FPGA Project - Report

Alireza Rostami 9832090
Sina Amini 9832081

1 Introduction

The function we are going to work with is defined as follows:

$$y = x^2 - x^3 \quad \text{where} \quad 0 < x < 1 \tag{1}$$

The equation can be simplified as:

$$y = x^2 \cdot (1 - x) \tag{2}$$

Since we have 8 bits to represent the input and output, and our numbers are in range of $0 < x < 1$, our number system can be thought of as $Q_{1.8}$ since the 8 bits we have will only be used to represent the fractional part. Therefore we can easily ignore the integer part since it will always be 0.

Since we have 8 bits, we have $2^8 = 256$ different number, therefore, the smallest number would be $\frac{(1-0)}{256} = 0.00390625$ and the difference of two consecutive number would also be to 0.00390625. The table of numbers would be as follows:

Index	Decimal	Binary
0	0.00000000	0.**00000000**
1	0.00390625	0.**00000001**
2	0.00781250	0.**00000010**
⋮	⋮	⋮
254	0.99218750	0.**11111110**
255	0.99609375	0.**11111111**
256	1.00000000	1.**00000000**

Table 1 – Table of numbers and their corresponding binary representation.

Note that in the table above, the cases 0 and 256 are not feasible, since x is strictly less than 1 and strictly greater than 0.

Since the LUT must be populated before we use it for interpolation, we must compute the values of y for the numbers shown in the table above. We can simplify the function; in our case, $1 - x$ would be equal to the two's complement of the 8-bit number. For example, for number 0.00000001, we would get:

$$\begin{array}{r} 1.\mathbf{00000000} \\ -\,0.\mathbf{00000001} \\ \hline 0.\mathbf{11111111} \end{array}$$

Figure 1 – Computation of $1 - x$ for $x = 0.00000001$

Therefore, we can write the function as:

$$y = x^2 \cdot \overline{x} \tag{3}$$

We can write a code snippet in a higher language (for example c) to pre-calculate the values for y.

2 Pre-Calculator

We will write our pre-calculator in *c*. Since our number is 8 bits, we will use a **char** variable. Recall that multiplying a $Q_{1.8}$ number by another $Q_{1.8}$ will result in an $Q_{2.16}$ number. Since the integer part is to be ignored, we only need to control bit-growth of the fractional part. Since we have 3 multiplications, we would have a 24-bits number; if we right-shift the number, we would get an 8-bits number.

```
unsigned char calculator(const unsigned char x)
{
    unsigned char x_bar = complement(x);
    unsigned short x_pow = (pow(x, 2));
    unsigned char y = (x_pow * x_bar) >> 16;
    return y;
}
```

The `complement` function can be implemented using the ~ operator. The ~ operator returns the 1's complement of the number; thus, by adding 1 to it, we get the 2's complement representation of the number.

```
unsigned char complement(const unsigned char x)
{
    unsigned char z = ~x + 1;
    return z;
}
```

Now we write a for-loop to calculate y for every number and then store the achieved output to an file which will be used in as a read-only memory (ROM) in our Verilog code.

```
void writeToROM(FILE * ROM, size_t const size, void const * const ptr)
{
    unsigned char *b = (unsigned char*) ptr;
    unsigned char byte;
    int i, j;

    for (i = size-1; i >= 0; i--) {
        for (j = 7; j >= 0; j--) {
            byte = (b[i] >> j) & 1;
            fprintf(ROM, "%u", byte);
        }
    }
    fputs("\n", ROM);
}

int main()
{
    FILE *ROM;
    ROM = fopen("ROM.bin","wb");
    if (ROM == NULL) {
        printf("Error!");
        exit(1);
    }
    unsigned char x = 0;
    for (int i = 0; i < 256; i++) {
        unsigned char y = calculator(x);
        x++;
        writeToROM(ROM, sizeof(y), &y);
    }
    return 0;
}
```

This snippet is implemented in the file named `PreCalculator.c`.

3 Reading from File in Verilog

We now read this file in our Verilog module.

```
reg [7:0] ROM [0:255];
initial $readmemb("ROM.bin", ROM);
```

By now, we have successfully implemented the first part of the project.

4 Implementing Linear Interpolation

Since $2^6 = 64$, therefore in the second part of the project we need 6 bits (x_{MSB}) for addressing the LUT. The other 2 bits are used for interpolation. The formula for the linear interpolation is as follows:

$$y = ROM[x_1] + \frac{x_{\text{LSB}} \times (ROM[x_2] - ROM[x_1])}{2^{N-P}} \tag{4}$$

Since $N = 8$ and $P = 6$, thus $N - P = 2$, thus:

$$y = ROM[x_1] + round(x_{\text{LSB}} \times (ROM[x_2] - ROM[x_1]), 2) \tag{5}$$

Since our LUT was 8-bit long, we do not need to go through all the process we did to get the original LUT once more. Instead, we just read by every 4 slot to get the new LUT.

```
integer i;
reg [7:0] ROM_64 [0:63];


// Initiating the 64-slots ROM.
initial begin
    for (i = 0; i < 64; i = i + 1) begin
        ROM_64[i] <= ROM[i*4];
    end
end

reg [7:0] y_1;
reg [7:0] y_2;
reg [7:0] y_3;

wire [5:0] XMSB;
wire [1:0] XLSB;
assign XMSB = x_input[7:2];
assign XLSB = x_input[1:0];

always @(posedge clk) begin
    y_1 <= ROM_64[XMSB+0];
    y_2 <= ROM_64[XMSB+1];
    y_3 <= ROM_64[XMSB+2];
end
```

Now, we pass the parameters to the interpolator to get the result.

```
module Linear_Interpolator (
    input [7:0] y_1,
    input [7:0] y_2,
    input [1:0] XLSB,
    output [7:0] y_linear
    );

    wire [9:0] product_linear_output;
    assign product_linear_output = XLSB * (y_2 - y_1);
    assign y_linear = y_1 + product_linear_output[9:2] + product_linear_output[1];

endmodule
```

By now, we have successfully implemented the second part.

5 Implementing Quadratic Interpolation

Like the last section, 6 bits are used for addressing and 2 bits are used for interpolation. The formula for quadratic interpolation is as follows:

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1} \times (x - x_1) + \frac{1}{x_3 - x_1} \times \{\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1}\} \times (x - x_1)(x - x_2) \tag{6}$$

$$= y_1 + \frac{x - x_1}{x_2 - x_1} \times (y_2 - y_1) + \frac{(x - x_1)(x - x_2)}{x_3 - x_1} \times \{\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1}\} \tag{7}$$

$$= y_1 + \frac{x - x_1}{2^{N-P}} \times (y_2 - y_1) + \frac{(x - x_1)(x - x_2)}{2^{N-P+1}} \times \{\frac{y_3 - y_2}{2^{N-P}} - \frac{y_2 - y_1}{2^{N-P}}\} \tag{8}$$

$$= y_1 + \frac{x - x_1}{2^{N-P}} \times (y_2 - y_1) + \frac{(x - x_1)(x - x_2)}{2^{2(N-P)+1}} \times \{y_3 - 2y_2 + y_1\} \tag{9}$$

Now, we have to find a relationship between x, x_1 and x_2. We assume $x_1 < x < x_2$.

Suppose x is the following:

$$\begin{aligned} x &= \boxed{0\ 0\ 0\ 0\ 0\ 1}\,\|\,\boxed{1\ 1} \\ x_1 &= \boxed{0\ 0\ 0\ 0\ 0\ 1}\,\|\,\boxed{0\ 0} \\ \Rightarrow\ x - x_1 &= \boxed{0\ 0\ 0\ 0\ 0\ 0}\,\|\,\boxed{1\ 1} \\ &= x_{LSB} \end{aligned}$$

$$\begin{aligned} x &= \boxed{0\ 0\ 0\ 0\ 0\ 1}\,\|\,\boxed{1\ 1} \\ x_2 &= \boxed{0\ 0\ 0\ 0\ 1\ 0}\,\|\,\boxed{0\ 0} \\ \Rightarrow\ x - x_2 &= \boxed{1\ 1\ 1\ 1\ 1\ 1}\,\|\,\boxed{1\ 1} \\ &= concat(-1,\ x_{LSB}) \\ &= concat(111111,\ x_{LSB}) \end{aligned}$$

Therefore, the final form of the formula is:

$$y = y_1 + \frac{x_{LSB}}{2^{N-P}} \times (y_2 - y_1) + \frac{x_{LSB} \cdot concat(111111,\ x_{LSB})}{2^{2(N-P)+1}} \times \{y_3 - 2y_2 + y_1\} \tag{10}$$

$$= y_1 + \frac{x_{LSB}}{2^2} \times (y_2 - y_1) + \frac{x_{LSB} \cdot concat(111111,\ x_{LSB})}{2^5} \times \{y_3 - 2y_2 + y_1\} \tag{11}$$

The first part of the quadratic interpolator is a linear interpolator. Therefore, we can instantiate a linear interpolator inside the quadratic interpolator module in order to avoid redundancy.

```
module Quadratic_Iterpolator (
    input [7:0] y_1,
    input [7:0] y_2,
    input [7:0] y_3,
    input [1:0] XLSB,
    output [7:0] y_quadratic
    );

    wire [7:0] y_output_linear;
    Linear_Interpolator LinearInterpolator (
        .y_1(y_1),
        .y_2(y_2),
        .XLSB(XLSB),
        .y_linear(y_output_linear)
    );

    wire [7:0] x_bar;
    wire [7:0] y_bar;
    wire [17:0] product_quadratic_output;
    assign x_bar = {6'b111111, XLSB};
    assign y_bar = y_3 - 2 * y_2 + y_1;
    assign product_quadratic_output = (x_bar * XLSB * y_bar) >> 5;
    assign y_quadratic = y_output_linear + product_quadratic_output[17:10];

endmodule
```

Keep in mind that we discarded bits from LSB to control the bit-growth for simplicity. A stochastic framework is required to analyze the average truncation/rounding error effect, which is out of the scale of this project.

6 Accuracy Comparison

256-slots LUT give the (approximately) exact value for each input. The error lies in the C program calculations. Linear interpolation works better with these specifications than quadratic, and is in par with the exact value with only 64 slots. Quadratic interpolation falls short because of over correction and bit-growth control. The following is a testbench for some value:

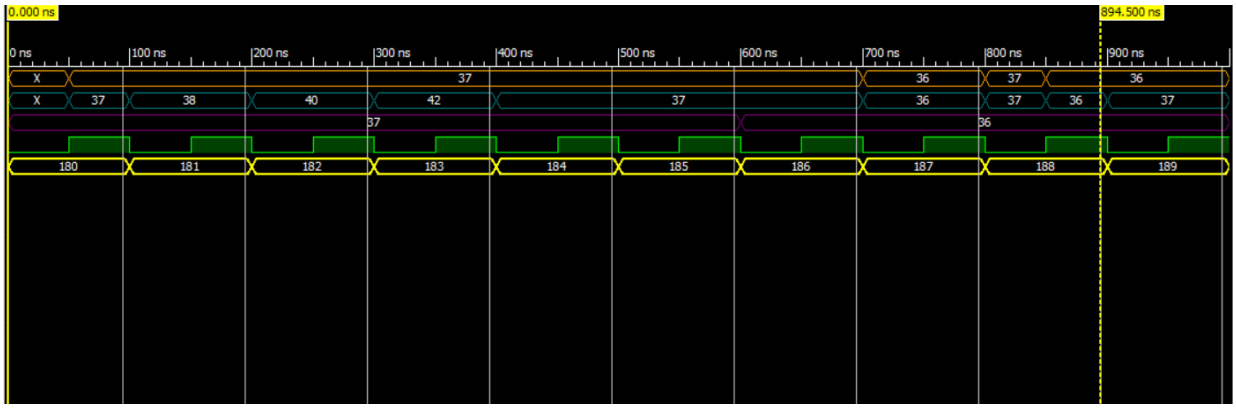

FIGURE 2 – Testbench for values = 180, 181, . . . , 189