THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

The School of Engineering

COMPSYS 301B
Engineering Design 3CS

# Fruits' Basket

Developed By: Niels Grootscholten (9810499)
Thusitha Mabotuwana (9790416)
Winnie Jiang (9850730)
David Wang (9749018)

Date of Report: 15.10.2003

## Summary

The aim of this project was to design and implement an interactive FPGA-based video game using Verilog Hardware Description Language and the Altera UP1 board. The board is equipped with 1,152 logic elements, 12,288 bits of RAM and has a 25MHz clock.

The game created is called ‘Fruits’ Basket’, and the aim is to catch as many falling fruits as possible by using the mouse to move the bowl into the correct position. Grapes, cherries, strawberries and blueberries, and have been arbitrarily chosen as the fruits, with a successful catch being worth 5, 10, 15 or 20 points respectively. Any misses are penalised by decrementing the score by 10 points. The user receives 5 lives at the beginning and loses one for every 5 fruits missed. Current and highest score along with the level of difficulty are displayed on the VGA monitor while the remaining number of lives is shown on the LED display on the UP1 board.

Resource usage of the final implementation is 75% memory and 82% Logic Cells. The game does not use any kind of video buffer, as this was found to be too resource intensive. As such, the colour for each pixel is generated as the electron gun of the VGA monitor scans across the screen, using combinational logic. Sequential logic is used to make the pictures move. A total of 213 person-hours and NZ$6900.00 were spent in the development process of the implemented final design.

This report documents the idea, design and details pertaining to the implementation of the game including possible future developments which could be made to enhance the challenge and excitement of the game.

## Glossary of Terms

| | |
|---|---|
| FPGA | Field Programmable Gate Array |
| LED | Light Emitting Diode |
| HDL | Hardware Description Language |
| GDF | Graphic Design File |
| RAM | Random Access Memory |
| SRAM | Static Random Access Memory |
| Hz | Hertz |
| MHz | Mega Hertz |
| AI | Artificial Intelligence |

**Table of Contents**



Appendices

## List of Figures

## 1.0 Introduction

The aim of this project was to design and implement an FPGA-based interactive video game on the Altera UP1 development kit, using Verilog HDL and graphic entry. Video, mouse and push buttons were to be utilised to enable the system to interact with human inputs.

The game that has been implemented is named 'Fruits' Basket' and the main idea is to catch as many falling fruits as possible and score the maximum number of points. Five lives are given to the player at the start which will decrement one by one for each 5 fruits missed. The game will automatically reset upon losing all the lives.

The Altera UP1 board is a SRAM-based Altera FLEX10K family FPGA which has 12,288 bits of RAM, 1,152 logic elements and a 25MHz clock [1]. This limited amount of resources was to be used efficiently along with the VGA synchronisation module and the mouse module that were provided [2].

This report discusses details of the design solution, including how the required functionalities were achieved to meet the project specifications. It also includes a detailed cost analysis followed by proposed future developments to the game. Verilog code, graphic design files and other simulations of the design are included in the appendices as referenced. A project schedule that was developed using Microsoft Project is included in VI.

## 2.0 Project Specifications

The main requirement of this project was to design and implement an FPGA-based interactive video game on the Altera UP1 board, using Verilog HDL and graphic entry. The style, colours and other particulars of the game were left unspecified in the project specifications [3].

The design was to have a reasonable level of AI, use at least two of the three inputs (the available inputs being mouse, pushbuttons and dipswitches) on the board, and display the output on a VGA monitor with at least two colours.

## 3.0 Details Pertaining to the Implemented Game

The main objective of the implemented game is to catch as many falling fruits as possible and score the maximum number of points. Four different kinds of fruits, namely blueberries, strawberries, grapes and cherries have been arbitrarily chosen with a catch being worth 5, 10, 15 or 20 points for each fruit respectively. Five lives are given to the player at the beginning which will decrement by one for every $5^{th}$ fruit that is not caught. Failure to catch a fruit also results in a 10-point penalty.

Five different levels have been implemented. The levels are a function of the score, and change every 200 points. The fruits fall at different speeds and will gradually start to fall faster as the user progresses through the various levels. The level of difficulty starts from one and goes up to five, with level one being the easiest and five the hardest. The current game score is displayed on the upper right hand corner while the highest score recorded is shown on the upper left. The current level of difficulty is displayed at the bottom right.

The seven-segment LED display on the UP1 board is used to show the number of remaining lives.

Since the level of difficulty is a function of the score, the user can move up the levels, as well as down. For example, if the current score is 450 and if the user misses 6 fruits without catching any new ones, the score will decrement to 390 and the level of difficulty will also move from 3 to 2.

## 4.0 Development of the Design

During the initial weeks of the design, it was realised that the resources available were highly limited. As such, the idea of a video-buffer of any kind was abandoned. A video buffer of 1 line, in 8 colours alone would take approximately 16% of the RAM, and a large number of logic cells needed to decode it. This overhead requirement was decided to be unnecessary since it does not directly contribute to the game. Instead, generating the colour output of each pixel as the electron gun scans across the screen was used and pictures were generated by the use of sprites to ensure that memory wastage was kept at a minimum. This method has proven to be more economical and simpler than having a video buffer to generate the pictures.

The aim was to make a visually appealing, playable game, keeping in mind the nature of HDL and the resources of the board, whilst making design decisions. For example it would be highly inefficient to design a game that extensively relied on multiplication or division, due to their high resource utilisation. Therefore considerable effort was put into minimising resource usage and finding a balance between using memory, and logic cells.

The following methods were used as rules of thumb in an effort to minimise resource usage:

- Avoid using too many registers.
- Avoid using integer data type and merge modules wherever possible.
- Complete *if* and *switch* statements by using the *else* and *default*, instead of adding an extra condition (ie. another *else if* or a case).
- When comparing (using >,<,== etc.) in *if* statements, use binary combinations instead of integer values.
- Use bitwise operations as much as possible instead of comparing whole numbers. Eg: If we want to check for value 9 in a mod10 counter, we can do so by checking if the $1^{st}$ and $4^{th}$ bits are 1s instead of checking if the number is equal to 9.
- Avoid using addition and/or subtraction as much as possible. Use case assignments instead.

## 5.0 Implementation of the Final Design

The final design has successfully been implemented to meet all the required project specifications. Most of the initially planned features have been incorporated in this design along with some additional features to enhance playability. Shown below in Figure 01 is a flowchart showing the overall implementation of the game. Please refer to Section 5.0 for deviations from the initial project plan.

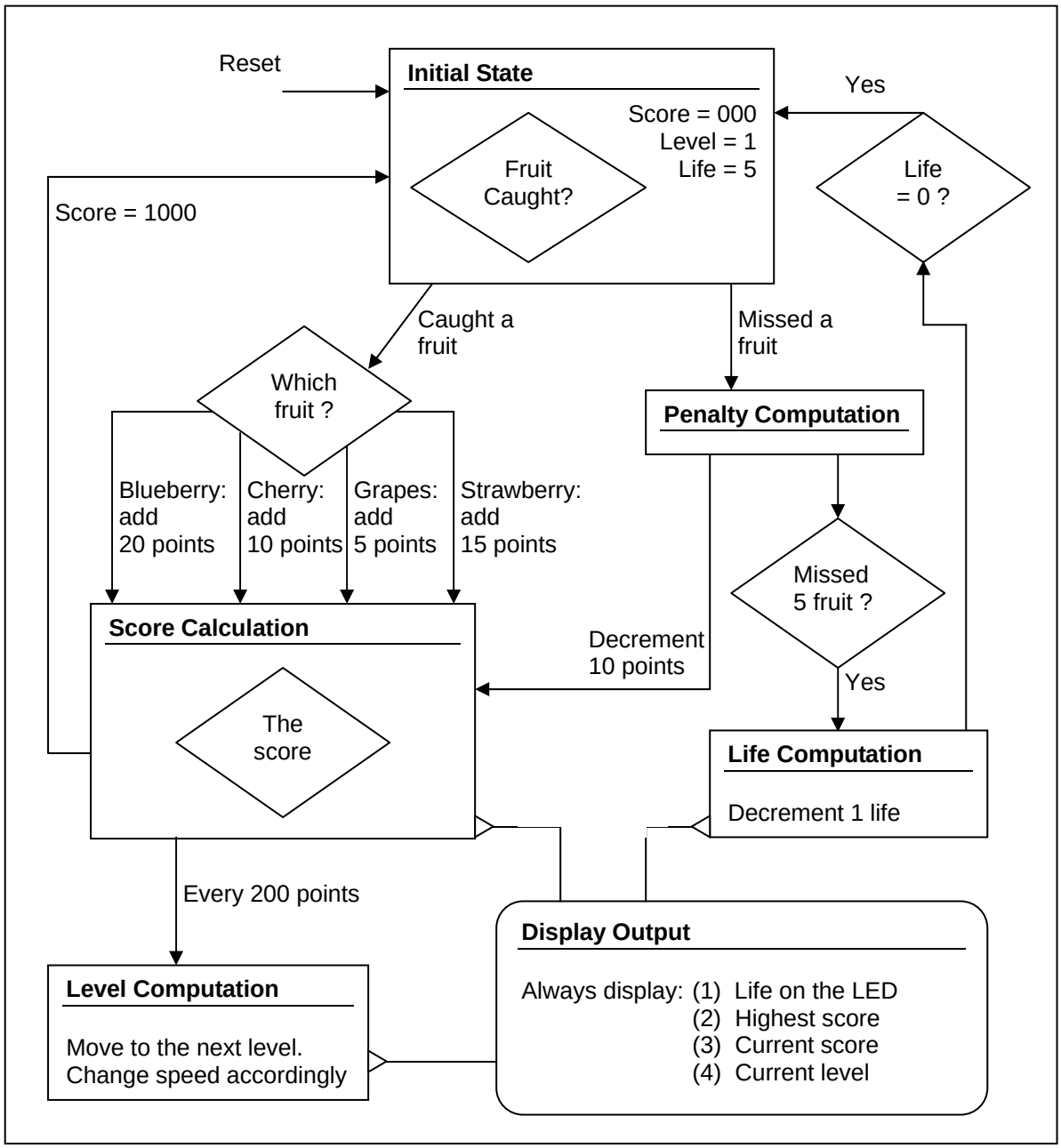



**Figure 01:** Flowchart showing overall implementation of the game

The final solution uses 75% of memory and 82% of the logic cells (The .rpt generated by MaxPlus10.1, containing the resource usage for the Altera FLEX10K EPF10K20RC240 chip can be found in Appendix V). This section contains a brief description and justification of various techniques used to implement each of the major components of the game.

**5.1**

## The Graphics

As shown below in Figure 02, the game area has several different components, each of which generates a colour output signal.

- Clouds
- Bowl (catcher)
- Grass
- Flowers
- Liana
- Fruits
- Current Score
- Maximum Score
- Display of the levels

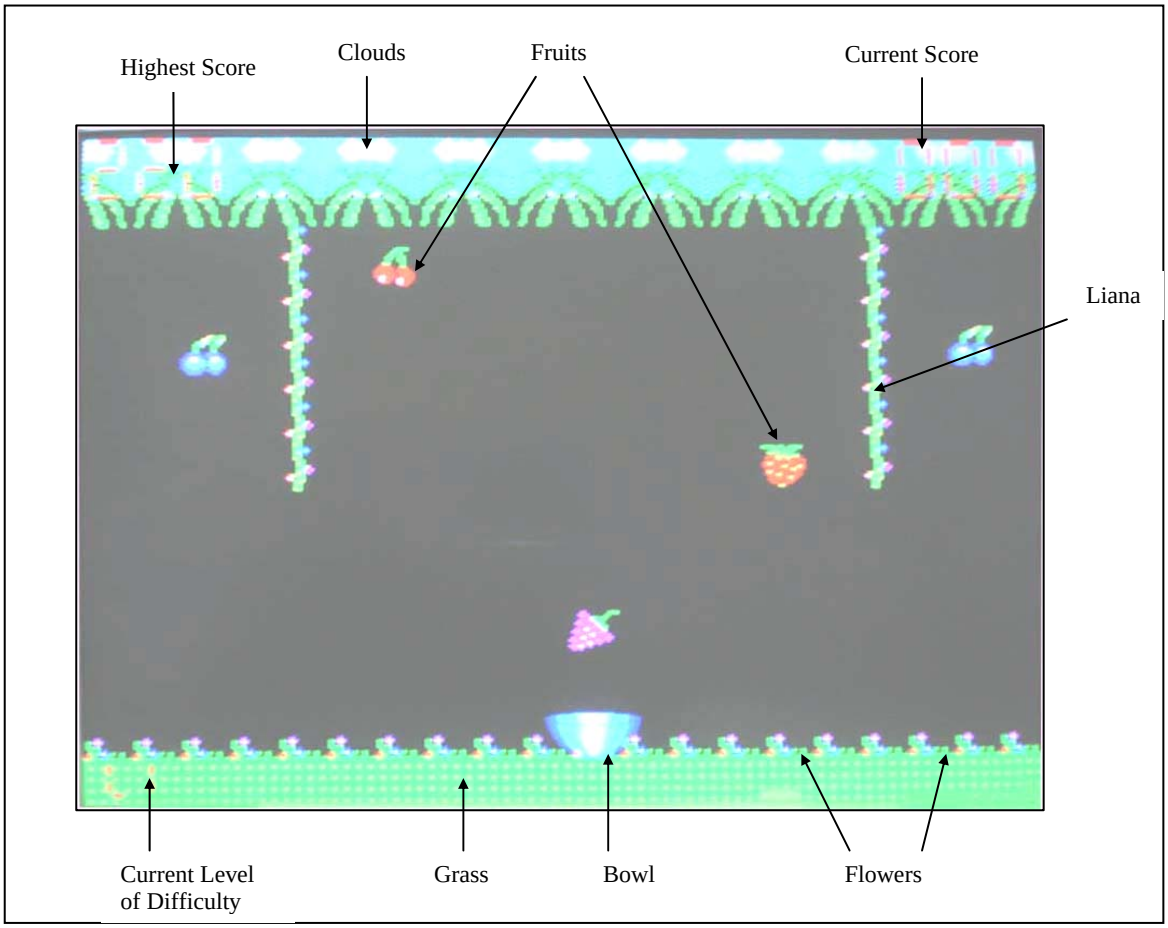



**Figure 02:** Screenshot of the play area

All components, except for the current score, highest score, level of difficulty and the grass are generated using the principles of Sprites. Sprites are an efficient way of displaying the same picture repeatedly, as the same memory is used, but only the x, and/or y starting position changes. An important optimisation technique we used is that every single sprite has width and height dimensions that are powers of 2 (this avoids subtraction and other mathematical operations). Sprites are implemented using one of the standard MaxPlus10.1 MegaFunctions called LPM_ROM, each created using the MegaFunctions Wizard.

### 5.1.1 Clouds, Lianas, Flowers

The clouds are based on a 3-bit per pixel, 32 x 16 pixel sprite. They are drawn across the top of the screen, for aesthetic pleasure. The address into the sprite is simply bits 1-5 of the row counter, and 1-4 of the column counter coming out of the *VGA_sync* module. This means that each pixel of the picture takes up 4 pixels on the screen. Mirroring the picture is achieved by inverting the column address when the pixel column bit 5 is high. The shadow effect is achieved using simple combinational logic to successively leave more and more pixels black as the column count increases. Further combinational logic ensures that the clouds are not drawn all the way down the screen. The flowers at the bottom of the screen and the lianas are achieved using similar techniques; however, these are only based on 16 x 16 pixel images.

### 5.1.2 Bowl (Catcher)

The catcher sprite is based on a 32 x 32 pixel image in 3-bit colour depth. This picture is mirrored along the x-axis, resulting in 32 x 64 pixels on the screen. Moving the mouse slides the bowl across the bottom of the screen. The column address for the bowl sprite is generated by subtracting mouse column from the pixel column. The row address is simply the appropriate bits of the *VGA_sync* module's *pixrow* output. Combinational logic ensures that the bowl is only drawn in one place on the screen.

The bowl makes use of a technique called dithering to generate extra colours. The colour output from the *ROM* module is decoded in a module called *ditherSelector*. This module has a register that toggles each frame. This is combined with using XOR operations on the least significant row and column bits, to create a chess-board pattern which is then flipped each frame (60 Hz). The bowl has the following colours:

- Black
- Dark Blue (blue mixed with black)
- Blue
- Light Blue (blue mixed with cyan)
- Cyan
- Light Cyan (cyan mixed with white)
- White

The use of these colours gives an appealing 'shiny' effect. A further module called *bowlColorShifter* swaps some of the above colours, depending on the level that the game is up to. There are four different colour states for the bowl.

### 5.1.3 Grass

The grass at the bottom of the screen is generated using only simple combinational logic.

### 5.1.4 Fruits

Each fruit is drawn based on a 16 x 16 pixel image in 8 colours. There are 4 different fruit sprites – one for each type of fruit. Each fruit sprite takes its column address straight from the *VGA_sync* column count. A multiplexer enables the appropriate fruit sprite's output, based on the column counter from the *VGA_sync* module. Each fruit moves at a different speed depending on the current level of difficulty. Movement is achieved using several

modules as shown in Appendix I. Some of the main modules involved in this process are the *Resetter*, *Ypos* modules and the *FiveToOneMux*, which are briefly described below.

#### *5.1.4.1 'Resetter' module*

This module is responsible for generating catch signals, which are then passed on to the appropriate fruit's *Ypos* module. A valid catch only occurs when the bowl is at the right place at the right time. The following conditions must be satisfied for a catch to count:

- The fruit must not yet have touched the ground.
- The fruit and the bowl must visually touch (ie overlap of colour).
- No part of the fruit must fall outside the bowl.

These conditions are checked for using combinational logic.

#### *5.1.4.2 'Ypos' module*

There is a *Ypos* module for each fruit that keeps track of its y-position. This is necessary to ensure that each fruit can be in a different position. The y-position is incremented every time the *Vert sync* signal goes low (60Hz). This *Ypos* module is capable of adding three bits to the y-position, resulting in 7 available speeds (0 cannot be considered acceptable).

The *Ypos* module is also responsible for resetting the fruit's y-position when it is caught. This reset however, cannot be done until the electron gun gets to the bottom of the screen. Therefore, the input signal from the *Resetter* module is converted to an appropriate signal, which causes the fruit to go back to the top of the screen if it was caught. As catching different fruits score different points, each *Ypos* module asserts a catch signal for one clock cycle when a catch occurs. These signals are fed into the score module, which then increments the score as explained in Section 4.2.1.

Failing to catch a fruit loses the player points, hence the *Ypos* module also generates a *fruit_missed* signal similar to the catch signal.

### 5.1.5 Combining the Colour Outputs

The outputs from all modules are combined in the *ColorCombiner* module. This module simply takes 6 colour inputs, and performs a logical OR operation on them. The score and level display module is given priority – that is, the pixels that are part of the score (displayed in red), have green and blue turned off.

## 5.2 Displaying the Score

The game score is displayed at the top right hand side of the screen. The range of the score was chosen to be from 0 – 999, hence 3 digits are displayed. The main score module has the main *clock*, *reset* and a *fruit_missed* signal as its inputs and a 3-bit *colourOut* bus as one of its outputs. This module and other sub modules inside (Figure 03) are used to display the current score as well as the highest score

The *scoreDecoder* module keeps track of the current score and passes the 3-digits separately (For example, if the score was 650, the digits 6, 5, 0 will be passed) to each of the 3 *singleDigit* modules which then decode these numbers and control the visibility of each segment on the screen. These segments are then fed into the *selectEnable* module which takes the pixel row and column into considerations and decides what needs to be drawn on the screen. The *busMux* is a 6-3 multiplexer which passes the highest score

digits if the electron gun is on the left half of the screen and the current score digits otherwise. Please refer to Appendix III for the Verilog HDL code implementation of this section.

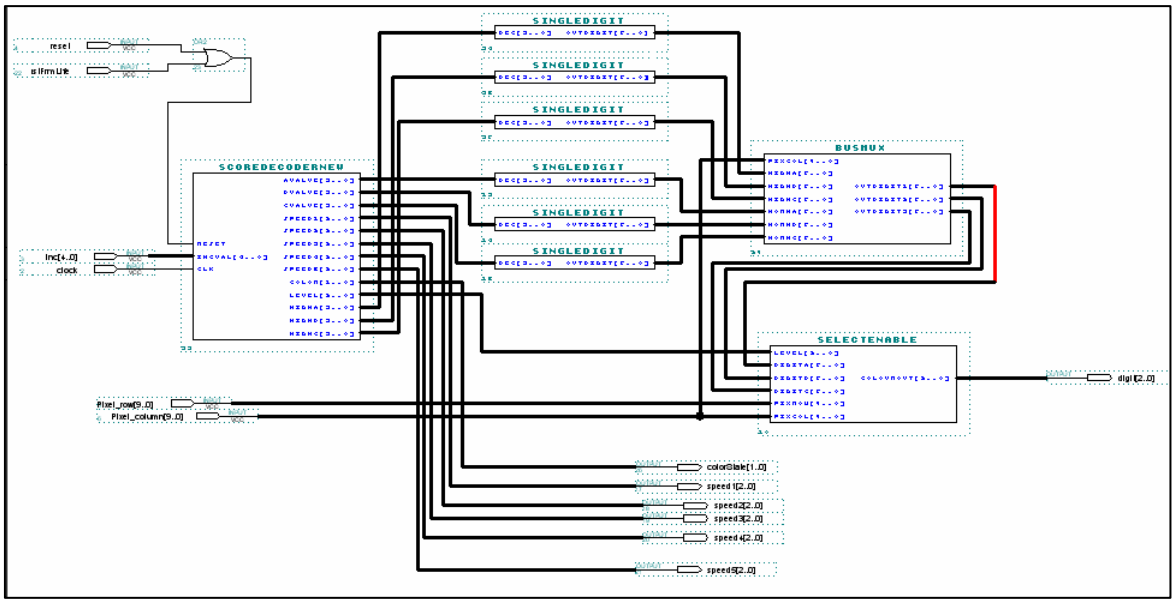



**Figure 03:** Sub-modules used to display the score and highest score.

### 5.2.1 Updating the Score

The score is updated each time a fruit is caught or missed. This signal is generated by the *mover* module (how a catch or a miss is detected is explained in Section 4.1.4.1) which is used by the *bus_output* module to determine by how many points the score has to be incremented. This *inc_val* signal is then passed onto the *scorenew* module as an input. The four different kinds of fruits, grapes, cherries, strawberries and blueberries are worth different points and upon a successful catch, the score will be incremented by 5, 10, 15 or 20 points respectively.

Since all the point combinations will be divisible by 5, score computation has limited possibilities. The last digit (*digit_C*) will always be either a 5 or 0 while the other 2 digits can take any value between 0-9. Logical statements have been implemented to check all possibilities for the addition of points. A maximum of 1,000 points are awarded and upon reaching this maximum, the score will ‘wrap around’. (Please refer to Appendix II for more details)

Each fruit that is missed is penalised and the score is decremented by 10 regardless of the fruit that was missed (however, the score will not go below 0). A simplified version of how the score is updated is shown Figure 04.

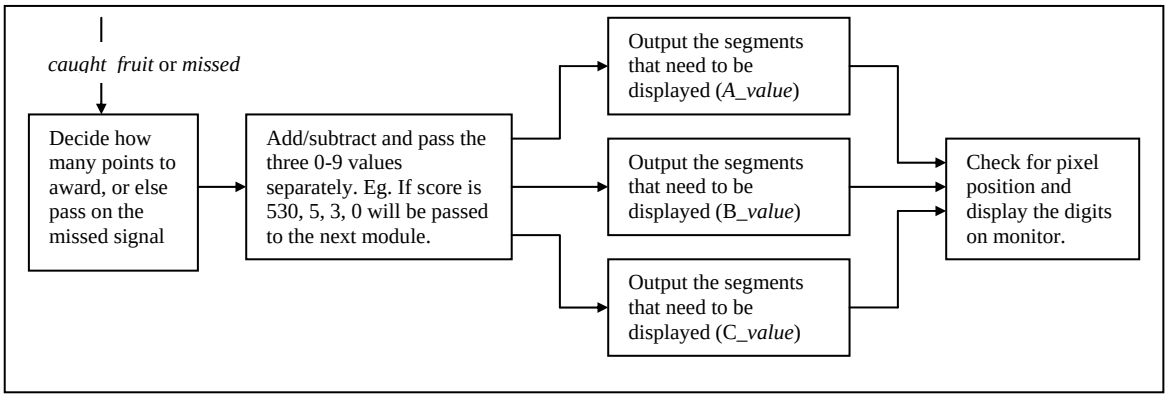



**Figure 04:** A simplified diagram showing of how the score is updated

### 5.2.2 Displaying Highest Score

The *scoredecoder* module keeps track of the highest score recorded so far. The highest score is updated each time the number of lives reaches zero, if the game score is higher than the last recorded highest score. Appendix II shows how the individual modules are connected to accomplish this.

## 5.3 Moving between Different Levels, Displaying Level of Difficulty on Screen and Colour Shift of Bowl

The level of difficulty is changed every 200 points and changes of the first digit (*digit_A*) are monitored to accomplish this. For example, if the first digit is 0 or 1 the corresponding level would be 1 and if the digit is 2 or 3, the level would be 2. The same *scoredecoder* module used for computing the score is used to detect these changes and output the current level of difficulty to the *selectenable* module, which will then draw this on the VGA monitor. The level is displayed on the bottom left and the same method used for displaying the score is used to determine the correct position on screen. Individual fruit speeds for different stages of the game are also generated by the *selectenable* module.

At any time, this module also outputs a *colour* signal corresponding to the level. This signal goes into the module responsible for rendering the bowl (see Section 4.1.2).

## 5.4 Displaying Number of Remaining Lives On the LED Display

As shown below in Figure 05, the number of lives starts at 5. The player loses a life for every 5 fruits missed. The number of lives remaining is shown on the LED display on the Altera UP1 board. Subtraction has to be performed in the *down5_count* module, hence switch case statements have been used in order to minimise resource usage. Appendix IV shows how this has been implemented in Verilog HDL.

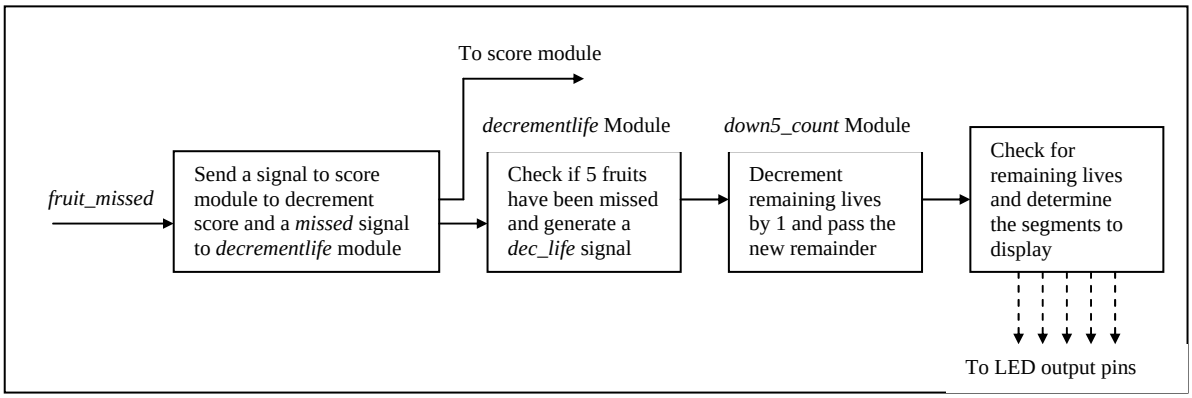



Figure 05: **Flowchart showing how the remaining number of lives is determined.**

## 6.0 Variations from the Initial Project Plan

Although most of the features implemented in the final design agree closely with what was initially suggested, the following modifications have been in order enhance the playability and visual appearance of the game.

- Initially, it was intended that the player had to repeatedly click the mouse button to change the colour of the bowl and match this colour with the colour of the fruit to achieve a successful catch. However, experimentation revealed that this concept made the game virtually impossible to play, hence the idea was modified so that the colour of the bowl automatically changes depending on the current level.
- Originally the fruits' falling speeds were to be adjusted using push-buttons, in order to cater for different levels of difficulty and this had to be done manually. This concept was changed in the final implementation and the speeds of the falling fruits are increased automatically, depending on the level of difficulty the player is currently in. This has made the game more challenging and interesting to play.
- The remaining number of lives was originally to be displayed on the screen, but in the final solution this was done using the LED display on the UP1 board in an effort to make use of more available interface resources. Also the given number of lives was decremented from 10 as was in the original plan, to 5, to make the game more challenging.
- It was felt that displaying the highest score so far would generate a greater interest in the player since he/she would be driven towards beating this score. This is the only completely new feature that has been implemented in the final solution, which was not part of the initial game plan.

## 7.0 Cost Analysis of Final Solution

Shown below is a record of number of hours contributed by each group member, itemised by task, along with a cost analysis table for the implemented final solution. The calculated total cost of the project is NZ$6900.00.

| Task No. | Task Name | Each members' working hours | | | |
|---|---|---|---|---|---|
| | | Niels | Winne | David | Thusitha |
| 1 | Have a catcher moving in x-direction by the mouse | 4 | | 10 | 4 |
| 2 | Have falling objects, with different speeds | 8 | | 8 | |
| 3 | Have some basic background, using combinational logic | 15 | | 5 | |
| 4 | Converting bitmaps into ".mif" files using MATLAB | 1 | 3 | | |
| 5 | Having all objects in ".mif" form using RAM_DQ manager | 4 | | | |
| 6 | Display the 3-digit score on the screen | | 10 | | 10 |
| 7 | Catch Detection, and catch signal generation | 3 | | | 2 |
| 8 | Having the miss signal generated when a "miss" is detected | 2 | | | |
| 9 | Increment score upon catch signal | | 4 | | 4 |
| 10 | Decrement score upon miss signal and update display | | 2 | | 3 |
| 11 | Introduce different levels according to the score and make fruits fall faster | | 4 | | 5 |
| 12 | Change catcher colour depending on current level | 8 | 1 | 4 | |
| 13 | Display remaining lives on the LED | | 3 | | 4 |
| 14 | Pause function | 3 | | | |
| 15 | Display the highest score | | 6 | | 4 |
| 16 | Performance, testing and debugging | 7 | 8 | 5 | 10 |
| 17 | Writing the final report | 8 | 10 | 6 | 13 |

| | | | | | |
|---|---|---|---|---|---|
| Total number of hours spent | 63 | 52 | 39 | 59 | 213 |
| Cost per hour for each employee | $30.00 | | | | |
| Total labour cost in '000s (Hours x Cost per hour) | $1.89 | $1.56 | $1.17 | $1.77 | $6.39 |

Instrument and equipment costs :

| | |
|---|---|
| VGA monitor | $180.00 |
| ALTERA UP1 board | $300.00 |
| mouse | $30.00 |

**Total actual costs incurred for the whole project = $6,900.00**

## 8.0 Proposed Future Developments

Currently, the resource usage is 82% logic cells, and 75% memory. It should be noted that the initial components of the game were highly optimised. As a result of this, there was 45% Logic Cells free before levels, highest score and lives were implemented – which currently use 27%. Optimisation of the aforementioned features should free up at least 5%, which leaves sufficient resources for future developments. The following is a list of suggested possibilities:

- Making the fruits rotate as they fall (in 90 degree hops).
- Make different fruits fall from different x-positions.
- Have falling objects other than the fruits that must be avoided rather than caught.
- Draw splattered fruit if the player misses a catch.

These features could all potentially be implemented, however, especially RAM will become a serious constraint as it is not possible to achieve 100% use. Simple encoding schemes will need to be used to free up some RAM.

## 9.0 Conclusions

The game that has been developed has a reasonable level of AI, uses approximately 30 colours, and is highly interactive. During the initial stages of development, design decisions were heavily biased towards optimisation. As such, the principle of the game is quite simple, with more resource intensive features added at later stages.

The aim of the game is to catch as many falling fruits as possible in a bowl using the mouse. The features in the game include 5 levels of difficulty, 3-digit current score as well as the highest score recorded, display of the current level and attractive graphics. The number of remaining lives is displayed on the LEDs.

The concept of using a video buffer was abandoned, as it was found to be highly inefficient and resource intensive. Instead, memory is conserved by extensive use of sprites. The picture is created as the electron gun scans across the screen using combinational logic. The picture is changed using sequential logic.

The design solution uses 82% of the available logic cells, and 75% of RAM. Some of the recent features, such as displaying the current and highest scores, have not been optimised, and as such, there are still resources available for future developments. “Fruits’ Basket” fulfills all the required project specifications and was a fun and an interesting project to work on.

## List of References


1. Altera UP1 User Guide. Retrieved on 25, August 2003, from http://www.cecil.edu/interface/cwiframes.asp?UserID=tmab001
2. *Cecil Resources folder*. Retrieved on 17, August 2003, from http://www.cecil.edu/interface/cwiframes.asp?UserID=tmab001
3. COMPSYS 301 Computer Systems Engineering Design 3CS - Project 3 Handout. Distributed by Dr. Abbas Bigdeli on 14th August 2003.
4. Students at Georgia Tech University http://users.ece.gatech.edu/~hamblen/ALTERA/altera.htm
5. Thomas D.E. and Moorby P.R., *The Verilog hardware description language*, 5th Edition. Norwell, Mass. : Kluwer Academic Publishers, 2002.

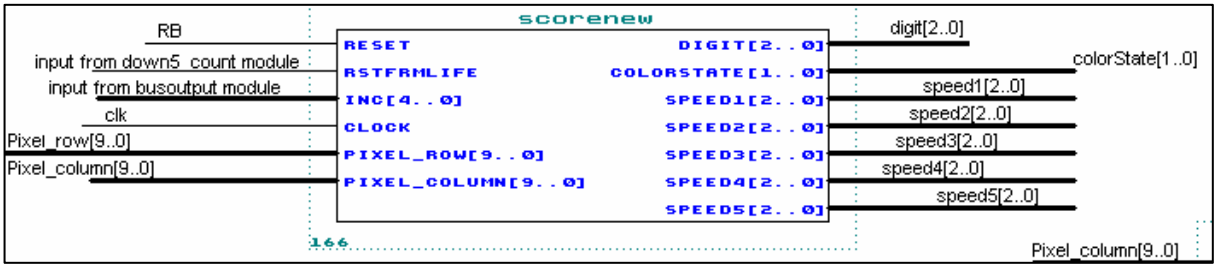
## Appendices

## Appendix I: A Hierarchical Overview of the Game

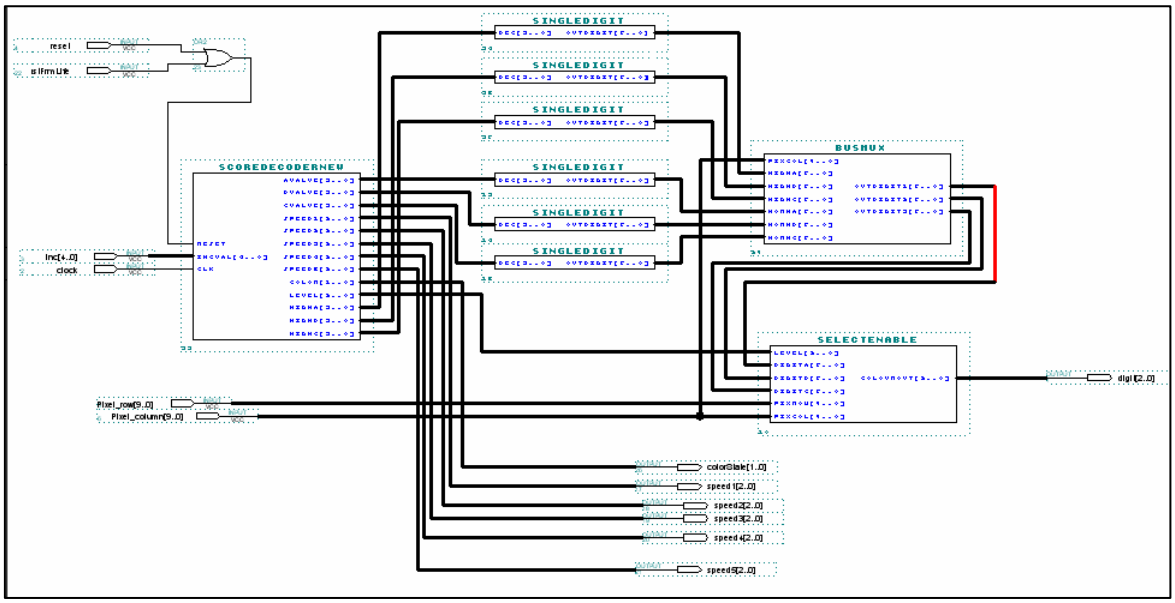
## Appendix II: Computation of the Current and Highest Score

The score is updated and the highest score is kept track of by the sub-modules in the *scorenew* module. How these sub-modules are connected within the *scorenew* module is shown below.

*Default Symbol for scorenew*




*Sub-modules within scorenew*




*Code Implementation Using Verilog HDL*

```
/*Code written for COMPSYS 301b – Engineering Design Project 3 by Thusitha Mabotuwana and Winnie Jiang of Auckland University.
Last modified on 08.10.03
 Score displaying on screen & Speed determining module */

module ScoreDecoderNew (reset,incVal,clk,Avalue,Bvalue,Cvalue,speed1,speed2,speed3,speed4,speed5, color, level);

        parameter N = 4;
        input reset;                                                    // Game
start/restart
        input clk;
        input [N:0] incVal;                                             // Score
assigning 5, 10 15 or 20

        output [1:0]color;                                              // Color changing
of the bowl at different levels
        output [N-1:0] Avalue;                                          // MSD digit
        output [N-1:0] Bvalue;
```

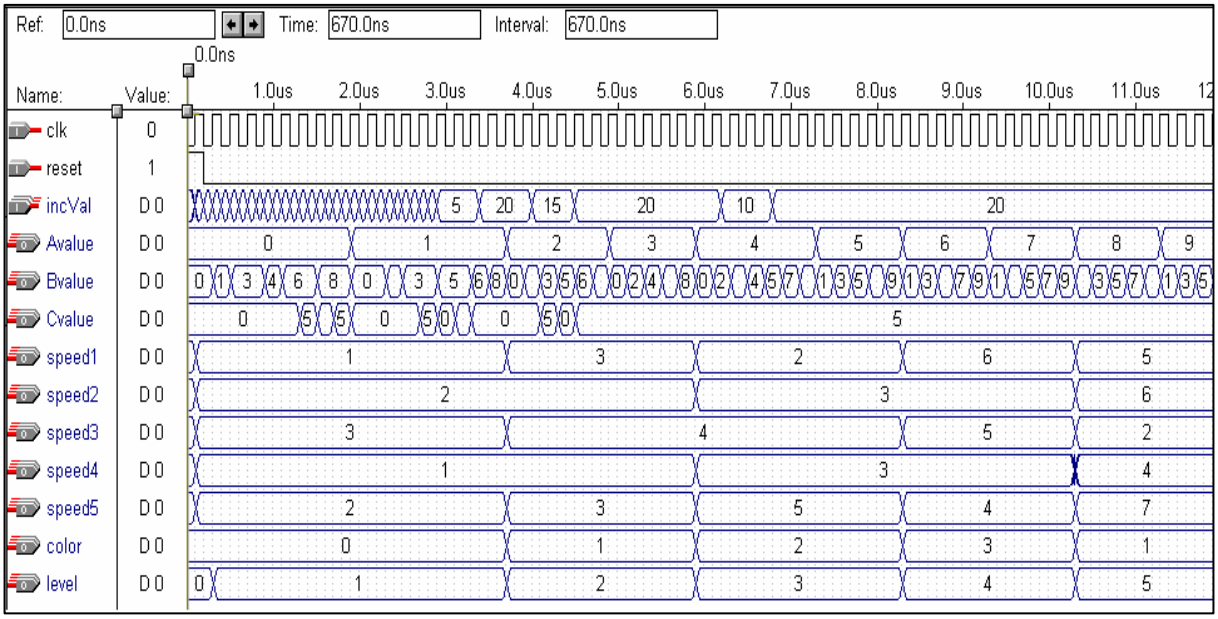
```
	output [N-1:0] Cvalue;
	output [N-2:0] speed1,speed2,speed3,speed4,speed5;		// Speed for the different fruits
	output [2:0] level;		// Level
controller

	reg [N-1:0] Avalue,highA,highB,highC,oldA,oldB,oldC;
	reg [N-1:0] Bvalue;
	reg [N-1:0] Cvalue;
	reg [N-2:0] speed1,speed2,speed3,speed4,speed5;
	reg [1:0] color;
	reg [2:0] level;

	always @ (posedge clk)
	begin
		if (reset==1)begin		// Starting of the
game, all values reset

				if (highA<oldA) begin //check for the digits that represents hundreds
					highA=oldA;
					highB=oldB;
					highC=oldC;
					end
				else if (highB<oldB && highA==oldA) begin //have to check for hundreds and tens. Eg.
630>620
					highB=oldB;
					highC=oldC;
					end
				else if (highC<oldC && highB==oldB && highB==oldB)
					highC=oldC;

				Avalue = 4'b0000;
				Bvalue = 4'b0000;
				Cvalue = 4'b0000;
				speed1=1;
				speed2=2;
				speed3=3;
				speed4=1;
				speed5=2;
				color = 2'b00;
			end


		else		//
	Depending on the score increment to assign
			begin		//
correct value to each digit A, B, C
			case(incVal)

				5: begin		/* Incoming
score is 5 */
					if (!Cvalue)		//
Digit C goes up in steps of 5
						Cvalue=5;
					else
						begin
						Cvalue = 0;
						Bvalue=Bvalue+1;		// B increments
when C = 10
						if (Bvalue==10)
							begin
							Avalue=Avalue+1;		// A increments
when B = 10
							Bvalue=0;
							end
						end
					end

				10: begin		/* Incoming
score is 10 */
					if (Bvalue<9)
						Bvalue=Bvalue+1;
					else
						begin
						Bvalue=0;
```

```
                    Avalue=Avalue+1;
                    end
            end

        15: begin                                   /* Incoming
score is 15 */
            if (Cvalue==0)
                    begin
                    Cvalue=Cvalue+5;                // if Digit C is 0
                    Bvalue=Bvalue+1;                // increment C by
5 and B by 1
                    if (Bvalue==10)
                            begin
                            Bvalue=0;
                            Avalue=Avalue+1;
                            end
                    end
            else                                            // if
Digit C is 5
                    begin
                    Cvalue=0;                       // set C to 0 and
increment B by 2
                    Bvalue=Bvalue+2;                // (i.e. 15 -> 30 )
                    if (Bvalue==10)
                            begin
                            Bvalue=0;
                            Avalue=Avalue+1;
                            end
                    else if(Bvalue==11)             // B overflows
                            begin                           // (i.e.
95 -> 115 )
                            Bvalue=1;
                            Avalue=Avalue+1;
                            end
                    end
            end

        20: begin                                   /* Incoming
score is 20 */
            Bvalue=Bvalue+2;
            if (Bvalue==10)
                    begin
                    Bvalue=0;
                    Avalue=Avalue+1;
                    end
            else if(Bvalue==11)
                    begin
                    Bvalue=1;
                    Avalue=Avalue+1;
                    end
            end
                                                    /*
Decrement score due to a miss signal */
        1: begin                                    // Score deducts
by 10 for each miss
            if (!Avalue && !Bvalue)                 // Current score
is < 10
                    Cvalue=0;

            else if (Bvalue!=0) begin               // Current score
is > 10 (i.e. Digit B != 0 )
                    case (Bvalue)
                            9: Bvalue=8;
                            8: Bvalue=7;
                            7: Bvalue=6;
                            6: Bvalue=5;
                            5: Bvalue=4;
                            4: Bvalue=3;
                            3: Bvalue=2;
                            2: Bvalue=1;
                            1: Bvalue=0;
                    endcase
                    end
```

```
                else if(!Bvalue) begin                          // Current score
is < 100
                    Bvalue=9;                                   // Start to
decrement Digit B by 10
                    case (Avalue)                                       //
Decrement Digit A by 1
                        9: Avalue=8;
                        8: Avalue=7;
                        7: Avalue=6;
                        6: Avalue=5;
                        5: Avalue=4;
                        4: Avalue=3;
                        3: Avalue=2;
                        2: Avalue=1;
                        1: Avalue=0;
                    endcase
                    end
                end
            endcase

            oldA=Avalue; //have to keep track of current score so that we can update highest later
            oldB=Bvalue;
            oldC=Cvalue;

            if (Avalue==10) begin                               /* Stage
Determining Section */
                Avalue=0;                                       // Score
overflows thus resets score & level
                Bvalue=0;                                       // Stage
increments every 200 points
                Cvalue=0;                                       // Bowl changes
color according to different stages
                color = 2'b00;
                level=0;
                end
            if (Avalue==0 || Avalue==1) begin                   // Stage 1
                speed1=1;
                speed2=2;
                speed3=3;
                speed4=1;
                speed5=2;
                color = 2'b00;
                level=1;
                end
            else if (Avalue==2 || Avalue==3) begin              // Stage 2
                speed1=3;
                speed2=2;
                speed3=4;
                speed4=1;
                speed5=3;
                color = 2'b01;
                level=2;
                end
            else if (Avalue==4 || Avalue==5) begin              // Stage 3
                speed1=2;
                speed2=3;
                speed3=4;
                speed4=3;
                speed5=5;
                color = 2'b10;
                level=3;
                end
            else if (Avalue==6 || Avalue==7) begin              // Stage 4
                speed1=6;
                speed2=3;
                speed3=5;
                speed4=3;
                speed5=4;
                color = 2'b11;
                level=4;
                end
            else if (Avalue==8 || Avalue==9) begin              // Stage 5
                speed1=5;
                speed2=6;
```

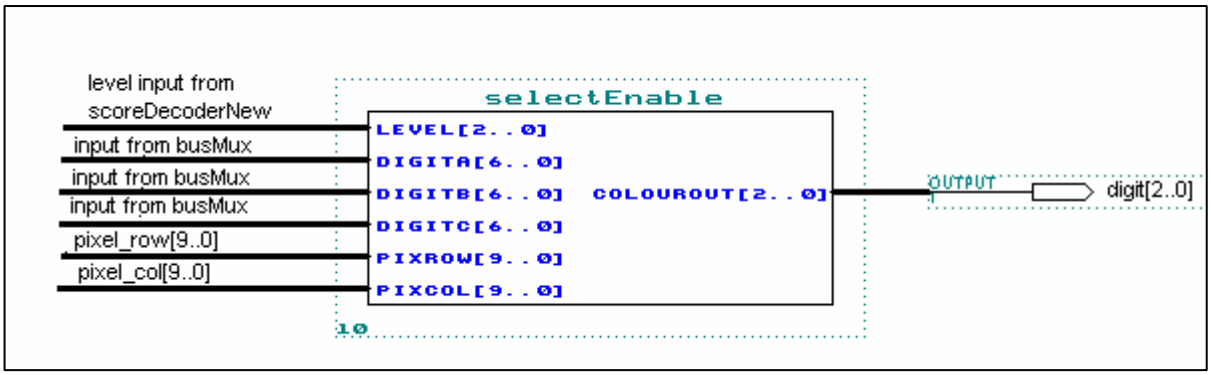
```
                                        speed3=2;
                                        speed4=4;
                                        speed5=7;
                                        color = 2'b01;
                                        level=5;
                                        end
                                end
        end
endmodule
```

## *Simulation for ScoreDecoderNew*



## Appendix III: Displaying and Updating the Score on the screen

The *selectenable* module shown below is used to check for the current pixel row and column positions and output the relevant colour after checking for the required segments that need to be displayed. It is also used to display the highest score and the current level of difficulty. The *Digit_A, Digit_B* and *Digit_C* inputs contain the segments of the digit that has to be displayed.




*Code Implementation Using Verilog HDL*

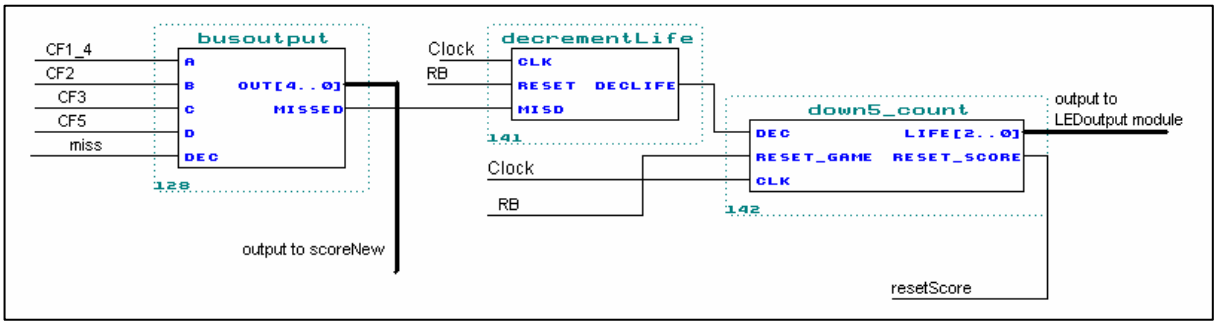
```
/*Code written for COMPSYS 301b – Engineering Design Project 3 by Thusitha Mabotuwana and Winnie Jiang of Auckland University.
Last modified on 08.10.03
 Drawing the 3 Digits & Level on the screen */

module selectEnable(level,digitA,digitB,digitC,pixRow,pixCol,colourOut);
        parameter N=10;
        input [2:0] level;
        input [N-1:0] pixRow, pixCol;
        input [6:0]digitA,digitB,digitC;
        output [2:0] colourOut;
        reg [2:0] colourOut;

        // Digit segements for the different level
        wire segA= (level==2 || level==3 || level==5);
        wire segB= (level==1 || level==2 || level==3 || level==4);
        wire segC= (level==1 || level==3 || level==4 || level==5);
        wire segD= (level==2 || level==3 || level==5);
        wire segE= (level==2);
        wire segF= (level==4 || level==5);
        wire segG= (level==2 || level==3 || level==4 || level==5);

        always @ (pixRow or pixCol)
                begin
                if (pixRow <4)  // checks for all the A segments
                        begin
                                if (pixCol>548 && pixCol<564 && digitA[0]==1)
                                        colourOut=3'b100;
                                else if (pixCol>580 && pixCol<596 && digitB[0]==1)
                                        colourOut=3'b100;
                                else if (pixCol>612 && pixCol<628 && digitC[0]==1)
                                        colourOut=3'b100;
                                else
                                        colourOut=3'b000;

                        end

                else if (pixRow >20 && pixRow <24)  // checks for all the G segments
                        begin
                                if (pixCol>548 && pixCol<564 && digitA[6]==1)
                                        colourOut=3'b100;
                                else if (pixCol>580 && pixCol<596 && digitB[6]==1)
                                        colourOut=3'b100;
                                else if (pixCol>612 && pixCol<628 && digitC[6]==1)
```

```
                        colourOut=3'b100;
                else
                        colourOut=3'b000;
        end

else if (pixRow >40 && pixRow <44)  //checks for all the D segments
        begin
                if (pixCol>548 && pixCol<564 && digitA[3]==1)
                        colourOut=3'b100;
                else if (pixCol>580 && pixCol<596 && digitB[3]==1)
                        colourOut=3'b100;
                else if (pixCol>612 && pixCol<628 && digitC[3]==1)
                        colourOut=3'b100;
                else
                        colourOut=3'b000;
        end

else if (pixRow> 4 && pixRow <20)  // checks for all the F and B segments
        begin
                if (pixCol>544 && pixCol<548 && digitA[5]==1)                   //F
                        colourOut=3'b100;
                else if (pixCol>576 && pixCol<580 && digitB[5]==1)
                        colourOut=3'b100;
                else if (pixCol>608 && pixCol<612 && digitC[5]==1)
                        colourOut=3'b100;

                else if (pixCol>564 && pixCol<568 && digitA[1]==1)//B
                        colourOut=3'b100;
                else if (pixCol>596 && pixCol<600 && digitB[1]==1)
                        colourOut=3'b100;
                else if (pixCol>628 && pixCol<632 && digitC[1]==1)
                        colourOut=3'b100;
                else
                        colourOut=3'b000;
        end

else if (pixRow >24 && pixRow <40)  //checks for all the E and C segments
        begin
                if (pixCol>544 && pixCol<548 && digitA[4]==1)                   //E
                        colourOut=3'b100;
                else if (pixCol>576 && pixCol<580 && digitB[4]==1)
                        colourOut=3'b100;
                else if (pixCol>608 && pixCol<612 && digitC[4]==1)
                        colourOut=3'b100;

                else if (pixCol>564 && pixCol<568 && digitA[2]==1)//C
                        colourOut=3'b100;
                else if (pixCol>596 && pixCol<600 && digitB[2]==1)
                        colourOut=3'b100;
                else if (pixCol>628 && pixCol<632 && digitC[2]==1)
                        colourOut=3'b100;
                else
                        colourOut=3'b000;
        end

// Displaying of Level at the bottom of the screen
//checking for the position (rows & columns) of where the level is going to appear

// Drawing the "L" shape on the screen
else if (((pixRow >450 && pixRow <458) || (pixRow >462 && pixRow <470)) && (pixCol>14 && pixCol<18))
        colourOut=3'b100;
else if (pixRow >470 && pixRow <474 && pixCol>18 && pixCol<26)
        colourOut=3'b100;

//Drawing digit from 1 ~ 5 on the screen

else if (pixRow >446 && pixRow <450 && pixCol>34 && pixCol<42 && segA)
        colourOut=3'b100;
else if (pixRow >450 && pixRow <458 && pixCol>42 && pixCol<46 && segB)
        colourOut=3'b100;
else if (pixRow >462 && pixRow <470 && pixCol>42 && pixCol<46 && segC)
        colourOut=3'b100;
else if (pixRow >470 && pixRow <474 && pixCol>34 && pixCol<42 && segD)
        colourOut=3'b100;
```

```
                else if (pixRow >462 && pixRow <470 && pixCol>30 && pixCol<34 && segE)
                        colourOut=3'b100;
                else if (pixRow >458 && pixRow <462 && pixCol>34 && pixCol<42 && segG)
                        colourOut=3'b100;
                else if (pixRow >450 && pixRow <458 && pixCol>30 && pixCol<34 && segF)
                        colourOut=3'b100;
                else
                        colourOut=3'b000;

        end
endmodule
```

## Appendix IV: Displaying Number of Remaining Lives on the LED Display

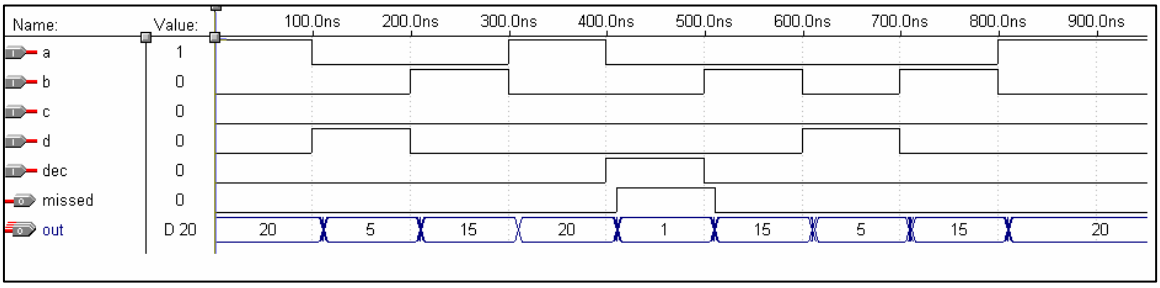
Each time a fruit is missed, a counter in *decrementLife* will increment and upon reaching 5, it will generate a signal for *down5_count* which will then decrement the remaining lives by one, and display this new value on the LEDs. This appendix contains the code implementation and the default symbols generated and shows how separate modules are connected to generate the final result.
Shown below is how the *busoutput*, *decrementLife* and *doown5_count* modules are connected to generate the remaining number of lives.




### *Code Implementation Using Verilog HDL*

```
/*Code written for COMPSYS 301b – Engineering Design Project 3 by Thusitha Mabotuwana and Winnie Jiang of Auckland University.
Last modified on 10.10.03
Determining the Calculation of Score display */

module busoutput(a,b,c,d,dec,out,missed);
        input a,b,c,d,dec;
        output [4:0] out;
        output missed;

        assign out=(a==1)? 5'b10100:    // 20 points
                                        (b==1)? 5'b01111:     // 15 points
                                        (c==1)? 5'b01010:     // 10 points
                                        (d==1)? 5'b00101:     // 5 points
                                        (dec==1)? 5'b00001:  //sending a miss signal
                                        5'b00000;

        assign missed=dec;
endmodule
```

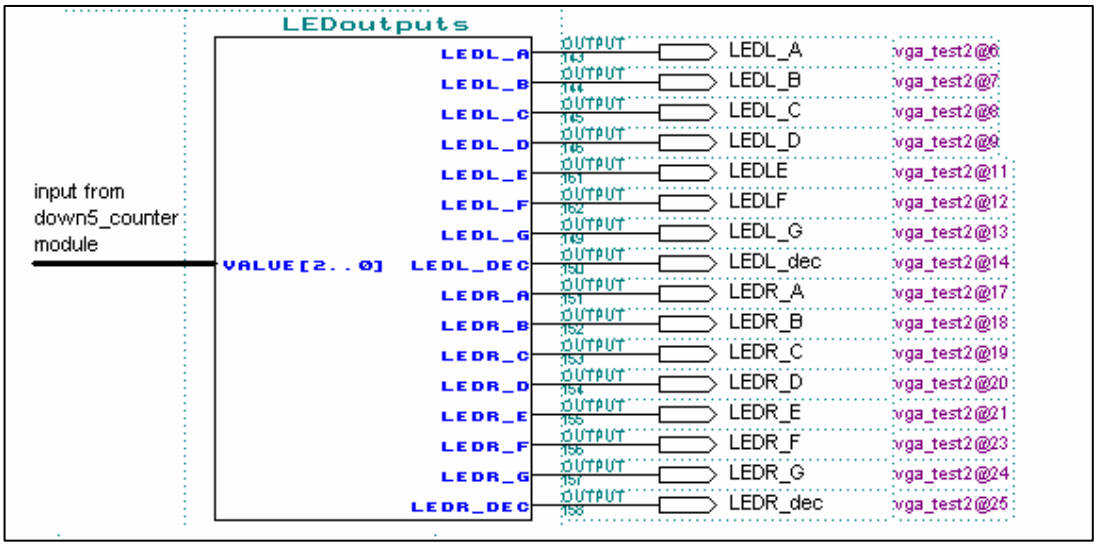
### *Simulation*



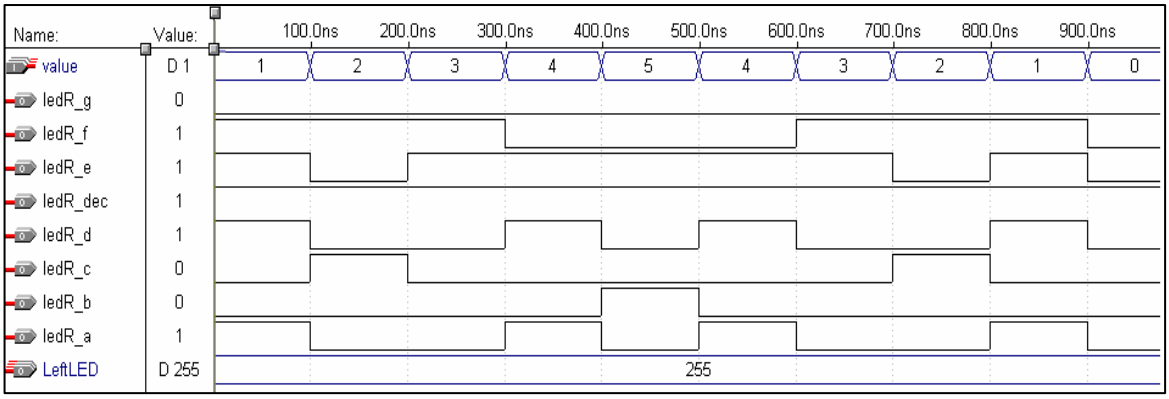
## *Code for LEDoutputs module*

```
/*Code written for COMPSYS 301b – Engineering Design Project 3 by Thusitha Mabotuwana and Winnie Jiang of Auckland University.
Last modified on 11.10.03
 The displaying of Life on the LED provided on UP1 board */

module LEDoutputs
(value,ledL_a,ledL_b,ledL_c,ledL_d,ledL_e,ledL_f,ledL_g,ledL_dec,ledR_a,ledR_b,ledR_c,ledR_d,ledR_e,ledR_f,ledR_g,ledR_dec);

        input [2:0] value;

        output
ledL_a,ledL_b,ledL_c,ledL_d,ledL_e,ledL_f,ledL_g,ledL_dec,ledR_a,ledR_b,ledR_c,ledR_d,ledR_e,ledR_f,ledR_g,ledR_dec;

        wire [6:0] ledsegs;

        // Only digit 1~5 is displayed, thus 6~0 format can be ignored.
        assign ledsegs = (value==0)? 7'b1000000:
                                                (value==1)? 7'b1111001:
                                                (value==2)? 7'b0100100:
                                                (value==3)? 7'b0110000:
                                                (value==4)? 7'b0011001:
                                                7'b0010010;                                     //vallue==5

        assign ledL_a=1;                                                                        //not using the left LED. ALL
GROUNDED
        assign ledL_b=1;
        assign ledL_c=1;
        assign ledL_d=1;
        assign ledL_e=1;
        assign ledL_f=1;
        assign ledL_g=1;
        assign ledL_dec=1;


        assign ledR_a=ledsegs[0];                                                               // Right digit LED
        assign ledR_b=ledsegs[1];
        assign ledR_c=ledsegs[2];
        assign ledR_d=ledsegs[3];
        assign ledR_e=ledsegs[4];
        assign ledR_f=ledsegs[5];
        assign ledR_g=ledsegs[6];
        assign ledR_dec=1;                      // Grounded decimal point

endmodule
```

## *Default Symbol for LEDoutputs*



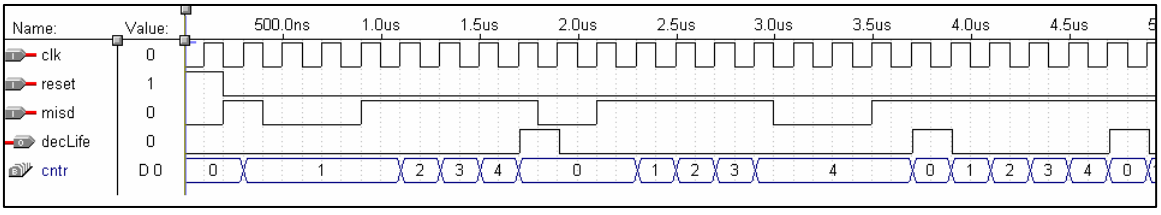
## *Simulation*




## *Code for decrementLife module*

```
/*Code written for COMPSYS 301b – Engineering Design Project 3 by Thusitha Mabotuwana and Winnie Jiang of Auckland University.
Last modified on 11.10.03
Life decrementing module */

module decrementLife(clk,reset,misd,decLife);
        input reset,clk,misd;                           // Life decrements every misses of 5 fruits
        output decLife;
        reg decLife;
        reg [3:0] cntr;

        always @(posedge clk) begin
                if (reset==1) begin
                        cntr = 4'b0000;
                        decLife = 1'b0;                 // decrement life signal disabled
                        end

                else if (misd == 1) begin               // A miss is detected
                        cntr = cntr + 1;

                        if (cntr == 4'b0101) begin      // 5 misses, decrements a life
                                cntr= 4'b0000;
                                decLife= 1'b1;
                                end

                        else
                                decLife = 0;                    // not 5 misses yet
                        end

                else
                        decLife = 0;                    // no misses
                end
endmodule
```

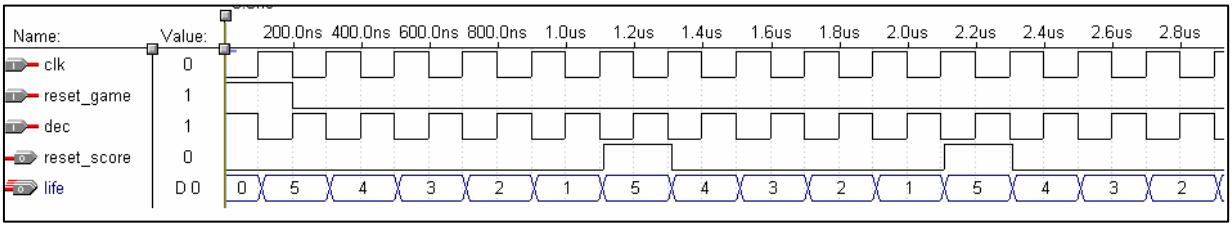
## *Simulation*



## *Code for down5_count module*

```
/*Code written for COMPSYS 301b – Engineering Design Project 3 by Thusitha Mabotuwana and Winnie Jiang of Auckland University.
Last modified on 11.10.03
Remaining of lives of game. */

module down5_count(dec,reset_game,clk,life,reset_score);
        input dec,clk,reset_game;
        output [2:0] life;
        output reset_score;
        reg [2:0] life;
        reg reset_score;

        always @(posedge clk) begin

                if (reset_game==1) begin                    // Game starts with 5 lives
                        life=3'b101;
                        reset_score=0;
                        end
                else if (dec==1) begin
                        reset_score = 0;
                        case(life)                          // Life decrements using case statements
                                5: life=3'b100;
                                4: life=3'b011;
                                3: life=3'b010;
                                2: life=3'b001;
                                1: begin
                                        life = 3'b101;      // life resets when user has 0 life left
                                        reset_score = 1;
                                        end
                                endcase
                        end
                end
endmodule
```

## *Simulation*



## Appendix V: The .rpt File Generated by Altera MaxPlus Showing Resource Usage of the Final Design

Shown below is part of the .rpt generated by Altera MaxPlus10.1 showing the main resource usage of the final implementation. However the sections following the 'File Hierachy' haven't been included since the whole report is around 140 pages even with a font size of 8 Times New Roman.

**Project Informationh:\project 3_09_10_03\project 3_09_10_03\vga_test\vga_test2.rpt**

**MAX+plus II Compiler Report File**
**Version 10.1 06/12/2001**
**Compiled: 10/13/2003 17:19:21**



******* Project compilation was successful**

**** DEVICE SUMMARY ****

| Chip/ POF | Device | Input Pins | Output Pins | Bidir Pins | Memory Bits | % Utilized | LCs | % Utilized |
|---|---|---|---|---|---|---|---|---|
| vga_test2 | | | | | | | | |
| | EPF10K20RC240-4 | 1 | 21 | 2 | 9216 | 75 % | 947 | 82 % |
| User Pins: | | 1 | 21 | 2 | | | | |

**§**
**Project Informationh:\project 3_09_10_03\project 3_09_10_03\vga_test\vga_test2.rpt**

**** PROJECT COMPILATION MESSAGES ****

**Warning: Flipflop '|grassLogic:119|:51' stuck at GND**
**Warning: Node 'LEDL_E' has assignments but doesn't exist or is a primitive array -- edit the project's ACF to fix the problem**
**Warning: Node 'LEDL_F' has assignments but doesn't exist or is a primitive array -- edit the project's ACF to fix the problem**
**Warning: Node '|21|LEDL_E' has assignments but doesn't exist or is a primitive array -- edit the project's ACF to fix the problem**

**§**
**Project Informationh:\project 3_09_10_03\project 3_09_10_03\vga_test\vga_test2.rpt**

**** PIN/LOCATION/CHIP ASSIGNMENTS ****

| User | Actual Assignments |
|---|---|

```
Assignments   (if different)     Node Name

vga_test2@238                    Blue
vga_test2@91                     Clock
vga_test2@237                    Green
vga_test2@240                    Horiz_Sync
vga_test2@6                     LEDL_A
vga_test2@7                     LEDL_B
vga_test2@8                     LEDL_C
vga_test2@9                     LEDL_D
vga_test2@14                     LEDL_dec
vga_test2@11                     LEDLE
vga_test2       ---------       LEDL_E
vga_test2@21    ---------        LEDL_E
vga_test2@12                    LEDLF
vga_test2@23    ---------        LEDL_F
vga_test2@13                    LEDL_G
vga_test2@17                    LEDR_A
vga_test2@18                    LEDR_B
vga_test2@19                    LEDR_C
vga_test2@20                    LEDR_D
vga_test2@25                    LEDR_dec
vga_test2@21                    LEDR_E
vga_test2@23                    LEDR_F
vga_test2@24                    LEDR_G
vga_test2@30                    MOUSE_CLK
vga_test2@31                    MOUSE_DATA
vga_test2@236                    Red
vga_test2@239                    Vert_sync
vga_test2       ---------       |21|LEDL_E

§
Project Informationh:\project 3_09_10_03\project 3_09_10_03\vga_test\vga_test2.rpt

** STATE MACHINE ASSIGNMENTS **


|MOUSE:56|mouse_state: MACHINE
      OF BITS (
        |MOUSE:56|mouse_state~6,
        |MOUSE:56|mouse_state~5,
        |MOUSE:56|mouse_state~4,
        |MOUSE:56|mouse_state~3,
        |MOUSE:56|mouse_state~2
      )
      WITH STATES (
             INHIBIT_TRANS = B"00000",
              LOAD_COMMAND = B"11000",
             LOAD_COMMAND2 = B"10100",
          WAIT_OUTPUT_READY = B"10010",
              WAIT_CMD_ACK = B"10001",
             INPUT_PACKETS = B"10000"
);


§
Project Informationh:\project 3_09_10_03\project 3_09_10_03\vga_test\vga_test2.rpt

** EMBEDDED ARRAYS **


|cloud:97|cloudSprite:3|lpm_rom:lpm_rom_component|altrom:srom|content: MEMORY (
          width       =    3;
          depth       =  512;
          segmentsize  =  512;
          mode        = MEM_READONLY#MEM_INITIALIZED;
          file        = "H:/Project 3_09_10_03/Project 3_09_10_03/VGA_test/cloudMirror.mif";
      )
      OF SEGMENTS (
          |cloud:97|cloudSprite:3|lpm_rom:lpm_rom_component|altrom:srom|segment0_2,
          |cloud:97|cloudSprite:3|lpm_rom:lpm_rom_component|altrom:srom|segment0_1,
          |cloud:97|cloudSprite:3|lpm_rom:lpm_rom_component|altrom:srom|segment0_0
);
```

```
|flowers:98|grassSprite:3|lpm_rom:lpm_rom_component|altrom:srom|content: MEMORY (
        width       =    3;
        depth       =  256;
        segmentsize  =  256;
        mode        = MEM_READONLY#MEM_INITIALIZED;
        file        = "H:/Project 3_09_10_03/Project 3_09_10_03/VGA_test/grass.mif";
    )
    OF SEGMENTS (
        |flowers:98|grassSprite:3|lpm_rom:lpm_rom_component|altrom:srom|segment0_2,
        |flowers:98|grassSprite:3|lpm_rom:lpm_rom_component|altrom:srom|segment0_1,
        |flowers:98|grassSprite:3|lpm_rom:lpm_rom_component|altrom:srom|segment0_0
);


|monkey:99|bowl4ColorSprite:11|lpm_rom:lpm_rom_component|altrom:srom|content: MEMORY (
        width       =    3;
        depth       = 1024;
        segmentsize  = 1024;
        mode        = MEM_READONLY#MEM_INITIALIZED;
        file        = "H:/Project 3_09_10_03/Project 3_09_10_03/VGA_test/bowlDither3232.mif";
    )
    OF SEGMENTS (
        |monkey:99|bowl4ColorSprite:11|lpm_rom:lpm_rom_component|altrom:srom|segment0_2,
        |monkey:99|bowl4ColorSprite:11|lpm_rom:lpm_rom_component|altrom:srom|segment0_1,
        |monkey:99|bowl4ColorSprite:11|lpm_rom:lpm_rom_component|altrom:srom|segment0_0
);


|grapesSprite:100|lpm_rom:lpm_rom_component|altrom:srom|content: MEMORY (
        width       =    3;
        depth       =  256;
        segmentsize  =  256;
        mode        = MEM_READONLY#MEM_INITIALIZED;
        file        = "H:/Project 3_09_10_03/Project 3_09_10_03/VGA_test/grapes.mif";
    )
    OF SEGMENTS (
        |grapesSprite:100|lpm_rom:lpm_rom_component|altrom:srom|segment0_2,
        |grapesSprite:100|lpm_rom:lpm_rom_component|altrom:srom|segment0_1,
        |grapesSprite:100|lpm_rom:lpm_rom_component|altrom:srom|segment0_0
);


|berriesSprite:101|lpm_rom:lpm_rom_component|altrom:srom|content: MEMORY (
        width       =    3;
        depth       =  256;
        segmentsize  =  256;
        mode        = MEM_READONLY#MEM_INITIALIZED;
        file        = "H:/Project 3_09_10_03/Project 3_09_10_03/VGA_test/blueberries.mif";
    )
    OF SEGMENTS (
        |berriesSprite:101|lpm_rom:lpm_rom_component|altrom:srom|segment0_2,
        |berriesSprite:101|lpm_rom:lpm_rom_component|altrom:srom|segment0_1,
        |berriesSprite:101|lpm_rom:lpm_rom_component|altrom:srom|segment0_0
);


|cherriesSprite:105|lpm_rom:lpm_rom_component|altrom:srom|content: MEMORY (
        width       =    3;
        depth       =  256;
        segmentsize  =  256;
        mode        = MEM_READONLY#MEM_INITIALIZED;
        file        = "H:/Project 3_09_10_03/Project 3_09_10_03/VGA_test/cherries.mif";
    )
    OF SEGMENTS (
        |cherriesSprite:105|lpm_rom:lpm_rom_component|altrom:srom|segment0_2,
        |cherriesSprite:105|lpm_rom:lpm_rom_component|altrom:srom|segment0_1,
        |cherriesSprite:105|lpm_rom:lpm_rom_component|altrom:srom|segment0_0
);
```

```
|strawberrySprite:106|lpm_rom:lpm_rom_component|altrom:srom|content: MEMORY (
        width       =    3;
        depth       =  256;
        segmentsize  =  256;
        mode        = MEM_READONLY#MEM_INITIALIZED;
        file        = "H:/Project 3_09_10_03/Project 3_09_10_03/VGA_test/strawberry.mif";
    )
    OF SEGMENTS (
        |strawberrySprite:106|lpm_rom:lpm_rom_component|altrom:srom|segment0_2,
        |strawberrySprite:106|lpm_rom:lpm_rom_component|altrom:srom|segment0_1,
        |strawberrySprite:106|lpm_rom:lpm_rom_component|altrom:srom|segment0_0
);


|lianaSprite:107|lpm_rom:lpm_rom_component|altrom:srom|content: MEMORY (
        width       =    3;
        depth       =  256;
        segmentsize  =  256;
        mode        = MEM_READONLY#MEM_INITIALIZED;
        file        = "H:/Project 3_09_10_03/Project 3_09_10_03/VGA_test/liana2.mif";
    )
    OF SEGMENTS (
        |lianaSprite:107|lpm_rom:lpm_rom_component|altrom:srom|segment0_2,
        |lianaSprite:107|lpm_rom:lpm_rom_component|altrom:srom|segment0_1,
        |lianaSprite:107|lpm_rom:lpm_rom_component|altrom:srom|segment0_0
);
```

## Appendix VI: Project Schedule Developed Using Microsoft Project