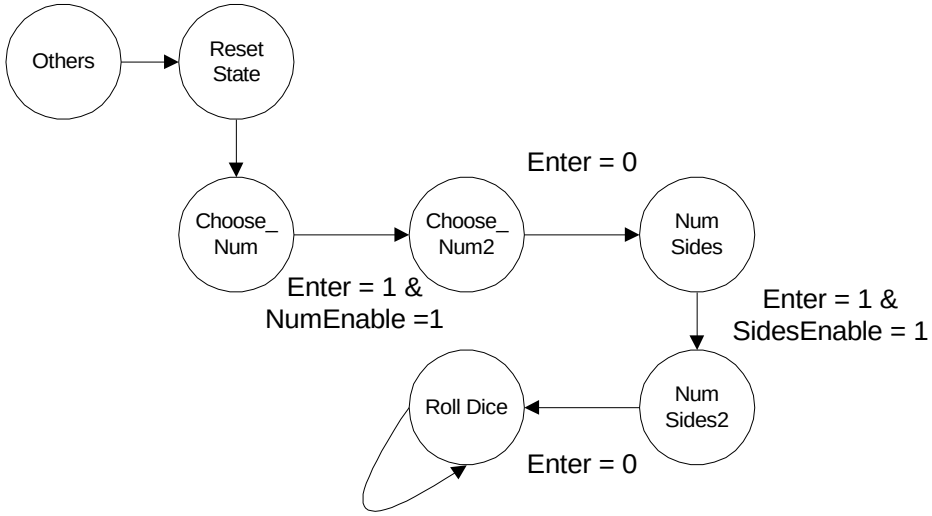

Figure 4.5

The dice entity was simulated using the Altera tools, and gave correct output. It was also tested with incorrect input to see if it would be robust and withstand any erroneous input a final user may subject it to, and again it simulated correctly. Some of the tests included not giving the number of dice or the number of sides, as well as testing whether the roll key would work before the counters were actually running and if other key presses would cause it to change states at the wrong time. Again, the dice passed this test.

ENTITY: MINTIME, MIN3TIME, and MIN5TIME

Our original plan was to have a countdown timer in which the user would enter the time to be counted down from, and the user would be given a choice of either letting the timer repeat the countdown as soon as it finished or the counter waiting for the user to restart the timer. Much time was spent on the countdown timer, but we could not make it simulate correctly. Therefore, the original countdown timer will be explained in detail, as well as the individual timers that took its place. One-minute, three-minute, and five-minute timers were designed. These timers start at the specified time and count down to 0:00. When they are done, the reset button can be pushed to choose that timer, a different timer, or the dice function.

To design the countdown timer, we first made a state diagram (see Figure 4.6). A case-when statement structure was used to implement the state diagram by assigning the outputs followed by the next state based on the inputs. A separate process was used to assign the state to the next state on the rising clock edge. Another process was used to increment the time by 1 minute or 10 seconds based on what the user enters. Two different approaches were taken for the timer itself. First, the timer was included in the main countdown entity as a process. This process changed the time on the 1 second rising clock edge. When it reached 0:00, a done signal was set so that the state machine could either restart the countdown or wait for the user to press enter. This approach did not work as the numerous acknowledge signals that were required became very difficult

to keep up with. The simulation failed and a different approach was taken. A separate entity, timer, was created to be a component used by the countdown entity. The timer itself was simulated and seemed to be functionally correct, however when called from within the countdown entity, the sec1 signal cycled through 1,0, and the sec2 signal cycled through 9,0. The states did not change correctly either. When the no_repeat function was chosen, the state machine followed the repeat function path. To remedy this, state assignments were given to the states, keeping in mind that states with asynchronous inputs should be adjacent to their following states. Unfortunately, this also did not work. The sec1 signal cycled through 0, 8, and 9. This was incorrect. Also, the state machine did not change states correctly. If "repeat" was chosen as the method of countdown, it would start in the repeating state, then jump over to the restarting state of no_repeat. At this point, it was decided that the countdown timer was not worth continuing, since it was unlikely that it would ever work.

Since the state machine in the countdown timer was not working, we chose to make the timer much simpler in order to have a working product at the end, rather than a feature-filled non-working product. A one-minute timer was written such that the reset signal on the board initializes the timer to 1:00. When enabled by the GamePal entity, the timer counts down to 0:00. This timer was simulated by itself and proved to be functionally correct. No acknowledge signals were necessary anymore as the timer only counts down and stops until reset is pressed so that a new function can be chosen. The one-minute timer was simulated with the GamePal entity and shown to be functionally correct. Since this timer was working, 3-minute and 5-minute versions of the timer were included. These timers work exactly the same way that the 1-minute timer works except that they start from a different initial time. Each timer requires 22 LCs. Before the

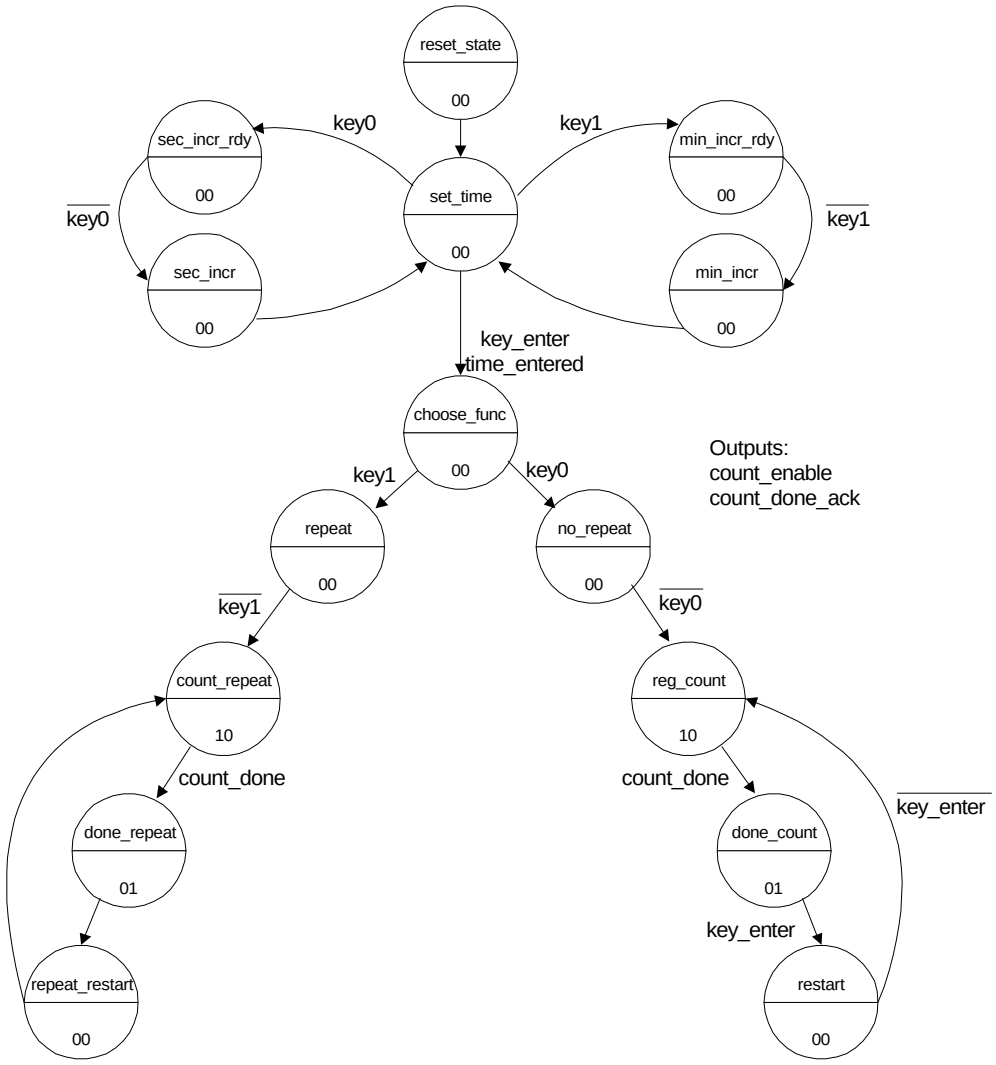

Figure 4.6

enable signal was added, 19 LCs were used. Thus adding an enable signal increased the LCs by 3 for each timer. All three timers simulate correctly alone and within the GamePal entity.

Testing Procedures

The Dice and Timers were tested through simulation. The simulations are located in Appendix C. Even though we simulated the dice and the timers we still had to tweak them slightly when we incorporated them into the GamePal entity in order to get them to work with the other components.

The keypad and video routines were tested through test prototypes which acted as if they were the entity GAMEPAL. The two files keytst.vhd and chip.vhd were used to test the keypad and video routines respectively (these two files are located in Appendix B). Due to the nature of the video and keypad interfaces, simulations would not have been a particularly effective way to test these routines. This is due to the fact that an in-depth knowledge of VGA timing would be necessary. Also the video code we used had been proven to work and it was not necessary to do any further testing of that code. The other testing was done on the fly by making changes, recompiling and testing by downloading to the Altera board.

One of the main reasons for applying the testing procedure mentioned above is due to the reprogrammable capability of the FLEX 10K20 chip. If we had only one time programmable components available we would not have attempted this type of project. We would have chosen something that would have simulated with a much higher degree of probability of working first try in the chip. Also the MAX+plus II simulation module is not nearly as capable as that of Mentor Graphics, however, we had to stick with the Altera tools due to the fact that we were programming their chip.

The true test is whether all the entities, when compiled together, give the desired output. They do, thus the testing, though perhaps not as strict as it should have been was successful in the end.

NOTE: In total we probably reprogrammed the FLEX10K20 about 150 – 200 times through the development of the project.