Marienbad Game

EE552 Final Project Report

Stangeland, Duane 357288
McDermott, Ashley 231634
Koziar, Kory 252655

1 Executive Summary

For this project an Altera Flex 10K20RC240-4 equipped UP1 board was used to implement a Marienbad game. The UP1 board includes VGA signal generation capabilities as well as a PS2 communications port. Also included on the UP1 board are multiple LED's, 7-segment displays, push button switches, as well as DIP switches.

A game of skill was implemented in which 2 players are pitted head-to-head, or the player can try their luck against the computer, to compete for the prize of the last pebble.

The game has evolved twice from its original implementation. In the beginning the game was implemented using a keypad and 7-segment displays. The user was asked to enter the number of pebbles they would like to take using the keypad and the number of remaining pebbles were displayed on the 7-segment displays, up to a maximum of 99.

The first design change came about when the 7-segment displays were replaced with VGA graphics. A VGA monitor is now used to display the game board containing 503 pebbles at a resolution of 640x480 pixels.

The second change came from the addition of the mouse replacing the push buttons. The mouse now selects the number of pebbles that a player would like to take for their turn, while the push buttons are used to reset the game, and invoke a computer player.

The game was produced using behavioral VHDL. The complete testing and development of the game was performed on the Altera Max plus II development platform.

This project report details the design of the Marienbad game. Game rules are explained and specific modules of the VHDL design are investigated in this report. General design and interface details are also discussed. In addition, the steps taken to verify proper operation of the major components of the game are also included.

2 IC Data Sheet

Figure 1 shows usage statistics for the resources of the Flex 10K20 chip that were used in this project. At one time the design used more than 95% of the available logic cells. After hand optimization of the VHDL code was performed the total number of logic cells used decreased to 82%.

.

Total dedicated input pins used:	1/6	(16%)
Total I/O pins used:	8/183	(4%)
Total logic cells used:	952/1152	(82%)
Total embedded cells used:	6/48	(33%)
Total EABs used:	2/6	(33%)
Average fan-in:	3.55/4	(88%)
Total fan-in:	3386/4608	(73%)

Figure 1 Project statistics from project.rpt

The external interface block diagram for this project (Figure 2) is very simple as the design uses only 3 external input connections, and all output is provided via 5 video signals. Inputs include 2 push button switches which are used to reset the game and to invoke the computer player. The third input is used to read input from the mouse. The system clock is included in the block diagram as it is used to provide various timing sources for mouse data capture and video signal generation.

As mentioned above, all output for the game is provided by a VGA monitor which only requires 5 signals from the 10K20.

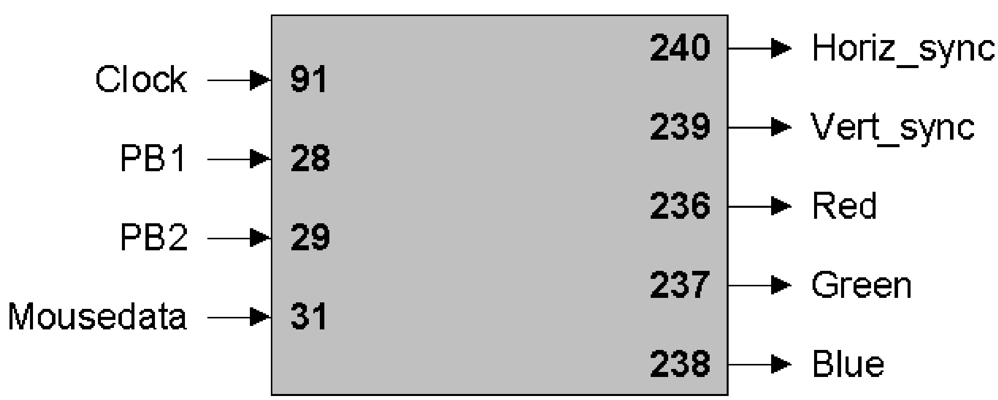

Figure 2 UP1 pin connections

Table of Contents

Table of Figures

3 The Game

Marienbad is a game of mathematical skill. The layout of the game may seem simple, and the game may seem silly, but that intuition may not last long once the game begins, and you realize your opponent is in a winning position. You can only pray your opponent does not see the winning move.

3.1 Rules

When the game begins, the first player has a special move available (**First Move Rule**). He or she may take any of the 503 pebbles on the playing board, except the final pebble. Doing so, they risk placing themselves in a losing position or the first player can act conservatively to ensure that the next player cannot win on the next move.

When player 1 has selected the number of pebbles that they wish to take, player 2 is then left to decide what to do. After the first move the rules of the game remain constant. The valid moves that are available are to select any number of pebbles between one and two times the amount that the previous player took.

Game play continues to alternate between players until the last pebble is taken. The player who takes the last pebble is declared the winner, and the game is over. The game must then be reset to continue.

If the player vs. computer option is selected, the second player is turned over to computer control. Once the human player has selected their pebbles to take, the computer will wait for the player to click the mouse button before taking the number of pebbles the computer would like. This delay was added to allow the user to view the playing field and see how many pebbles the computer takes. When the system is reset, and the computer player option is selected, player 1 will always go first followed by the computer. The computer option can be selected at any point in game play, and will take over the role of player 2.

To make the counting of the pebbles easier it should be noted that there are 36 pebbles per line, except on the first line, where there is only 35 pebbles. There are 14 rows of pebbles. The total number of pebbles is 503.

3.2 Interface

The external interface to the Marienbad game utilizes a mouse and a VGA monitor. The game board is displayed on the monitor, and players use the mouse to select the number of pebbles they would like to take.

To select a pebble, position the mouse over the pebble and click the left mouse button. If the move is determined to be valid, the screen will update, and control will be turned over to the next player. If the move is not valid, no change will occur on the display. A valid move is determined using the rules as stated above in §3.1. If the mouse is clicked and no display update occurs, the move is either too many pebbles are being taken, or the mouse was clicked where no pebbles exist.

To reset the system, the right hand push button must be pressed on the Altera UP1 board. If the system is reset, all previous game information is lost. If the player wishes to play against the computer or if player 2 quits, the computer can be activated using the right hand push button on the UP1 board. When the system is reset, the game defaults to player vs. player mode, and the computer must be re-activated. For more information on the push buttons, please see §9.

4 Main Project Hierarchy

4.1 Introduction

Project.vhd is the top-level VHDL module in this design. Included in this module is the constant's library. The constant's library is used throughout the design and is called My_Constants. Also included in this top-level module is the logical link between the game and the Altera UP1 board. This external interface is defined in the project.acf file. The Project module also acts as a gateway to the rest of the design hierarchy.

The Project module links to the Display module, where further propagation of external interface signals takes place. The main inputs to the system from the UP1 board are PB1, PB2, Clock, and the mouse data signal. PB1 and PB2 are the FLEX push buttons provided on the Altera UP1 board, Clock is the 25.175MHz system clock, and MouseData is the information obtained from the PS/2 port from the mouse.

The interface between the Altera UP1 board, the Project module, and the remaining modules of the code is given in Figure 3. Following Figure 3, a text description of the signals passed between modules is given in Figure 4 and Figure 5.

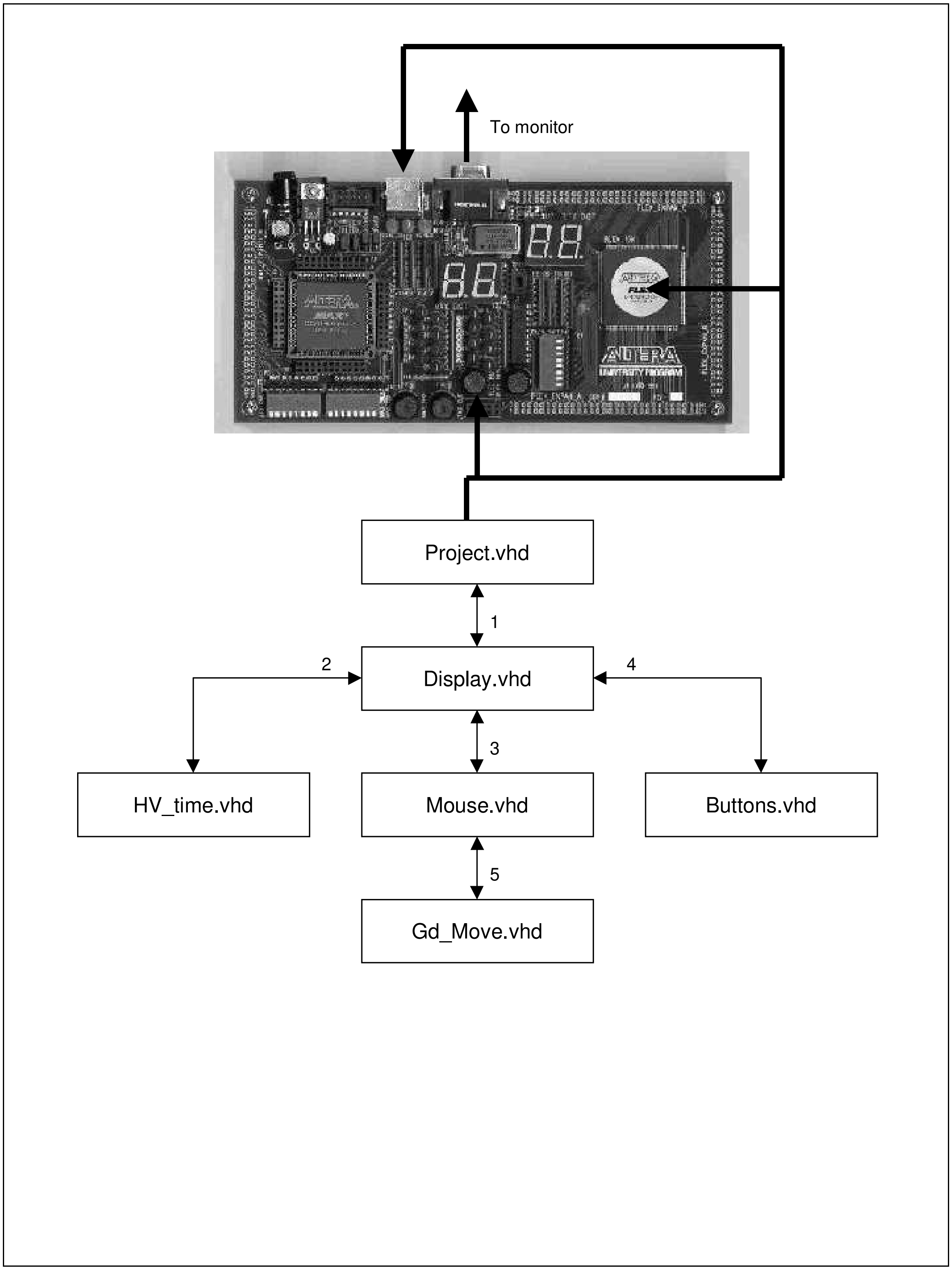

Figure 3 Graphical representation of design hierarchy.

```
0.  Project <-> Altera UP1

        signal PB1              : in std_logic;   -- Push Button 1,       pin #28
        signal PB2              : in std_logic;   -- Push Button 2,       pin #29
        signal Clock            : in std_logic;   -- System clock,        pin #91
        signal MouseData        : in std_logic;   -- Mouse data,          pin #31

        signal Red              : out std_logic;  -- VGA red,             pin #236
        signal Green            : out std_logic;  -- VGA green,           pin #237
        signal Blue             : out std_logic;  -- VGA blue,            pin #237
        signal Horiz_sync       : out std_logic;  -- Horiz signal,        pin #240
        signal Vert_sync        : out std_logic   -- Vert signal,         pin #239


1.  Project <-> Display

        signal PB1              : in std_logic;  -- Push Button 1
        signal PB2              : in std_logic;  -- Push Button 2
        signal Clock            : in std_logic;  -- System Clock
        signal MouseData        : in std_logic;  -- Data from the mouse

        signal Red              : out std_logic; -- Red for VGA
        signal Green            : out std_logic; -- Green for VGA
        signal Blue             : out std_logic; -- Blue for VGA
        signal Horiz_sync       : out std_logic; -- Horizontal for VGA
        signal Vert_sync        : out std_logic  -- Vertical for VGA

2.  Display <-> HV_Time

        signal Clock            : in std_logic;
        signal Power_On         : in std_logic;

        signal Horiz_sync       : out std_logic;
        signal Vert_sync        : out std_logic;
        signal Video_on_H       : out std_logic;
        signal Video_on_V       : out std_logic;

        signal Pixel_Col_Count  : buffer std_logic_vector(5 Downto 0);
        signal Pixel_Row_Count  : buffer std_logic_vector(5 Downto 0);
        signal Col_Address      : buffer std_logic_vector(5 Downto 0);
        signal Row_Address      : buffer std_logic_vector(5 Downto 0)
```

Figure 4 Text description of module interfaces.

```
3.  Display <-> Mouse

      signal Clock                : in std_logic;
      signal MouseData            : in std_logic;
      signal Power_On             : in std_logic;
      signal Computer_Play        : in std_logic;

      signal PointerX             : out std_logic_vector(6 downto 0);
      signal PointerY             : out std_logic_vector(6 downto 0);

      signal Player               : buffer std_logic;
      signal MouseX               : buffer std_logic_vector(6 downto 0);
      signal MouseY               : buffer std_logic_vector(6 downto 0)


4.  Display <-> Buttons

      signal Clock                : in std_logic;
      signal Power_On             : in std_logic;
      signal PB1                  : in std_logic;
      signal PB2                  : in std_logic;

      signal Reset_Ready          : out std_logic;
      signal Computer             : out std_logic


5.  Display <-> Gd_Move

      signal Power_On             : in std_logic;                         --Reset signal
      signal MouseL               : in std_logic;                         --Left mouse button
      signal MouseX               : in std_logic_vector(6 downto 0);      --Mouse x-position
      signal MouseY               : in std_logic_vector(6 downto 0);      --Mouse y-position
      signal Computer_Play        : in std_logic;                         --Against computer=1

      signal PointerX             : out std_logic_vector(6 downto 0);     --Last x-location
      signal PointerY             : out std_logic_vector(6 downto 0);     --Last y-location
      signal Player               : buffer std_logic                      --Whos turn
```

Figure 5 Text description of module interfaces (continued).

5 The Mouse

A mouse is used to select the number of pebbles that players would like to take from the game board on the VGA screen. This chapter details how the provided mouse communicates with the UP1 board, and how the position of the mouse cursor is calculated.

5.1 Mouse Introduction

The mouse provided in the lab is not a true PS2 mouse. The brand name of the mouse provided in the lab is Starmouse. It was purchased from Compusmart at a cost of approximately $15.00.

The properties of a true PS2 mouse are briefly discussed in one of our application notes which is available for viewing at:

http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/98w/Altera_UP1_MOUSE/

It is believed that the mouse provided in the lab is a hybrid mouse, which can operate either on a PS2 communications bus, or a standard RS232 serial communications interface.

The design group was unable to get the mouse to function in PS2 mode, however we were able to configure the mouse in such a way as recognizable data transmissions were provided by the mouse.

It is the belief of the design group that PS2 mice require an initialization string from the host in order to operate properly. Similar PS2 devices, such as keyboards, require the host to send initialization strings. A classmate suggested that a Linux driver be located for a PS2 mouse to see if the code in the driver provides PS2 mice with initialization strings. Our group was unable to locate such a driver in the provided time frame. We encourage future groups to continue the search at ftp://ftp.cdrom.com/pub/linux/.

5.2 Mouse Configuration and Data Packets

The mouse provided in the lab would provide data transmissions if Pin 6 was connected to Vcc. Data transmissions from the mouse have a structured format. This data format is given in Figure 6. Note that data transmissions from the mouse only occur when the mouse has something to report, not when the mouse is in a steady state.

All data from the mouse is inverted. This inversion of data was done so that the data conformed to a more logical representation of information. For example, before the inversion took place, a bit representing a mouse button would be high when the button was not pushed, and low when the button was pushed.

Bit #	Static Data	
0	0	Start bit
1		X direction flag
2		X overflow flag
3		Y direction flag
4		Y overflow flag
5		Right mouse button
6		Left mouse button
7	1	Unknown purpose
8	1	Unknown purpose
9	1	Unknown purpose
10	0	Unknown purpose
11		X movement data bit 0 (LSB)
12		X movement data bit 1
13		X movement data bit 2
14		X movement data bit 3
15		X movement data bit 4
16		X movement data bit 5
17		X movement data bit 6 (MSB)
18	0	Unknown purpose
19	1	Unknown purpose
20	1	Unknown purpose
21	0	Unknown purpose
22		Y movement data bit 0 (LSB)
23		Y movement data bit 1
24		Y movement data bit 2
25		Y movement data bit 3
26		Y movement data bit 4
27		Y movement data bit 5 (MSB)
28		Stop bit

Figure 6 Mouse data transmission packet.

5.2.1 Data Packet: Start and Stop Bits

Bits #0 and #28 of the mouse data packet were termed the start and stop bits. Each data packet send by the mouse has a start bit. The start bit was used in the VHDL code that implements the data capture algorithm to indicate that new data from the mouse was about to arrive. The minimum time delay between a packet stop bit, and a following data packets' start bit is 1.6 ms.

5.2.2 Data Packet: Mouse Buttons

The left and right mouse buttons are represented in a mouse data packet by bits 6 and 5 respectively. When a button is depressed, the data bit corresponding to the button pressed is set. When a mouse button is released, or is not being pressed, the logical state of the button bits are 0.

It should be noted, regardless of the movement of mouse, a new data packet is sent to the host by the mouse each time a mouse button is depressed and another packet is send to the host when a mouse button is released.

5.2.3 Data Packet: Movement Indicators

Movement of the mouse is described by the mouse using a right-handed Cartesian coordinate system. Bits 17 to 11 are used to describe the number of mouse "pixels"[1] moved corresponding to a change in position of the mouse along the X axis. Bits 27 to 22 are used by the mouse to indicate the number of mouse "pixels" the mouse has moved along the Y axis. The X and Y counters were termed the movement counters by the design team.

Movement in the X direction is described by 1 more bit than movement in the Y direction. It is postulated that due to the fact that most computer monitors have more pixels in the X direction than in the Y direction, a higher count description of motion is required. More bits used for the X direction would allow more accurate movement of the mouse cursor on the display screen before overflow would occur in the movement counter.

Movement in the X direction is described by the mouse using 9 bits in each data packet sent to the host. The data sent by the mouse represents not a position on the coordinate axis, but the number of mouse "pixels" moved in the X direction since the last data transmission. From experiments done with the mouse, it appears that the mouse counts the number of pulses from sensors, and then at a predefined time threshold, relays this count to the host.

---

[1] The term "mouse pixels" is used in this chapter to describe the distance traveled by the mouse in a particular direction.

The direction bit corresponding to movement in the X direction will be at logic 0 if movement is along the positive X axis. If movement is along the negative X axis the direction bit will be at logic 1.

The overflow bit tied to the movement counter in the X direction will be at the same logic level as the direction bit if no overflow has occurred. Once an overflow has occurred the state of the overflow bit will be the complement of the direction bit.

The X movement data bits in a mouse data packet represents the number of mouse "pixels" the mouse has moved since the last data transmission. The further the mouse has moved, the larger the count will be. Note that the X movement data bits use the two's complement number system. Therefore when the X direction bit is not set, the count represents a positive number. When the X direction bit is at logic 1, the count is in two's complement form and must be converted into a positive magnitude.

Due to the large snap grid used in this project, it was necessary to modify the original VHDL code so that the magnitude of motion of the mouse was not considered by the code. Before this change was made, the mouse was jittery and very difficult to control. Now, when the mouse is moved we only check for movement at a certain threshold count. If movement is detected, one is added or subtracted from an internal position counter.

Movement in the Y direction is described by the mouse using the same techniques used by the mouse to quantify motion in the X direction.

5.3 Data Capture From Mouse: Timing

Each bit in a data packet sent from the mouse has a duration of 800 us. Data is captured from the mouse by first looking for the start bit of a data packet. Once the start bit has been detected, the VHDL code will delay for 400us and then sample the data from the mouse. This captures the start bit. After the start bit has been captured, each of the subsequent 28 data bits in the data packet are sampled by the VHDL code every 800 us. This method of capturing data from the mouse samples each bit in the middle of each bit pulse to satisfy setup and hold times. This method of data sampling is shown graphically in Figure 7.

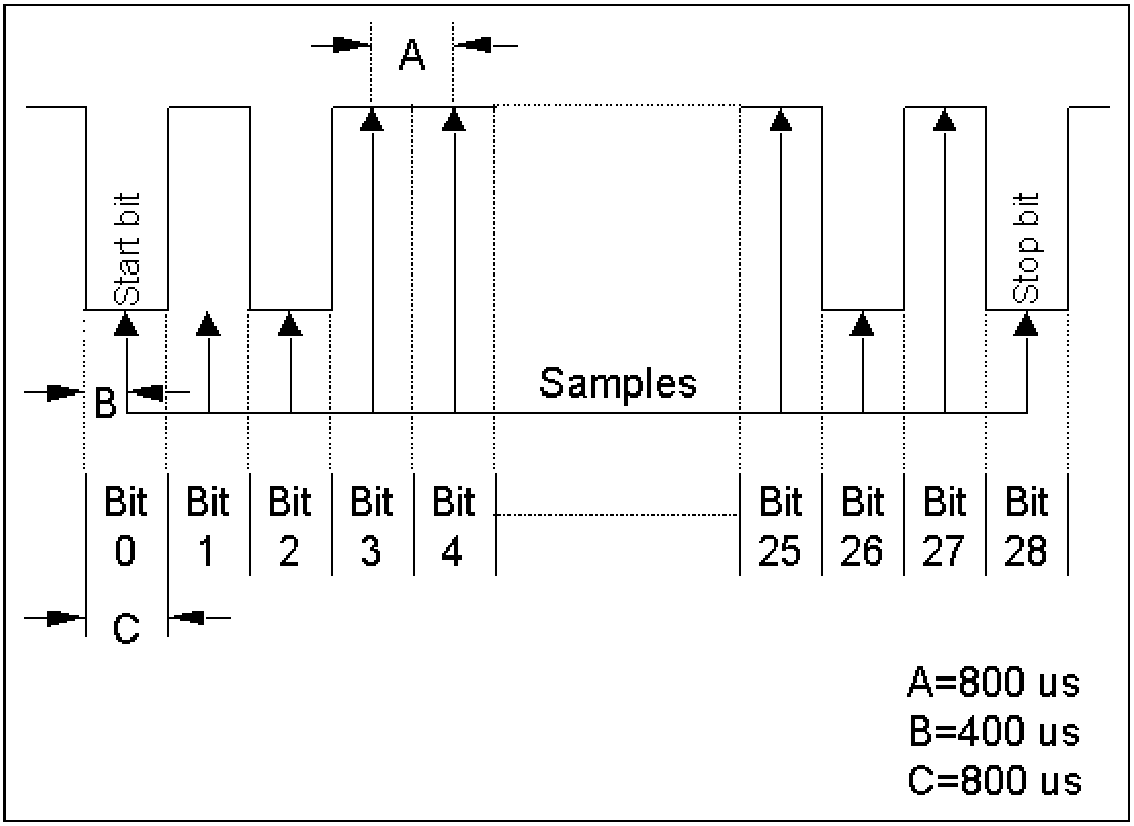

Figure 7 Sampling from mouse data transmission.

5.3.1 Data Capture From Mouse: VHDL

The VHDL algorithm used to capture mouse data is described graphically in Figure 8 and Figure 9. The calculation of the mouse cursor position is also described in these figures and discussed further in §5.4 The Mouse Cursor.

In general, Counter1 is used to reduce the 25.174 MHz system clock into a clock with a period of 800 us and a 50% duty cycle. 800us was chosen because the mouse produces data bits at timed intervals of 800us.

Counter2 provides an index into the data array. Each of the 29 bits in a mouse data packet are placed into the data array. Counter2 is incremented each time the Counter1 clock divider times out. Not all bits of the data array are used at this time, however at differing points in the development of the project different data bits were examined. Logic space could have been saved by only capturing data bits of interest, however the logic required to select only the currently required bits would far out weigh the and logic savings of not using an array.

Once all 29 mouse data bits have been captured, the mouse algorithm recalculates the current mouse cursor position. This is discussed in the next section of this report.

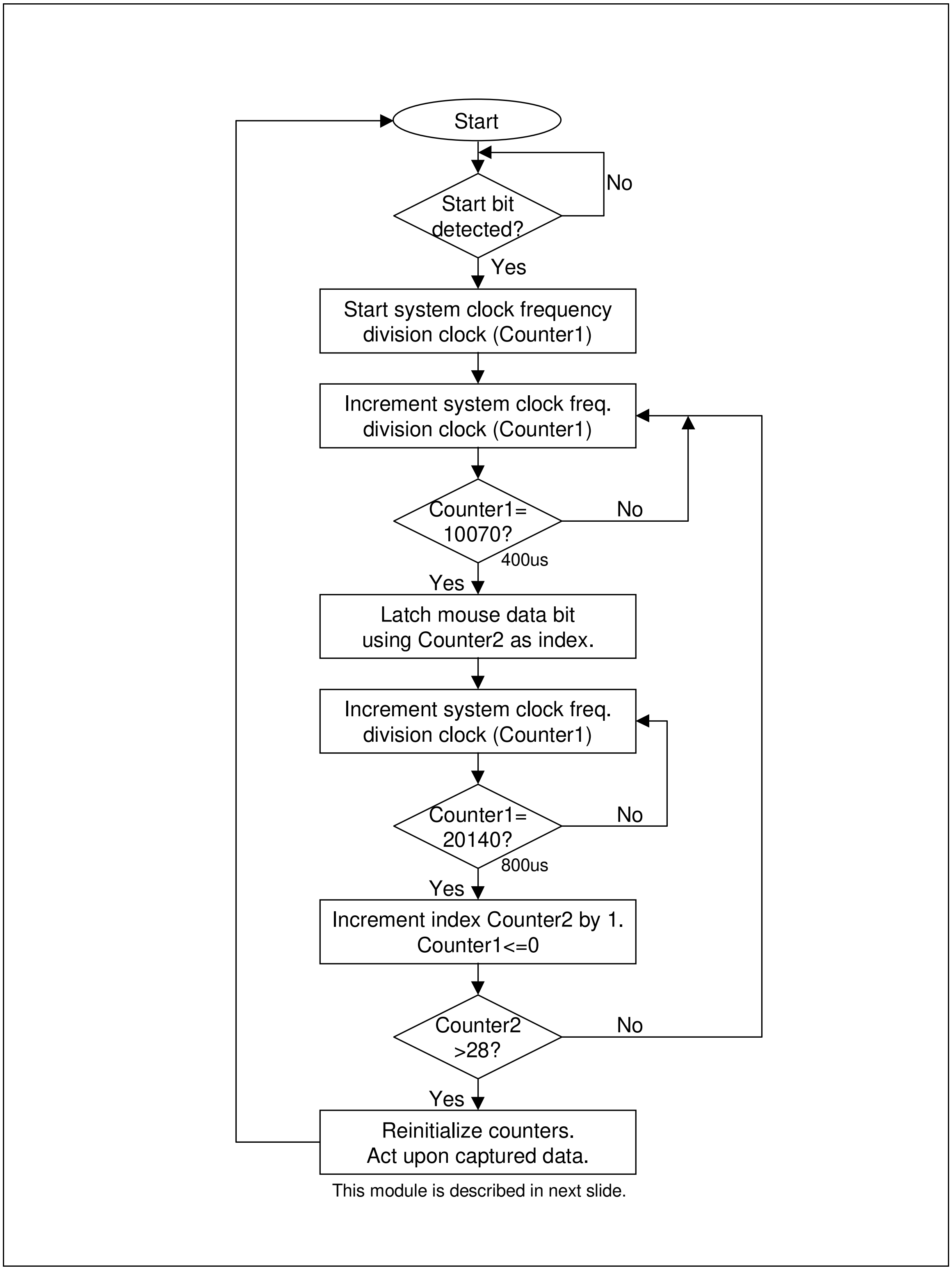

Figure 8 Mouse data capture algorithm – slide 1.

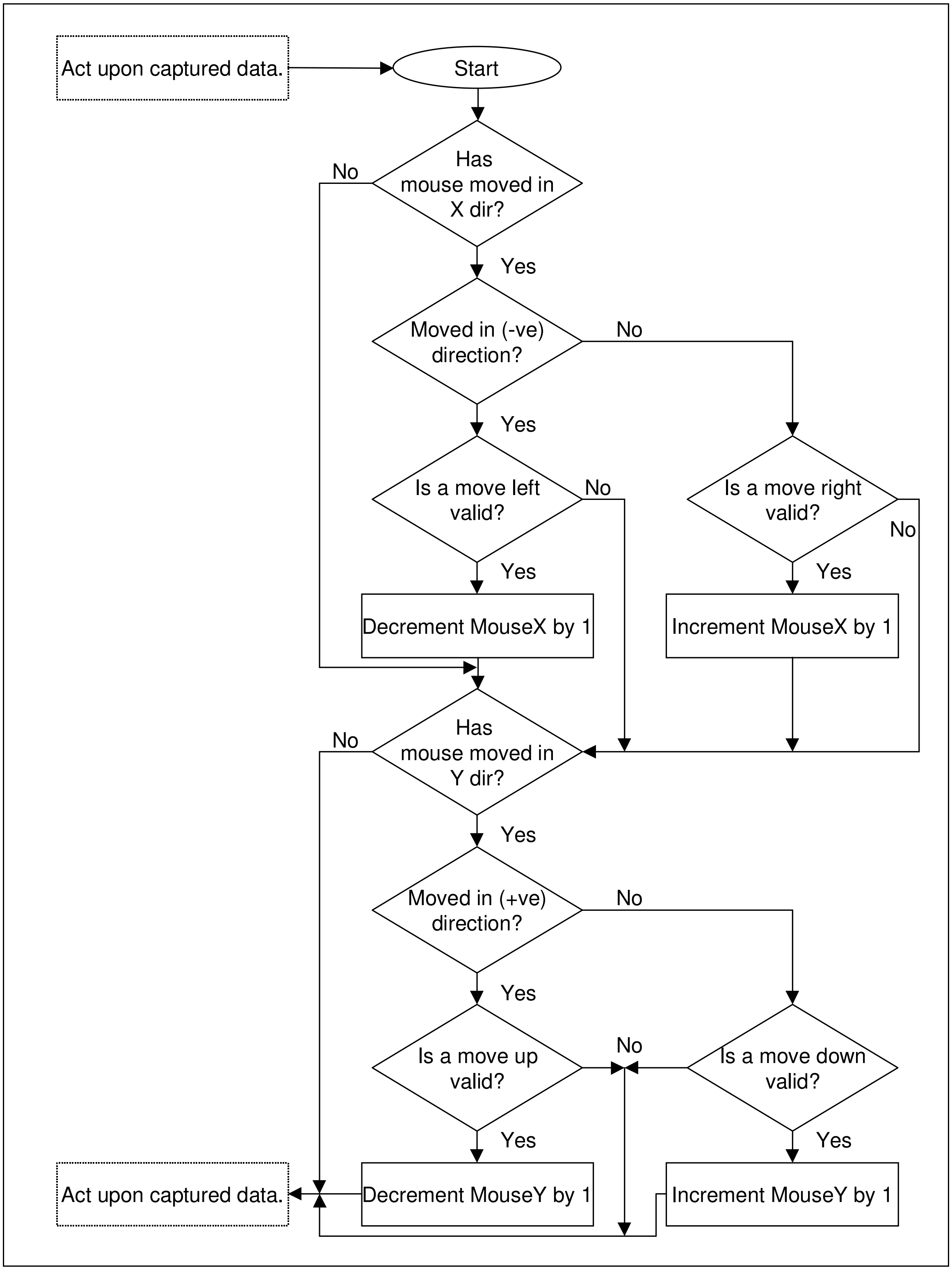

Figure 9 Mouse data capture algorithm – slide 2.

5.4 The Mouse Cursor

The position of the mouse cursor on the VGA screen in calculated within the mouse module. The calculation of the position of the mouse was done in this module because the design team had already created a working model of the game using the Flex push buttons in a mouse like fashion. This working model accepted X and Y coordinates for the selection cursor, and reuse of the main code was essential for time reasons.

There are 36 pebbles displayed in each row, and there are 14 rows on the game board. The mouse cursor can be within any valid column and row address. Valid row values range between 2 and 37. Valid column values are allowed to range from 3 to 29.

When movement of the mouse is detected, variables MouseX and MouseY are adjusted accordingly. The mouse module does not change the current value of MouseX and MouseY if the move would place the mouse cursor outside of the 14x36 grid. In other words the mouse is not allowed to move outside of the game board.

6 Display

6.1 Introduction

The Display.vhd module serves three purposes:

- To control the display of player information at the top of the screen.
- To control the display pebbles on the game board.
- To control the display of the mouse cursor.

Display instantiates three components, HV_Time, Buttons and Mouse. Display routes the necessary signals to Project.vhd to display the results on the VGA monitor.

6.2 Input / Output

Display has four inputs that are received from the top-level module Project.vhd. These four signals are PB1, PB2, Clock and MouseData. PB1 is the system reset. If PB1 is pressed, the initial board will be displayed with all signals set to initial values. PB2 is used to invoke player vs. computer play. If PB2 is pressed, the "Player 2" string will be changed to "C_Player 2" to let the user know that the system is in computer mode. Clock is the system clock, which operates at a frequency of 25.175 MHz. The MouseData input carries the data received from the mouse that describes mouse movement and the state of mouse buttons

The five outputs of Display link to the Altera UP1 Board via Project.vhd. These five outputs result in the display of the game on the VGA screen. The link between the top-level module Project.vhd and Display is found in Project.vhd, through the use of a component statement and a port map.

6.3 Specifics

The Display module manipulates the colors on the screen by using simple logic statements to get different colors displayed at different times. This is accomplished through the use of signal assignments in specific locations. For example, the color of the mouse is different from all other objects on the board. By assigning Mouse_Color to 1 only when the column and row address equal the location of the mouse, otherwise Mouse_Color is assigned to 0 and the mouse cursor is displayed. The signals rom_mux_output and Rev_video are also used to determine the color, depending on the current location the screen is being drawn. Rev_video is set if the current row_address is less then or equal to 2. This gives a title bar at the top of the VGA screen that is a different color then the rest of the screen.

The different characters that are displayed on the screen are accessed from tcgrom.mif by using the Rom_Address variable. Rom_Address is a standard logic vector that is nine bits long. The first six bits are assigned by the programmer depending on the character that is to be displayed. The variable Pixel_row_count is assigned to the least significant three bits of Rom_Address.

The rest of the file is the implementation to display the graphics on the screen. This includes displaying the pebbles, mouse pointer, "Player 1" and "Player 2" strings, the "C_Player 2" string if computer mode is currently set, all of the spaces to fill the rest of the screen, and the "1 win" or "2 win" message if a winner has been declared. This is accomplished by comparing the col_address and row_address with the current video scan location. If they are equal, the rom_address(8 downto 0) assignment will get the character to be displayed from tcgrom.mif.

7 Hv_time module

7.1 Introduction

This file was copied from:

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

The HV_Time module is responsible for the generation of the horizontal and vertical synchronization signals used to display the information in the display module. It does this through the use of two counters:

H_count – responsible for the horizontal pixel count
V_count – responsible for the vertical pixel count

The design team agreed that this file met our needs for VGA signal generation and left it unmodified such that the proper credit is given to the original authors. The design team did however learn how the file worked so that it could be modified to meet our specific needs.

8 Gd_Move Module

8.1 Introduction

The Gd_move module is responsible for the math calculations required to decide if a player's move is valid. Additionally, if the computer flag is set, GD_move is used to calculate the number of pebbles that the computer should take.

The computer moves are calculated in the Check_move process. The math operations that are calculated are:

- Convert the location of the mouse into a row and column number.
- Decide how many pieces the computer can take based on the previous move.
- If the number of pieces the computer would like to take is greater then the number remaining on the current row, how many rows it would have to take, as well as the number of pieces taken from the new row.
- Validate players requested move regardless if the mouse cursor is on the same line as the last pebble taken, or if it is on the next line, or if multiple lines have been selected.

Once the required math is ready, the Check_mouse process then takes this information and does one of two things:

1. If the system is in player vs. player mode, Check_mouse will check to see if it is a valid move, and if it is a valid move will update the last pebble pointers, and the previous taken number taken variable.
2. If the math operations are not complete or if the move has been flagged as invalid, Check-mouse will not change any of the above mentioned variables.

If the system is in Player vs. Computer mode, it will calculate the location of the new PointerX and PointerY location, and will switch players. The validity of the move for the computer does not have to be checked, as it is guaranteed to be correct from the Check_move process. (See information regarding the computer player in §10.3).

8.2 Check_Move

Check_move is responsible for the math required for Check_mouse. Check_move is described graphically in Figure 10, Figure 11 and Figure 12. The first thing Check_move does is convert MouseX, MouseY, PointerX, and PointerY into column and row numbers. The X corresponds to the column that must be calculated in the range 1 to 36.

```
MXI2 <= conv_integer(MouseX);
MXI <= MXI2 - 1;
PXI2 <= conv_integer(PointerX);
PXI <= PXI2 - 1;
```

To convert the Y locations to a row number, the Y count must be divided by 2. Since our game displays one row of pebbles followed by a blank row the range is 1..14.

```
MYI2 <= conv_integer(MouseY);
MYI <= (MYI2 - 1) / 2;
PYI2 <= conv_integer(PointerY);
PYI <= (PYI2 - 1) / 2;
```

If in computer mode the process then calculates the number of pieces the computer should take. At this point, the computer will take twice the number previously taken, if it cannot win on it's turn, or if it can win, will take the last pebble. It should be noted that there is still a small timing error in the computer calculations, as every so often, the amount wanted will be a invalid number.

To save on the number of logic cells that the computer would require, we implemented the move that the computer should take as a long list of if / elsif / else statements, as otherwise division would be required. Once the computer knows how many pieces it should take, it must then calculate the new PointerX, PointerY locations. To do this, we check to see how many rows must be removed based on the number still remaining on the present row, and how many pebbles would be left on the new row, and calculate a Ystep so that the Y pointer can be calculated by that amount.

This process will also check if the players requested amount is valid. This is done in three stages, but is simplified by the use of multiplication, as we need to multiply by 36, which can be split into 32 and 4, so that only a shift register need be used. The first thing that is checked, is if the user selected pebbles on the same line as the previous player. If this is the case, the amount of pebbles that are requested is just the MouseX location minus the PointerX location. If the user selected pebbles on the next line as the previous move, then the amount of pebbles that is requested is 36 minus PointerX plus MouseX. If however, there are one or more full lines between the previous value of PointerX and MouseX, the value is calculated the same as step 2, but multiplies 32 + 4 by the difference in the y position.

8.3 Check_Mouse Process

The Check_mouse process does one of two things:

- Validate the players move
- Compute the new location of the pointers based on the computer's move.

If the mode of play is player vs. player, or it is player 1's turn during a computer match, the module will take the numbers calculated from above, and compare it with 2 times the previous move, or to see if the mouse was clicked in the white space. If either of these conditions fail, the system will not update and wait for the same user to select a new number of pebbles. If however, the move is valid, the previous amount will be replaced with the current amount, and the player flag will be switched. See Figure 13 and Figure 14 starting on page 28.

If the system is calculating the new position of the pointer based on the guaranteed move from the computer, it will check one of 4 things:

Was the computers move to take the last pebble, or
Did the Y pointer get moved beyond the max, or
Did the computer pick a move still on the board, or
Was the pieces taken by the computer on the same row as before

If the computers move was the last pebble, the win flag is set and the game ends. If the Y pointer was moved beyond the maximum amount, the Y pointer is reset to the last line, and the number of X locations that was calculated is added to the pointer. If the computer picked a move still on the board, then the pointers are updated. And finally if the computer picked a small enough amount of pebbles that the new X, Y pointers are still on the same row as before, the X amount is just added.

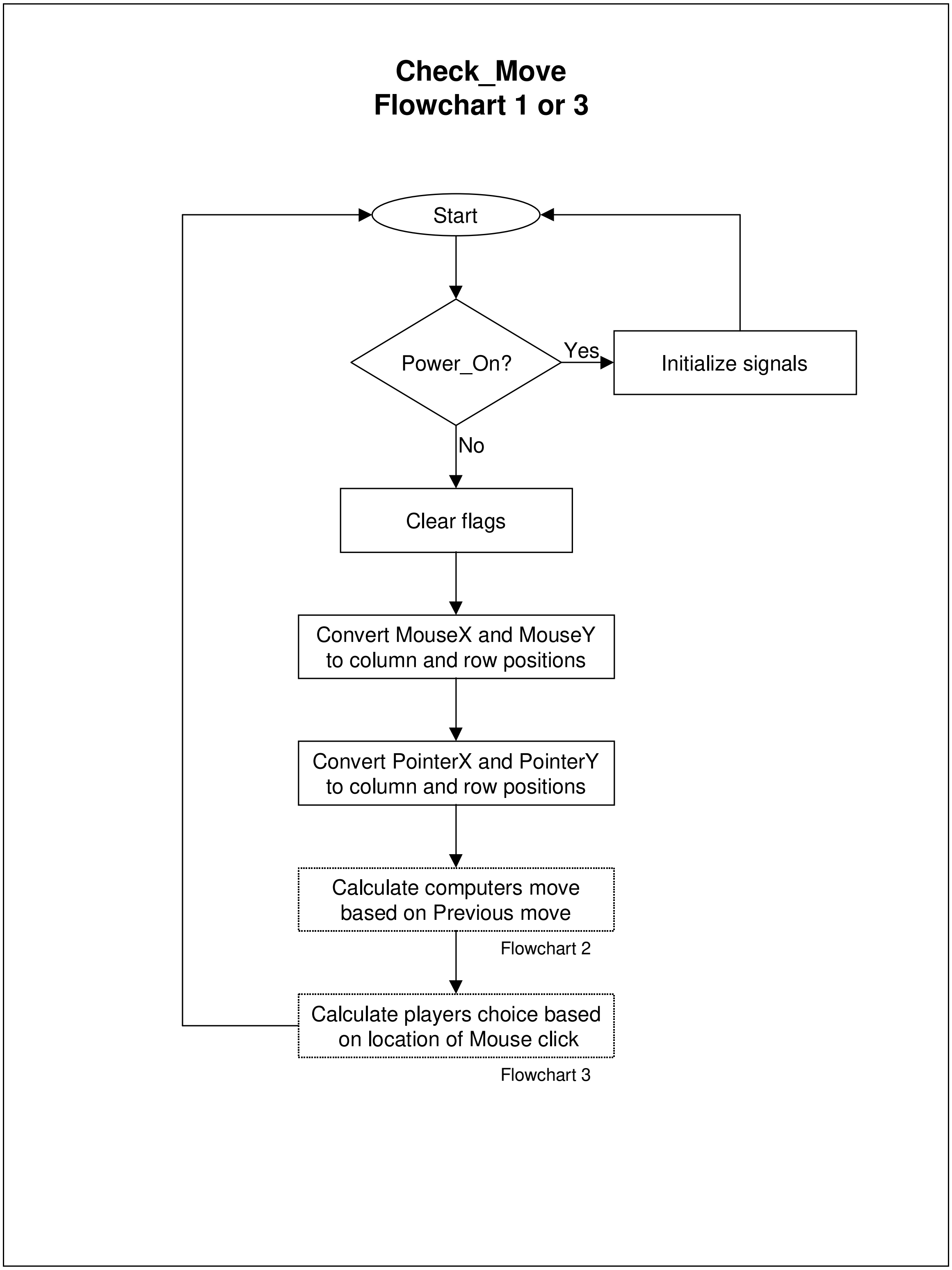

Figure 10 Check_move flowchart 1 of 3.

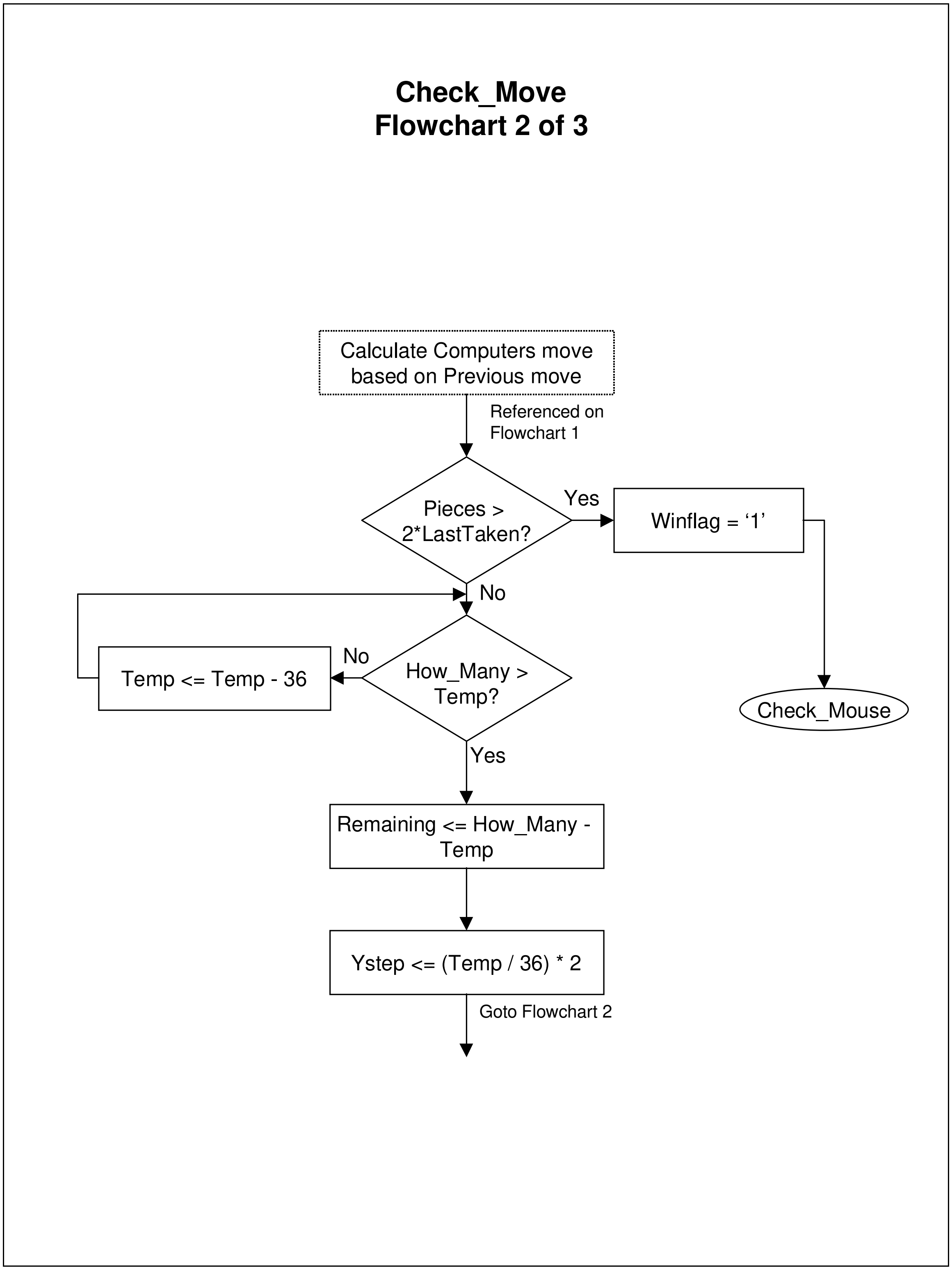

Figure 11 Check_move flowchart 2 of 3.

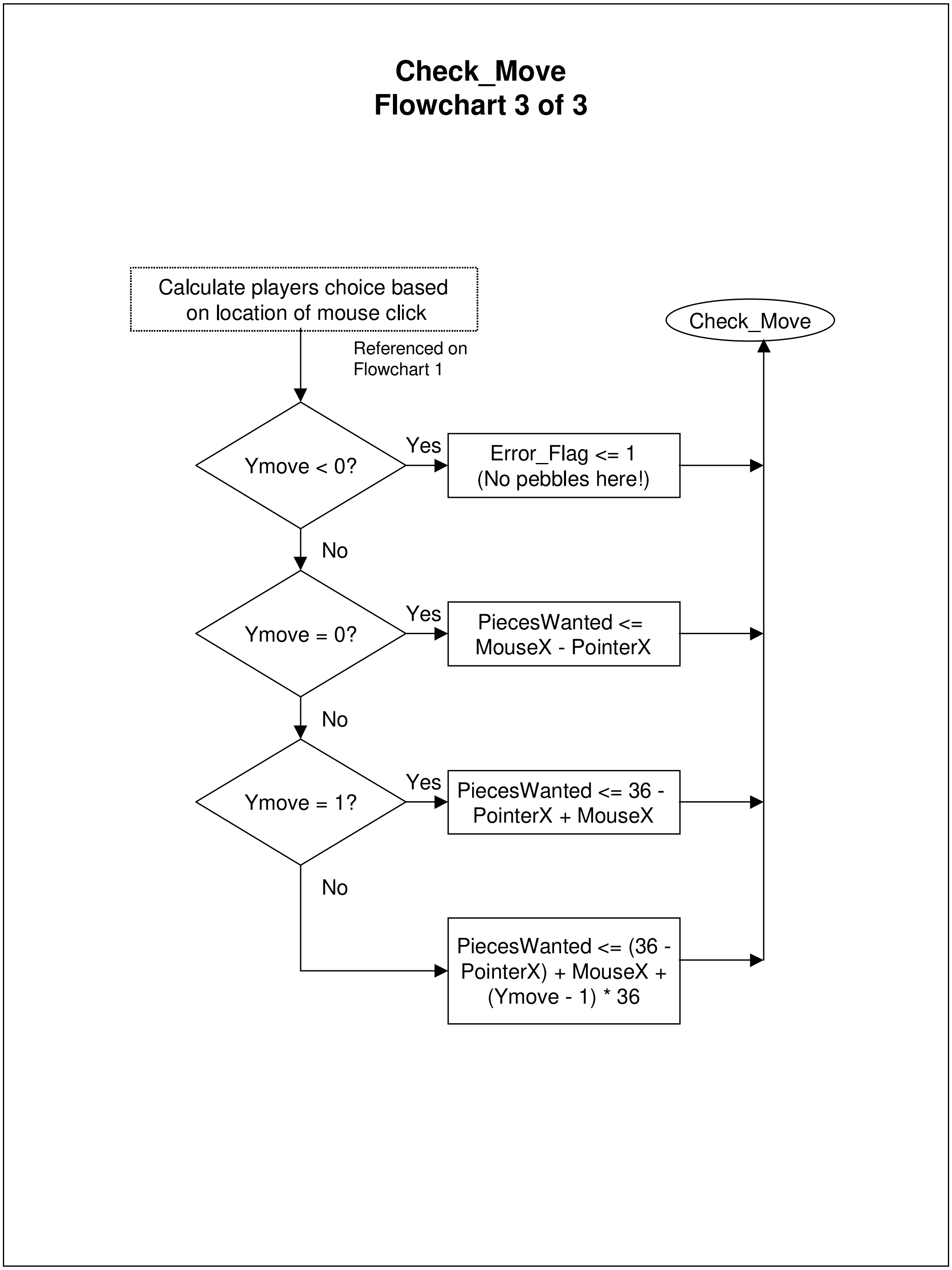

Figure 12 Check_move flowchart 3 of 3.

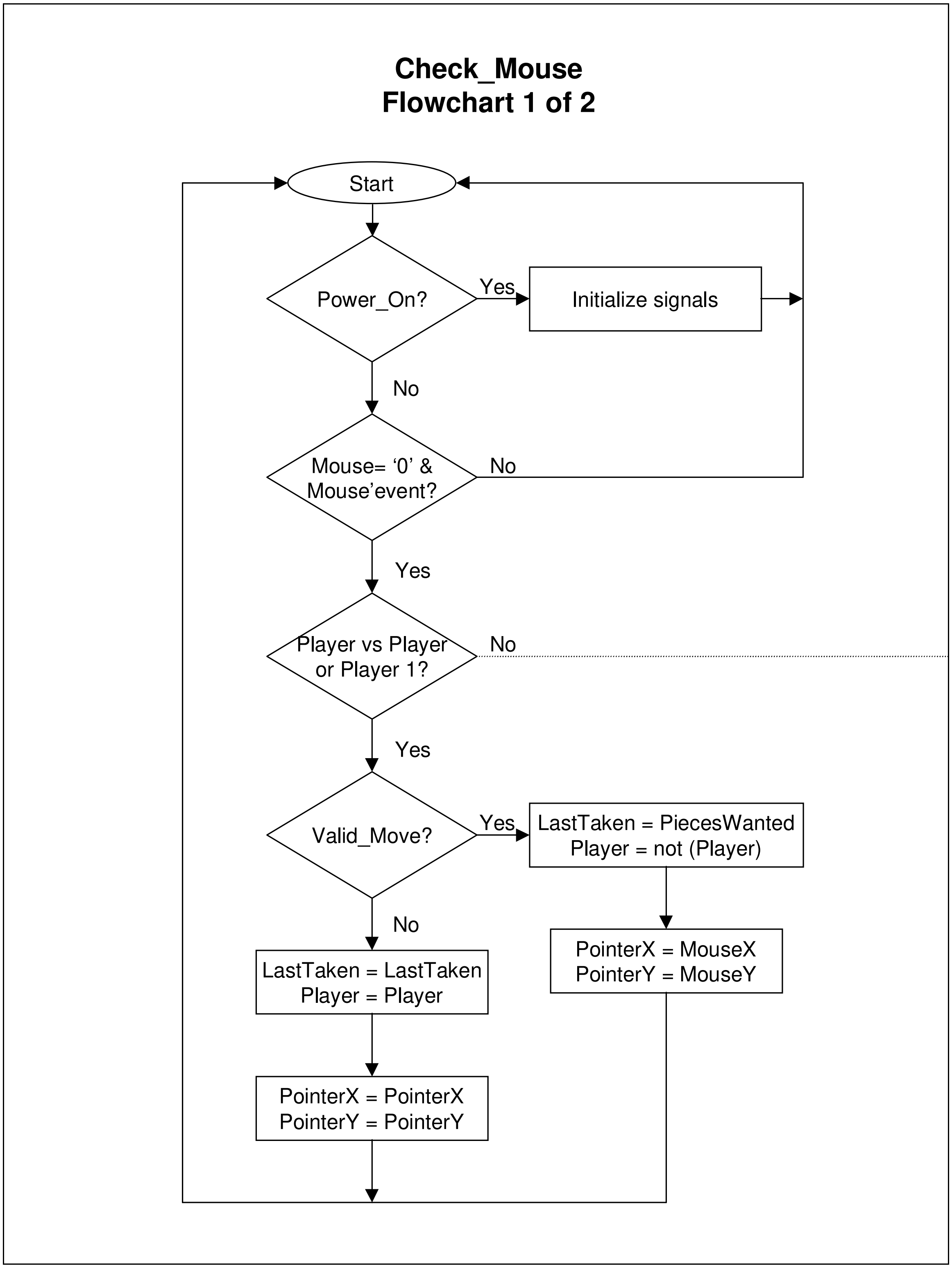

Figure 13 Check_mouse flowchart 1 of 2.

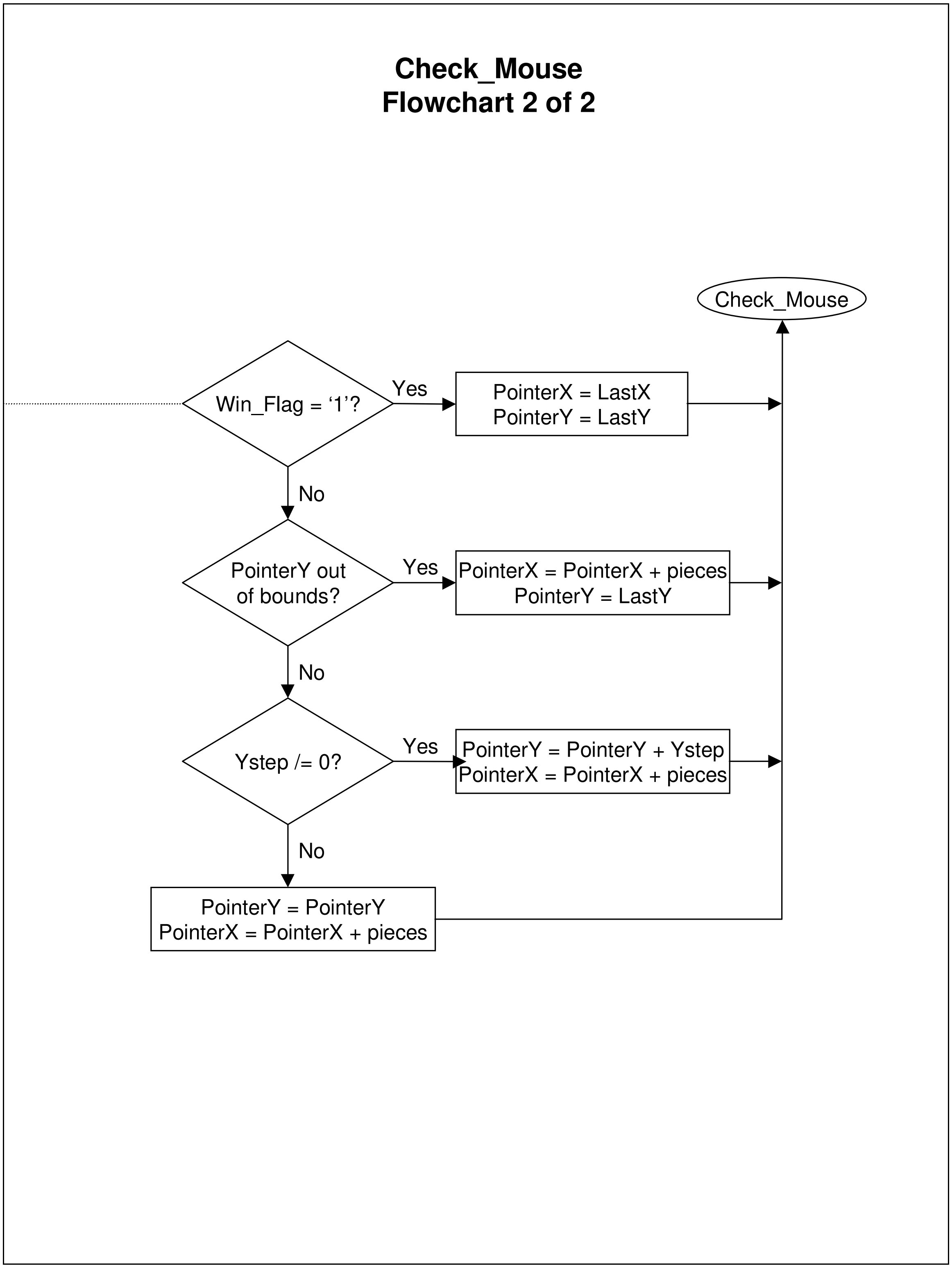

Figure 14 Check_mouse flowchart 2 of 2.

9 Push Buttons

Push buttons 1(PB1) and 2(PB2) are used by this project. PB1 was used to reset the game. PB2 was used to select player-player or computer-player mode.

No special logic is required to use the push buttons except that the push buttons on the Altera UP1 board are not debounced. The buttons are therefore debounced within the Flex chip.

The algorithm used to debounce the push buttons is as follows:

1. If power is being applied for the first time, latched signal values are forced to 0, and the internal debounce counter is set to 0.
2. One each system clock cycle, PB1 is checked to see if it has been pressed. (Remember that the push buttons are active low).
3. If PB1 has been pressed, PB1_Latch is set, and a counter is started. After 1ms has elapsed, all signal bounces are assumed to have died out and the Reset_Ready signal is asserted. At this time the game will reset itself.
4. If PB2 was pressed, the same 1ms delay is evoked. After the 1ms delay has elapsed, the computer will begin to play once the Computer signal is asserted.

This algorithm is graphically illustrated in Figure 15 below.

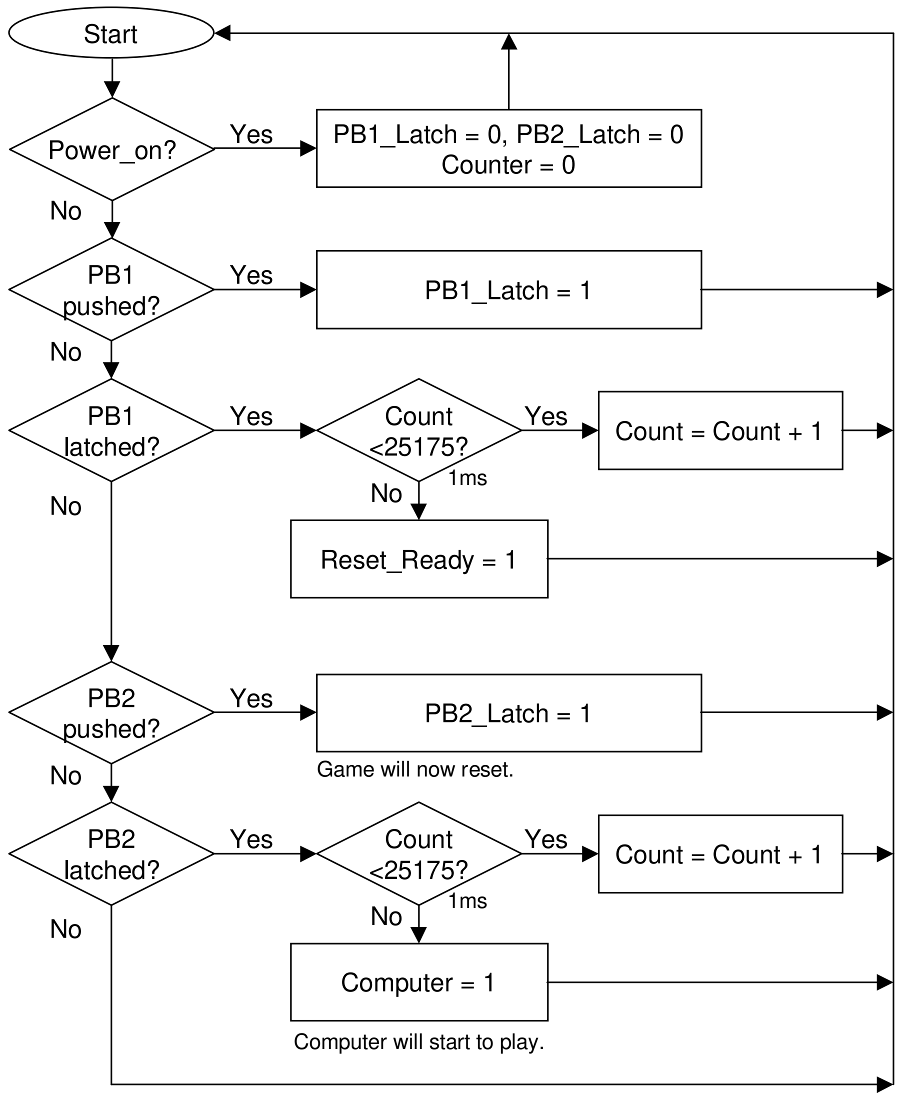

Figure 15 Debouncing of UP1 push buttons

10 Design Verification and Simulation

The design verification and simulation chapter is divided into two parts. This division was necessary to highlight simulations and verifications that were completed, and those that were not possible to empirically complete. Finally, the computer player implemented in this project is briefly discussed.

10.1 Verification Introduction – What was done

The mouse, VGA graphics, and internal game playing logic compose the three major components of this project. The testing of these three modules will be examined in the following sections of this report.

10.1.1 Testing of the mouse

There are two aspects that must be considered when discussing the incorporation of the mouse in this project:

1. Mouse data capture
2. Mouse cursor movement

10.1.1.1 Mouse Data Capture

Correct capturing of data from the mouse was tested using the code provided with one of this groups' application notes available at:

http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/98w/Altera_UP1_MOUSE/

The code provided with this application note captures data from the mouse in the manner explained in §5.3 of this report. The output of the code provided in the application note is a string of 29 bits displayed on a VGA monitor.

The 29 bits captured from the mouse are displayed as 1's and 0's on the screen. These bits were examined after each iteration of the tests given in Table 1. The method used to capture data from the mouse proved to operate correctly in 100% of the tests.

It should be noted that the original interpretation of the bits (see Figure 6) provided by the mouse was also a product of the code given in the application note. The tests performed to ensure proper data capturing only tested for repeatability of data capture from the mouse and did not test for the actual meaning of the bits captured.

Test	Result
Bit 0 was always 0	Bit 0 was always 0
Bit 28 always 0	Bit 28 was always 0
Bit 10 always 0	Bit 10 was always 0
Left mouse button pressed	Bit 6 – 1
Left mouse button release	Bit 6 – 0
Right mouse button pressed	Bit 5 – 1
Right mouse button released	Bit 5 – 0
Left and Right mouse button pressed	Bit 5 – 1 Bit 6 – 1
Middle mouse button pressed	No new data packet sent by mouse
Middle mouse button released	No new data packet sent by mouse
Slow movement in (+)ve Y direction	Bit 3 – 0 Bit 4 – 0 Bits [22..27] – varied
Slow movement in (–)ve Y direction	Bit 3 – 1 Bit 4 – 1 Bits [22..27] – varied
Slow movement in (+)ve X direction	Bit 1 – 0 Bit 2 – 0 Bits [11..17] – varied
Slow movement in (-)ve X direction	Bit 1 – 1 Bit 2 – 1 Bits [11..17] – varied
Movement and right button pressed	Bit 5 – 1
Movement and left button pressed	Bit 6 – 1

Table 1 Tests used to verify proper mouse data capturing.

10.1.1.2 Mouse cursor movement

Movement of the mouse cursor on the VGA screen was tested visually. The way that the mouse module was set-up, movement of the mouse is interpreted only by the mouse module.

The position of the mouse cursor is calculated within the mouse module. The game board consists of 36 columns and 14 rows of pebbles. The mouse cursor is not permitted to leave the grid of pebbles.

Three tests were used to verify the proper operation of the movement of the mouse within the pebble grid:

1. After reset, the mouse cursor was placed on top of every pebble in odd screen columns to ensure that the mouse module would allow the cursor to be positioned within the game board properly
2. After reset, in the middle of a game, and after the game was won, the mouse cursor was first dragged along the pebbles which comprise the outer perimeter to ensure that the mouse module would never allow the cursor to leave the game board. Secondly, the mouse was moved erratically at high velocities to make sure that the arithmetic functions performed in the mouse module could complete quick enough as to not allow movement outside of the game board. (Inequalities were used to define the edges of the game board in case the mouse ever left the game board).
3. In the middle of a game, with approximately ½ of the pebbles taken the mouse was placed on every pebble in even screen columns. This was done to further ensure that the mouse module would allow positioning of the mouse cursor properly once a game has started.

The mouse module proved to position the cursor properly in all cases. Ironically, an error not related to the positioning of the mouse cursor was discovered. The discovered error results in the mouse having an increased sensitivity to motion in negative directions than in positive directions.

When the mouse is moved in a negative direction the bits describing the movement count in that particular direction are given in two's complement form. The problem that arises in the project code, but not in the code provided with the application note, is that movement is considered only if the movement count provided by the mouse is of a certain magnitude.

For example some of the least significant bits are truncated. Therefore, when the mouse is moved in a positive direction we check to see if any of the most significant bits are non-zero. If the most significant bits compose a non-zero number the movement of the mouse is considered valid. The problem of using this method is when movement is in the negative direction, small movements of the mouse produce a non-zero number in the upper bits because the number is in two's complement form. This produces the increased sensitivity of the mouse in the negative direction.

The project code was not modified to correct this error to date. The fix would be to consider only movement in which the most significant digits contained one or more zeros. This could be done with an inequality comparison.

10.1.2 Proper Operation of VGA Graphics

General verification of the proper operation of the VGA graphics was very simple. If the Flex chip was configured, and the VGA monitor was connected, the graphics would appear.

At no time in the development of this project did an image fail to appear on the VGA monitor. The image may not have been what we wanted it to be at all times, but there was always a image.

Occasionally, the output shown in the image was questioned. However, after the learning curve of character placement and image generation was surpassed, the output on the monitor screen was never miss-predicted.

As an example of the ease of use and reliability of the character generation methods used in this project, the 1's and 0's in the mouse application note provided by this group were added to the VHDL code in less than 20 minutes.

10.1.3 Internal Game Playing Logic

The development of the final version of the game took the following route:

1. VGA graphics added to the project.
2. The mouse replaced the push buttons previously used.
3. The rules of the game were programmed.
4. Addition of a computer player.

The rules, or internal game playing logic, was added after VGA graphics and the mouse were operational. The design team decided that the rules of the game could be tested using observational methods only. This decision was based upon the following reasons:

1. The rate at which the rules could be tested was exponentially higher than that which could have been obtained using a simulator. Movement of the mouse, clicking a mouse button and pressing reset was much quicker than simulation and produced highly understandable results.
2. The interaction of the three main modules (VGA graphics, mouse, and game logic), could be tested concurrently ensuring no timing violations or other irregularities existed in the code of all three modules.
3. The simplicity of the game did not appear to warrant endless hours of simulation. Proper operation could be verified quickly and accurately without a simulation.

The game logic proved to operate flawlessly for all tests performed.

10.2 Verification - What was not done

Providing verification and simulation documents for any large project such as ours is difficult. Due to the use of VGA graphics, we feel that we are unable to meet one of the major requirements of this final project report.

The final report specification requires that simulation and verification documentation be provided for major portions of the project. VGA graphics generation is one of three major modules in this project.

Our design group has discussed this part of the document carefully and would like to offer the following explanation as to what was not presented in the preceding sections of this chapter. Additionally, we would like to offer a suggestion for future projects using VGA graphics.

We feel that the document specification provided on the web page for this course does not adequately take into consideration projects which use VGA graphics as the only output device. With the exception of the horizontal and vertical synchronization signals, the project can only be verified using qualitative techniques.

Our design group, as with other groups, did not create the module that generates the horizontal and vertical synchronization signals. This module was sourced from the MIPS program. The synchronization module was not stolen with reckless disregard for the underlying logic however!

At the start of development for this project the design group researched the signals required by VGA monitors to display a image. Information from the Altera UP1 manual and addendum was studied so that we would have a adequate understanding of what a VGA monitor requires to display a image. The MIPS program was then studied to discover how video signals were generated in that project.

The VHDL code generating the video signals in the MIPS program is simple. However, we were unable to “see the big picture”, and a simulation was done to see how the 5 VGA signals interacted with each other with respect to time. The simulation was not done to confirm proper operation, but to see graphically how the 5 VGA signals behave.

Due to the time expended on the addition of the mouse to the project, the design group did not believe that reproducing a previous simulation of synchronization signal generation for this document would prove proper operation of our design. Simply, without the proper operation of the stolen (re-used) signal generation module the VGA graphics would not appear on the monitor.

We do fully realize that in a project where the synchronization signal generator was created from scratch, a full simulation would be required to ensure proper operation.

If we could suggest for the next EE 552 project groups that the final document specification acknowledge projects which incorporate VGA graphics.

Perhaps by requiring groups to produce their own synchronization signals, there would be a more natural need to fully simulate and document that portion of their design. Also it may be beneficial to future groups if a set of guidelines were developed to help guide groups in the areas of simulation and verification of designs which incorporate VGA graphics.

10.3 Computer Player Verification

The final addition to the project was a computer player. A mode of play was included that allowed one human player and the Altera chip to go head to head.

The addition of a computer player to the project is incomplete. The computer player as implemented will play a game with limited intelligence. There is also a suspected timing violation that occurs intermittently which results in the computer trying to take an invalid number of pebbles.

11 References

References for any VHDL code re-use are given with the code file which uses someone else's ideas or code.

The Marienbad game designed in the lab was described in Bassard & Brately, "Fundamentals of Algorithmics". Prentice Hall, 1996. p285-291. This was a CMPUT 204 text book and may still be available from the Reserve Reading Room at the Cameron Library.

General mouse information was obtained from the following sources.

> Altera: PS2 mouse data packets and board pinouts.
>
> Rob Chapman (EE 552 student) Continuation & further explanation of Altera PS2 mouse documentation
> http://www.compusmart.ab.ca/rc/studentnotes/mousenotes.html
>
> General PS2 information & code examples
> http://www.insightsystems.com/gatech/ps2keyboard.html
>
> Georgia Tech students Doug McAlister, Gauthier Philippart, and Michael Sugg.
>
> Miscellaneous mouse information: Tomi Engdahl then@delta.hut.fi
> http://www.hut.fi/~then/mytexts/mouse.html
>
> Logitech: General technical information from Application note 1410
> ftp://ftp.logitech.com/pub/TechSupport/MOUSE/HELP/1410.txt

Video generation code and ideas were taken from the MIPS program available at http://www.ece.gatech.edu/users/hamblen/ALTERA/mips.zip.

12 Appendix – VHDL code