University of Toronto, St. George

CSC258H1

Computer Organization

Final Project Report

Authors:
Makram Kamaleddine
Theresa Ma

Supervisor:
Steve Engels

April 4, 2014

1 Introduction

What follows is the report on our CSC258 project, which involved designing and implementing the classic game brick-breaker using the Altera DE2 as the device to run it on and the Verilog Hardware Description Language as a language to implement it in. In this section we describe, albeit briefly, the idea behind why this is an interesting thing to do on the DE2 (to students of this class and anybody else outside of it) as well as what technical aspects of it are in common with what was done in the course. We will expand on this later on in the report in greater detail.

1.1 Why is it cool?

The Altera DE2 board is a great tool for what it was designed for: teaching students. It comes equipped with the ability to display onto the VGA (video graphics array) display hardware, and also can play sound, receive input from the keyboard, and has a built-in LED screen on it as well. Doing Brickbreaker involved all of these things initially: a game is not just a computer program that runs in the command line (well, they used to be, but not anymore). A modern game requires, at the minimum, the following things:

- A visual depiction of the game itself. This is typically referred to as the graphics aspect of the game. Animations, art, realistic physics, and everything similar or derived from these things falls under this category.
- Sound. The best games aren't just games with incredible graphics or gameplay, but also with a great soundtrack and realistic (or extremely unrealistic) sound effects. All modern games put a huge effort into this category.
- Input. Are the controls the game is using intuitive? Are they getting in the way of you enjoying the game? Keyboard input and mouse input are by far the most common input peripherals for all modern games on a computer.
- Rudimentary "engine" to do all the game logic crunching. Virtually all games possess some sort of 'logic' behind their execution: they always have a finite number of *states*: for example, in a first-person shooter, you can either be alive, almost-dead (in your last stand, for example), almost-alive (about to respawn), or dead, and your power within the games is really affected by these fundamental states. This is just a simplified example, but you can see how this scales into much more states and much more in-game control.

So, if the above didn't convince you, it turns out that making the above work in harmony is actually very difficult, but extremely rewarding. Thousands and thousands of pages have been written about game development, and the summary given above really does not do the field justice at all. Many games are released every year, and with each coming year, they keep getting better at displaying graphics, integrating sound, and coming up with revolutionary control schemes and systems (such as the Microsoft Kinect and Playstation Move, and more recently Oculus Rift) to make the game experience richer in every way. If that's not cool, I don't know what is!

1.2 Relevance to the course

The game was written in the Verilog Hardware Description Language, from now on referred to just as Verilog. Verilog lets the programmer declare and define combinational as well as sequential circuits in a very natural and (somewhat) verbose way. It has features from a high-level language like C++, such as addition, subtraction, conditionals, etc., but also features that are somewhat unfamiliar, such as declaring registers, wires, clock configurations, and weird pseudo-high-level "loops", as well as concurrency support via non-blocking statements. Verilog was taught by Steve in the second or third week of the course, and has been used in the labs ever since.

The game made use of the following ideas from the course: finite-state machines, synchronous circuits, as well as addition/subtraction of signed numbers in binary, and sequential circuits. The game is in essence a large finite state machine: there are many states that must be dealt with manually in Verilog because the abstraction provided by the language is extremely low: i.e organizing the code in a manner that would be logical to a game is somewhat difficult, so the whole game was thankfully simple enough to fit in one module (I say 'fit' here in a more human sense, as in it is still decipherable: obviously the compiler doesn't care how big your module is).

Also, we made extensive use of the concurrency support provided by Verilog as well as the `always` and the `initial` block, as well as built-in clock support. As we will see, the clock is very important to the execution of our code.

2 Details

In this section we go into the details of the design and the implementation of the game. Brickbreaker is a game almost as old as the world of video games itself. Here we will also state some of the technical design and implementation decisions made during the implementation of the game.

2.1 Designing Brickbreaker

The premise of Brickbreaker is very simple: on the screen you have a bunch of bricks, drawn in some particular configuration. The player controls a paddle at the very bottom of the screen, and the paddle is used to deflect a ball, which is what can break the bricks, after colliding into them. When all the bricks are broken (i.e destroyed) the player wins the current level, and possibly the game. When you break it down into game entities, here is what you get:

- *The ball.* The ball is the only thing in the game that can break the bricks, but it is not being controlled by the player! The ball initially starts out on the paddle, and when the game starts, is released in one of two directions: left or right. After this it can only be deflected by the paddle in such a way that it can destroy the bricks on the screen by colliding with them. The ball can be in many positions during the game. This made the implementation somewhat difficult.

- *The paddle.* The paddle is what is being controlled by the player. The paddle is a relatively simple object: it can only move left or right. The simplicity of the code used to control the paddle reflects this, as we shall see later.

- *The bricks.* The bricks are the adversary in this game. The goal is to destroy all the bricks so that we can reach the next level. The bricks were single-handedly the most complicated thing to implement in Verilog, simply because there is no supported way of defining an object with attributes in Verilog. The most we can do is define a bunch of local parameters via `localparam` or parameters that can be changed via the `parameter` keyword.

2.2 Implementing Brickbreaker

Alright, enough design talk. Lets get down to the code part: how was this done? Well, there are really two parts. The first part is something that we wrote nothing of, and that is the VGA controller and VGA sync modules, which is what was actually used to display the game onto the monitor screen. For this complete credit goes out to Dr. John Loomis [?]. We used one of his VGA labs for the Altera DE2 board, called `vgalab1`. It was the starting point of our project.

The main file that we created and edited throughout the implementation process is a file called `paddle.v`, which you can find on GitHub [?]. We show excerpts of it here and explain parts of the implementation here and there, but the full file is too long to fit in this report (and the syntax highlighting is better on GitHub).

2.2.1 The clock

The clock was a very important aspect of our implementation. We essentially use the built-in clock support of Verilog three times, and in a very important manner: to move the paddle and to draw the bricks on the screen, as well as to detect collisions. Here are the interfaces for these three times:

```
// declare variables (registers to hold brick positions, paddle positions, etc.)
initial begin
//initialize all variables
end
always @(posedge moveleft or posedge moveright)
begin
// code to move the paddle
end
//...
always @(posedge clk)
begin
// code to detect collisions with bricks
end
//...
always @(x, y, clk, moveleft, moveright)
begin
// code to draw the bricks on the screen
end
```

And here is the interface for our module in it's final form:

```
module paddle(input [9:0] x, input [9:0] y,
input [9:0] red, input [9:0] green, input [9:0] blue,
input clk, input moveleft, input moveright);
```

The `clk` input was given the built-in 50MHz clock on the DE2 board, but it had to be slowed down to 1MHz in order to get a ball and paddle speed that was acceptable to the user. The `moveleft` and `moveright` inputs are mainly for the paddle: the first will move the paddle left and the second will move the paddle right. As you can see in the first code sample, we move it at the positive edge of the input, which is ideal for keystrokes.

To slow down the clock we did something similar to what we did in one of the labs: we declared a register large enough to store a number up to 25 million (because initially that's what we thought would be the ideal clock pulse rate), and when the desired number of clock pulses is reached, then we execute our code for collisions and whatnot. Here's a snippet of that code:

```
reg[25:0] slowclk = 0;
// other variable declarations here...
always @(posedge clk)
begin
    slowclk = slowclk + 1; // notice the blocking assignment

    if(slowclk == 26'hf4240)begin
        slowclk = 0; // another blocking assignment
        // actual collision code
    end
end
```

The blocking assignment is crucial here because we want to increment the value of `slowclk` first before we execute anything inside the `if` block. When we satisfy the `if` condition, we immediately use another blocking assignment to set `slowclk` to 0, so that we can keep using this slower clock.

We also used this positive edge of the clock idea to move the paddle, as you can see with this code here. `paddlex1`, `paddlex2` represent the leftmost and rightmost x values of the paddle on the screen, and `paddley1` and `paddley2` represent the top and bottom y values of the paddle on the screen. The intricacies of this will be explained in a later section.

```
always @(posedge moveleft or posedge moveright)
begin

if (moveleft)
begin
    if (paddlex1 > 10)
        paddlex1 = paddlex1 - 10;
    end

if (moveright)
begin
    if (paddlex1 < 510)
        paddlex1 = paddlex1 + 10;
    end
end
```

Notice that these are all blocking statements. Although this may not be necessary upon first sight, but Verilog does not allow the programmer to mix blocking and non-blocking statements for the same register (and rightfully so). We will see in the next section that in our initial statement, we wanted to draw the paddle and the ball before we drew the bricks.

2.2.2 Displaying onto the screen, collisions

We needed to understand exactly what was going on with the screen before we could implement collision detection. It turns out that the VGA screen according to the DE2 has it's origin in the top left corner (i.e $(x, y) = (0, 0)$ in the top left corner), so that when the ball is falling down, it's y value is actually *increasing*, not decreasing. So keeping this in mind, we had to draw things onto the screen in such a way that we keep the following invariants in our code: the ball is always either moving left or right as well as moving up or down, and the bricks are in two possible states: alive or dead. This observation is what made it possible to come up with a finite state machine (albeit with many states and conditions) to program our collision. Here are some examples:

```
reg[9:0] paddlex1, paddlex2, paddley1, paddley2; // paddle coordinates
reg[9:0] ballx1, ballx2, bally1, bally2; // ball coordinates
reg[9:0] ballvx, ballvy; // ball speed
reg[9:0] brick1x1, brick1y1, brick1x2, brick1y2; // brick coordinates
reg brick1vis = 0; // if 0 then brick is visible; otherwise it's not visible
reg[9:0] brick2x1, brick2y1, brick2x2, brick2y2;
reg brick2vis = 0;
//... more declarations...
reg[9:0] brick7x1, brick7y1, brick7x2, brick7y2;
reg brick7vis = 0;
reg[9:0] brick8x1, brick8y1, brick8x2, brick8y2;
reg brick8vis = 0;
```

All of these register values are assigned in the `initial` statement, and we see from this snippet that we wanted the ball and the paddle to be drawn to the screen first:

```
initial begin
    paddlex1 = 220;
    paddley1 = 450;
    paddlex2 = 120;
    paddley2 = 30;

    ballx1 = 310;
    bally1 = 230;
    ballx2 = 20;
    bally2 = 20;

    ballvx = 1;
    ballvy = 1;
    // more definitions...all nonblocking
end
```

Finally, with all this put together, we use the third `always` block to draw the bricks:

```
always @(x, y, clk, moveleft, moveright)
begin
    if ((paddlex1 < x) & (x < paddlex2 + paddlex1)
    & (paddley1 < y) & (y < paddley2 + paddley1))
    begin
        idx = 3'd0;
    end
    else if ((ballx1 < x) & (x < ballx2 + ballx1)
     & (bally1 < y) & (y < bally2 + bally1) )
    begin
        idx = 3'd1;
    end

    else if ((brick1x1 < x) &
    (x < brick1x2 + brick1x1) & (brick1y1 < y)
    & (y < brick1y2 + brick1y1) )
    begin
        if (brick1vis == 1)
            idx = 3'd7;
        else if (brick1vis == 0)
            idx = 3'd2;
     end
     // ... more conditions and code
end
```

In the end, the `always` blocks worked in a sort of crooked harmony, and the code was fairly organized (for beginner's Verilog code): we had one block to move the paddle, one block to detect collisions with the ball, and the last block was to draw the things on the screen. Finally, we owe it to Dr. Loomis again for the piece of code that made all of this work:

```
wire [9:0] r, g, b;

paddle c1(mCoord_X, mCoord_Y, r, g, b, CLOCK_50, SW[17], SW[16]);

wire [9:0] gray = (mCoord_X<80 || mCoord_X>560? 10'h000:
(mCoord_Y/15)<<5 | (mCoord_X-80)/15);

wire s = SW[0];
assign mVGA_R = (s? gray: r);
assign mVGA_G = (s? gray: g);
assign mVGA_B = (s? gray: b);
```

This is the code that actually displays everything onto the VGA display.

3 Conclusion

All in all the project was a good experience: we learned a whole lot about the naivete of the design phase and the difficulty of the implementation phase. We want to thank Steve and the TA's in our lab for putting up with the students. It's been a fun semester!

References

[1] Dr. John S. Loomis, `http://johnloomis.org/`, University of Dayton, Dayton, OH.

[2] Theresa Ma, `https://github.com/theresama/another-brick-in-the-wall`, University of Toronto St. George, Toronto, ON.