Matrix Multiplier

Scott Madeux 12/19/2018 ECEN 340

Project Design

For this project I wanted to use three 3-dimentional vectors, which represent the three corners of a triangle in 3D space. I wanted to be able to transform these vectors on the Basys-3 using matrix multiplication and speed things up a lot by doing those operations in parallel. The transformations I chose to apply to these vectors were scaling, translation in the X direction, translation in the Y direction, and translation in the Z direction. With more time, more transformations could have been added such as rotation, shearing, and reflection.

Here is an idea of how these transformations are done:

$$\begin{pmatrix} S & 0 & 0 & X \\ 0 & S & 0 & Y \\ 0 & 0 & S & Z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} AS+X & BS+X & CS+X \\ DS+Y & ES+Y & SF+Y \\ SG+Z & SH+Z & SI+Z \\ 1 & 1 & 1 \end{pmatrix}$$

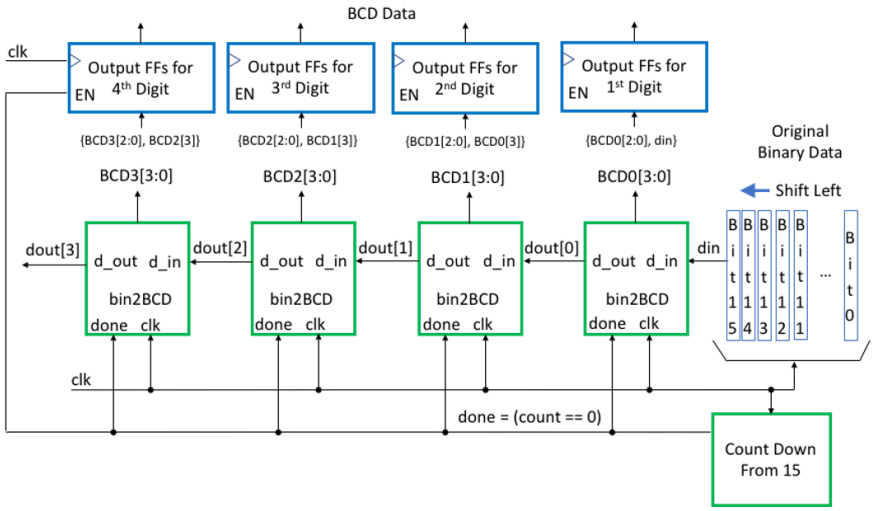
S is the number that the vectors will be scaled by.

X, Y, and Z are the amount the vectors will be translated in each direction.

$[A\ D\ G]^T$, $[B\ E\ H]^T$, and $[C\ F\ I]^T$ are the three vectors to be transformed.

As you can see this transformation takes a total of 9 multiplications and if we were to add more kinds of transformations it would take 27+ multiplications. This program will be able to all of these multiplications in parallel. This means the same principle would be applied much larger matrices and the time to do the calculations would be the same.

I used a 16-bit number for each entry in the matrices, so I needed a way to convert those binary numbers into something more human readable. For this I used a binary to binary coded decimal converter. This used a slightly modified version of the double dabble algorithm to reduce the number of shifts necessary to convert the binary number.



The 16 bits of the binary number are shifted in one by one into the binary to BCD converter.

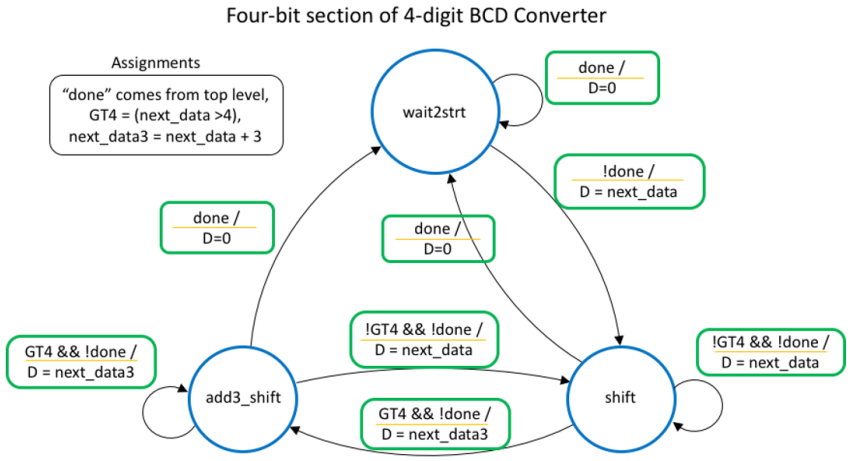


Figure 4 – State Diagram for Preferred 4-bit binary to Single-Digit BCD Conversion

Here is a state machine diagram for each 4-bit section of the converter.

The binary to BCD converter also uses the seven-segment display on the FPGA board to display the number that it has converted.

The three technologies this project includes are state machines, use of the seven-segment display, and higher-level math functions (multiplication).

Here is an overview of the controls for the matrix multiplier:

sw[3:0] - select the entry of the matrix you want to view/edit in binary (0-8)

sw[9] - view/edit the matrix holding the three vectors to be translated

sw[10] view the matrix holding the three vectors after they have been transformed

sw[11] - view/edit translation in the Z direction

sw[12] - view/edit translation in the Y direction

sw[13] - view/edit translation in the X direction

sw[14] - view/edit how much the vectors will be scaled

btnU – increment the value of the item that is being viewed

btnD – decrement the value of the item that is being viewed

btnL & btnC together – perform matrix multiplication

Code

Other code I used was the binary to BCD converter and the seven-segment display, which has been used previously in the semester. To save space, I only included the code used in my top level module.

matrix_multiplier.v

```
module matrix_multiplier(
    input clk,
    input [3:0] matPos,
    input origMat,
    input modMat,
```

```
input scaleSel,
input transXSel,
input transYSel,
input transZSel,
input btnU,
input btnD,
input btnC,
input btnR,
input btnL,
output [3:0] an,
output dp,
output [6:0] seg
);

//debounced clock for buttons
wire clk_deb;

//The five debounced buttons
wire numUp;
wire numDown;
wire bMultiply;
wire bRst;
wire bExecute;

//Button presses delayed by 1 clock cycle
reg numUp_tm1;
reg numDown_tm1;

//Singal to increment or decrement what is being viewed
reg inc_num;
reg dec_num;

//Singal to do matrix multiplication
reg multiply_mat;

//The 16 bit number currently being viewd
reg [15:0] current_num_view;

//Which entry in the matrix
reg [4:0] addr;

//Matricies
reg [15:0] origin_mat3x3 [8:0];
reg [15:0] mod_mat3x3 [8:0];
reg [15:0] trans_mat4x4 [15:0];

//Set the debounce clock and debounce the buttons
debounce_div div1(.clk(clk), .clk_deb(clk_deb));
btn_debounce up(.clk(clk_deb), .btn_in(btnU), .btn_status(numUp));
btn_debounce down(.clk(clk_deb), .btn_in(btnD), .btn_status(numDown));
```

```
btn_debounce left(.clk(clk_deb), .btn_in(btnL), .btn_status(bMultiply));
//Add reset button incase I need it in the future
btn_debounce right(.clk(clk_deb), .btn_in(btnR), .btn_status(bRst));
btn_debounce center(.clk(clk_deb), .btn_in(btnC), .btn_status(bExecute));

//Send the number currently being vied to the Binary to BCD converter to be displayed
//on the seven segment display
bin2BCD_4dig diplay_num(.clk(clk), .sw(current_num_view), .seg(seg), .dp(dp), .an(an));

//Assign first four switches to addr
always@(matPos)
begin
    if(matPos < 4'h9) addr = matPos;
    else addr = 4'h8;
end

//Choose which number goes to the seven segment display
always@(*)
begin
    if(origMat) current_num_view = origin_mat3x3[addr];
    else if(modMat) current_num_view = mod_mat3x3[addr];
    else if(transZSel) current_num_view = trans_mat4x4[4'hb];
    else if(transYSel) current_num_view = trans_mat4x4[4'h7];
    else if(transXSel) current_num_view = trans_mat4x4[4'h3];
    else if(scaleSel) current_num_view = trans_mat4x4[addr];
    else current_num_view = 16'h0000;
end

//Increment signal delayed by 1 cycle
always@(posedge clk)
begin
    numUp_tm1 <= numUp;
    numDown_tm1 <= numDown;
end

//What is happening based on button presses
always@(*)
begin
    multiply_mat = bMultiply && bExecute;
    inc_num = numUp && !numUp_tm1;
    dec_num = numDown && !numDown_tm1;
end

//Procedures for each button press function
always@(posedge clk)
begin
    if(inc_num)
    begin
        //Increment what is being viewed
        if(origMat) origin_mat3x3[addr] = origin_mat3x3[addr] + 1;
```

```
            else if(transZSel) trans_mat4x4[4'hb] = trans_mat4x4[4'hb] + 1;
            else if(transYSel) trans_mat4x4[4'h7] = trans_mat4x4[4'h7] + 1;
            else if(transXSel) trans_mat4x4[4'h3] = trans_mat4x4[4'h3] + 1;
            else if(scaleSel)
            begin
                trans_mat4x4[4'h0] = trans_mat4x4[4'h0] + 1;
                trans_mat4x4[4'h5] = trans_mat4x4[4'h5] + 1;
                trans_mat4x4[4'ha] = trans_mat4x4[4'ha] + 1;
            end
        end
        else if(dec_num && current_num_view > 16'h0000)
        begin
            //Decrement what is being viewed if it is greater than 0
            if(origMat) origin_mat3x3[addr] = origin_mat3x3[addr] - 1;
            else if(transZSel) trans_mat4x4[4'hb] = trans_mat4x4[4'hb] - 1;
            else if(transYSel) trans_mat4x4[4'h7] = trans_mat4x4[4'h7] - 1;
            else if(transXSel) trans_mat4x4[4'h3] = trans_mat4x4[4'h3] - 1;
            else if(scaleSel && current_num_view > 16'h0001)
            begin
                trans_mat4x4[4'h0] = trans_mat4x4[4'h0] - 1;
                trans_mat4x4[4'h5] = trans_mat4x4[4'h5] - 1;
                trans_mat4x4[4'ha] = trans_mat4x4[4'ha] - 1;
            end
        end
        else if(multiply_mat)
        begin
            //Steps for multiplying matricies in parallel
            mod_mat3x3[4'h0] <= trans_mat4x4[4'h0]*origin_mat3x3[4'h0] + trans_mat4x4[4'h3];
            mod_mat3x3[4'h1] <= trans_mat4x4[4'h0]*origin_mat3x3[4'h1] + trans_mat4x4[4'h3];
            mod_mat3x3[4'h2] <= trans_mat4x4[4'h0]*origin_mat3x3[4'h2] + trans_mat4x4[4'h3];
            mod_mat3x3[4'h3] <= trans_mat4x4[4'h5]*origin_mat3x3[4'h3] + trans_mat4x4[4'h7];
            mod_mat3x3[4'h4] <= trans_mat4x4[4'h5]*origin_mat3x3[4'h4] + trans_mat4x4[4'h7];
            mod_mat3x3[4'h5] <= trans_mat4x4[4'h5]*origin_mat3x3[4'h5] + trans_mat4x4[4'h7];
            mod_mat3x3[4'h6] <= trans_mat4x4[4'ha]*origin_mat3x3[4'h6] + trans_mat4x4[4'hb];
            mod_mat3x3[4'h7] <= trans_mat4x4[4'ha]*origin_mat3x3[4'h7] + trans_mat4x4[4'hb];
            mod_mat3x3[4'h8] <= trans_mat4x4[4'ha]*origin_mat3x3[4'h8] + trans_mat4x4[4'hb];
        end
    end

endmodule
```

Peer Reviews

First Peer Review: Johnathan Ashcraft and Jake Robertson

Some feedback that was given was that it would be nice to have a user friendly way to display the matrix instead of just using the seven segment display. I totally agree, it would be great to be able to connect a small LCD display to the Basys-3 to be able to see an entire matrix at one time rather than seeing only one entry at a time.

Second Peer Review: Johnathan Ashcraft

The feedback that was given for the follow up peer review was that a good next step would possibly be to implement negative numbers into the transformations. This would probably involve changing everything to two's compliment, using a multiplier that could handle two's compliment and changing the binary to BCD converter to also deal with two's compliment.

Conclusion

This project took a lot of trial and error to get just right without making things too complicated. I tried to using things like memory modules with bidirectional data busses to store my matrices, but since the multiplications eventually needed to happen in parallel, have 9 or 16 bidirectional busses would make things a little more complicated than they needed to be.

The final product turned out great and everything is functioning properly. The vectors are all being scaled and translated the way they should. I went and analyzed the timing for the multiplier and found that the worst negative slack is 2.393ns making the maximum clock frequency this could run at 131Mhz. Also, if you took some single core processor that could do multiplications at the same rate as this circuit, this matrix multiplier would be 9 times faster because it is doing these multiplications in parallel, while a single core processor could only do one at a time. This can be scaled up to and is only limited by the space available on the basis 3 FPGA.