The MANDELBROT Machine

Jesús Arias Alvarez

Dpto. de Electicidad y Electrónica. E.T.S.I. Telecomunicación.

Universidad de Valladolid

1 Introduction to the Mandelbrot set computation

The Mandelbrot set has been an obsession for me since the first time, in the eighties, when it was published in the magazine "Scientific American". Of course, I had to compute it and to see it with my very eyes, first with a BASIC code running in an Amstrad CPC that took many hours to fill the screen. Then using faster computers and C compilers the hours were reduced to minutes, and even seconds. But all these programs used floating point arithmetic. It wasn't until well into this millennium when I managed to code a fixed point version. It was intended to run in an ARM7TDMI core without floating point hardware and it performed extremely well, especially when taking into account that the clock frequency was under 100MHz. Since that version I never tough about another floating point implementation.

When I started playing with FPGAs and built a computer with a 6502 core the Mandelbrot set was also a kind of benchmark for its EH-BASIC interpreter, and it was also very slow because of its floating point arithmetic (BASIC only allows floating point numbers). I also built my own CPU core, the GUS16, and I managed to code a fixed point version in assembler language. That version was a lot faster, but still it was much slower than the ARM code, mainly due to the lack of multiplication instructions in the GUS16 core.

At that point I tough about all the time wasted in loops, subroutine calls, and so on, and I wondered if something could be done in order to reduce it. A single iteration of the Mandelbrot algorithm was taking many clock cycles and I wanted to reduce it to a single one. Of course, the GUS16 core was of no use now. What I'm pretending to do is to build a digital system intended only for the calculation of the Mandelbrot set, something not a single chip manufacturer is willing to do, but very appropriate for FPGA synthesis.

The first thing we have to discuss is the calculation itself. The Mandelbrot set is defined by means of the following number sequence:

$$Z_i = Z_{i-1}^2 + C$$

Where Z and C are complex numbers and $Z_0 = 0$. If the magnitude of Z_i remains always bounded the point C belongs to the set. If that magnitude diverges to infinity the point C is outside the set.

Also, Mandelbrot (the mathematician, not the set) proved that if $|Z_i| > 2$ the sequence is going to diverge. So, we can stop our iteration if that magnitude is reached. And if not reached we are going to stop the iteration after a maximum number of cycles anyway. From our practical point of view if the magnitude of Z remains below 2 after, lets say, 256 iterations, the number C belongs to the set. If our number is outside the set we can use the number of iterations reached to assign a particular color to the corresponding pixel in the screen and in that way a colorful Mandelbrot set can be drawn (The set itself is a plain white or black color, all the beauty of the Mandelbrot set lies near, but outside, the set)

So, taking into account these numbers are complex, and that they can be defined as:

$$Z = (a + jb) \text{ and } C = (c + jd)$$

were j is the imaginary unit (We, homo electronicus, like to name the imaginary unit ($\sqrt{-1}$) as j because the variable i is very often used to denote an electrical current). Using this definition we get the following recurrence equations:

$$a_{next} = a^2 - b^2 + c$$

$$b_{next} = 2ab + d$$

Also, the iteration ending condition can be rewritten as:

$$a^2 + b^2 > 4$$

In summary, we will need 3 multipliers, four adders, and little else, in order to build the datapath of the Mandelbrot machine. The multipliers are for:

$$a^2 = a * a$$

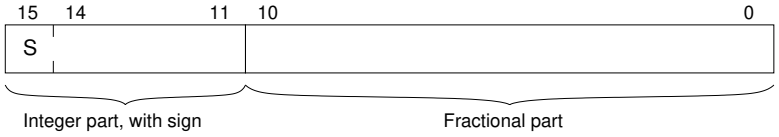
$$b^2 = b * b$$

$$ab = a * b$$

The $2\times$ factor in the b_{next} equation can be implemented by shifting the bits one position to the left, so no logic is required at all. Also, for the >4 condition we only have to look to the most significant bits of the $a^2 + b^2$ result. In this case, apart from the adder, we only need a simple OR gate.

1.1 Fixed point numbers

Also called fractional numbers because they include an implicit fraction were the denominator is a fixed power of two. Their format is (from the GUS16 code):



In this figure 16 bits are used for variables, with 11 bits reserved for the fractional part and 5 bits for the integer part. From these 5 bits the most significant is used for the sign, and therefore, only 4 bits remains for the absolute value of the integer part. These numbers have a value that is:

$$val = \frac{number\_as\_integer}{2^{11}}$$

And in this example the range of these number goes form -16.0 to +15.99951. I choose to have 5 bits for the integer part because we can get arithmetic overflows with less bits. (We can still get overflows if we check pixels far away from the set, but these areas are of little interest). Also, notice that the integer part must have

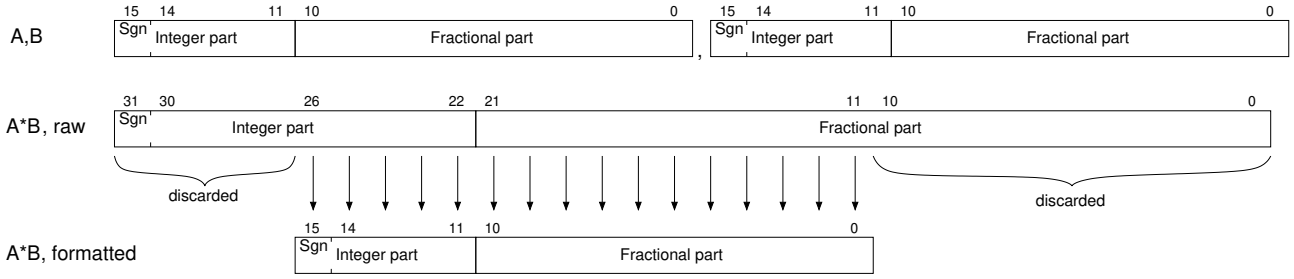


Figure 1: Multiplication of two 16 bit values with 11 bits fractional parts

always the same number of bits in order to keep the range, but the fractional part can be increased as desired in order to reduce truncation errors. (The ARM code used 32 bits variables with 27 fractional bits)

Using this format numbers are added or subtracted as simple binary values, but multiplications are different, as we can see:

$$a+b=\frac{A}{2^{11}}+\frac{B}{2^{11}}=\frac{(A+B)}{2^{11}}$$

$$a*b=\frac{A}{2^{11}}*\frac{B}{2^{11}}=\frac{A*B}{2^{22}}=\frac{\frac{A*B}{2^{11}}}{2^{11}}$$

In other words, as it is shown graphically in figure 1, a 16 bit by 16 bit multiplication gives a 32 bit result, but the 11 less significant bits have to be discarded in order to get the same fractional part, even if this will result in a truncation error. The 5 most significant bits also have to be discarded to fit the integer part into a 5 bit result. The trick is to align the boundaries between the integer and the fractional part (the fixed points) in both the raw 32 bit result and the formatted final 16 bit result.

The idea of computing bits to be discarded later isn't very attractive. In an existing CPU, like the ARM, it is done this way because the hardware resources are already at hand, but for a FPGA implementation this looks like a waste of logic. This is the main reason for the design of an specific fractional multiplier for the Mandelbrot Machine.

2 The Machine

In figure 2 the design of the Mandelbrot machine is presented as composed of several related blocks. Of all the blocks the innermost one, "Pixel computation", is the most critical of the design and it is also presented in more detail in figure 2. This logic is responsible for the iteration algorithm for each pixel. Outside this logic, the "Image generation" block is responsible for changing the values of the c or d variables and writing the resulting pixel color to memory for each 'init' pulse. The initial values of these variables, and their change step is controlled by the "Position & zoom" block, where the buttons of a game controller are used to modify their values.

Alongside the Mandelbrot Machine are two other blocks: one of them is responsible for the generation of a VGA compatible video signal and the other provides the required serial to parallel conversion for the NES-like game controller. These blocks can be considered as a human interface and not part of the Mandelbrot Machine properly.

The most expensive components in terms of complexity and delay time are without any doubt the multipliers of the "Pixel computation" block. These are combinational circuits that can be built by stacking lots of adders. Their design is described next.

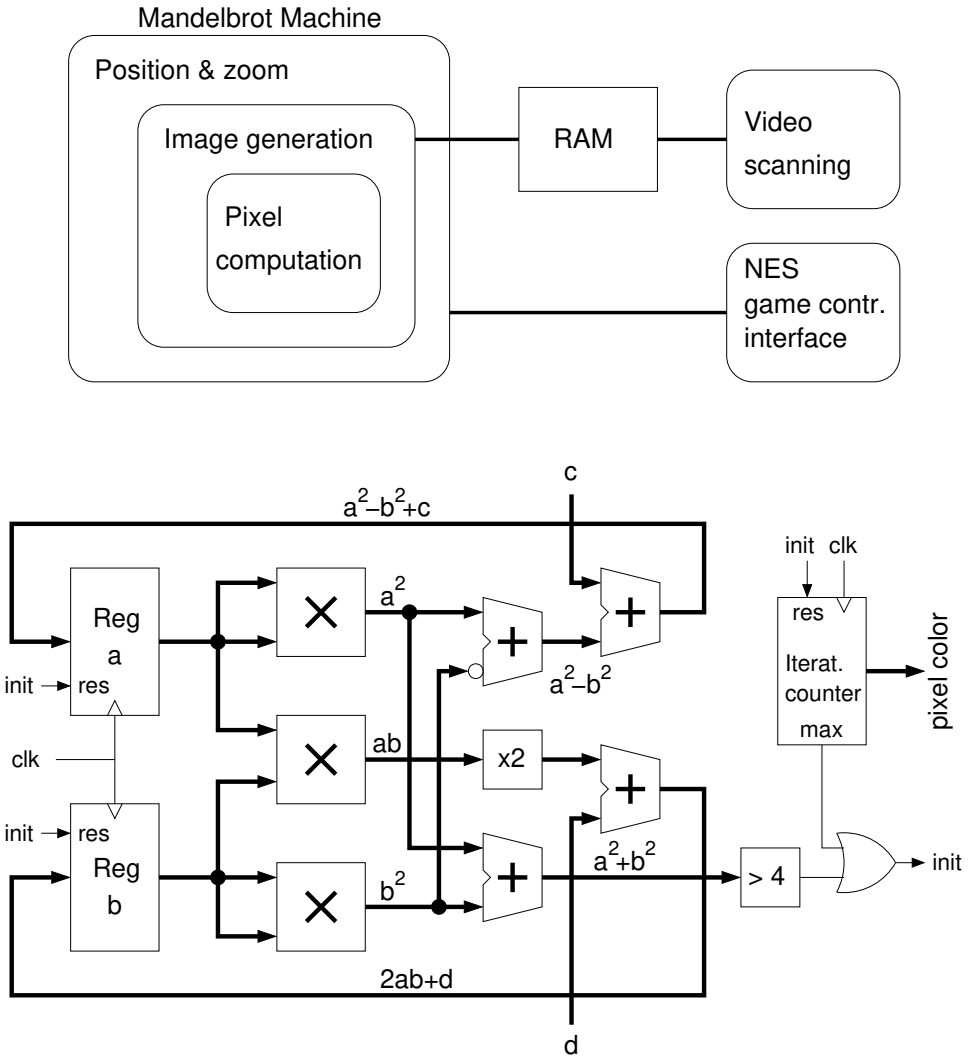


Figure 2: Design hierarchy of the Mandelbrot Machine and diagram of its inner datapath (Pixel computation)

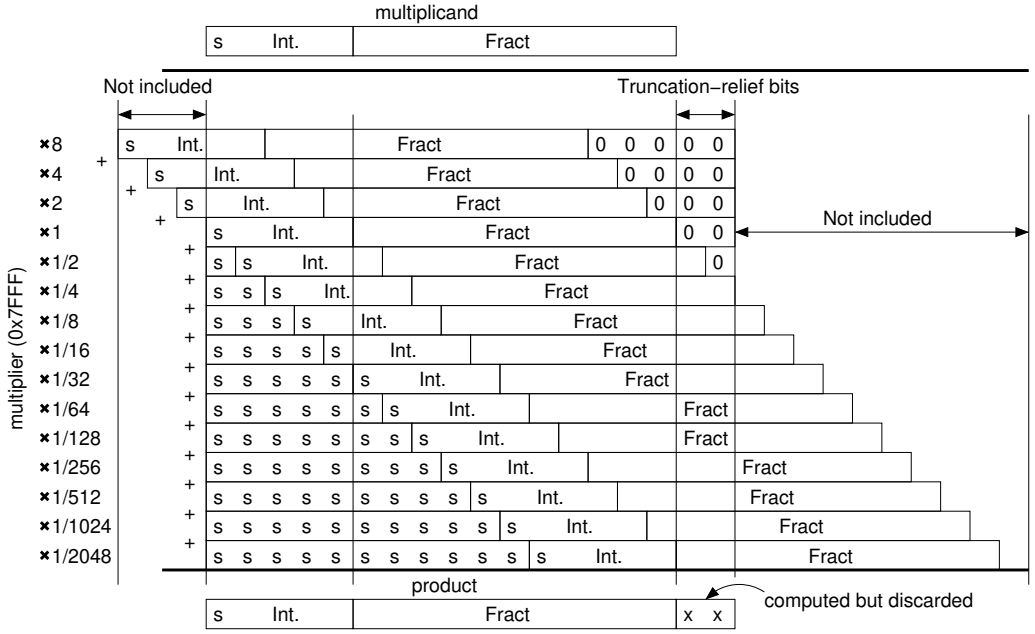


Figure 3: Detail of the fractional multiplication as 14 additions (maximum unsigned multiplier)

2.1 Fractional multipliers

Each multiplier has two 16 bit inputs and one 16 bit output. All signals are fractional integers with sign and an 11 bit fractional part. The first operation the multiplier has to do is to deal with the sign of its two inputs. In order to simplify things we want to have a positive multiplier, so, if the multiplier is negative the two's-complement of the multiplicand and the multiplier are used instead of the inputs. The sign adjust will result in two arrays of 16 XOR gates and half adders placed before the actual multiplier. Also, the multiplier is now always positive and this means its most significant bit is always zero, or in other words: the multiplier is now 15 bit wide.

The multiplication itself can be implemented as 14 additions with shifted multiplicand values. A diagram is shown in figure 3 for the case where all multiplier bits are one. For other multiplier values a bit in zero means no addition (or the addition of a zero value) for the corresponding shifted multiplicand. This will be translated as arrays of AND gates in series with the multiplicand bits controlled by the bits of the multiplier.

The multiplicand is a signed value and, therefore, its most significant bit has to be copied into the lower bits when shifting in order to get the same sign in the shifted value.

The adders used in the multiplier are 18 bit wide. The two extra bits are intended to reduce the truncation error with respect to 16 bits adders. A simulation was carried out in order to test the amount of the truncation error for different adder widths and some results are presented in figure 4. Here we can see that two extra bits in the adders will lower the error noticeably. Still, the error introduced is a bit higher than that of a multiplier with 32 bits adders where the error is bounded to the -0.5 to +0.5 LSB range. The GUS16 code used only 16 bits for the additions and results were good, so, I think that using 18 bits we can improve the accuracy of computations a bit without adding much more hardware. Also, these two bit can hold an initial value that results in an offset in the truncation error (a binary value of 11, in the case of figure 4). This helps in achieving an average truncation error near zero, or said in another words: a rounding instead of a truncation.

2.1.1 Multiplier code in Verilog

The Verilog code of the fractional multiplier is listed next:

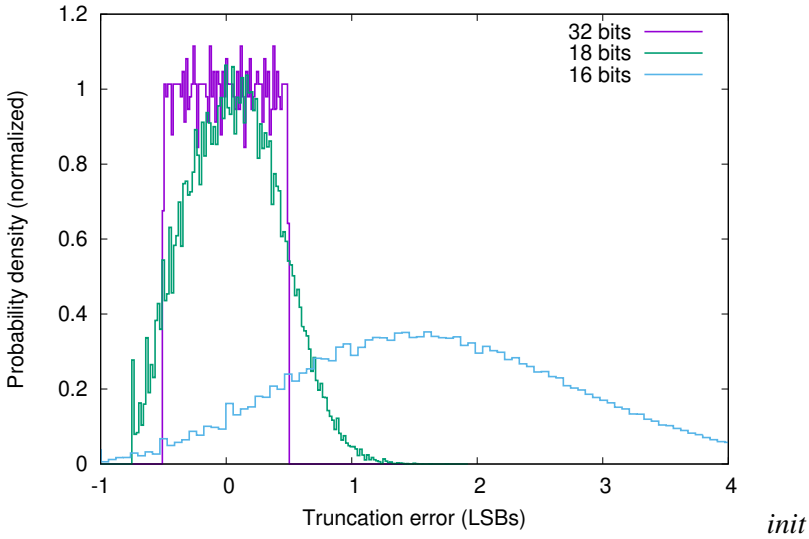


Figure 4: Truncation error probability for 16, 18, and 32 bit adders. The 18 bit adder gives a good compromise between precission and complexity.

```
//-----------------------------------------------------------------------------
//-- Fractional Multiplier for Mandelbrot Machine
//-----------------------------------------------------------------------------
module fmul(
    input   [15:0]md,   // Multiplicand
    input   [15:0]mr,   // Multiplier
    output  [15:0]out   // Product
);

wire [15:0]d;       // Multiplicand (sign adjusted)
wire [14:0]r;       // Multiplier (positive)
wire [17:0]sum;     // result with 2 extra bits

// Make multiplier positive
assign r=mr[15]? ((~mr[14:0])+1) : mr[14:0];
assign d=mr[15]? ((~md)+1) : md;

wire s; // sign bit of multiplicand
assign s=d[15];

assign sum= 18'h3 +     // (= 0.75 LSB) for rounding instead of truncating
 (r[14]? {d[12:0],5'h0}: 18'h0) +
 (r[13]? {d[13:0],4'h0}: 18'h0) +
 (r[12]? {d[14:0],3'h0}: 18'h0) +
 (r[11]? {d[15:0],2'h0}: 18'h0) +
 (r[10]? {s,d[15:0],1'h0}: 18'h0) +
 (r[9]?  {s,s,d[15:0]}: 18'h0) +
 (r[8]?  {s,s,s,d[15:1]}: 18'h0) +
 (r[7]?  {s,s,s,s,d[15:2]}: 18'h0) +
 (r[6]?  {s,s,s,s,s,d[15:3]}: 18'h0) +
 (r[5]?  {s,s,s,s,s,s,d[15:4]}: 18'h0) +
 (r[4]?  {s,s,s,s,s,s,s,d[15:5]}: 18'h0) +
 (r[3]?  {s,s,s,s,s,s,s,s,d[15:6]}: 18'h0) +
 (r[2]?  {s,s,s,s,s,s,s,s,s,d[15:7]}: 18'h0) +
 (r[1]?  {s,s,s,s,s,s,s,s,s,s,d[15:8]}: 18'h0) +
 (r[0]?  {s,s,s,s,s,s,s,s,s,s,s,d[15:9]}: 18'h0) ;

assign out=sum[17:2];

endmodule
```

In this code the sign of the multiplier is adjusted first, and then follows the Verilog implementation of the diagram of figure 3, where the multiplicand is shifted left or right (with sign extension), or made zero if its

corresponding bit in the multiplier is zero, and then added to a 18 bit result, "sum". The synthesis tool is left with the task of reducing this expression to a minimum number of logic cells in the FPGA. A synthesis was made for a Lattice ICE40HX FPGA using the Icestorm tools, that gave the following estimates for the multiplier:

Logic Cells	637
Delay	16.8ns

As expected, the multiplier takes a lot of logic cells, and its delay only allows clock frequencies below 50MHz when both its inputs and output are registered (a flip-flop includes a 1.46ns delay). Still, these numbers promises a much better performance for the Mandelbrot set calculation than any software alternatives.

2.2 The Pixel computation datapath

The Verilog code of the inner datapath (figure 2) is listed next:

```
////////////////////////////
// Pixel computing datapath
////////////////////////////
reg [15:0]a=0;      // real part of Z
reg [15:0]b=0;      // imaginary part of Z

wire [15:0]a2;
wire [15:0]b2;
wire [15:0]ab;
wire [16:0]abs2;

// Multipliers
fmul fmul0 (.md(a),.mr(a),.out(a2));
fmul fmul1 (.md(b),.mr(a),.out(ab));
fmul fmul2 (.md(b),.mr(b),.out(b2));

assign abs2=a2+b2;  // Square of the absolute value of Z
wire plus4;         // higher than 4
assign plus4=abs2[16]|abs2[15]|abs2[14]|abs2[13];

reg [8:0]icount=0;  // Iteration counter
wire init;          // Init signal
assign init=plus4 | (icount[8]);

// Sequential logic
always @(posedge clk) begin
    a<=init ? 0 : a2-b2+c;
    b<=init ? 0 : {ab[14:0],1'b0}+d;
    icount<= init ? 0 : icount+1;
end
```

This simple code hides the huge cost of the multipliers. A synthesis of it resulted in 1761 logic cells an a maximum clock frequency of 39.5 MHz

2.3 The image generation logic

The innermost datapath requires two fractional values, c, and d, and generates an *init* pulse when the pixel computation is over before beginning another iteration. The image generation logic scans the c, and d values according to the video resolution and writes the resulting iteration count to the video memory on each *init* pulse.

The video mode uses the standard VGA, 640×480, mode for timings, but the resolution is reduced to 512×480. The reasons for this are two: First, the amount of memory available for video (128 Kbytes), is less than the 150 Kbytes required for full resolution at 16 colors per pixel, and second, the fact that using a power of two for the horizontal resolution simplifies the design of the logic.

The Verilog code of the image generation block is listed next. Here we can remark the use of the variables, x, and y, that are the fractional values of the center of the image, and *delta*, which correspond to a single pixel step, both for the c, and d, variables. We can also remark the mapping between the 256 possible values of the iteration counter and the 16 colors of the pixel that follows a logarithmic scale. Finally, there is also a detail

that deserves another comment: Video memory is 16 bit wide and a word stores 4 pixels. The video generator therefore reads a word every 4 clock cycles during the visible part of the image and the image generator also writes to the memory after the computation of 4 pixels. But, if the write happens at the same time than a memory read it has to be delayed one cycle because the external memory is a single port one. Temporary registers (*pix4* and *waddr*) are thus used to hold the data and address values until they could be written to RAM.

```
    ////////////////////////////////////////////////
    //              Image scanning
    ////////////////////////////////////////////////
    reg [15:0]c=16'd57344;      // Real part of C
    reg [15:0]d=16'd3200;       // Imaginary part of C

    reg [15:0]x=(65536-96*12);              // Center Position (real)
    reg [15:0]y=0;              // Center Position (imaginary)
    reg [4:0]delta=5'd12;       // pixel step

    reg [11:0]pix3;             // three last pixels
    reg [15:0]pix4;             // buffer for 4 pixels
    reg [17:0]pixcnt=0;         // Pixel counter
    reg [15:0]waddr;            // write address
    reg wpend=0;                // write pending

    wire [3:0]pix;              // Pixel color
    // Logarithmic scale for colors
    assign pix= (icount[7:0]>133)?15 : (
                (icount[7:0]>96)?14 : (
                (icount[7:0]>69)?13 : (
                (icount[7:0]>50)?12 : (
                (icount[7:0]>36)?11 : (
                (icount[7:0]>26)?10 : (
                (icount[7:0]>19)?9 : (
                (icount[7:0]>13)?8 : (
                (icount[7:0]>10)?7 : (
                (icount[7:0]>=7)?6 : (
                (icount[7:0]>=5)?5 : icount[3:0] ))))))))));

    // sequential logic
    always @(posedge clk) begin
        if (init) begin
            pixcnt<=(pixcnt==(512*480-1))?0:pixcnt+1;   // pixel count
            pix3<={pix3[7:0],pix};              // pixel shifting
            // update: c=x-256*delta or c=c+delta
            c<=(pixcnt[8:0]==511)? (x-{delta,8'h0}):(c+delta);
            if (pixcnt[8:0]==511) begin
                // update: d=y+240*delta or d=d-delta
                d<=(pixcnt[17:9]==479)?(y+{delta,8'h0}-{delta,4'h0}):(d-delta);
            end
            if (pixcnt[0]&pixcnt[1]) begin  // every 4 pixels...
                pix4<={pix3,pix};
                waddr<=pixcnt[17:2];
                wpend<=1;
            end
        end
        if (wpend) begin
            if (~vrd) wpend<=0; // write pending can be delayed if video reads memory
        end
    end


    ////////////////////////////////
    // VIDEO generator
    ////////////////////////////////
    wire vrd;       // video reads memory if 1
    wire [15:0]va;  // video address for read

    videomodule video0(.clk(clk), .d(xdi), .a(va), .vrd(vrd),
        .hsyn(hsyn), .vsyn(vsyn), .video(video)
    );

    ////////////////////////////////
    // external RAM
    ////////////////////////////////
    assign xdo=pix4;            // write data
    assign xa=vrd?va:waddr;     // memory address
    assign xoe=~vrd;            // output enable (active low)
    assign xwe=~((wpend&(~vrd))&(~clk));    // Write enable (active low)
```

2.4 The game controller interface and position logic

Finally, the position and zoom of the displayed set is controlled via a NES-like game controller. This controller has 8 buttons whose levels are sampled by a CD4021 parallel to serial converter. This IC has an asynchronous load input, *glat*, and a clock signal, *gclk*. I choose to use the vertical synchronism signal of the VGA as clock, resulting in 134ms between samples and a button repetition rate of 7 per second. The recovered bits are stored in the *gbut* shift register, and if the corresponding bits are active the value of the *x*, *y*, or *delta* variables are changed accordingly.

The Verilog code of this block is listed next:

```
////////////////////////////////////////
// NES game controller and position logic
////////////////////////////////////////

reg [2:0]gcnt=0;
reg [7:0]gbut;

assign gclk=vsyn;
assign glat=gclk&(gcnt==0);

always @(posedge gclk) begin
    gcnt<=gcnt+1;
    if (gcnt==7) begin
        if      (gbut[0]) x<=x-{delta,4'h0};
        else if (gbut[1]) x<=x+{delta,4'h0};
        if      (gbut[2]) y<=y+{delta,4'h0};
        else if (gbut[3]) y<=y-{delta,4'h0};
        if      (gbut[7]) delta<=delta-1;
        else if (gbut[6]) delta<=delta+1;
    end
end
always @(negedge gclk) begin
    gbut<={gbut[6:0],~gin};
end
```

2.5 The Video generator

The image memory has to be displayed on a screen and a video generator is required for this. The code listed next generates a 512×480 image following more or less the timing of the standard 640×480, 60Hz, VGA mode. Starting from a 25MHz clock signal this Verilog code generates the horizontal and vertical synchronism pulses, and also the video address. It reads the video memory and shifts out the pixels. During retrace times no reads are performed and the video output is low. The color depth is 4 bits per pixel, giving a maximum of 16 colors from a CGA-like palette: The four pixel bits are I, R, G, B, with I (Intense) being an extra level of white added to the combinations of the Red, Green, and Blue components. The 4-bit pixel is converted into three 4-bit red, green, and blue signals before driving the color DACs of the board. The R, G, and B bits result in a 67% level of color when active, and the I bit adds another 33% to the level of the three colors.

The video memory is a 1Mbit SRAM, external to the FPGA, and configured as 64Kwords of 16bits. This memory is shared with the Image generation logic block.

```
////////////////////////////////////////////////////////////////
//                      VIDEO generation
////////////////////////////////////////////////////////////////

module videomodule (
    input clk,      // clock input @25MHz
    input [15:0]d,  // data read from external RAM
    output [15:0]a, // external RAM address
    output hsyn,    // Horizontal sync. pulse (active low)
    output vsyn,    // Vertical sync. pulse (active low)
    output [11:0]video, // Video output (12 bits/pixel)
    output vrd,     // video read
    output hblk,    // Horizontal blank
    output vblk,    // Vertical blank
    output [9:0]hc  // Horizontal counter
);

// Horizontal section
// Total: 800 pixels (32 us)
// Displayed: 512 pixels
// hsync width: 96 pixels
reg [9:0]hc=0;  // Horizontal counter
reg hsyn=1;     // sync
reg hblk=0;     // blank output (active if pixel >= 512)
always @(posedge clk) begin
    if (hc==10'd799) begin hc<=0; hblk<=0; end else hc<=hc+1;
    if (hc==10'd591) hsyn<=0;
    if (hc==10'd687) hsyn<=1;
    if (hc==10'd511) hblk<=1;
end

// Vertical section
// Total: 525 lines
// Displayed: 480 lines
// vsync width: 2 lines
reg [9:0]vc=0;  // Vertical Counter
reg vsyn=1;     // sync.
reg vblk=0;     // blank output (active if line >=480)
always @(negedge hblk) begin
    if (vc==10'd524) begin vc<=0; vblk<=0; end else vc<=vc+1;
    if (vc==10'd489) vsyn<=0;
    if (vc==10'd491) vsyn<=1;
    if (vc==10'd479) vblk<=1;
end

// Global Blank
wire blk;
assign blk = hblk|vblk;

// Memory Address
assign a = {vc[8:0],hc[8:2]};
reg dblk=0; // blk delayed
always @(posedge clk) dblk<=blk;

// Video read
assign vrd = (~blk) & (~hc[1]) & (~hc[0]);
reg [15:0]vsh;
always @(posedge clk) vsh <= vrd ? d : {vsh[11:0],4'bxxxx};

wire [3:0]pixel;
assign pixel = dblk ? 4'b0000 : vsh[15:12];
```

```
// Color palette: CGA-like

assign video[11]=pixel[2];  // R[3]
assign video[10]=pixel[3];  // R[2]
assign video[ 9]=pixel[2];  // R[1]
assign video[ 8]=pixel[3];  // R[0]

assign video[ 7]=pixel[1];  // G[3]
assign video[ 6]=pixel[3];  // G[2]
assign video[ 5]=pixel[1];  // G[1]
assign video[ 4]=pixel[3];  // G[0]

assign video[ 3]=pixel[0];  // B[3]
assign video[ 2]=pixel[3];  // B[2]
assign video[ 1]=pixel[0];  // B[1]
assign video[ 0]=pixel[3];  // B[0]

endmodule
```

3 Results, toughs, conclussions.

After some debugging this design performed as expected and the Mandelbrot set was displayed on a monitor (Figure 5). The total FPGA usage was 2355 out of the 7680 available logic cells, and the maximum estimated clock frequency was 35.6 MHz, even if the clock was actually running at 25 MHz due to VGA timing requirements. The whole Mandelbrot Machine requires more or less the same FPGA space than a Z80 core.

The computation was pretty fast. The first image displayed in figure 5 took only a fraction of a second to be drawn. A worst case image with all pixels black (all belonging to the Mandelbrot set) is going to take 2.5 seconds to be drawn. For a 25 MHz clock this is lightning speed.

But now the bad news. With only 11 fractional bits in the variables it is impossible to zoom too deep into the set (After only a few zooming steps our *delta* variable is reduced to 1 and there is no more zooming possible). More than 11 bits could be used for the fractional part of variables, but this will result in huge multipliers because the complexity of such circuits increases quadratically with the number of bits. Maybe a redesign using 24 bit variables would still be possible, but for 32 bit variables this approach is out of question.

The multiplier design is therefore the critical part of the Mandelbrot machine. Here I resorted to the simplest approach: a combinational one. But in the literature there are more hardware efficient multipliers to be explored (search for Booth's, Wallace's, or Dadda's multiplication algorithms)

Another direction for improvement is the video memory. It would be desirable to have at least 8 bit per pixel and full 640×480 resolution. Even an 800×600 resolution with a 36 MHz clock would be possible if more RAM is available (a 4 Mbit RAM instead of 1 Mbit will suffice)

In conclussion, an specific hardware for the Mandelbrot set computation was built in a FPGA, resulting in a design not much complex than a vintage CPU, while its performance surpassed even that of modern processors like ARMs. Yet, the accuracy of the calculation is limited by the use of 16 bit variables, which reduces the zooming to a maximum of ×16, and the design is hard to scale to higher resolutions without resorting to a more hardware efficient multipliers.

4 Update: Take it to the limit...

I did a redesign of the Mandelbrot Machine using 24 bit variables instead of 16. Because we are multiplying the data width by 1.5 I'm expecting the new multipliers would require about $1.5^2 = 2.25$ times the size of a 16 bit one, or about 1440 logic cells each, resulting in a total above 4300 logic cells. On the other hand, the delay is multiplied only by 1.5, and the new maximum clock frequency should be around 24 MHz, close to the desired 25 MHz (it may require a little overclocking). Anyway, these are coarse estimates because the place & route

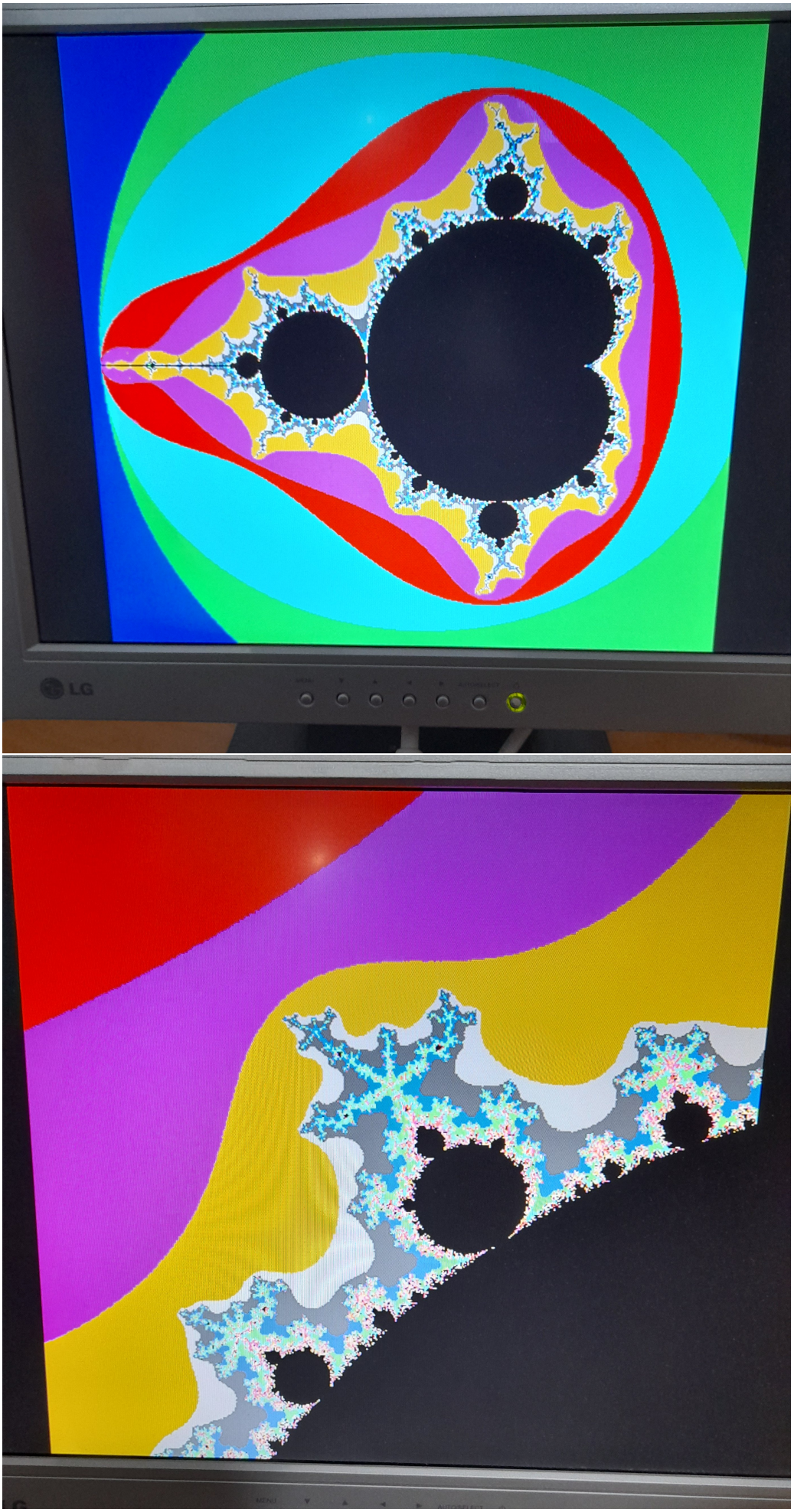


Figure 5: Snapshots of the Mandelbrot set displayed on a VGA monitor

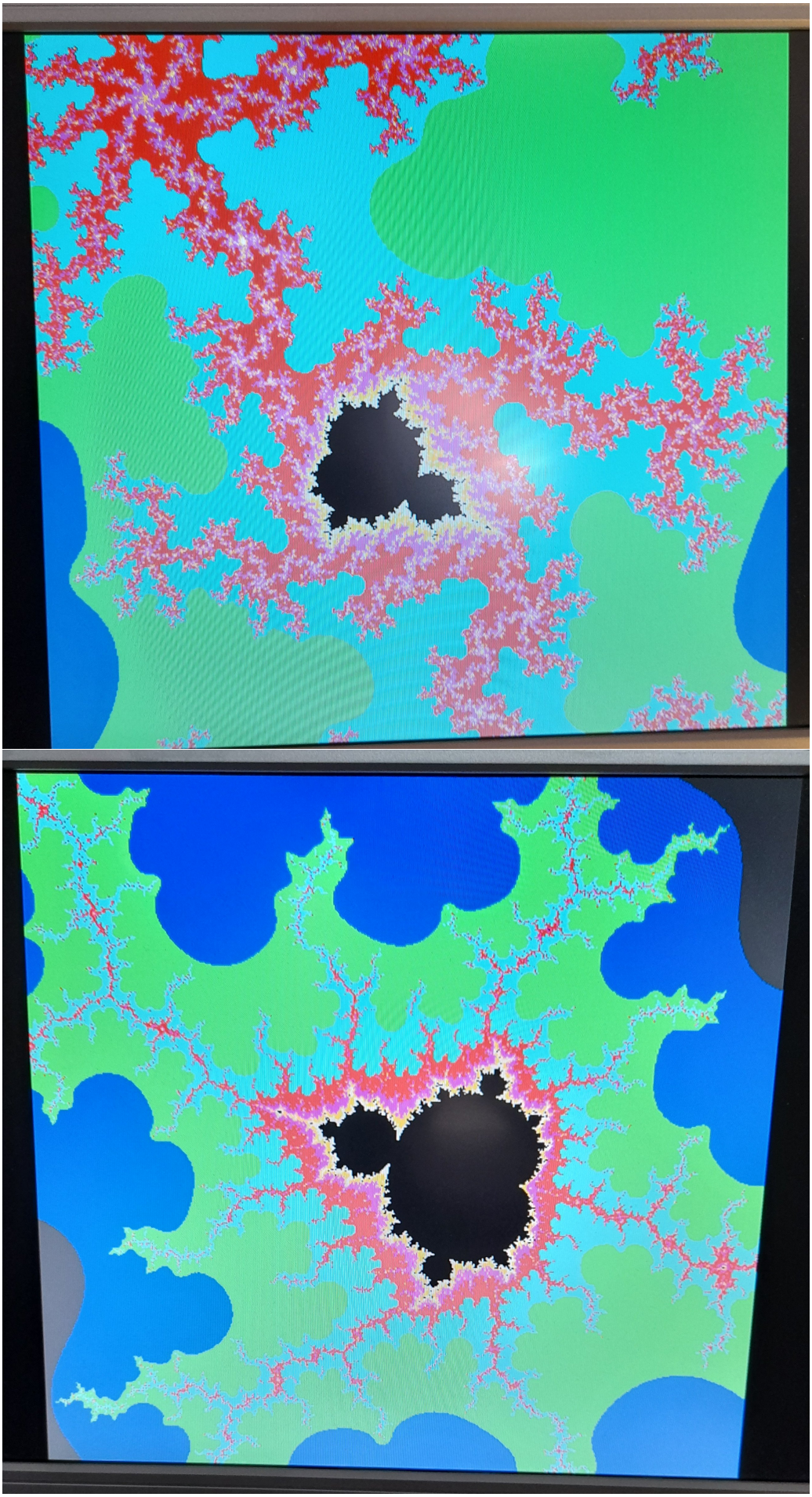


Figure 6: Snapshots of the Mandelbrot set computed with 24 bit variables

tool can map the new logic cells in a different way and results may vary. In fact, the results after a complete redesign were:

Total Logic Cells	4539
Max. Clock Frequency	28.78 MHz

So, the new 24 bit design will still fit into the 7680 logic cell FPGA, and no overclocking will be needed (but only by a narrow margin).

Now, the variables have a 19 bit fractional part and this will allow for a considerable deeper zooming into the Mandelbrot set: The maximum zoom is now $\times 4096$ instead of $\times 16$ (the zoom control is now logarithmic instead of linear to cover easily the increased range). This allows a much finer detail of the set to be displayed. For instance take a look to figure 6 where the zoom settings were near the maximum and Mandelbrot-like subsets are still clearly recognizable.

4.1 One more time?

I'm afraid not. With 24 bits in the multipliers we have a 60% of the FPGA space occupied and the clock frequency is barely enough to run, so, this is really our limit unless a bigger and faster FPGA is used or a better approach is found for the multipliers.

But the fascinating thing about the Mandelbrot set is that you'll never get enough zooming. No matter how deep you descend into the set boundary, there are always some areas that deserve a higher resolution. And there are several problems that limit that resolution:

- The maximum number of iterations, 257 in our case (I added one more bit to the counter to get the lower 8 bits in zero when inside the set). When more iterations are needed the perimeter of the Mandelbrot subsets looks rounded. If this happens the iteration limit have to be increased, even if this is translated into a longer computation time.

- Truncation errors. Currently more or less under control thanks to the two extra bits in the adders inside multipliers. If these errors are dominant the perimeter of the subsets appears dotted, like a cloud of black points.

- The number of fractional bits in the variables. This defines the maximum achievable zoom. it would be desirable to have a big number of fractional bits, lets say 59 out of 64 bits. Yet, this can only be done by software in a computer, not in a small FPGA like the one of the SIMRETRO board.

Meanwhile, look at the many awesome details of the Mandelbrot set with the available resolution and listen to The Eagles ;)

In Valladolid (Spain)
Year 2 AC (After COVID)