Lily Wang and Igor Geller
6.111 Final Project Report
December 14, 2007
TA mentor: Brian

Tetris

1. Introduction

We implement a tetris game that is played on a 10-column, 20-row grid on the video display. The seven standard tetris shapes fall from the top of the screen in a random sequence generated by a random number generator in Verilog. The object of the game is to manipulate these shapes by translation and rotation to create a horizontal line at the bottom with no gaps. When such a line is created, it disappears, earning 10 points. Whenever one player clears rows on their terminal, the game sends notification through a serial port connection between the two labkits to increase the other player's speed by a number proportional to the number of rows cleared. When the stack of tetris shapes reaches the top of the grid and no new tetris shapes are able to enter, the player loses and his opponent wins. So far the game is missing some of the functionality. As of now the full rows do not clear but cause the game to stop. Also, for the simplicity of debugging, the only shape currently being generated at the top once the previous piece reaches the bottom is the bar. The colors of the bar are in fact randomly generated and “symbolize” the random generation of the pieces. So the pieces move properly, do not go beyond the border of the allowed 10 by 20 screen, and stop once they can no longer move down. There is also a next piece showing on the right side of the screen that correctly predicts the color of the next piece (since all the pieces are bars).

Shown below is the block diagram of the modules involved in this project.

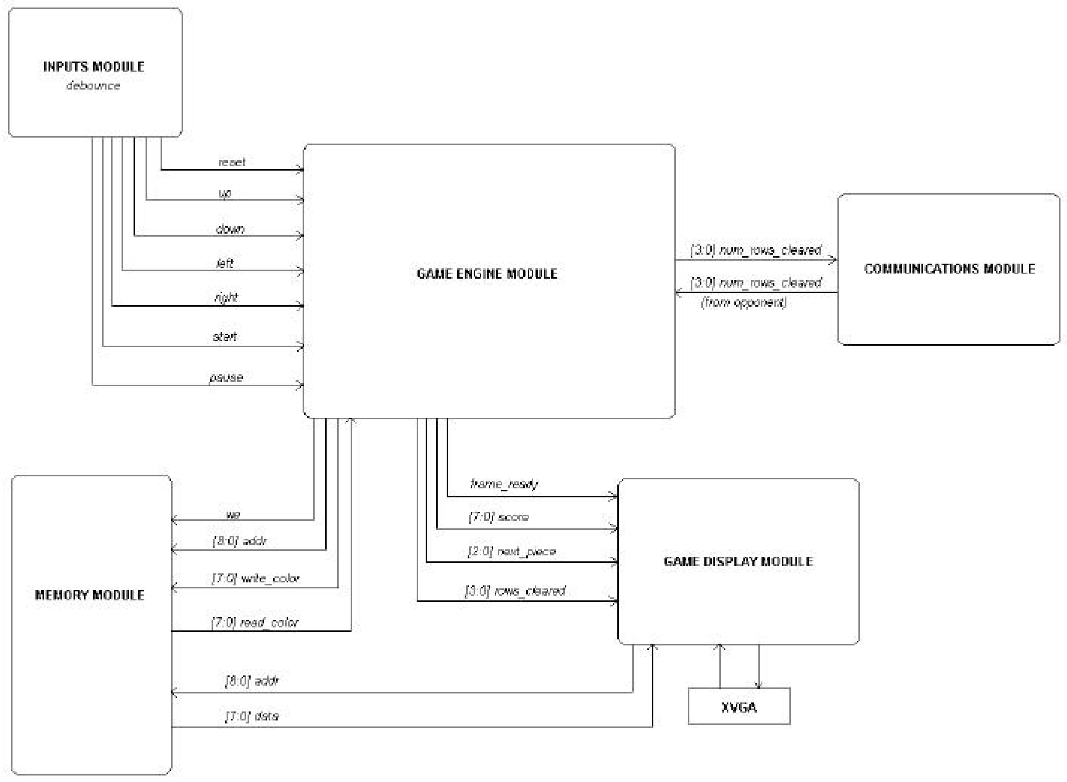

Figure 1. Simple block diagram of the modules in the proposed system.

2. Module Descriptions

2.1. The Game Engine Module - Igor

The game engine module consists of three major states, that are fundamental for the functionality of the basic tetris game and two additional that enable the row deletion and initialization of the new piece's environment squares.

The whole implementation is based on the coordinates **x_coordinate** and **y_coordinate** that indicate the location of the current "active piece". The piece is defined by the 4 by 4 square shown below and the (x,y) square is marked in red:

X					X
X		X			X
X					X
X					X
	X	X	X	X	

In the first state, the program reads the color values stored in the memory, indicated by the black Xs above. When those squares are outside of our 10 by 20 region, the values are read as “occupied”. These values are then used in the second state, the update state, to determine the changes in the game state depending on the inputs and the time variable (for moving down). Notice that for determining if the piece can move in a particular way the 16 squares defining the active piece in addition to the 12 surrounding squares marked by Xs is enough. Moreover notice that whenever the active piece moves, the 4 by 4 square defining it is fully known and can be properly re-assigned without being read from the memory. For example, suppose the initial values read during the reading state are stored into variables **to_left[3], to_left[2], to_left[1], to_left[0], to_right[3], to_right[2], to_right[1], to_right[0], row5[3], row5[2], row5[1], row5[0]**. The values of inactive pieces of the inside 4 by 4 square are being kept in variables **ina_row1[3:0], ina_row2[3:0], ina_row3[3:0], ina_row4[3:0]**. Let us say, the input to the module was the left arrow. Prior to moving the piece to the left, the program should check that the movement is possible. That means that for a particular piece it has to make sure that particular squares (marked by a blue C) are free.

X	C	X			X
C	X	X			X
X	C	X			X
X					X
	X	X	X	X	

In case that the movement is in fact possible, the active piece, including the red square defining the coordinate location of the active piece will move one square to the left:

X	X				X
X	X				X
X	X				X
X					X
	X	X	X	X	

For this movement to take place, two things have to be done. The new values of the shifted squares have to be written into the memory, assigning **to_left[2]**, **row1[3]**, **row2[3**], and **row3[3]** to 1 and **row1[2]**, **row2[2**], and **row3[2]** to 0. Also, the variables storing the inactive pieces inside of the 4 by 4 square have to be shifted because the square itself was shifted to the left and inactive pieces haven’t changed. But since we have read the 12 encircling squares during the read state, we have enough information to accommodate the shift:

```
ina_row1[3] <= to_left[3];
ina_row1[2] <= ina_row1[3];
ina_row1[1] <= ina_row1[2];
ina_row1[0] <= ina_row1[1];

ina_row2[3] <= to_left[2];
ina_row2[2] <= ina_row2[3];
ina_row2[1] <= ina_row2[2];
ina_row2[0] <= ina_row2[1];

ina_row3[3] <= to_left[1];
ina_row3[2] <= ina_row3[3];
ina_row3[1] <= ina_row3[2];
ina_row3[0] <= ina_row3[1];

ina_row4[3] <= to_left[0];
ina_row4[2] <= ina_row4[3];
ina_row4[1] <= ina_row4[2];
ina_row4[0] <= ina_row4[1];
```

As we can see, in a typical cycle, we have no need to read the values of inactive pieces in the 4 by 4 square defined by the active piece. The only time this is essential is when the new piece is generated at the top of the screen after the old piece has stopped moving. We will come back to that point later.

Now, we keep track of inactive values in the 4 by 4 square, how do we know which pieces are occupied by the active piece itself? For that we have a state machine inside of the update state. The upper layer of the state machine has 8 states. States 1 through 7 correspond to a particular shape of the active piece. State 0 is a transitional shape where new piece is generated at the top. States one through seven have four sub-states each, corresponding to 4 different rotations of the particular shape. That way, once we know which sub-state we are in, we know exactly what squares are occupied by the active piece, and can store that information into the memory. For example, suppose we entered a horizontal state of a Z piece, upon entering, we define the values of its 4 by 4 square as follows:

```
row1[3] <= piece_color;
row1[2] <= piece_color;
row1[1] <= ina_row1[1];
row1[0] <= ina_row1[0];

row2[3] <= ina_row2[3];
row2[2] <= piece_color;
row2[1] <= piece_color;
row2[0] <= ina_row2[0];

row3[3] <= ina_row3[3];
```

```
row3[2] <= ina_row3[2];
row3[1] <= ina_row3[1];
row3[0] <= ina_row3[0];

row4[3] <= ina_row4[3];
row4[2] <= ina_row4[2];
row4[1] <= ina_row4[1];
row4[0] <= ina_row4[0];
```

Now we update the state in accordance we the inputs we have and the time parameter value set for moving down. For example rotation will require the following re-assignment (that overwrites the previous assignment in the same clock cycle):

```
row1[3] <= 0; (black)
row1[2] <= 0;
row1[1] <= piece_color;
row3[2] <= piece_color;
```

Notice that for rotation we do not need to shift the inactive squares since the coordinate square does not move.

Now, in the third state we need to write the changes that happened to the game state into the memory. To properly do that we need to write all the 28 values back (in order to not be shape, rotation, and movement specific) since any of the 28 squares could be overwritten in the process (that is actually not true, but it is simpler to make this assumption than to neat-pick the squares narrowing them down one by one). I will talk about the mechanism of reading from and writing to the memory in more depth in the difficulties and resolutions section.

While the above three states enable the pieces to be properly controlled and re-generated at the top, there are two fundamental parts missing: checking for and deleting the full rows and checking for the GAME OVER. Both of those should only be done when one active piece cannot fall any further and another one is about to be generated at the top. State 4 is called to check the 4 rows that were occupied by the previously active piece from top to bottom. This is accomplished by having another state machine inside the 4th state. The variable **row_state**, initially assigned to 1, gets re-assigned to 2 and calls the main state 5, that will check the particular row specified by the old y_coordinate of the previously active piece reduced by 1 (to get the top row of the 4 by 4 square). State 5, will read through all 10 squares of the particular row from the memory and will go back to state 4 in case any of the squares are empty. If, all the squares in the row are in fact occupied, state 5 calls state 6, previously assigning the number of the checked row to the proper variable. State 6 will fill the values of the row with zeroes if the row number is 0. Otherwise, it will call state 7 and decrement the input row by one. State 7 will read the values from the row, one above the input row, write those values into the input row, and return back to state 6. After state 6 is done, it will return to state 4 to check for the next row.

State 8 is used for assigning the inactive squares to the new piece about to be generated. This requires reading from the memory the values stored in the 4 by 4 square of the newly generated piece. The **initial_X_value** and **initial_Y_value**, that are set to 4 and 1 as of now, are used to read the values from the memory. The values in the X region 3-6 and Y region 0-3 are stored into **ina_row1 [3:0]** to ina**_row4[3:0]**. This way, before generating the new piece, it is possible to check if there is place to generate it. If there is no free space for the newly generated piece, the game is terminated.

Now we have covered in some detail all the basic functionality of the program. There is a next piece feature that is based on the randomly generated values. These values come from a variable that is incremented up to 13 every clock cycle and re-assigned to 0 once it reaches 13. The next piece is chosen every time the previous piece is generated at the top of the screen. 0 and 1 values correspond to I, 2 and 3 – to J, 4 and 5 – L, 6 and 7 – O, 8 and 9 – T, 10 and 11 – S, 12 and 13 – Z.

There is also a reset button that calls another upper level state and writes 0s to all the values in the BRAM.

2.2. The Memory Module

For our memory module we have used the dual-port BRAM. The Game Engine had to have both write and read privileges, while the Display module only had to read from memory. The dual-port was used to store the values for the colors of the squares in the 10 by 20 grid used for the game. The accessing format was the concatenation of the X coordinate of the square and the Y coordinate of the square, so the address was 4 + 5 bits long (4 to store values from 0 to 9 and 5 to store values from 0 to 19). The size of each entry was 8 bits even though we ended up using only the last 3 bits of those values. The BRAM was automatically generated according to the above specifications.

2.3. The Display Module - Lily

This module takes as input the BRAM holding the colors of each square in the game grid.

The display is refreshed at every update of the game. To access the color of each square, for each x from 1 to 10 and for each y from 1 to 20, the module reads from the address resulting from the concatenation of the x and y coordinates. The output is a pixel at the appropriate square of the corresponding color.

2.3.1 The Seven Tetris Shape Submodules (L, J, I, T, S, Z, Sq):

Each of the seven tetris shapes is created in a module that outputs to the right panel of the display. The module that controls which one shows at a certain time takes as input next_piece from the game engine and updates itself accordingly.

2.3.2 Music Module

This module plays a simple four-note tune (C, G4, G3, G4) from the speakers connected to the labkit throughout the game. If we had more time, the music could easily be extended to sound on different events, such as clearing a row or hitting a particular high score.

2.3. The Input module

We had 7 inputs implemented in the game. All of them were properly debounced prior to use. The left and right buttons would move the active piece in the appropriate direction if the game rules allow such a movement. Otherwise the inputs are ignored. In the Game Engine Module there is a parameter specifying how many times in a row should the button be pressed for a particular press to be counted as an input since obviously if the Game Engine would react to every input, the movement would be too fast for a human to control. The up button would rotate the active piece, once again only if possible. The down button would speed up the movement of the piece in the vertical direction. The reset and start buttons would basically clear the memory and start the game from the beginning. The pause button would pause the game putting it in an intermediate state that would only be broken out of if the pause button, start button or reset buttons are pressed.

Conclusion:

As of now, clearing the rows functionality does not work and the game pauses once a full row is found. The score variable is also not assigned as the result. The speed variable is not being incremented since initially its value was supposed to depend on the success rate of the opponent playing on another labkit.

Other than that, the functionality problems were overcome and it is actually possible to play the game.

Testing:

We tested the display module by creating a tester module that feeds in as input a test BRAM filled with hardcoded data, and checked with the switches to see if it displayed as expected on the screen.

After finishing the display module, we were able to plug it directly into the game engine module for testing.

Hex-display was rather unsuccessfully used for debugging the Game Engine at an earlier stage. It was successfully used for the input checking though.

Memory module was tested together with display module and separately using the hex-display.

Difficulties and Resolution

The biggest problem I ran into was the bug in the reading memory state. I had a case statement that would store the proper address into the address specified to the BRAM and then read the value passed by the BRAM during the next cycle. To do that I was incrementing the state variable passed to the case statement. The method did not seem to work and was creating highly complicated bugs. In order to debug the problem I have decided to limit myself to only one shape (a bar) and test different cases by supplying constant inputs to some variables instead of reading them from the memory. This has led to even more interesting bugs, until finally I was forced to disable the reading functionality altogether and integrate the other functions. The reason behind this curious behavior was discovered by Chris during the demonstration. It takes one cycle for the address to be assigned the proper value and one more cycle for the BRAM itself to process the new address and spit out the proper output. To I needed a two cycle gap between assigning the address and reading its corresponding value from memory.

Other confusions were minor. We have decided to pass the frame_ready variable from the Game Engine Module to the display module since we did not want the display module to start drawing the current state until the Game Engine module is done changing it. This variable was completely useless and having it caused some display glitches.

Another glitch was in random generator which I initially assigned to be a 3 bit value incremented every cycle. Because it was a power of two, the shapes generated at the top would repeat themselves (actually the color would repeat, since the shape was set to constant anyways). The easy way to fix the problem was assigning the counter to 0 and clearing it once it reaches 13, that way the 14 values can properly be distributed among the potential 7 shapes.

Appendix A. Python Code used to generate Verilog (Lily)

```
print 'always @ (posedge vclock) begin'
print ' if (frame_ready) begin'
print '     if ((vcount >= (y+0*HEIGHT)) && (vcount < (y+20*HEIGHT))) begin
//large-scale boundary for y'
print '     if((hcount >= (x+0*WIDTH)) && (hcount < (x+10*WIDTH))) begin
//large-scale boundary for x'

print '              if ((hcount >= (x+' + str(0) + '*WIDTH)) && (hcount < (x+'
+ str(1) +'*WIDTH))) begin'
print "                 pixel <= bram_doutb[2:0];"
print '                 if ((vcount >= (y+' + str(0) + '*HEIGHT)) && (vcount <
(y+' + str(1) + '*HEIGHT))) begin'
print "                     bram_addrb <= {4'd" + str(0) + ",5'd" + str(0) +
"};"
#print '                    pixel = bram_doutb;'
#print "                     pixel = 3'b100;"
print '                 end//if vcount'
for n in range(1, 20):
    print '                 else if ((vcount >= (y+' + str(n) + '*HEIGHT)) &&
(vcount < (y+' + str(n+1) + '*HEIGHT))) begin'
    print "                     bram_addrb <= {4'd" + str(0) + ",5'd" + str(n)
+ "};"
    #print '                    pixel = bram_doutb;'
    #print "                     pixel = 3'b111;"
    print '                     end//if vcount'
print '                 end//if hcount'#initial conds

for m in range(1, 10):
    print '      else if ((hcount >= (x+' + str(m) + '*WIDTH)) && (hcount <
(x+' + str(m+1) +'*WIDTH))) begin'
    print "             pixel <= bram_doutb[2:0];"
    print '             if ((vcount >= (y+' + str(0) + '*HEIGHT)) && (vcount <
(y+' + str(1) + '*HEIGHT))) begin'
    print "                 bram_addrb <= {4'd"  + str(m) + ",5'd"  + str(0) +
"};"
    #print '                 pixel = bram_doutb;'
    #print "                 pixel = 3'b101;"
    print '                 end'
    for n in range(1, 20):
        print '             else if ((vcount >= (y+' + str(n) + '*HEIGHT)) &&
(vcount < (y+' + str(n+1) + '*HEIGHT))) begin'
        print "                 bram_addrb <= {4'd"  + str(m) + ",5'd"  + str(n)
+ "};"
        #print '                 pixel = bram_doutb;'
        #print "                 pixel = 3'b110;"
        print '                 end//if vcount'#repeat this block
    print '             end//if hcount'


print '             end//if WIDTH'
print '         else'
print "             pixel <= 3'b0;"
print '     end//if HEIGHT'
print '         else'
print "             pixel <= 3'b0;"
print ' end//ready'
print 'end//always'
```

Appendix B. Display.v (Selected segments, original file too long)

```
///////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version)
//
///////////////////////////////////////////////////////////////////////////////

module debounce (reset, clock_65mhz, noisy, clean);
   input reset, clock_65mhz, noisy;
   output clean;

   reg [19:0] count;
   reg new, clean;

   always @(posedge clock_65mhz)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 650000) clean <= new;
     else count <= count+1;

endmodule



   ////////////////////////////////////////////////////////////////////////////
   //
   // The tetris game
   //
   ////////////////////////////////////////////////////////////////////////////

   // use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)
   wire clock_65mhz_unbuf,clock_65mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
                 .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;

   // ENTER button is user reset
   wire reset, user_reset;

   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank);

//Tetris Game Instantiations
   wire [2:0] pixel;
   wire thsync,tvsync,tblank;

       wire frame_ready;
       wire [9:0] score;
```