ECE4530 Fall 2016
Codesign Challenge - Star Map Plotter

Assignment posted on 10 November
Solutions due on 2 December 11:59PM

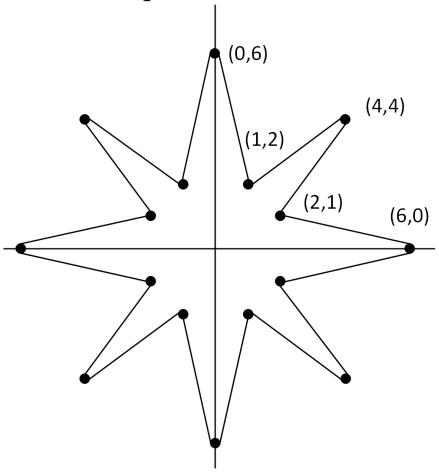
The Codesign Challenge is the final assignment in ECE 4530. This project is an exercise in performance optimization: you will start from a reference application on an ARM processor. You have to improve the performance of the reference application as much as possible, using the hardware/software codesign 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.

The major constraints in the design are as follows.

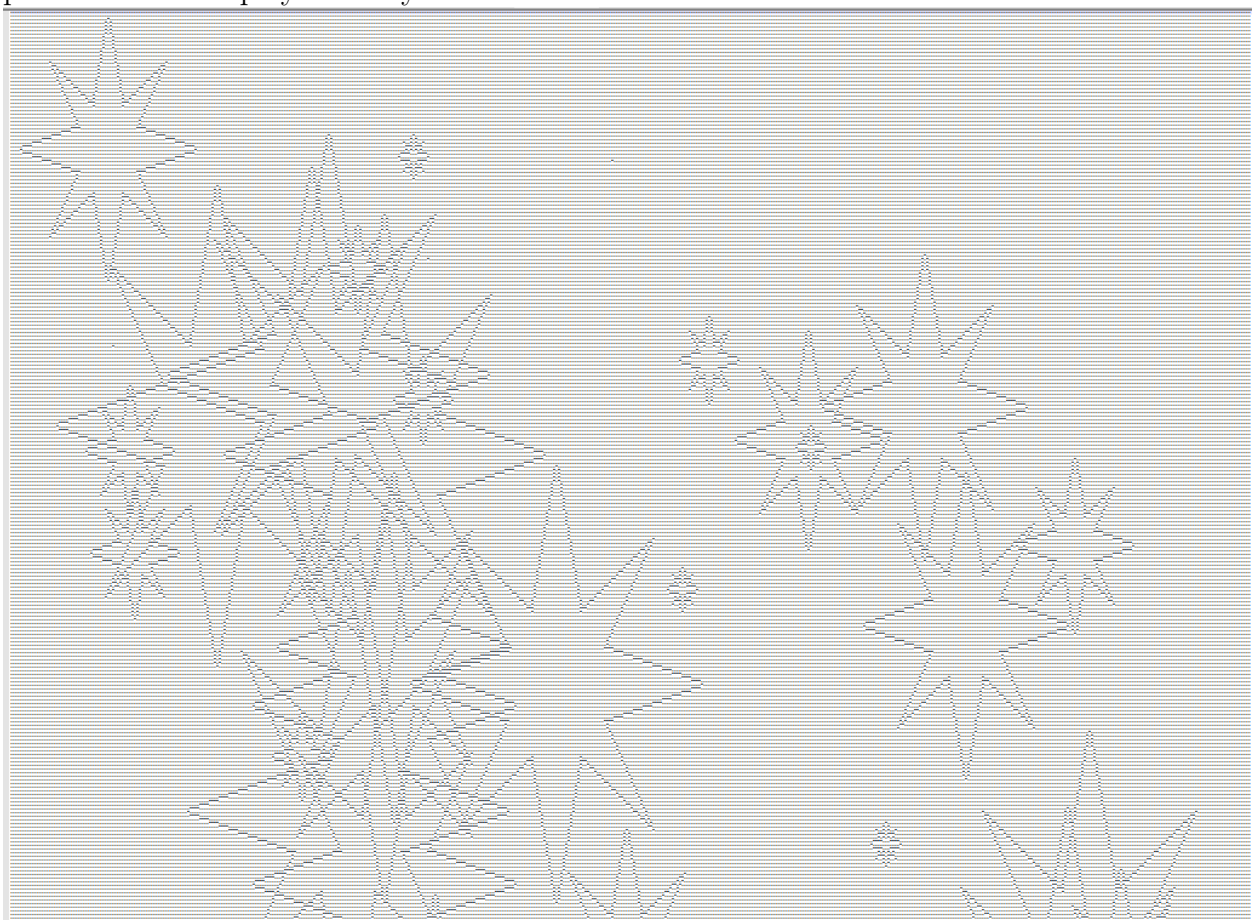
- The final result has to run on a DE1-SoC board
- The final result has to be turned in before the submission deadline, 2 December 11:59PM
- The testbench needs to remain in C on the ARM core; the assignment defines the required API for the accelerated design.

Application: Star Map Drawing

The assignment is an application of a line drawing algorithm over a raster-display. The raster display is a memory of 64 Kilobytes, storing the pixels of a 1024 (X-dimension) by 512 (Y-dimension) display. Your objective is to plot 100 stars of varying origin and size on this display as quickly as possible. 'Plotting a star' means turning on all the pixels that lie on the edge of a star. Each star has a shape that is isomorph to the figure shown below. Note the symmetry in this start. By computing the pixels from only two edges of this star, you are able to directly derive all other pixels.



The reference implementation writes pixels into the memory. By dumping the contents of the memory after the testbench of 100 stars completes, you can get a sense of the data present in the display memory.



The materials you receive for the assignment include the reference hardware implementation, and a reference software testbench.

Implementation of the raster display

The raster display is emulated in a 64 Kilobyte on-chip memory in the FPGA-AXI bus. The on-chip memory has a 64-bit port, even though each pixel is implemented as a single bit. The following functions demonstrate how to set and get pixels from such a memory-mapped display.

```
volatile unsigned char *screen;

#define SCREENSIZEX 1024
#define SCREENSIZEY  512
#define SCREENBYTES (SCREENSIZEX * SCREENSIZEY / 8)
```

```
void setpixel(unsigned x, unsigned y) {
  unsigned bytenum = (y * SCREENSIZEX + x)/8;
  unsigned bitnum  = x % 8;
  if (x > (SCREENSIZEX-1))
    return;
  if (y > (SCREENSIZEY-1))
    return;
  screen[bytenum] |= (1 << bitnum);
}

unsigned getpixel(unsigned x, unsigned y) {
  unsigned bytenum = (y * SCREENSIZEX + x)/8;
  unsigned bitnum  = x % 8;
  if (x > (SCREENSIZEX-1))
    return 0;
  if (y > (SCREENSIZEY-1))
    return 0;
  if (screen[bytenum] & (1 << bitnum))
    return 1;
  else
    return 0;
}
```

Plotting stars

The plotting of a star makes use of a line drawing algorithm called Bresenham's algorithm. This algorithm allows to draw slanted lines using only integer operations - and this makes it quite amendable to hardware implementation. The main function that plots a star the following.

```
void plotstar (unsigned xorig,
               unsigned yorig,
               unsigned starsize) {
  plotstaredge(xorig, yorig, 2*starsize, -starsize, 6*starsize,          0);
  plotstaredge(xorig, yorig, 2*starsize,  starsize, 4*starsize, 4*starsize);
}
```

This function plots a star at center point `(xorig, yorig)`. The star dimensions are scaled by `starsize`, an integer value between 1 and 12. The `plotstar` function plots in fact only 2 edges of a star, but further lower-level functions will replicate these plotting operations in each of the 8 octants of the star, thereby rendering all 16 edges.

The function to plot an edge follows the Bresenham algorithm (which we will explain in class on 11/10).

```
void plotstaredge (unsigned xorig,
                   unsigned yorig,
                   unsigned x0,
                   unsigned y0,
                   unsigned x1,
                   unsigned y1) {

  // Plots the edge of a star at (xorig, yorig)
  // The algorithm requires that (x1 - x0) > 0 and (y1 - y0) > 0

  int y, x, dx, dy, delta;

  dx    = x1 - x0;
  dy    = y1 - y0;

  if (dx > dy) {
    delta = 2*dy - dx;
    y = y0;
    for (x = x0; x <= x1; x++) {
      plotoctants(xorig, yorig, x, y);
      if (delta > 0) {
        y = y + 1;
        delta = delta - dx;
      }
      delta = delta + dy;
    }
  } else {
    delta = 2*dx - dy;
    x = x0;
    for (y = y0; y <= y1; y++) {
      plotoctants(xorig, yorig, x, y);
      if (delta > 0) {
        x = x + 1;
        delta = delta - dy;
      }
      delta = delta + dx;
    }
  }
}
```

Finally, the plotoctants function replicates a generated pixel over all 8 octants.

```
void plotoctants(unsigned xorig,
                 unsigned yorig,
                 unsigned x,
                 unsigned y) {
  setpixel(xorig + x, yorig + y);
  setpixel(xorig - x, yorig + y);
  setpixel(xorig + x, yorig - y);
  setpixel(xorig - x, yorig - y);
  setpixel(xorig + y, yorig + x);
  setpixel(xorig - y, yorig + x);
  setpixel(xorig + y, yorig - x);
  setpixel(xorig - y, yorig - x);
}
```

Testbench and performance measurement

The testbench generates the coordinates for 100 stars, and plots each start sequentially.

```
unsigned dx[NUMSTARS];
unsigned dy[NUMSTARS];
unsigned ds[NUMSTARS];

void referencedata() {
  unsigned i, starsize;
  for (i=0; i<NUMSTARS; i++) {
    starsize = random() % (SCREENSIZEY / 40);
    dx[i] = 6*starsize + random() % (SCREENSIZEX - 12*starsize);
    dy[i] = 6*starsize + random() % (SCREENSIZEY - 12*starsize);
    ds[i] = starsize;
  }
}

void testbench_sw() {
  unsigned i;
  for (i=0; i<NUMSTARS; i++) {
    plotstar(dx[i], dy[i], ds[i]);
  }
}

void main() {

  ...
```

```
  referencedata();

  //==== reference software implementation

  long long time_start_sw = 0;
  long long time_end_sw   = 0;

  clearscreen();

  time_start_sw = cyclecount();
  testbench_sw();
  time_end_sw = cyclecount();

  printf("Reference Plotting time is  %lld\n", time_end_sw - time_start_sw);
  printf("Reference Checksum is       %d\n",   chksumscreen());

  ...

}
```

The testbench then repeats the same for `testbench_hw()` which, by default, will call `testbench_sw()`.

Objective and rules of implementation

- The accelerated design have the same functionality as the reference software design; the `chksumscreen()` function, which computes a checksum over all bytes stored in the raster display, will be used to verify that the pixel map for the reference software and the accelerated design are identical.
- You are allowed to change anything beyond the call to `testbench_hw()` function call. This call needs to make use of the list of coordinates for stars, stored in `dx[], dy[] and ds[]`. However, beyond this list of input data (which needs to be identical for reference software and accelerated hardware), everything is customizable.
- In particular, it's completely your decision on how you will accelerate `testbench_hw()`. This can include building hardware versions of the line drawing algorithm, optimizing the software of the line drawing algorithm, designing a more efficient version of the onchip memory. Note that changes to the onchip memory structure may require adaptions to the `chksumscreen()` function; this is allowed as long as you guarantee that it computes the same operations as the original `chksumscreen()` (that is, mod-256 accumulation of screen bytes).

- You are allowed to change tool settings and enable optimizations as you wish. You are allowed to change the operating frequency of the board as you wish.
- The speedup of your design is equal to the execution time of the original reference design (129.8 million cycles) to the accelerated design. Your speedup will be computed as (129800000 / accelerated plotting time). The value 129800000 can be considered absolute and will not be affected when you use software optimization compiler flags. Obviously, your checksum value must match that of the reference testbench.

Ranking Criteria

All designs will be strictly ranked from best to worst. 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. Working means: your accelerated design computes the correct checksum.
- Metric 1: The speedup of your design following the formula given above. Higher is better.
- Metric 2: The area efficiency of the resulting design, expressed in ALMs. An ALM is an Adaptive Logic Module, a unit of hardware in a Cyclone V FPGA. A lower ALM cell count corresponds to a smaller design. You can find the ALM count in the 'Flow Summary' of Quartus.
- 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 be 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, the speedup (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, on 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 efficiency 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.

- 60 points of the grade are determined by the ranking of your project as described above. The best design gets 60 (out of 60) points, the worst design gets 30 (out of 60) points, and all other designs are linearly distributed between 30 and 60 points.
- 40 points of the grade are determined by the quality of the written documentation you provide with the solution.

Acceleration Strategies

The main objective of this assignment is to make the testbench run as fast as possible, using hardware software codesign. There are several layers to this problem and it is worthwhile to carefully think them through.

- First, there is the Bresenham line drawing algorithm itself. You can study the code of the reference implementation and observe that this is a simple while loop that sequentially generates the pixels from an octant.
- Next, there is a sequence of 100 stars, each with a different centerpoint and size. You cannot predict these specifications; your accelerator will need to be able to accept flexible input.
- The access to the pixel memory is sequential, since the memory is attached as a slave to the Avalon Memory Bus. Hence, building an accelerator needs to consider not only the computation tasks, but also the memory access bottleneck.
- The reference testbench takes over 129 million clock cycles to generate just 100 stars. That is more than a millions cycles per star. The reference implementation is inherently inefficient.

Given these layers, there are several optimization mechanisms to consider.

- Avoid using a random lets try this and see strategy. Very likely you will forget something, or it will lead you into a local optimum while loosing sight of the big picture. Always keep an objective in mind (eg. a desired acceleration factor), and measure your progress in reaching your objective. Such a final objective should be derived through a reasonable back-of-the-envelope calculation. Always look for the bottleneck in the overall system.
- The line drawing algorithm could be completely converted into an FSMD.

- The pixel memory itself can be customized as well. Observe that the number of parallel write-ports and read-ports to the memory is your design decision, and that it is possible to partition a single memory into many smaller memories which could be written into simultaneously. As long as the software can call the `chksumscreen()` function on your memory, you are allowed to make architectural changes. 'Being able to call `chksumscreen()` ' means that your raster screen memory, regardless of your optimizations, needs to have an Avalon Memory Mapped Slave Port.

- The key to succeed in this assignment is NOT in a monster session of back-to-back all-nighters a few days before the deadline. The key is to work on this design in small steps over a longer period. Design is a creative process, and that takes time. Ideas take time to develop. Think about your strategy and discuss with your colleagues. This is an open-ended assignment, and you are competitively graded, so its not in your advantage to disclose all your ideas. However, a brainstorm session or two with your peers may do wonders to sharpen your insight in this assignment.

What to turn in

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

- A rbf (bitstream) file of your final design, and a compiled executable (elf) of your final design.

- The source code of your optimized testbench driver, as a C file.

- The source code of your Verilog design, in case you added a new coprocessor to the QSYS system.

- A PDF document that describes your resulting design. Please explain your design strategy, the architecture of your hardware/software solution, and overall observations on the design. Note that the PDF document counts for 40 % of the grade, so it's worth to do this carefully. **In the PDF, make sure to clearly state the results you obtained for each of the three metrics.**