

E&CE 427 Project: Kirsch Edge Detector

2007t1 (Winter)

| Deliverable | Due Date | Submission Method |
|------------------|---------------------------------|-------------------|
| Dataflow Diagram | Monday, Mar. 5 6:00pm | Drop box |
| Main Project | Thursday, Mar. 22 11:59pm | Electronic |
| Report | 8:30am after project submission | Drop box |
| Demo | TBD | Signup |

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 3 |
| 2 | Edge Detection | 3 |
| 3 | Requirements | 7 |
| 3.1 | System Modes | 7 |
| 3.2 | System Initialization | 9 |
| 3.3 | Input/Output Protocol | 9 |
| 3.4 | Row Count of Incoming Pixels | 10 |
| 3.5 | Memory | 10 |
| 4 | Provided Code | 11 |
| 4.1 | top_kirsch.vhd and lib_kirsch.vhd | 12 |
| 4.2 | Project: kirsch.uwp and top_kirsch.uwp | 13 |
| 4.3 | Reference Model: spec/kirsch.vhd | 13 |
| 4.4 | Packages | 13 |
| 4.5 | Testbench: kirsch_tb.vhd | 14 |
| 4.6 | Test Cases | 14 |
| 4.7 | PC and FPGA communication: kirsch.exe | 14 |
| 5 | Design and Optimization Procedure | 15 |
| 5.1 | Part 1: Explore the Reference Model | 15 |
| 5.2 | Part 2: High-Level Model | 16 |
| 5.3 | Part 3: Optimization | 17 |
| 5.4 | Part 4: Suggested Development/Optimization Process | 18 |
| 5.5 | Part 5: FPGA Synthesis and Timing Simulation | 19 |
| 5.6 | Part 6: Implementation on an FPGA | 20 |
| 6 | Deliverables | 21 |
| 6.1 | Dataflow Diagram | 21 |
| 6.2 | Overview of Project Submission | 22 |
| 6.3 | Directory Structure | 22 |
| 6.4 | Submission Command | 23 |
| 6.5 | Design Report | 23 |
| 7 | Marking | 24 |
| 7.1 | Functional Testing | 24 |
| 7.2 | Performance Testing | 25 |
| 7.3 | Optimality Calculation | 25 |
| 7.4 | Marking Scheme | 26 |

7.5 Late Penalties 26

1 Introduction

The purpose of this project is to explore the digital design process by implementing the Kirsch edge detector algorithm in VHDL. The design process includes exploring the provided reference model of the algorithm, creating a high-level model, optimizing the design, and finally implementing the optimized design on a target device, which is a Cyclone II 2C35 FPGA on the Altera DE2 Board.

Note: *You shall work as groups of four students.*

2 Edge Detection

In digital image processing, each image is quantized into pixels. With gray-scale images, each pixel indicates the level of brightness of the image in a particular spot: 0 represents black, and with 8-bit pixels, 255 represents white. An edge is an abrupt change in the brightness (gray scale level) of the pixels. Detecting edges is an important task in boundary detection, motion detection/estimation, texture analysis, segmentation, and object identification.

Edge information for a particular pixel is obtained by exploring the brightness of pixels in the neighborhood of that pixel. If all of the pixels in the neighborhood have almost the same brightness, then there is probably no edge at that point. However, if some of the neighbors are much brighter than the others, then there is a probably an edge at that point.

Measuring the relative brightness of pixels in a neighborhood is mathematically analogous to calculating the derivative of brightness. Brightness values are discrete, not continuous, so we approximate the derivative function. Different edge detection methods (Prewitt, Laplacian, Kirsch, Sobel etc.) use different discrete approximations of the derivative function. In the E&CE-427 project, we will use a modified version of Kirsch edge detector algorithm to detect edges in 8-bit gray scale images of 256×256 pixels. Figure 1 shows an image and the result of the Kirsch edge detector applied to the image.

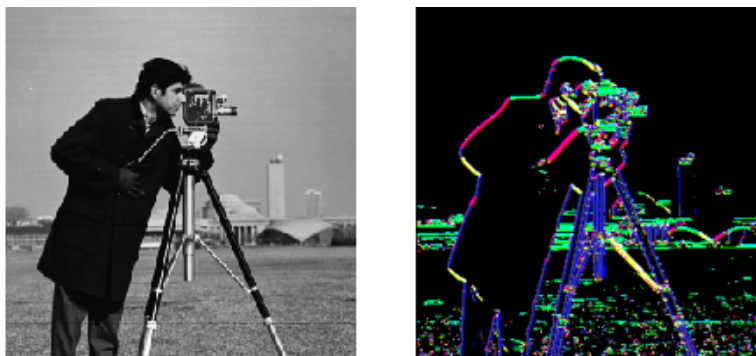
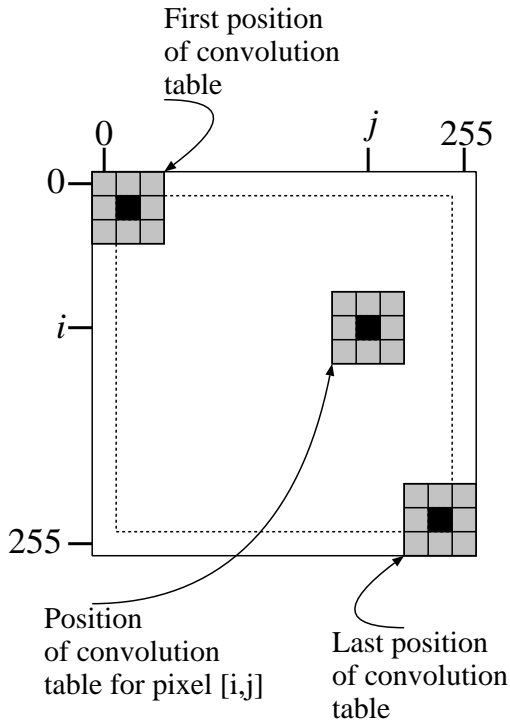


Figure 1: Cameraman image and edge map

The Kirsch edge detection algorithm uses a 3×3 table of pixels to store a pixel and its neighbors while calculating the derivatives. The 3×3 table of pixels is called a *convolution table*, because it moves across the image in a convolution-style algorithm.

Figure 2 shows the convolution table at three different locations of an image: the first position (calculating whether the pixel at [1,1] is on an edge), the last position (calculating whether the pixel at [254,254] is on an edge), and at the position to calculate whether the pixel at $[i, j]$ is on an edge.



| | | |
|----------------|--------------|----------------|
| $Im[i-1, j-1]$ | $Im[i-1, j]$ | $Im[i-1, j+1]$ |
| $Im[i, j-1]$ | $Im[i, j]$ | $Im[i, j+1]$ |
| $Im[i+1, j-1]$ | $Im[i+1, j]$ | $Im[i+1, j+1]$ |

Figure 3: Contents of convolution table to detect edge at coordinate $[i, j]$

Figure 2: 256x256 image with 3x3 neighborhood of pixels

```

for i = 1 to 255-1 {
  for j = 1 to 255-1 {
    for m = 0 to 2 {
      for n = 0 to 2 {
        table[m,n] := image[i+m-1, j+n-1];
      }
    }
  }
}

```

| | | |
|-------|-------|-------|
| [0,0] | [0,1] | [0,2] |
| [1,0] | [1,1] | [1,2] |
| [2,0] | [2,1] | [2,2] |

Figure 5: Coordinates of 3x3 convolution table

Figure 4: Nested loops to move convolution table over image

Figure 3 shows a convolution table containing the pixel located at coordinate $[i, j]$ and its eight neighbors. As shown in Figure 2, the table is moved across the image, pixel by pixel. For a 256x256 pixel image, the convolution table will move through 64516 (254×254) different locations. The algorithm in Figure 4 shows how to move the 3x3 convolution table over a 256x256 image. The lower and upper bounds of the loops for i and j are 1 and 254, rather than 0 and 255, because we cannot calculate the derivative for pixels on the perimeter of the image.

The Kirsch edge detection algorithm identifies both the presence of an edge and the direction of the edge (Figure 6). There are eight possible directions: north, northeast, east, southeast, south, southwest, west, and northwest.

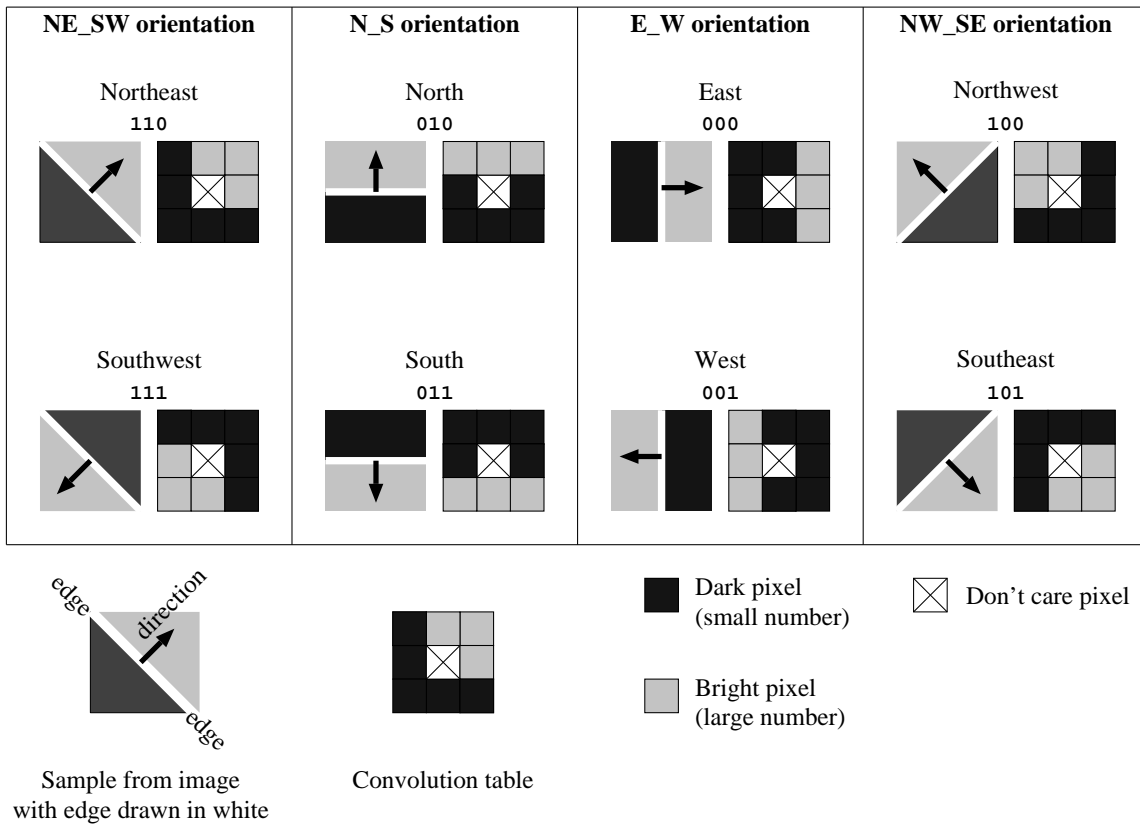


Figure 6: Four orientations and eight directions

For each direction, Figure 6 shows an image sample, a convolution table, and the encoding of the direction. In the image sample, the edge is drawn in white and direction is shown with a black arrow. Notice that the direction is *perpendicular* to the edge. The trick to remember the edge direction is that the direction points to the brighter side of the edge. The eight directions are grouped into four orientations: NE_SW, N_S, E_W, and NW_SE.

For a convolution table, calculating the presence and direction of an edge is done in three major steps:

1. Calculate the derivative for each of the eight directions. The equations for the derivatives are written in terms of elements of a 3x3 table, as shown in Figure 5.

$$\begin{aligned}
 Deriv_{NE} &= 5 \times (table[0, 1] + table[0, 2] + table[1, 2]) - \\
 &\quad 3 \times (table[0, 0] + table[1, 0] + table[2, 0] + table[2, 1] + table[2, 2]) \\
 Deriv_{SW} &= 5 \times (table[2, 1] + table[2, 0] + table[1, 0]) - \\
 &\quad 3 \times (table[2, 2] + table[1, 2] + table[0, 2] + table[0, 1] + table[0, 0]) \\
 Deriv_N &= 5 \times (table[0, 0] + table[0, 1] + table[0, 2]) - \\
 &\quad 3 \times (table[1, 0] + table[2, 0] + table[2, 1] + table[2, 2] + table[1, 2])
 \end{aligned}$$

$$\begin{aligned}
 \text{Deriv}_S &= 5 \times (\text{table}[2,2] + \text{table}[2,1] + \text{table}[2,0]) - \\
 &\quad 3 \times (\text{table}[1,2] + \text{table}[0,2] + \text{table}[0,1] + \text{table}[0,0] + \text{table}[1,0]) \\
 \text{Deriv}_E &= 5 \times (\text{table}[0,2] + \text{table}[1,2] + \text{table}[2,2]) - \\
 &\quad 3 \times (\text{table}[0,1] + \text{table}[0,0] + \text{table}[1,0] + \text{table}[2,0] + \text{table}[2,1]) \\
 \text{Deriv}_W &= 5 \times (\text{table}[2,0] + \text{table}[1,0] + \text{table}[0,0]) - \\
 &\quad 3 \times (\text{table}[2,1] + \text{table}[2,2] + \text{table}[1,2] + \text{table}[0,2] + \text{table}[0,1]) \\
 \text{Deriv}_{NW} &= 5 \times (\text{table}[1,0] + \text{table}[0,0] + \text{table}[0,1]) - \\
 &\quad 3 \times (\text{table}[2,0] + \text{table}[2,1] + \text{table}[2,2] + \text{table}[1,2] + \text{table}[0,2]) \\
 \text{Deriv}_{SE} &= 5 \times (\text{table}[1,2] + \text{table}[2,2] + \text{table}[2,1]) - \\
 &\quad 3 \times (\text{table}[0,2] + \text{table}[0,1] + \text{table}[0,0] + \text{table}[1,0] + \text{table}[2,0])
 \end{aligned}$$

2. Find the value and direction of the maximum derivative.

EdgeMax = Maximum of eight derivatives
 DirMax = Direction of EdgeMax

Note: The following priority order determines which direction gets picked if more than one derivative have the same magnitude.

- (a) Deriv_W
- (b) Deriv_{NW}
- (c) Deriv_N
- (d) Deriv_{NE}
- (e) Deriv_E
- (f) Deriv_{SE}
- (g) Deriv_S
- (h) Deriv_{SW}

This means that if, for instance, Deriv_N and Deriv_E are equal, Deriv_N must be picked.

3. Check if the maximum derivative is above the threshold.

```

if EdgeMax > 400 then
  Edge = true
  Dir = DirMax
else
  Edge = false
  Dir = 000
  
```

3 Requirements

Your circuit (`kirsch`), will be included in a top-level circuit (`top_kirsch`) that includes a UART module to communicate through a serial line to a PC and a seven-segment display controller (`ssdc`) to control a 2-digit seven-segment display. The overall design hierarchy is shown in Figure 7. The entity for `kirsch` is shown in Figure 8.

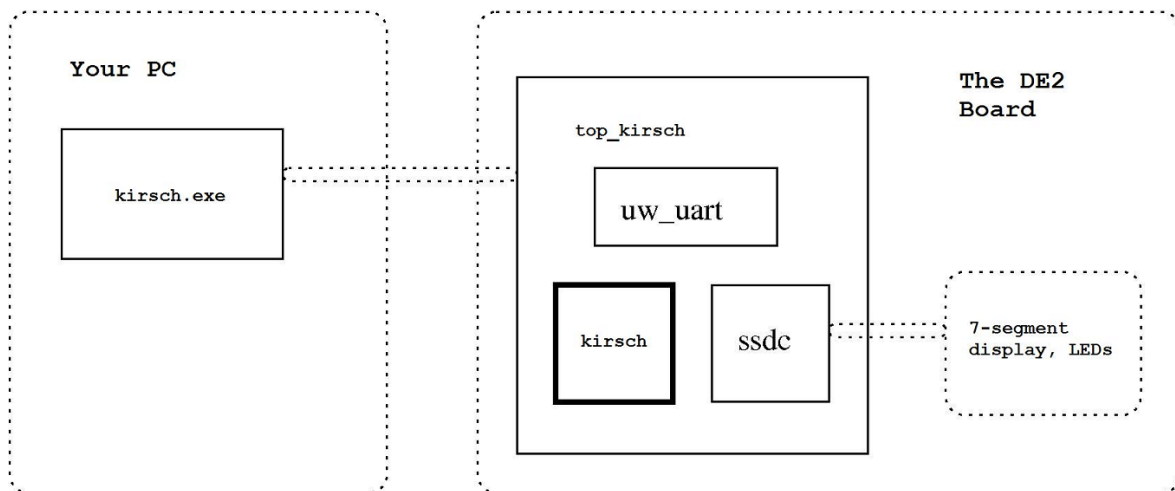


Figure 7: PC-FPGA Communication

Your design will be implemented on the Altera DE2 board with a 50MHz clock.

Note: To work correctly on the DE2 board which has a Cyclone II and to earn full marks in the demonstration, your circuit shall function correctly at a clock frequency of at least 50MHz. The clock speed shown after running `uw-elab` is for Stratix II and not Cyclone II. You have to run `uw-fpga` on your project and the clock information can be found in `LOG/uw-fpga.log`

3.1 System Modes

The circuit shall be in one of three modes: idle, busy, or reset. The encodings of the three modes are shown in Table 1 and described below. The current mode shall appear on the `o_mode` output signal. The `o_mode` signal is connected to the LEDs, which can be useful for debugging purposes.

| mode | o_mode |
|-------|--------|
| idle | "10" |
| busy | "11" |
| reset | "01" |

Table 1: System modes and encoding

- Idle mode: When the circuit is in idle mode, it either has not started processing the pixels or it has already

```

entity kirsch is
  port (
    i_clock      : in std_logic;           -- input clock
    i_valid      : in std_logic;           -- is input valid?
    i_pixel      : in std_logic_vector(7 downto 0); -- 8-bit input
    i_reset      : in std_logic;           -- reset signal
    o_edge       : out std_logic;           -- 1-bit output for edge
    o_dir        : out std_logic_vector(2 downto 0); -- 3-bit output for direction
    o_valid      : out std_logic;           -- is output valid?
    o_mode       : out std_logic_vector(1 downto 0); -- 2-bit output for mode
    o_row        : out std_logic_vector(7 downto 0); -- row number of the in-
    put image
  );
end entity;

```

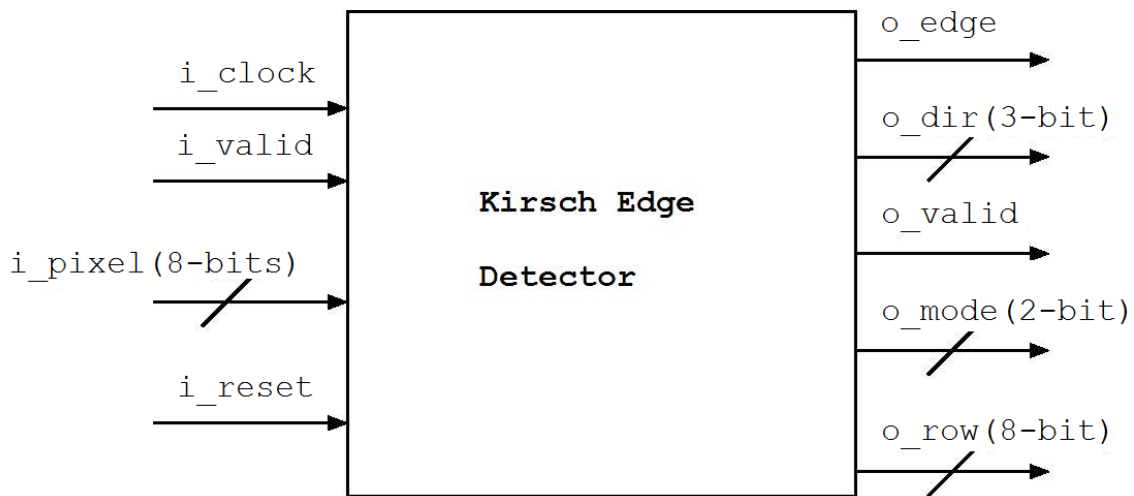


Figure 8: Entity for kirsch edge detector

finished processing the pixels.

- **Busy mode:** Busy mode means that the circuit is busy with receiving pixels and processing them. As soon as the first pixel is received by the circuit, the mode becomes busy and it stays busy until all the pixels (64KB) are processed after which, the mode goes back to idle state.
- **Reset mode:** If `i_reset = '1'` on the rising edge of the clock, then `o_mode` shall be set to "01" (and your state machine shall be reset) in the clock cycle after reset is asserted. The mode shall remain at "01" as long as reset is asserted. In the clock cycle after reset is deasserted (`i_reset = '0'` on the rising edge of the clock), mode shall be set to idle and the normal execution of your state machine shall begin. You may assume that reset will remain high for at least 5 clock cycles.

3.2 System Initialization

On the FPGA used in this project, all flip-flops are set to zero when your program is downloaded to the board. As such, your "Power-up" state should be represented by all zeros. For example, if you use three bits to represent your state, then the power-up state will be "000". This way, when you download your design, you will enter the power-up state automatically.

Your power-up state may correspond to idle mode, or you may have separate states for power-up and idle. After powerup, the environment will always assert reset before sending the first pixel.

Note: *In this project (but not in general), to make your simulation behaviour consistent with the hardware behaviour, you should assign an initial value of zero to your state register in your VHDL code.*

3.3 Input/Output Protocol

Pixels are sent to the circuit through the `i_pixel` signal byte by byte. The input signal `i_valid` will be '1' whenever there is a pixel available on `i_pixel`. The signal `i_valid` will stay '1' for exactly one clock cycle and then it will turn to '0' and wait for another pixel.

In general, the rate at which valid data arrives is unpredictable. There is a constant value in `kirsch_tb.vhd` called "**bubbles**" that determines the number of clock cycles of invalid data (bubbles) between valid data. Your design shall work with any value of "**bubbles**" that is at least 7. Having 7 clock cycles with no valid input data ("**bubbles=7**") allows you to benefit from the unused slots between valid input data to optimize your design. When you download your code into the FPGA, you will use a PC based program to send the data to the FPGA through the serial port, which is slow compared to the maximum frequency of the board. Using the serial port, several hundred clock cycles may pass before the next valid data arrives.

Note: *Your circuit shall work correctly for all data rates between one valid pixel every eight clock cycles (**bubbles=7**) and one valid pixel every 201 clock cycles (**bubbles=200**).*

Whenever an output pair (`o_edge` and `o_dir`) become ready, `o_valid` shall be '1' for one clock cycle to indicate that the output signals are ready to be sent back to the PC. If the pixel under consideration is located on an edge, `o_edge` shall be '1', otherwise the signal should be '0'. `o_dir` shall be "000" if `o_edge` is '0' (no

edge) and it shall show the direction of the edge if `o_edge` is '1'. The values of the signals `o_edge` and `o_dir` are don't cares if `o_valid` is '0'.

Note: Your circuit shall **not** output a result for the pixels on the perimeter. That is, for each image, your circuit shall output $254 \times 254 = 64516$ results with `o_valid='1'`.

3.4 Row Count of Incoming Pixels

The output signal `o_row` shall show the row number (between 0 and 255) for the most recent pixel that was received from the PC. The signal `o_row` shall be initialized to 0. When the last pixel of the image is sent to the FPGA, `o_row` shall be 255. The seven-segment controller in `top_kirsch` architecture displays the value of `o_row` on the seven segment display of the FPGA board.

3.5 Memory

256×256 bytes (=65536 pixels) will be sent to the Kirsch circuit byte by byte either by a testbench (for functional and timing simulation) or by PC to the FPGA (for real test on FPGA board) through the serial port. As illustrated below, you can do Kirsch edge detection by storing only a few rows of the image at a time.

To begin the edge detection operations on a 3×3 convolution table, you can start the operations as soon as the element at 3rd row and 3rd column is ready. Starting from this point, you can calculate the operations for every new incoming byte (and hence for new 3×3 table), and generate the output for edge and direction.

Some implementation details are given below, where we show a 3×256 array. Other memory configurations are also possible.

1. Read data from input (`i_pixel`) when new data is available (i.e. if `i_valid = '1'`)
2. Write the new data into the appropriate location as shown below. The first byte of input data (after reset) shall be written into row 1 column 1. The next input data shall be written into row 1 column 2, and so on. Proceed to the first column of the next row when the present row of memory is full.

256 bytes

| | | | | | | | | | | | | | | | | |
|--------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|-----|-----------|-----------|
| | a_1 | a_2 | a_3 | a_4 | a_5 | a_6 | a_7 | a_8 | a_9 | a_{10} | a_{11} | a_{12} | a_{13} | ... | a_{255} | a_{256} |
| 3 rows | b_1 | xx | xx | xx | xx | xx | xx | xx | xx | xx | xx | xx | xx | ... | xx | xx |
| | xx | xx | xx | xx | xx | xx | xx | xx | xx | xx | xx | xx | xx | ... | xx | xx |

3. The following shows a snapshot of the memory when row 3 column 3 is ready.

| | | | | | | | | | | | | | | | | | |
|---------|--|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|------|-----------|-----------|
| Row Idx | | a_1 | a_2 | a_3 | a_4 | a_5 | a_6 | a_7 | a_8 | a_9 | a_{10} | a_{11} | a_{12} | a_{13} | ... | a_{255} | a_{256} |
| 1st | | a_1 | a_2 | a_3 | a_4 | a_5 | a_6 | a_7 | a_8 | a_9 | a_{10} | a_{11} | a_{12} | a_{13} | ... | a_{255} | a_{256} |
| 2nd | | b_1 | b_2 | b_3 | b_4 | b_5 | b_6 | b_7 | b_8 | b_9 | b_{10} | b_{11} | b_{12} | b_{13} | ... | b_{255} | b_{256} |
| 3rd | | c_1 | c_2 | c_3 | xx | xx | xx | xx | xx | xx | xx | xx | xx | xx | | xx | xx |

4. At this point, perform the operations on the convolution table below:

| | | |
|-------|-------|-------|
| a_1 | a_2 | a_3 |
| b_1 | b_2 | b_3 |
| c_1 | c_2 | c_3 |

Note: This requires 2 or 3 memory reads to retrieve the values from the memory (depending on how you design your state machine). Come up with a good design so that the above write and read can be done in parallel.

5. When the next pixel (c_4) arrives, you will perform the operation on the next 3×3 convolution table:

| | | |
|-------|-------|-------|
| a_2 | a_3 | a_4 |
| b_2 | b_3 | b_4 |
| c_2 | c_3 | c_4 |

6. When row 3 is full, the next available data shall be overwritten into row 1 column 1. Although physically this is row 1 column 1, virtually it is row 4 column 1. Note that the operations will not proceed until the 3rd element of 4th row (d_3) is available in which case the operation will be performed on the following table based on the virtual row index as depicted in the following figure.

Virtual
Row Idx

| | | | | | | | | | | | | | | | | |
|-----|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|-----|-----------|-----------|
| 4th | d_1 | d_2 | d_3 | a_4 | a_5 | a_6 | a_7 | a_8 | a_9 | a_{10} | a_{11} | a_{12} | a_{13} | ... | a_{255} | a_{256} |
| 2nd | b_1 | b_2 | b_3 | b_4 | b_5 | b_6 | b_7 | b_8 | b_9 | b_{10} | b_{11} | b_{12} | b_{13} | ... | b_{255} | b_{256} |
| 3rd | c_1 | c_2 | c_3 | c_4 | c_5 | c_6 | c_7 | c_8 | c_9 | c_{10} | c_{11} | c_{12} | c_{13} | ... | c_{255} | c_{256} |

the convolution table:

| | | |
|-------|-------|-------|
| b_1 | b_2 | b_3 |
| c_1 | c_2 | c_3 |
| d_1 | d_2 | d_3 |

7. Moving the 3×3 table over the 256×256 memory and performing the operation is in fact a convolution process. Regarding that the operations will start at the 3^{rd} row of 256×256 memory and at the 3^{rd} element of each row, the number of 3×3 tables on which the operations will be performed, is calculated by: $254 \times 254 = 64516$.

Your memory arrays shall be formed using instances of the 1×256 entry memory (provided in Ram.vhd), where each entry is 8 bits wide.

4 Provided Code

You are given a set of files that can be copied to your local account. The procedure below shows how this can be done if the desired target directory is `/home/your_userid/ece427/proj`.

```
% cd ~/ece427
% cp -r /home/ece427/public_html/cur/proj/proj proj
```

There are three directories:

```
spec  Specification.
hlm   High-level model
opt   Optimized code and support software to run on PC for communication with FPGA
```

Directories

| spec | hlm | opt | File | |
|------|-----|-----|------------------------|--|
| | | ✓ | top_kirsch.uwp | Project file with kirsch, uart, ssdc |
| | | ✓ | top_kirsch.vhd | Top level code for final synthesis |
| | | ✓ | lib_kirsch.vhd | Source code for uart, uw_uart, ssdc and sevensegment |
| ✓ | | | kirsch-report.vhd | Reference model with extra debugging functionality |
| ✓ | ✓ | ✓ | kirsch.uwp | Project file |
| ✓ | ✓ | ✓ | kirsch.vhd | Source code for edge-detection design |
| | ✓ | ✓ | Ram.vhd | VHDL code for the memory used for simulation and synthesis |
| ✓ | ✓ | ✓ | kirsch_synth_pkg.vhd | Constants and types (synthesizable) |
| ✓ | ✓ | ✓ | kirsch_unsynth_pkg.vhd | Constants, types, reading/writing images (unsynthesizable) |
| ✓ | ✓ | ✓ | string_pkg.vhd | Functions for string conversion |
| ✓ | ✓ | ✓ | kirsch_tb.vhd | Testbench |
| ✓ | ✓ | ✓ | kirsch_tb.sim | Simulation script |
| | ✓ | | README | Group information |
| ✓ | ✓ | ✓ | tests (directory) | 4 test images in different formats: *.txt input images in text format for simulation *.bmp input images in bitmap format for running on FPGA |
| | | | | Also: kirsch.exe a PC-based program to transmit and receive images |

Note: *The kirsch.vhd in the spec directory is a reference model that you may use to evaluate the correctness of your code.*

Note: *Do not modify the following files:*

```
lib_kirsch.vhd           top_kirsch.vhd           Ram.vhd
kirsch_synth_pkg.vhd   kirsch_unsynth_pkg.vhd   string_pkg.vhd
```

When your design is marked, we will use the original versions of these files, not the versions that are from your directory.

4.1 top_kirsch.vhd and lib_kirsch.vhd

When you wish to download your design to the FPGA, you will use `top_kirsch.uwp`, `top_kirsch.vhd`, and `lib_kirsch.vhd` to build a complete system that includes a UART and a driver for the seven-segment display.

For this project, Table 2 shows the mapping between the top-level entity ports (`top_kirsch.vhd`) and the physical pins on the DE2 FPGA board. The `uw-fpga` script does these pin mappings automatically.

| Description | Entity Port | Pad Location | Direction |
|---|-------------|---|-----------|
| 50 MHZ Clock | CLK | Pin_N2 | input |
| Seven-segment display, LEDs | o_sevenseg | Pin_M[2-5], Pin_C22, Pin_P[3-4, 9], Pin_R2, Pin_N9, Pin_P[2,3,6,7,9], Pin_AA20, Pin_Y18 | output |
| KEY0, Reset. 1 when the button is pressed, 0 otherwise. | nRST | Pin_G26 | input |
| UART data transmit | TXFLEX | Pin_B25 | output |
| UART data receive | RXFLEX | Pin_C25 | input |

Table 2: Ports and pins for `top_kirsch`

4.2 Project: `kirsch.uwp` and `top_kirsch.uwp`

If you create additional files for your source code, you will need to add the names of these files to `kirsch.uwp` and `top_kirsch.uwp`. For more information, see the web documentation on the format of the UWP project file.

4.3 Reference Model: `spec/kirsch.vhd`

You are given a reference model (`kirsch.vhd`) for a Kirsch edge detector, which is located in the `spec` directory. You are required to simulate the reference model with the given four different test cases and generate four set of results (which will be stored in text format such as `test1_res.edg` and `test1_res.dir`). You will use these results to verify the correctness of your high level model code.

Note: *The behavioral code uses a 256×256 2-dimensional array to store the image data. You are required to build your memory using the memory component in `Ram.vhd`.*

Note: *The simulator might not be able to display the whole 256×256 memory array in waveforms. When simulating the reference model, do not choose the “memory” signal.*

4.4 Packages

There are three libraries (`string_pkg.vhd`, `kirsch_synth_pkg.vhd` and `kirsch_unsynth_pkg.vhd`) that are used by the reference model (`kirsch.vhd`) and the testbench (`kirsch_tb.vhd`). To simulate the reference model you will need all these libraries. However, you may or may not need them for your high level model. Just note:

- `kirsch_tb.vhd` needs to have access to the three files, so whenever you run functional or timing simulation, you will need the three libraries.
- `string_pkg.vhd` and `kirsch_unsynth_pkg.vhd` are not synthesizable and therefore, your synthesizable code cannot use these libraries.

4.5 Testbench: `kirsch_tb.vhd`

You are given a testbench (`kirsch_tb.vhd`) that can be used to simulate both the given reference model code and your high level model (or optimized) code (both functional and timing simulation). The testbench reads a 256×256 image data from a text file, passes the data to Kirsch circuit (`kirsch.vhd`) byte by byte, receives the outputs of the circuit (edges and directions) and stores them in three separate text files (e.g. `test1_res.dir`, `test1_res.edg`, `test1_res.ted`).

Each number in the `<filename>.dir` and `<filename>.edg` files corresponds to the presence of an edge (0 or 1) and the direction of the edge (0 to 7 for each direction) respectively. In the `<filename>.ted` file, there are four columns: edge (0 or 1), direction (0 to 7 for each direction), row number (in the image), and column number. The ted file can be converted into a bitmap and vice versa using `kirsch.exe`. `kirsch.exe` will be discussed in more detail in sections 4.7 and 5.6.

Note: *You are encouraged to modify the testbench to increase the productivity of your functional verification. For example, automatically checking if results are correct, running multiple test cases in a row, exercising corner cases, etc.*

4.6 Test Cases

The data of four real 256×256 images are provided in text format for functional simulation of reference model, high level model and functional and timing simulation of optimized code. The text files (`test1.txt`, etc.) can be loaded by the testbench. To switch between test cases, open the testbench file (`kirsch_tb.vhd`) and change the filename in line 29. For example, to run test case `test2.txt`, change the filename to `test2` as shown below:

```
constant test_name : string := "test2";
```

Note: *You are encouraged to develop your own test cases, possibly very small test cases, for initial debugging, fast turnaround on simulation, and exploring corner cases*

4.7 PC and FPGA communication: `kirsch.exe`

A PC based program (`kirsch.exe`) will send four bitmap images as test cases (`test1.bmp` ... `test4.bmp`) to FPGA and receive the results of your circuit for evaluation purpose. `kirsch.exe` will be discussed in more

detail in section 5.6.

Note: *The four test files in text format (test1.txt, test2.txt, etc.) and the ones in bitmap format (test1.bmp, test2.bmp, etc.) are in fact two different formats of the same images.*

kirsch.exe is a PC based program that reads the bitmap test files (test1.bmp ... test4.bmp) and sends their pixels to your circuit in FPGA. You can switch between the test cases by choosing from four different radio buttons on kirsch.exe interface. When the pixels are sent to FPGA and the results are received back, kirsch.exe displays the original image and the edge map. When the original image is displayed, just close it to see the edge map image. When you close the edge map image, the percentage of error will be displayed on kirsch.exe interface. kirsch.exe uses the test*_spec.bmp files to evaluate the result of your Kirsch code sent back to the PC.

Note: *kirsch.exe and all eight bitmap files should be in the same directory in PC.*

5 Design and Optimization Procedure

Your design and optimization should proceed through the following steps:

5.1 Part 1: Explore the Reference Model

In Part 1, you will use the reference model code of Kirsch with the provided testbench (located in spec directory) to run four test cases and produce the results. You will use these results in the following parts to verify the functionality of your VHDL code.

1. The provided codes for this part are located in spec directory
2. Run the reference VHDL code in kirsch.vhd using the testbench kirsch_tb.vhd.
3. Try all four test cases and store the results.
4. To switch between test cases, open the testbench file kirsch_tb.vhd and change the filename in line 29.
5. Try to understand the Kirsch algorithm by exploring the behavioral code.
6. Run the simulation using the following command:

```
% uw-sim kirsch.uwp
```

Note: To run the command line version of *uw-sim* without opening the graphical interface, append *-c* to the command:

```
% uw-sim kirsch.uwp -c
```

Note: To pass generic to the simulation script, append *-G<name>=<value>* to the command, for example:

```
% uw-sim kirsch.uwp -c -Gtest_name="test2"
```

Note: When simulating the reference model, the "image" signal in *kirsch.vhd*, which is a 256x256 array, should NOT be selected. Otherwise, the simulator might run out of memory, since it is a large 2-dimensional array with so many elements.

Note: If you want to see how an image edge map (e.g. *test1.edg*, which can be the result of either the reference model or your own code) looks like and you have access to Matlab (you can use Matlab on E2-2363 machines), follow the instructions below.

1. Run Matlab
2. Change the current directory in Matlab to the directory where the edge map file (e.g. *test1.edg*) is located
3. In Matlab command window type: `edge=load('test1.edg');` Press enter
4. Type: `imshow(edge);` Press enter

This will display the edge map of the original image.

The goal of simulating the reference model is to produce results (*testx.edg* and *testx.dir*) for four tests cases. You will use these results in the next steps to verify the functionality of your VHDL code. In addition, you may go through the reference model and try to understand the Kirsch algorithm by exploring the behavioral code. Once you are comfortable with the algorithm, you will have enough information to implement your high level model and optimized code.

5.2 Part 2: High-Level Model

In this part, you will create a high-level model of Kirsch algorithm. Your high-level model does not need to be synthesizable, but it must use memory arrays as internal storage. You should partition your computation into clock cycles, so that you can predict the latency through your final circuit and begin to estimate the clock speed that you will achieve.

Note: The measurement for latency begins as soon as the first 3x3 table becomes available. For example, the latency will be 1, if the corresponding outputs of the first table become available in the immediate next clock cycle following the first valid table. In other words, as soon as the pixel in the third row and third column of the image arrives, the measurement for the latency should begin. The number of "bubbles" does not have anything to do with overall latency.

To verify the functionality of your code, you will use the test results that you have already created using the reference model code in Part 1.

Note: *When first debugging your model, fast turnaround time on simulations is very important. You should develop some small test cases (e.g. 8×8 arrays) and modify the testbench and your design to work with these smaller arrays.*

Note: *The functionality of your high level model should exactly match the provided reference model*

1. The provided codes for this part are located in `h1m` directory
2. Create you own high level model by completing (`kirsch.vhd`), which is located in `h1m` directory.
3. Use the VHDL code of the RAM as provided in `Ram.vhd`.
4. Simulate your design using the provided testbench.
 - To verify that your high level model code is correct, try all 4 of the provided test cases.
 - To switch between test cases, open the testbench file `kirsch_tb.vhd` and change the filename in line 29.
 - Run the simulation using the following command:

```
% uw-sim kirsch.uwp
```

Note: *Some groups made good use of Perl and other scripting languages to generate tests and evaluate the correctness of results.*

Note: *Many groups used one or more of the following programs to compare their output files against the specification: diff, cmp, BeyondCompare, vimdiff*

Note: *To verify the datapath while avoiding any bugs that might be present in the pipeline-control circuitry, send one parcel at a time through the pipeline.*

Note: *The small amount of extra time required to write the assertions may be rewarded with a significant reduction in the time required to find the causes of bugs.*

When you are confident that your code is functionally correct, proceed to Part 3.

5.3 Part 3: Optimization

In Part 3, you will write synthesizable code and optimize. Once you ensured that your code is functionally correct and synthesizable, you will apply the optimization techniques that you have learned in E&CE427 course to improve the design performance and reduce the consumed area.

1. The provided codes for this part are located in `opt` directory

2. Use the RAM component provided in `Ram.vhd`.
3. Check that your design is synthesizable using the following `uw-elab` command:

```
% uw-elab kirsch.uwp
```

The `uw-elab` command also displays the clock speed and area of your design if it were to be run on a Stratix II FPGA.

4. Begin your optimizations by working on latency, then area, and finally clock speed. More details are in Part 4. Return to step 3 until you are satisfied with the speed and area.

5.4 Part 4: Suggested Development/Optimization Process

1. Predict the slowest operation (e.g. memory access, 8-bit subtract, etc).
2. Estimate the maximum clock speed that can be achieved if a pipeline stage contains just the slowest operation. This gives you an upper bound on the clock speed.
3. Choose your optimality goal. From your maximum clock speed and optimality target, calculate the maximum area that you can use.
4. Decompose your design into pipeline stages that are predicted to satisfy your clock speed target.
5. Do area optimizations to reduce your area to your area target.
6. After you are within 5-10% of your area target, or are decreasing your area by less than 10% per day, change your focus to clock speed optimizations.

Note: *Once your optimizations begin to change your area and clock speed by less than 10%, optimizations often have unpredictable effects on area and clock speed. For example, combining two separate optimizations, each of which helps your design, might hurt your design.*

Note: *You are probably wise to stop your optimizations at the point where your optimizations are improving the optimality by 2–3%.*

Note: *Many groups kept logs of their design changes and the effect on optimality scores. The vast majority felt that these logs were very helpful in achieving good optimality scores.*

Some observations on optimization:

- Reusing components can increase the amount of control circuitry.
- Identify common subexpressions after exploring algebraic modifications to equations (e.g. play with associativity to increase number of common subexpressions).
- Aim for an equal number of registers and lookup tables. If your area is dominated by lookup tables, there is no point in trying to save area by reducing the number of registers.

- Minimize the number of flip-flops that are reset. For example, usually there is no need to reset datapath registers. What that means is that you do not need a reset in all clocked *processes*, only in each clocked *circuit*. The purpose of the reset signal is to get the circuit into a known state from which it can begin processing. So, you almost assuredly need to reset your state machines, but probably do not need to reset your datapath registers. Removing reset from flops that don't need a reset might save you some area, and might boost your clock speed.
- Avoid using the type `signed` unless you need negative numbers. Comparison of signed numbers is more complicated than unsigned. Datapaths that use signed data are often slower than those that use unsigned data.
- When comparing a signal to a value, it may not be necessary to look at the full width of the signal. For example, if the value being compared to was 64 and the signal width was 8 bits, only bits 6 and 7 need to be examined to determine if the signal is greater than or equal to 64.

5.5 Part 5: FPGA Synthesis and Timing Simulation

In Part 5, you will synthesize your code for timing simulation. You will ensure that your code is functionally correct regarding timing simulation. Since the `uw-fpga` command uses the information of the Cyclone II FPGA, if your code works under the timing simulation, it will have a higher chance of running correctly on the DE2 Board.

1. The provided codes for this part are located in `opt` directory
2. Use the RAM component provided in `Ram.vhd`.
3. Synthesize your design using the following `uw-fpga` command:

```
% uw-fpga kirsch.uwp
```

4. Perform the timing simulation of your design, by:

```
% uw-timsim kirsch.uwp
```

5. Try all 4 test cases to ensure your code is working properly with back-annotated delay information. To switch between test cases, follow the same instructions as described in Section 5.2.

Note: For timing simulation, note that you need to drive all the i/o signals in `kirsch.vhd` entity. In other words, if you do not assign anything to, for example, `o_mode`, timing simulation will fail due to signal mismatch between `kirsch_tb.vhd` and `kirsch.vho`.

Note: Timing simulation can be very time consuming due to the large number of pixels to be processed (65536). Modify the image size in the package, so that instead of sending 256×256 pixels to your code, it sends a small portion of data (e.g. 16×16)

5.6 Part 6: Implementation on an FPGA

In Part 6, you will re-synthesize your code with the top-level files and will download your design to one of the Altera DE2 boards in the lab, and test your design on an FPGA chip.

1. In order for the PC to send test cases to the board and to your state machine, the support of UART serial communication is required. To reduce the complexity of the project, we have provided you two wrapper files `lib_kirsch.vhd` and `top_kirsch.vhd`. Figure 7 illustrates the hierarchy.
 - The entity `ssdc` is a Seven Segment Display Controller. It controls and communicates with the seven segment display on the Altera DE2 board.
 - The entity `uw_uart` is a UART controller for sending and Receiving information to and from PC.
 - All 3 blocks (`kirsch`, `uw_uart`, `ssdc`) are embedded in `top_kirsch`.

Each signal in `top_kirsch.vhd` is mapped to a pin on the FPGA chip. The mappings will be performed automatically by the `uw-fpga` script based on the contents in `/home/ece427/lib/uw/DE2-pins.tcl`. For resetting your state machine(s) in the FPGA, press the switch labelled “**KEY0**“ on the Altera DE2 board. The `i_reset` pin is mapped to this switch.

Now all you need to do is the following (in the same directory):

```
% uw-fpga top_kirsch.uwp
```

2. The `uw-fpga` script should generate a file called `top_kirsch.sof`. Download this file to the board and test your design. If you have not done the background reading, please read “Downloading Designs to the DE2 board”.
3. The next step is to setup your PC for downloading test cases to the board. A program called `kirsch.exe` is given and can be found in `opt/tests` directory. You need to copy the following files from the `opt/tests/` directory in your UNIX account to your Windows machine desktop.

```
kirsch.exe
test1_org.bmp      test2_org.bmp      test3_org.bmp      test4_org.bmp
test1_spec.bmp     test2_spec.bmp     test3_spec.bmp     test4_spec.bmp
```

Note: *The bitmap test files must be copied into the same directory on the desktop of PC where you execute `kirsch.exe` in Windows.*

4. For information about mapping your UNIX drive into Windows, see http://www.ece.uwaterloo.ca/mach_www/SMB-use.html.
5. Run `kirsch.exe` on your PC that is connected to the Altera DE2 board, select a test case from the radio buttons and press start. The program will send the data to the board. The result that your circuit generates will be sent back to PC and be displayed on monitor. `kirsch.exe` will also display the percentage of error that your design result contains. During the communication between PC and FPGA,

- The seven segment will display the row count of image being sent to FPGA that will be displayed on numeric digits.
- LEDG7 and LEDG6 will display o_mode(1) and o_mode(0) respectively.

Note: *The communication between PC and FPGA board is through the serial port of the PC by polling. Therefore, if the CPU is busy with other jobs while sending and receiving data to/from FPGA board, data might be lost. Do not run any other program on the PC when you are running your circuit on the FPGA and using kirsch.exe for sending and receiving data.*

6 Deliverables

The deliverables contains five parts:

1. Dataflow diagram (Described in Section 6.1)
2. High-level model (VHDL code)
3. Optimized implementation (VHDL code)
4. Design Report (Described in Section 6.5)
5. Design demonstration

6.1 Dataflow Diagram

Please note that this is a separate deliverable from the rest of the project. Its due date is outlined on the first page of this document. Submit this report in the drop box marked "ECE 427", located directly across the hall from E2-3344.

Your dataflow diagram should show the calculations of the derivatives of the eight directions. Your dataflow diagram should refer to Figure 9. The steps to create a dataflow diagram are outlined in Section 2.6 in the course notes and an example is given in Section 2.9.

It is not expected that your design will be completely optimized at this point. In your final report, you will have an opportunity to discuss how you improved and optimized this high-level model to arrive at your final design.

No explanatory text is needed to show how you arrived at your final dataflow diagrams and no extra marks will be given for this. No fancy cover page is needed, but the diagram should include the name and students ID number of each group member. Your dataflow diagram will be counted towards your design report.

| | | |
|-------|-------|-------|
| a_1 | a_2 | a_3 |
| b_1 | b_2 | b_3 |
| c_1 | c_2 | c_3 |

Figure 9: Convolution table

6.2 Overview of Project Submission

All submissions will be processed as soon as they are received. First, submissions will be tested for the proper directory structure. Second, a simple “Dead or Alive” test will be run on the code in the `opt/code` directory to ensure that basic functionality is present. An email will be sent to the submitter indicating whether or not the submission was successful. If there are warnings or errors, you should attempt to fix them and resubmit to ensure that your design can be properly processed for full functionality testing.

Two important notes:

- Do not rely on the “Dead or Alive” test as the only testing mechanism for your design. This is a simple test which only ensures that you have followed the proper naming requirements and that your design has basic functionality. Full functional testing will not be performed until all final submissions have been received.
- You can submit your design as many times as you want without penalty. Each submission will replace the previous design and **only the final submission will be marked**.

6.3 Directory Structure

Submissions shall include the following in the specified directory structure. Other files and directories may be present, but they will not be submitted. The submission script will gather only the following directories and files.

hlm/

README – Names and UWuserids for project members, plus any additional information to help us reproduce your results (e.g. If you know that your design produces Xs in timing simulation, mention that here.).

kirsch.vhd – Unoptimized High Level model: prior to code optimization and synthesis. This is for simulation and may or may not be synthesizable.

opt/

***.vhd** – All VHDL files needed to synthesize and simulate your design.

kirsch.uwp – project file for kirsch

top_kirsch.uwp – project file for top_kirsch, which includes the UART and seven-segment-display controller.

6.4 Submission Command

To submit your design, enter the following command **from the location containing the above directory structure**:

```
ece427-project-submit
```

Unless a message is displayed indicating that the submission has been aborted, your submission has been sent and you will receive an email containing your submission results. It may take up to 30 minutes for this email to be sent as the submission first needs to be processed and tested.

Don't forget to turn in your report by 8:30am after you submit your project!

6.5 Design Report

Maximum: 3 pages, no cover page (just put project members and UWuserids), minimum 11pt font.

We will only mark the first 3 pages!

Your audience is familiar with the project description document and the content of E&CE 427. Don't waste space repeating requirements from the project description or describing standard concepts from the course (e.g. pipelining).

Good diagrams are very helpful. Pick the right level of detail, layout the diagram carefully, describe the diagram in the text.

Design reports are to be turned in to the E&CE 427 drop box by 8:30am after you submit the final version of your project. For example, if you submit your project at 10pm on March 18, then you must turn in your report by 8:30am on March 19. If you submit your project at 7:30am on March 19, then you must submit your report by 8:30am on March 19.

Note: *Most of the report can be written before beginning the final optimizations. Most groups have the report done, except for their final optimality calculation, before beginning their final optimizations. This allows them to submit their report as soon as they are satisfied with their optimality score.*

As described in the next few sections, the report should address design strategy, performance and optimization, and validation.

6.5.1 Design Goals and Strategy

Discuss the design goals for your design, and the strategies and techniques used to achieve them. Discuss any major revisions to your initial design and why such revisions were necessary. Present a high-level description of

your optimized design using one or more dataflow diagrams, block diagrams, or state machines.

6.5.2 Performance Results versus Projections

Include performance estimates made at the outset of your design and compare them to the results achieved. Describe the optimizations that you used to reduce the area and increase the performance of your design. Describe design changes that you thought would optimize your design, but in fact hurt either area or performance. Explain why you think that these “optimizations” were unsuccessful.

Include a summary of the area of your major components, the maximum clock speed of the design, the critical path, latency, and throughput of your design. In addition, include an overall optimality calculation.

6.5.3 Validation Plan and Test Cases

Summarize the validation plan and test cases used to verify the behaviour of your design and comment on their effectiveness as well as any *interesting* bugs found.

7 Marking

7.1 Functional Testing

Designs will be tested both in a demo on the Altera DE2 Boards and using an automated testbench hooked up to the entity description of your design.

The design is expected to correctly detect all the edges of input images and exhibit correct output behaviour on both functional simulation, timing simulation and on the FPGA board. For full marks, designs shall work correctly on the FPGAs on the Altera DE2 Development Boards in the labs as well as in test simulations of the post-layout design with back-annotated delays.

A *Functionality Score* will be used to quantify the correctness of your design. This score will range from 0 (no functionality) to 100 (perfect functionality). For calculating *Functionality Score*, we will run a series of tests to evaluate common functionality and corner cases, then scale the mark as shown below:

| Scaling Factor | Type of simulation |
|----------------|--|
| 100 | we will first try timing simulation (<code>uw-timsim</code>) of the post-place-and-route design (<code>opt/kirsch.vho</code>). |
| 80 | if that fails, we'll try the post-place-and-route design (<code>opt/kirsch.vho</code>) with zero-delay simulation (<code>uw-vhosim</code>) |
| 60 | if that fails, we'll try the design prior to place and route (<code>opt/kirsch.vhd</code>) |
| 50 | if that fails, we'll try the high-level model (<code>hlm/kirsch.vhd</code>) |

7.2 Performance Testing

Throughput is a performance measure for stream-processing tasks, such as digital-signal processing and image processing. Throughput is dependent upon the clock frequency and the number of clock cycles that must elapse between valid data entering the system. For this project, we have constrained the number of clock cycles between valid data to be 7: that is your circuit must work correctly if valid data is received every 8 clock cycles. Thus, throughput is dependent solely upon clock frequency.

Clock Frequency represents the maximum clock frequency as reported by uw-elab.

Note: For calculations, express the Clock Frequency in megahertz.

For stream-processing tasks, latency is largely irrelevant. However, it is important to use your clock cycles wisely and not be wasteful. For this reason, we will measure the latency through your system and the latency will factor into the performance calculation. For the performance tests, we will measure the number of clock cycles by simulating your optimized design prior to place and route (`opt/kirsch.vhd`).

7.3 Optimality Calculation

The optimality of each design will be evaluated based on functionality score, throughput, as well as the design's area (LE count). In your report you will need to calculate clock speed, latency, and optimality (Equation 1). Systems with an excessive latency (more than 15 clock cycles) will be penalized as shown below:

If Latency \leq 15 then

$$\text{Optimality} = \text{Functionality} \times \frac{\text{ClockSpeed}}{\text{LE Count}} \quad (1)$$

If Latency $>$ 15 then

$$\text{Optimality} = \text{Functionality} \times \frac{\text{ClockSpeed}}{\text{LE Count}} \times e^{\frac{-(\text{Latency}-15)}{20}} \quad (2)$$

Note: Assuming perfect functionality, if you achieve an optimality score of 500, you are guaranteed to obtain at least 90% of the optimality mark.

Note: For area, we count the number of FPGA cells that are used. This is usually the greater of either the number of flip-flops that are used or the number of 4:1 combinational lookup tables (or PLAs) that are used. Our area calculations do not take into account the ESBs used by memory arrays, but do take into account the normal FPGA cells for address decoding, etc that are used by the memory arrays.

7.4 Marking Scheme

| | | | | | |
|---|--------------------------|-----|---|-----|-----|
| Design Report <ul style="list-style-type: none"> • Clearness, completeness, conciseness, information analysis | 15% | | | | |
| Submission <ul style="list-style-type: none"> • Correct signal, entity, file, and directory names • Correct I/O protocol | 5% | | | | |
| Demo <table style="width: 100%; border: none;"> <tr> <td style="width: 80%;">Test image functionality</td> <td style="width: 20%; text-align: right;">15%</td> </tr> <tr> <td>Discussion about your design and design process</td> <td style="text-align: right;">15%</td> </tr> </table> | Test image functionality | 15% | Discussion about your design and design process | 15% | 30% |
| Test image functionality | 15% | | | | |
| Discussion about your design and design process | 15% | | | | |
| Optimality (see Equation 1) <ul style="list-style-type: none"> • High speed, small area, functionality | 50% | | | | |

7.5 Late Penalties

To mimic the effects of tradeoffs between time-to-market and optimality, there is a regular deadline and an extended deadline that is one week after the regular deadline.

Projects submitted between the *regular deadline* and *extended deadline* will receive a multiplicative penalty of 0.05% per hour. Projects submitted at the extended deadline will receive a multiplicative penalty of 8.4%. Projects submitted after the extended deadline will be penalized with a multiplicative penalty of 8.4% plus 1% per hour for each hour after the extended deadline. Penalties will be calculated at hourly increments.

Example: a group that submits their design on Wed Mar 21 at 1:26pm is 2 days and 13 hours after the regular deadline. This gives them a penalty of: $(2 \times 24 + 13) \times 0.05 = 3.05\%$. Thus, if they earn a pre-penalty mark of

85, their actual mark will be $85 \times (100\% - 3.05\%) = 82$.

Example: a group that submits their design on Tue Mar 27 at 3:42am is 1 day 4 hours after the extended deadline. This gives them a penalty of: $8.4\% + (1 \times 24 + 4) \times 1\% = 36.4\%$. Thus, if they earn a pre-penalty mark of 85, their actual mark will be $85 \times (100\% - 36.4\%) = 54$.