ECE-327: Digital Systems Engineering
Lab Manual
Project DFD and Simulation

2017t1 (Winter)

Mark Aagaard

University of Waterloo
Department of Electrical and Computer Engineering

February 27, 2017

Table of Contents

ECE-327	Digital Systems Engineering	2017t1 (Winter)

Project: Kirsch Edge Detecter

1 Overview

The purpose of this project is to implement the Kirsch edge detector algorithm in VHDL. The design process includes exploring a reference model of the algorithm, creating a dataflow diagram, implementing the design in VHDL, optimizing and verifying the design, and finally downloading the optimized design to the FPGA Board. The project is to be done in groups of four.

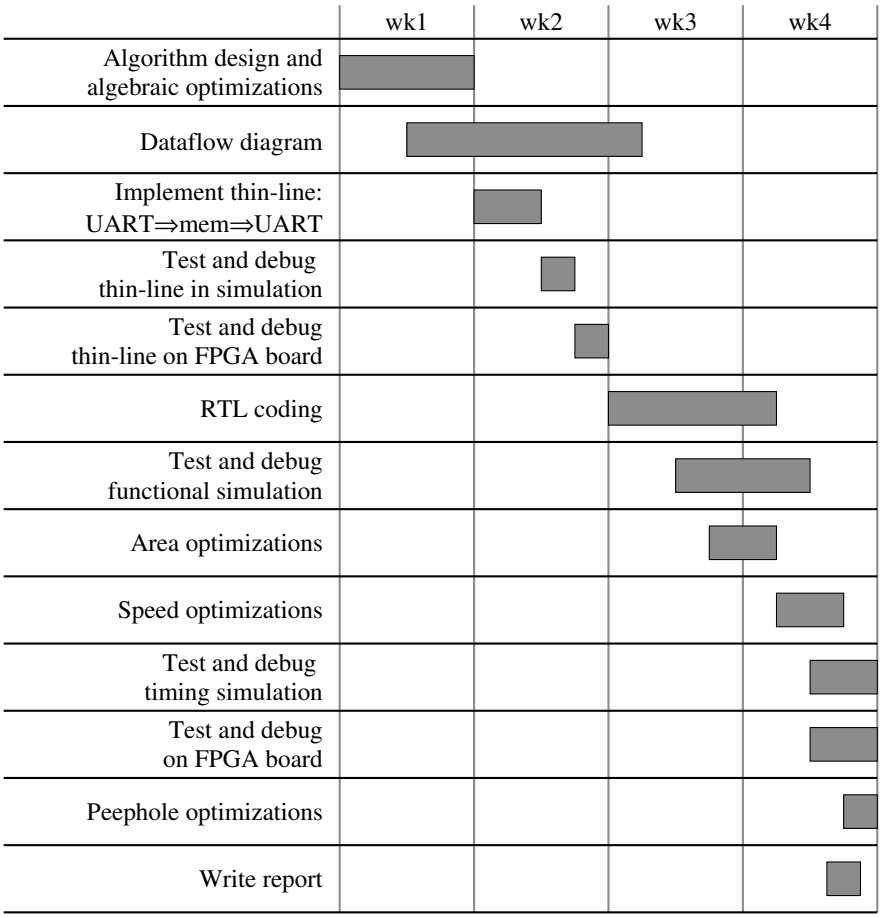

Figure 1: Suggested schedule

There are five deliverables for the project. The list below describes each deliverable, the method of submission, and the approximate due date. The specific due dates are listed on the course web site.

Deliverable	Due Date	Submission Method
Group registration	End of week 2	Coursebook
Dataflow diagram	End of week 2	`ece327-submit-dfd`
Main Project and Report	End of week 4	`ece327-submit-proj`
Demo	1 week after project due date	Coursebook

1.1 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-327 project, we will use the Kirsch edge detector algorithm to detect edges in 8-bit gray scale images of 256×256 pixels. Figure 2 shows an image and the result of the Kirsch edge detector applied to the image.

Figure 2: 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 3 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.

Figure 4 shows a convolution table containing the pixel located at coordinate $[i, j]$ and its eight neighbors. As shown in Figure 3, the table is moved across the image, pixel by pixel. For a 256×256 pixel image, the convolution table will move through 64516 (254×254) different locations. The algorithm in Figure 5 shows how to move the 3×3 convolution table over a 256×256 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.

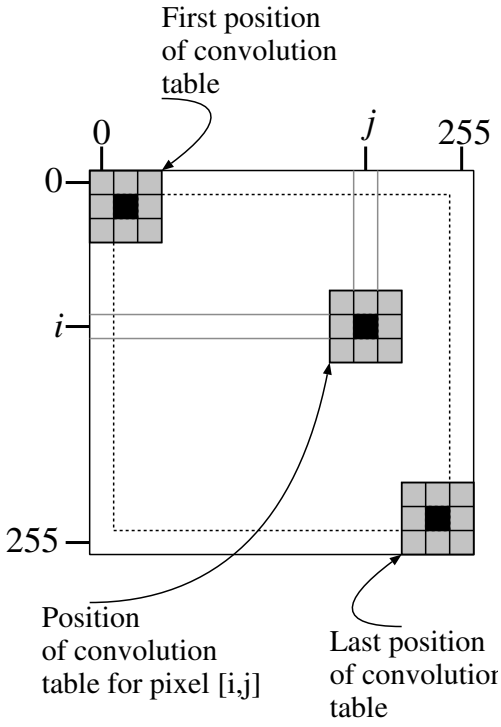

Figure 3: 256×256 image with 3×3 neighborhood of pixels

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

```
for i = 1 to 254 {
  for j = 1 to 254 {
    for m = 0 to 2 {
      for n = 0 to 2 {
        table[m,n] = image[i+m-1, j+n-1];
      }
    }
  }
}
```

Figure 5: Nested loops to move convolution table over image

[0,0]	[0,1]	[0,2]
[1,0]	[1,1]	[1,2]
[2,0]	[2,1]	[2,2]

Figure 6: Coordinates of 3×3 convolution table

a	b	c
h	i	d
g	f	e

Figure 7: Short names for elements of 3×3 convolution table

The Kirsch edge detection algorithm identifies both the presence of an edge and the direction of the edge. There are eight possible directions: North, NorthEast, East, SouthEast, South, SouthWest, West, and NorthWest. Figure 8 shows an image sample for each direction. In the image sample, the edge is drawn in white and direction is shown with a black arrow. Notice that the direction is *perpindicular* to the edge. The trick to remember the direction of the edge is that the direction points to the brighter side of the edge.

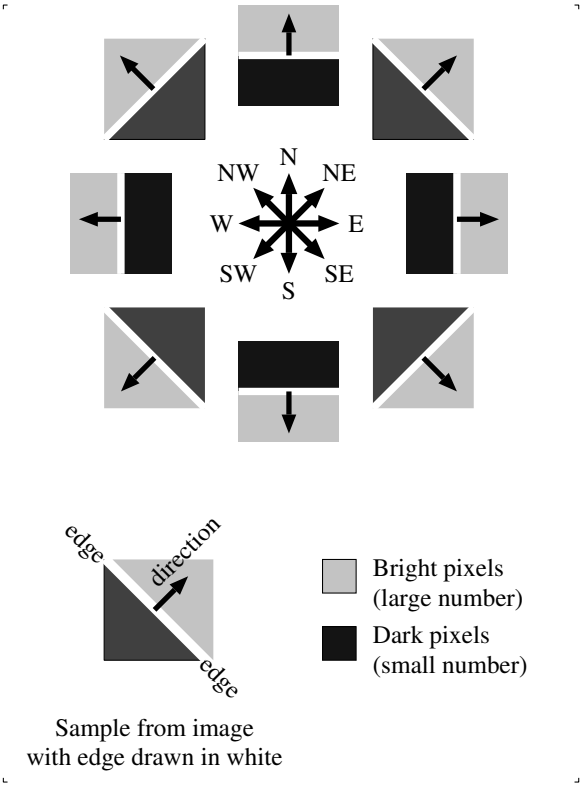

Figure 8: Directions

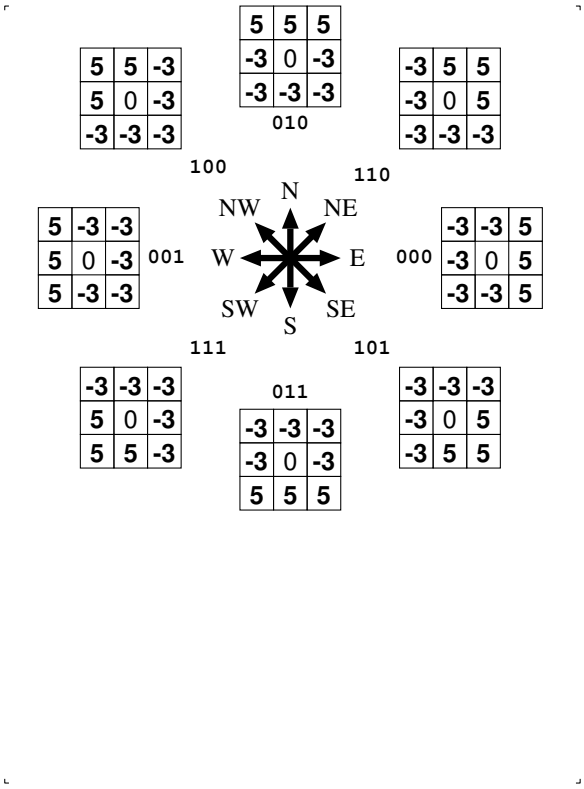

Figure 9: Masks

The equation for each derivative is defined in terms of a *convolution mask*, which is a 3×3 table of constants that are used as coefficients in the equation. Figure 9 shows the convolution mask and encoding for each direction. For example, the equation for the Northeast derivative is given below using coordinates shown in Figure 6:

$$\begin{aligned} Deriv\_NE &= 5\times(\text{table}[0,1]+\text{table}[0,2]+\text{table}[1,2])- \\ &\quad 3\times(\text{table}[0,0]+\text{table}[1,0]+\text{table}[2,0]+\text{table}[2,1]+\text{table}[2,2]) \end{aligned}$$

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

$$\begin{aligned} \text{EdgeMax} &= \text{Maximum of eight derivatives} \\ \text{DirMax} &= \text{Direction of EdgeMax} \end{aligned}$$

Note: *If more than one derivative have the same value, the direction is chosen based on the following order, from highest priority to lowest priority:*
W, NW, N, NE, E, SE, S, SW.
For example, if Deriv_N and Deriv_E are equal, Deriv_N shall be chosen.

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

```
if EdgeMax > 383 then
    Edge = true
    Dir = DirMax
else
    Edge = false
    Dir = 000
```

1.2 System Implementation

Your circuit (`kirsch`), will be included in a top-level circuit (`kirsch_top`) 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 10. The entity for `kirsch` is shown in Figure 11. To reduce the complexity of the project, we have provided you two wrapper files `kirsch_lib.vhd` and `kirsch_top.vhd` that contain the seven-segment display and UART.

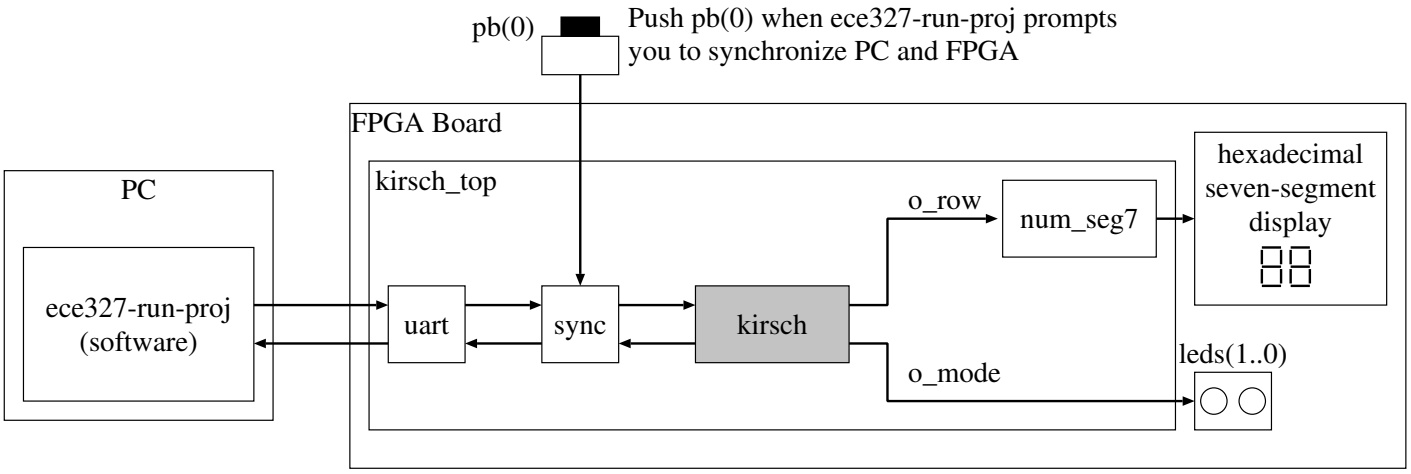

Figure 10: Kirsch system

1.3 Running the Edge Detector

You will run the Kirsch edge detector both in simulation and on the FPGA board. For simulation, you will use the provided testbench (`kirsch_tb.vhd`), which you may modify, or write your own testbench. For running on the FPGA, you will use the program `ece327-run-proj` to send an image to the FPGA board and receive the results.

Four 256×256 images are provided in text format for simulation and bmp format viewingand for download to the FPGA.

The testbench (`kirsch_tb.vhd`) can be used for functional simulation of the reference model and both functional and timing simulation of your design. The testbench reads an image from a text file, passes the data to the Kirsch circuit byte by byte, receives the outputs of the circuit (edges and directions) and stores them in a `.ted` text file.

In the `.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 `ece327-run-proj`. The extension "ted" means "text edge direction".

There are several programs to analyze and debug `ted` files:

```
entity kirsch is
   port (
      clk        : in std_logic;                    -- clock
      reset      : in std_logic;                    -- reset signal
      i_valid    : in std_logic;                    -- is input valid?
      i_pixel    : in unsigned(7 downto 0);         -- 8-bit input
      o_valid    : out std_logic;                   -- is output valid?
      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_mode     : out std_logic_vector(1 downto 0); -- 2-bit output for mode
      o_row      : out unsigned(7 downto 0));       -- row number of the input image
      o_col      : out unsigned(7 downto 0))        -- col number of the input image
   );
end entity;
```

Figure 11: Entity for kirsch edge detector

diff_ted compares two `.ted` files and outputs the lines where they differ.

diff_ted_to_bmp Generates a `.bmp` bitmap image showing the differences between two `.ted` files:

`diff_ted_to_bmp` *file1.ted file2.ted diff.bmp*.

The colors in the resulting image are:

file1 edge	file2 edge	direction	bmp
1	1	same	white
1	1	diff	green
0	0	same	black
0	0	diff	pink
1	0	–	red
0	1	–	blue

ted_to_bmp converts a `.ted` file to a `.bmp` bitmap image. Each pixel in the input image is converted to a 2x2 block of pixels in the output, so that details in ouput image will be easier to see.

txt_to_bmp converts a `.txt` file to a `.bmp` image. NOTE: there might be some bugs in this code.

bmp_to_txt converts a `.bmp` image to a `.txt` file.

1.4 Provided Code

Top-level wrappers	
`kirsch_top.uwp`	Project file with kirsch, uart, num_seg7 for download to FPGA board
`kirsch_top.vhd`	Top level code for for download to FPGA board
`kirsch_lib.vhd`	Source code for uart, ssdc.
Components and Packages	
`mem.vhd`	VHDL code for the memory
`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
Testbenches and simulation scripts	
`kirsch_tb.vhd`	Main testbench
`kirsch_tb.sim`	Simulation script
`kirsch_spec.xls`	Spreadsheet reference model for debugging.
Kirsch core	
`kirsch_spec.uwp`	Project file for Kirsch reference model specification
`kirsch_spec.vhd`	Reference model specification for Kirsch core
`kirsch.uwp`	Project file for your Kirsch core
`kirsch.vhd`	Source code for your Kirsh core
`tests (directory)`	4 test images: `*.txt` input images in text format for simulation `*.bmp` input images in bitmap format for running on FPGA

Note: *Do **not** modify the following files:*
`kirsch_lib.vhd` `kirsch_top.vhd` `mem.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.

Note: *You may use any of the above files in your testbench. You may use `kirsch_synth_pkg.vhd` in your design. The files `string_pkg.vhd` and `kirsch_unsynth_pkg.vhd` are not synthesizable and therefore, your synthesizable code cannot use these files.*

2 Requirements

2.1 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.

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

mode	o_mode
idle	“10”
busy	“11”
reset	“01”

Table 1: System modes and encoding

- Reset mode: If `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 (`reset='0'` on the rising edge of the clock), the 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.

2.2 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.

Your design shall support an unpredictable number of bubbles parcel schedule where the minimum number of bubbles is In general, the rate at which valid data arrives is unpredictable. Your design shall work with any number of bubbles at least 3.

When you run your design on the FPGA, there will be several hundred bubbles between parcels. The large number of bubbles is because the communication with the PC is quite slow compared to the FPGA clock speed.

The maximum latency through the edge detector *without penalty* is 8. Latencies greater than 8 will be penalized, as described in Section 5.3. 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 another words, as soon as the pixel in the third row and third column of the image arrives, the measurement for the latency begins. The number of “bubbles” is completely independent from the latency.

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. If the pixel under consideration is located on an edge, `o_edge` shall be ’1’, otherwise the signal shall 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’.

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’.

At the end of sending an image, at least 20 clock cycles will elapse before the first pixel of the next image will be sent.

Your circuit shall be able to process multiple images without a reset being asserted between the end of one image and the start of the next image.

2.3 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 seven-segment controller in `kirsch_top` architecture displays the value of `o_row` on the seven segment display of the FPGA board.

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. At the end of processing an image, `o_row` shall remain at 255 until reset is asserted or pixels from the next image are received.

2.4 Memory

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 3^{rd} row and 3^{rd} 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. It is also possible to use a 2×256 array.

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

3 rows														...		
	a_0	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_{254}	a_{255}
	b_0													...		
														...		

3. The following shows a snapshot of the memory when row 2 column 2 is ready.

Row Idx																
0	a_0	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_{255}	a_{256}
1	b_0	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_{255}	b_{256}
2	c_0	c_1	c_2											....		

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

a_0	a_1	a_2
b_0	b_1	b_2
c_0	c_1	c_2

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 write and read can be done in parallel.

5. When the next pixel (c_3) arrives, you will perform the operation on the next 3×3 convolution table:

a_1	a_2	a_3
b_1	b_2	b_3
c_1	c_2	c_3

6. When row 2 is full, the next available data shall be overwritten into row 0 column 0. Although physically this is row 0 column 0, virtually it is row 3 column 0. Note that the operations will not proceed until the 3rd element of the 4th row (d_2) 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																
3	d_0	d_1	d_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	...	a_{254}	a_{255}
1	b_0	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_{254}	b_{255}
2	c_0	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_{254}	c_{255}

b_0	b_1	b_2
c_0	c_1	c_2
d_0	d_1	d_2

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

Memory arrays shall be used only for storage of pixels, they shall not be used to mimic computation. In the past, a few groups have tried *unsuccessfuly* to reduce the number of lookup tables in their design by using memory arrays to mimic computation. While it is theoretically possible to use memory arrays for some types of computation, this approach is not relevant to Kirsch edge detection.

3 Design and Optimization Procedure

Follow the suggested schedule in Figure 1.

Begin by copying the files to your local directory. The procedure below shows how this can be done if the desired target directory is `/home/your_userid/ece327/proj`.

```
% cd ~/ece327
% cp -rL /home/ece327/proj  proj
```

3.1 Reference Model Specification

Use the reference model code of Kirsch with the provided testbench to run the four test cases and produce the results. To switch between test cases, set the value of the generic when running a simulation by passing a `-Gtestnum=`i flag to the simulation program.

The suffix of the output `.ted` file is by default "sim" for "simulation. You may change this suffix using the `result_suffix` generic. When simulating the reference model, you should set the suffix to "spec", because `run-ece327-proj` will evaluate the correctness of your results by comparing them against the "spec" file.

To run the command line version of `uw-sim` without opening the graphical interface, use the `--nogui` flag.

```
% uw-sim --nogui -Gtest_num=1,result_suffix=spec kirsch_spec.uwp
```

You will use the resulting images to verify the functionality of your VHDL code. The reference model specification uses the same entity as your implementation. The reference model is not synthesizable.

3.2 Equations and Pseudocode

The Kirsch edge-detection algorithm was designed to be implemented simply and efficiently in hardware. You can do significant area optimizations very early in the design cycle by finding efficient ways to calculate the derivative equations. For example, identify algebraic optimizations and common subexpressions in the equations.

Several useful equations to apply in the optimizations are:

$$\begin{aligned} 5a - 3b &= 8a - 3(a+b) \\ \max(a-c, b-c) &= \max(a,b) - c \\ \max(a+b, b+c) &= b + \max(a,c) \end{aligned}$$

3.3 Dataflow Diagram

The dataflow diagram shall show the calculation for the presence of an edge. The input to the dataflow diagram shall be the 3×3 window of pixels for the current position. The output shall be `o_edge`.

The pixels in the 3×3 window shall be labeled as:

a	b	c
h	i	d
g	f	e

The dataflow diagram does not need to include the calculation of `o_dir`. The dataflow diagram does not need to include writing to or reading from memory, updating the row and column counters, or the state machine.

In addition to the standard VHDL arithmetic operations, the dataflow diagram may use a MAX datapath component. For your dataflow diagram, you do not need to show the implementation of your MAX component.

In your VHDL code, you will need to implement your own MAX component, function, or procedure, because VHDL does not include a MAX operator. One complication to take into account is that your edge detector must compute both `o_edge` and `o_dir`, which means that your MAX will need to propagate *two* different values (a derivative and a direction). A function is probably the most concise option. But, a function may have only one result, so you will need to put the derivative and direction together into a record. A procedure may have multiple outputs and is slightly more concise than a component. A component does not require learning any new VHDL, but is the most verbose option.

The suggested process to go from your pseudocode to a dataflow diagram is:

1. Draw a data dependency graph.
2. Predict the slowest operation (e.g. memory access, 8-bit subtract, etc).
3. Estimate the maximum clock speed that can be achieved if a clock cycle contains just the slowest operation. This gives you an upper bound on the clock speed.
4. Based upon your maximum clock speed dataflow diagram, estimate the total latency of your system, including updating the memory arrays and convolution table. Compare your latency against the latency goal in Section 2.2. If your latency is excessive, reschedule the operations in your dataflow diagram, and potentially increase your clock period.
5. Choose your optimality goal. From your maximum clock speed and optimality target, calculate the maximum area that you can use.
6. Pick your optimality strategy: high-performance or low-area, and then pick your area and clock speed targets.
7. Based upon the throughput requirement and your dataflow diagram, decompose your design into pipeline stages.
8. Estimate the datapath resources and registers that your dataflow diagram uses.
9. Estimate the total area of your system by including allowances for the circuitry to update memory, the convolution table, and the control circuitry. (For most datapaths, an efficient implementation of the circuitry for control and memory can be done in about 160 FPGA cells.)
10. Estimate optimality of entire system.

> **Note:** *A week of all-night panicked debugging can save a day of thoughtful data-flow-diagram designing.*

3.4 Implementation

Begin with a thin-line implementation that simply loads data into memory, reads data from memory, and loads the data into the convolution table.

Do not begin coding the main part of your design until you are satisfied with your dataflow diagram and have the thin-line implementation working on the FPGA board.

Your VHDL code should be based on a dataflow diagram: either the one that you submitted for the first deliverable, or an improved diagram.

Use the memory component provided in `mem.vhd`.

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

3.5 Functional Simulation

To simulate your design, use the following command, with the number of the test that you want to run:

```
% uw-sim  -Gtest_num=3 kirsch.uwp
```

Use `diff_ted` or `diff_ted_to_bmp` to compare your resulting `.ted` file to the file produced by the reference model. For a line that is different in the two files, `diff_ted` uses $\star$ to show which fields are different.

```
% diff_ted tests/test3_spec.ted tests/test3_sim.ted
```

Use `ted_to_bmp` to convert a `.ted` file to a `.bmp` file and visually compare two `.bmp` files. The visual comparison can be very useful in detecting patterns in bugs (e.g. images are shifted or skewed).

When debugging your design, 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. To make it easy to change the size of the image that the circuit is using, your design should use the constants `image_height` and `image_width`, which are defined in `kirsch_synth_pkg.vhd`, rather than hardcoded values for the image size.

- Some groups made good use of Python and other scripting languages to generate tests and evaluate the correctness of results.
- The spreadsheet `kirsch_spec.xls` is designed for you to modify so that you can check the internal calculations of your design.
- The testbench contains a constant named "**bubbles**" that determines the number of clock cycles of invalid data (bubbles) between valid data. Make sure that your design works with any value of bubbles that is 3 or greater.

3.6 High-level Optimizations

- 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.
- Do area optimizations to reduce your area to your target before worrying much about clock speed. Area optimizations involve larger changes to a design than timing optimizations.
- Too much reuse of components can increase the number of multiplexers and amount of control circuitry.
- Estimate if your design has more flip-flops or lookup tables. If your area is dominated by lookup tables, there is no point in trying to save area by reducing the number of flip-flops. Conversely, if you area is dominated by flip-flops, then do not optimize the lookup tables.
- 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.
- For clock speed optimizations look at the 3–5 slowest paths listed in the timing report. Your slowest paths might be different from what you predicted from your dataflow diagram. Try to understand the cause of the difference (e.g., control circuitry), then look for simplifications and optimizations that will decrease the delay. Retiming and state-encoding are two techniques that can often be used to decrease delay.
- 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. At this point, it is usually wise to transition to peephole optimizations.

3.7 Logic Simulation

Synthesize your design and then perform logic simulation using the following commands:

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

> **Note:** *For logic simulation, your kirsch core needs to drive all of the output signals in the* `kirsch` *entity. If you do not drive an output, Precision RTL will optimize it away and the output port will not appear in* `uw_tmp/kirsch_logic.vhd`*. The missing port will cause logic simulation to fail.*

- Logic simulation can be very time consuming due to the large number of pixels to be processed. 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)
- If you have bugs in logic simulation, read the handout on debugging in timing simulation.

3.8 Peephole Optimizations

- Minimize the number of flip-flops that are reset. For example, usually there is no need to reset datapath registers.
- Use the minimum width of data that is necessary to prevent overflow. Related to this, avoid using the type `signed` unless you need negative numbers.
- 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.
- You are probably wise to stop your optimizations at the point where your optimizations are improving the optimality by 2–3%.

4 Deliverables

4.1 Group Registration

Register your group on Coursebook.

4.2 Dataflow Diagram

See Section 3.3 for information on the input and output, design process, and analysis for your dataflow diagram.

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 the dataflow diagram to arrive at your final design.

The dataflow diagram shall list the resources used (adders, subtracters, comparators, and registers). The dataflow diagram shall state the latency and throughput. The dataflow diagram shall include estimates for the total area, clock period, and optimality, with *brief* explanations of and/or equations for how the calculations were done.

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. The report *must* be a PDF file. Hand-drawn images are fine. Marks will be deducted for images that are poor quality, blurry, too-dark, *etc.*.

Submit with `ece327-submit-dfd` *`dfd-name.pdf`*

4.3 Design Report

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

> **Note:** *We will only mark the first 3 pages!*

Your audience is familiar with the project description document and the content of ECE-327. 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 included as a PDF file named `report.pdf` in the project directory.

> **Note:** *Your report* must *be in PDF and **must** be named* `report.pdf`

> **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.*

The report should address design goals and strategy, performance and optimization, and validation.

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.

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 (look in the area report files), the maximum clock speed of the design, the critical path (look in the timing report files), latency, and throughput of your design.

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

4.4 Implementation

You may 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**. To submit your design, enter the following command **from the directory containing your design**:

```
% ece327-submit-proj
```

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.

When your submission is sent, a simple "Dead or Alive" test will be run 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.

Do not rely on the "Dead or Alive" test as the only testing mechanism for your design. The "Dead or Alive" test simply runs the testbench for 1 ms and checks whether `o_valid='1'` happens.

4.5 Demo

Your group will demonstrate the design on the FPGA board to a TA or an instructor and will answer questions about the design and verification process. The demonstration will last approximately 30 minutes. It is important for everyone in the group to be present and to participate.

At the demo, we will give you instructions for how to copy the .sof file that we synthesized from your project submission. Then, you will demo several example images and answer questions about your project.

You'll be marked on quality, clarity, and conciseness of answers; and on group participation. Don't memorize prepared answers! If it appears that a group is just regurgitating prepared responses, we will change the questions in the middle of the demo.

5 Marking

5.1 Functional Testing

Designs will be tested both in a demo on the FPGA 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 FPGA board as well as with `uw-sim --chip`.

A *Functionality Score* will be used to quantify the correctness of your design. This score will range from 0 (no functionality) to 1000 (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
1000	we will first try timing simulation (`uw-sim --timing`) of the post-place-and-route design (`kirsch_chip.vho`).
800	if that fails, we'll try the post-place-and-route design (`kirsch_chip.vho`) with zero-delay simulation (`uw-sim --chip`)
600	if that fails, we'll try the design prior to place and route (`kirsch.vhd`)

5.2 Optimality Testing

Optimality will be evaluated based on *logic synthesis* results for a Stratix FPGA for area and clock speed, and behavioural simulation for latency.

5.3 Performance and Optimality Calculation

The optimality of each design will be evaluated based on functionality score, performance, and area (FPGA cell count). Performance will be measured by clock frequency in megahertz as reported by `uw-synth --opt`. For area, we count the number of FPGA cells that are used. This is 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. Systems with an excessive latency (more than 8 clock cycles) will be penalized as shown below:

If $Latency \leq 8$ then

$$Optimality = Functionality \times \frac{ClockSpeed}{Area} \tag{1}$$

If $Latency > 8$ then

$$Optimality = Functionality \times \frac{ClockSpeed}{Area} \times e^{\frac{-(Latency-8)}{20}} \tag{2}$$

5.4 Marking Scheme

Dataflow diagram •Clarity of drawing •Optimality of design	5%
Submission •Correct signal, entity, file, and directory names •Correct I/O protocol	5%
Optimality (see Equation 1) •High speed, small area, functionality	45%
Design Report •Clearness, completeness, conciseness, analysis	15%
Demo Test image functionality 15% Discussion about your design and design process 15%	30%

5.5 Late Penalties

For the dataflow diagram, late submissions will receive a mark of 0 for the dataflow diagram.

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 0.1% per hour. Projects submitted at the extended deadline will receive a multiplicative penalty of 16.8%. Projects submitted after the extended deadline will be penalized with a multiplicative penalty of 16.8% 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 2 days and 14 hours after the *regular* deadline will receive a penalty of: $(2 \times 24 + 14) \times 0.1 = 6.2\%$. Thus, if they earn a pre-penalty mark of 85, their actual mark will be $85 \times (100\% - 6.2\%) = 80$.

Example: a group that submits their design 1 day 4 hours after the *extended* deadline will receive a penalty of: $16.8\% + (1 \times 24 + 4) \times 1\% = 44.8\%$. Thus, if they earn a pre-penalty mark of 85, their actual mark will be $85 \times (100\% - 44.8\%) = 47$.