ECE4530 Fall 2012:
The Codesign Challenge
Drawing the Mandelbrot Fractal

Assignment posted on 15 November 8AM
Solutions due on 6 December 8AM

The Codesign Challenge is the final assignment in ECE 4530. This project is an exercise in performance optimization: you will start from a given reference application on a Nios-II processor. You have to improve the performance of the reference application as much as possible, using the hardware/software codesign techniques covered in this course. Typically, you would design a hardware coprocessor. In addition, you would also optimize the driver software, and/or modify the system architecture. There is an important constraint: the final result must execute on the DE2-115 board under a predefined performance-evaluation testbench. The testbench is written in C, so you have to keep a Nios-II processor present in the final implementation.

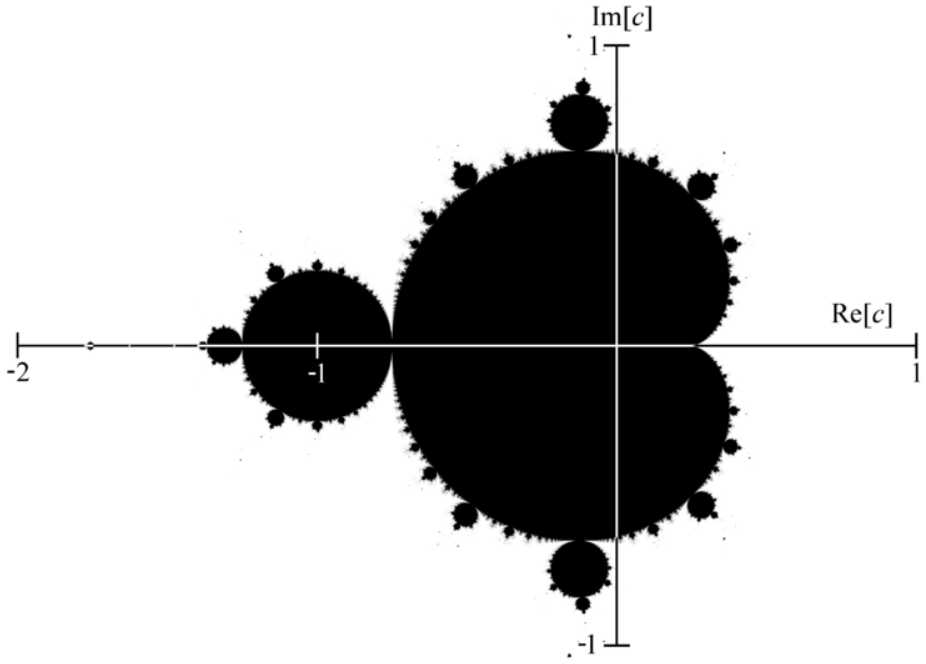

Figure 1: Mandelbrot Set: the black part of the figure are included in the set

Application: Drawing the Mandelbrot Set Fractal

The Mandelbrot Set Fractal is a set of points in the two-dimensional plane, whose inclusion in the set is determined by an iterated complex equation. It is a well known Fractal Set, and its 'gingerbread man' shape is an icon associated with research in Fractals. The Mandelbrot Set is defined as follows. Consider a sequence of complex points defined by

$$z_n = \begin{cases} 0 & if n = 0 \\ {z_{n-1}}^2 + c & otherwise \end{cases} \quad (1)$$

For a given complex number c, a point is in the Mandelbrot set of z_n stays bounded as n goes to infinity. The Mandelbrot gingerbread man is created by choosing different values of c in the complex plane, and coloring a point black if it remains bounded. This results in Figure 1.

A quick test to evaluate if the sequence is bounded or not is to test if the absolute value of z_n becomes higher than 2. If it does, we can abort the iteration and conclude that the sequence computed at this particular c does not converge.

A colored representation of the set is possible by associating a color for the number of iterations after which the absolute value of z_n grows above 2. This leads to the following pseudo algorithm for drawing a colored Mandelbrot Set fractal.

```
for each pixel on the screen do {

  // c and z are complex values mapped into variables as follows
  //    c = x0 + j.y0
  //    z = x + j.y
  //
  // each iteration computes:
  //    z = z^2 + c
  //      = (x^2 - y^2 + x0) + j.(2.x.y + y0)

  x0 = scaled x coordinate of pixel
  y0 = scaled y coordinate of pixel
  x = 0
  y = 0
  iteration = 0
  while ( x*x + y*y < 2*2  AND  iteration < max_iteration ) {
    xtemp = x*x - y*y + x0
    y = 2*x*y + y0
    x = xtemp
    iteration = iteration + 1
  }
  color = iteration % colormap;
  plot(x0,y0,color)
```

```
}
```

`max_iteration` bounds the number of iterations; `colormap` reflects the number of available colors. There are several important observations one can make, from the viewpoint of performance optimization:

- The algorithm is compute intensive. Each pixel iteration involves three data multiplications. Up to `max_iteration` iterations may be needed per pixel. If `max_iteration = 30`, we are looking at up to 23 million data multiplies for a full VGA frame (640 by 480 pixels).
- The algorithm is *embarassingly* parallel. There are no dependencies among pixels, and each of them can be computed in parallel, if sufficient hardware would be available.

Fixed Point Arithmetic

The reference implementation is written with fixed point arithmetic; every computation is done with 28 fractional bits and 4 integer bits. We call such a data type a `fixed428`.

A real number v is mapped as follows into `fixed428`:

```
typedef int fixed428;
float v;
fixed428 a;

a = (int) (v * (1 << 28));
```

The multiplication of a `fixed428` with another `fixed428` yields a number of type `fixed856`. It can be converted to a `fixed428` by a downshift over 28 positions.

```
typedef int fixed428;
typedef long long fixed856;
fixed428 a, b;
fixed856 c;
fixed428 casted;

c      = a * b;
casted = (fixed428) (c >> 28);
```

Note that the casted result may loose some bits at the msb side. Indeed, a downshift of a `fixed856` over 28 positions results in a `fixed828` (a 36-bit datatype), but we only have room for 32-bits. However, overflow will not occur during the computation as the values are continuously tested for convergence to the Mandelbrot Set criterion, and because we will make the drawing only for a bounded region of the complex plane.

Hence, the pseudocode algorithm discussed earlier can be mapped as follows in C.

```
int calc_frac_point_soft(fixed428 cx, fixed428 cy, alt_u16 max_itr) {
  fixed428 x, y, xx, yy, xy2;
  fixed856 xx_raw, yy_raw, xy_raw;
  int itr;

  x = cx;
  y = cy;
  itr = 0;
  do {
    xx_raw = (fixed856)(x) * (fixed856)(x);
    xx = (fixed428)(xx_raw >> 28);
    yy_raw = (fixed856)(y) * (fixed856)(y);
    yy = (fixed428)(yy_raw >> 28);
    xy_raw = (fixed856)(x) * (fixed856)(y);
    xy2 = (fixed428)(xy_raw >> 27);
    x = xx - yy + cx;
    y = xy2 + cy;
    itr++;;
  } while (((xx+yy)<0x40000000) && (itr<max_itr));
  return(itr);
}
```

Video Interface

The implementation makes use of a VGA video driver, implemented in hardware. The video driver is constructed from modules in Altera's University Program library. VGA resolution is 640 pixels horizontal by 480 pixels vertical, and uses 24 bits of color information per pixel.

Each pixel has three color components (R, G, B), each 8 bits in resolution. The pixel value `0xFFFFFF` would be an all-white pixel, the pixel value `0xFF0000` would be an all-red pixel, the pixel value `0xFF00` would be an all-green pixel, and the pixel value `0xFF` would be an all-blue pixel. Other combinations can be achieved by mixing colors, and the reference implementation includes a colormap with 32 predefined colors.

The display associates pixel coordinate $(0, 0)$ with the top left of the screen, and $(639, 479)$ with the bottom right of the screen. Of course, to display the Mandelbrot set we will choose the direction of X and Y to point right and up, respectively.

The Video Interface comes with several driver functions that can be used to plot pixels on the screen.

```
#include "altera_up_avalon_video_pixel_buffer_dma.h"
alt_up_pixel_buffer_dma_dev *video_device;

// initialize the video device
```

```
video_device = alt_up_pixel_buffer_dma_open_dev(VIDEO_PIXEL_BUFFER_DMA_0_NAME);

// draw a box from top left (x0, y0) to bottom right (x1, y1)
alt_up_pixel_buffer_dma_draw_box(video_device, x0, y0, x1, y1, color, 0);
```

Additional functions can be consulted in the documentation for this peripheral (check Video.pdf on Scholar).

The hardware implementation can be used as is and deserves no particular optimization; the main optimization will be to improve the computation speed of the Mandelbrot Set convergence loop. Should you make modifications, then keep the following in mind.

The VGA display controller needs a RAM memory so store pixel values. The off-chip static RAM is used for this. The address of this RAM is hard-coded into the vga controller peripheral, and is currently mapped to `0x20000000`.

The VGA display controller generates pixels at 25MHz, half of the clock frequency of the processor. The hardware implementation uses a clock boundary fifo for this.

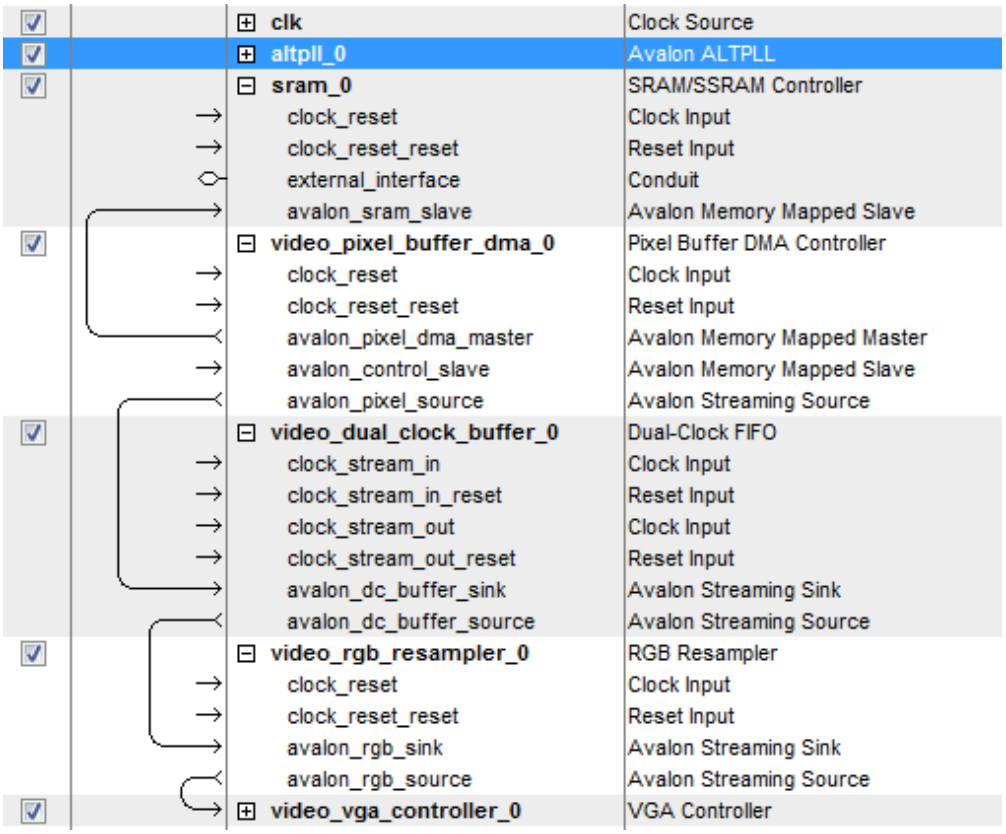

Figure 2: VGA Video Generation

Figure 2 shows the Video Controller configuration. Pixels are stored in the SRAM; they are written into the SRAM by the Nios processor (not shown). The `video_pixel_buffer_dma` module scans out the SRAM and forwards the pixels to a dual-clock buffer. This buffer is

a FIFO that is written in a 50MHz clock domain and read out in a 25MHz clock domain (the VGA pixel rate). The output of the dual-clock buffer is fed into an RGB resampler module. The VGA controller is designed for 30-bit pixels. Hence, the 24-bit pixels coming from the dual-clock buffer are converted to 30-bit resolution. Note that, even though the VGA controller is designed for 30-bit RGB, the DE2-115 board physically only supports 24-bit RGB (24-bit video DAC). Hence, the top-level module of this system includes only a 24-bit RGB output.

Testbench

The testbench draws the Mandelbrot fractal starting from coordinates (-2, -1), and spanning a horizontal width of 3. Thus the lower left of the display maps the coordinate (-2, -1) and the lower right of the display maps the coordinate (1, -1).

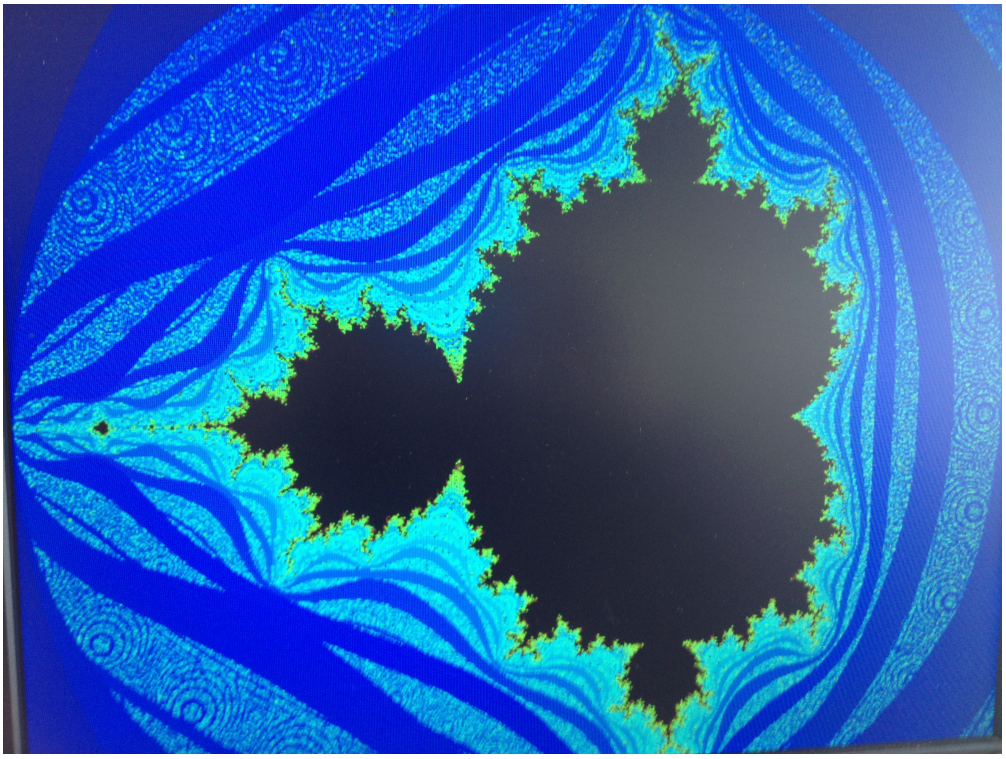

Figure 3: Testbench output

The testbench measures the time (in 50MHz clock cycles) to draw a complete frame. Time is measured with a 64-bit counter integrated in the system. The reference implementation computes the algorithm on a Nios/e processor. This takes 135 billion cycles to draw (45 minutes at a 50MHz clock!).

Clearly there is room for optimization. Simply substituting the Nios/e with a Nios/f processor, for example, reduces the computation time to 1.775 billion cycles per frame.

Your job is to optimize this number as much as possible, using hardware/software codesign techniques.

Among the things you could try to exploit are, for example:

- Use software compiler optimization (O3);
- Increase cache sizes;
- Add a hardware coprocessor (custom-instruction or memory-mapped);
- Add computational parallelism: multiple Nios processors, multiple coprocessors;
- Use pipelining in the computation of pixels;
- Anything else you can think of within the contraints of a DE2-115 board.

You can change the testbench to fit your system design; the only criteria is that your testbench must end by drawing a complete frame as shown in the Figure, and it must display the cycle count to generate that drawing on a Nios2 terminal. However, there are some minimal quality criteria that should be followed.

- The lower bound at which an iteration can be aborted for a convergent pixel is 30. Your graph must be able to display 30 colors.
- The number of pixels to compute is 640 by 480; you cannot extrapolate pixels or copy them.
- You cannot pre-compute pixels. You have to write your algorithm such that it could start from an (x0, y0), and at any suitable level of resolution that can be captured in a `fixed428`.

These items will be checked by hand in the source code of your program.

Ranking Criteria

All designs will be strictly ranked from best to worst. This section defines what 'best' means. We will use the following metrics to evaluate the rank of your design.

- Functional Correctness: This requirement is mandatory for all designs. Your design has to work, in order to be considered for ranking. If it does not work, you will be automatically ranked last.
- Metric 1: The cycle count of generating the fill drawing, as executed on the DE2-115 board, measured with a clock frequency of 50MHz. A lower cycle count is better.

- Metric 2: The area of the resulting design, expressed in terms of 'Total Logic Elements' of a Cyclone IV FPGA. A lower LE cell count is better.
- Metric 3: The turn-in time of your design and report, as measured by the turn-in time on Scholar. Turning in the solution earlier is better. Note that, if you turn in the design multiple times, only the latest turn-in time will be used.

Two designs will be compared as follows, to determine their ranking order. If a design is functionally not correct (i.e. fails the testbench), it will automatically moved to the last rank. If multiple designs are not operational, they will all share the same lowest rank. All functionally correct designs will get a better and unique rank.

- First, their cycle count (Metric 1) will be compared. If there is a difference of more than 5% between them (with the fastest design considered 100%), the fastest design will get a better rank. If, one the other hand, the difference is smaller than 5%, Metric 2 will be used as tie-breaker.
- Metric 2 will be used in a similar way to compare two designs. If the difference in area between two designs is larger than 5%, then the smallest design gets the better rank. Otherwise, if they are separated less than 5%, Metric 3 will be used as tie-breaker.
- Metric 3 will be used as a final metric in case the ranking decision cannot be made using Metric 1 and Metric 2 alone. In this case, the design turned-in earlier wins.

The grade for your project is determined as follows.

- 70 points of the grade are determined by the ranking of your project as described above. The best design gets 70 (out of 70) points, the worst design gets 30 (out of 70) points, and all other designs are linearly distributed between 30 and 70 points.
- 30 points of the grade are determined by the quality of the written documentation you provide with the solution. The requirements are listed below.

Getting Started

On Scholar, you will find two designs. The first design, `ginger`, is a reference design against which other implementations will be measured. This should be your baseline. The second design, `ginger_niosf` is the baseline design in which the Nios/e processor was replaced with a Nios/f processor. You can try both and experience the difference.

A quick start is as follows.

1. First, make sure that you have the latest collection of University Program modules for this project (v12.0). Download the `University_Program.zip` from Scholar and install the library in the `ip` subdirectory from Quartus. Make sure to replace (overwrite) the old versions of modules which are there.

2. Try to get hold of a VGA monitor. Your computer monitor should work fine for this; the CEL may also have a couple of VGA monitors. If you have major trouble getting access to a VGA monitor, please let me know (I don't have a collection of loaner monitors available, but I'll try to help out where I can).

3. Start quartus and expand the project ginger.qar. The archive contains precompiled files, you don't need to compile anything.

4. Open two Nios2 command shells. In the first one, download the bitstream to the board with

   ```
   nios2-configure-sof de2_basic_computer.sof
   ```

5. In the second Nios2 command shell, start a Nios2 terminal.

6. In the first Nios2 terminal, download the software (app.elf in the software subdirectory) with

   ```
   nios2-download app.elf --go
   ```

7. You will see that the drawing process on the VGA monitor starts. In the Nios2 terminal, you will see an output similar to the following.

   ```
   nios2-terminal: connected to hardware target using JTAG UART on cable
   nios2-terminal: "USB-Blaster [7-1]", device 1, instance 0
   nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

   Opened pixel buffer device
   X Y resolution 640 480
   Buffer start 20000000
   Back Buffer start 21000000
   Color Mode 3
   Addressing Mode 0
   Line 0
   Line 1
   Line 2
   ```

8. If you have all this working, you can now start optimizing the design. Sit back, and THINK FIRST. The due date is 1.5 weeks after the end of the break, so I am not expecting you to work on coding during your break. Instead, think about the design, and about possible optimization strategies. Take small steps at a time: it's very hard to get things completely optimized in one take. Most importantly, work in incremental steps, and test each modification you make.

If you replace the Nios/e processor with a Nios/f processor, the tools will generate a time-limited bitstream. Such a bitstream is called `name_time_limited.sof`, where the orginal bitstream would have been called `name.sof`. You can download this bitstream just as a normal one, but you need to ensure that you do not close the bitstream download program.

What to turn in

You have to deliver the following items for the project result:

- A qar file of your final design. It must contain a Nios processor that is capable to execute the evaluation testbench. Make sure it is a complete system, and, pay particular attention to the following items.
  - Include the driver software for the resulting system as well. By default, the software is not included as part of a qar file - you have to include it by hand.
  - Include the bitstream (sof file) for your design. This can be selected as an option during qar file generation.
- Any GEZEL code that you've developed. It is not mandatory to use GEZEL coding, and you may directly code your design in VHDL or Verilog. However, if you have not done HDL coding before, it's strongly recommended to develop your coprocessor in GEZEL, and follow one of the methods explained in the previous Homework.
- A PDF document that describes your resulting design. Your report must include, at the very least, the resulting cycle count for your `bmmdriver_hw()`, and a screen shot of a NiosII terminal to prove your claim. Additionally, please explain your design strategy, the architecture of your hardware/software solution, and overall observations on the design.

Good Luck!!