ECE532 Digital Systems Design

Winter Semester 2016

The Turret Master 5000

INDIVIDUAL REPORT

April 8, 2016

Submitted by

Emily Ng

University of Toronto

Acknowledgements

First off, I'd like to thank my team. Everyone did a great job and I could not have asked for better teammates.

In this project I used a lot of the skills and knowledge I gained during my PEY at Altera. I'd like to thank my team at work for teaching me about FPGA CAD flow, timing analysis, working with simulation, general debugging and above all being very patient and willing to explain things to me.

I would like to thank Professor Chow for organizing this course in such a way that it encourages us to explore what's out there, build something that we're proud of, and learn a lot. Finally, I would like to thank Charles for his continual support and advice.

Contents

List of Figures

Section 1

What I did

Responsibilities

Overview

> I was tasked with selecting an algorithm to do what we needed for our project, prototyping it on software and implementing it in hardware. I allocated one week for research, one week for prototyping, two weeks for building and two weeks for integration, which left one week buffer.
>
> Secondly, as I was prototyping in software, which in itself was a challenge, I was supposed to consider the feasibility of hardware implementation. At the time, I thought that meant to avoid recursion and graph traversal. I hadn't really considered memory accesses and the challenge they would present as I moved to hardware implementation.

Our aim was to identify objects and be able to recognize friends and foes. The objects were to be on a plain monochrome background and non overlapping. This task was broken down into the following smaller tasks: (1) locate non-background objects (2) perform a binary classification of friend or foe.

This was performed in the following steps:

- grayscale
- edge detection
- flooding
- connected components analysis
- data extraction

The full top level diagram showing all these components is available in Figure B.2 in Appendix B.

Grayscale

Grayscale was calculated using the ITU-R Rec. 601

$$I = 0.299R + 0.587G + 0.114B$$

Sobel

Image Filter

Image filters can produce interesting effects, such as blur, sharpen, emboss, or outline. This diverse range of effects is achieved with a very simple and compact computation. It is performed by convolving an *image kernel* with an input image to produce an output image. The kernel is a small matrix that defines the resultant pixel as a weighted summation of the pixels in the window surrounding it. The output is computed by convolving at each input pixel a window of the input with the kernel.

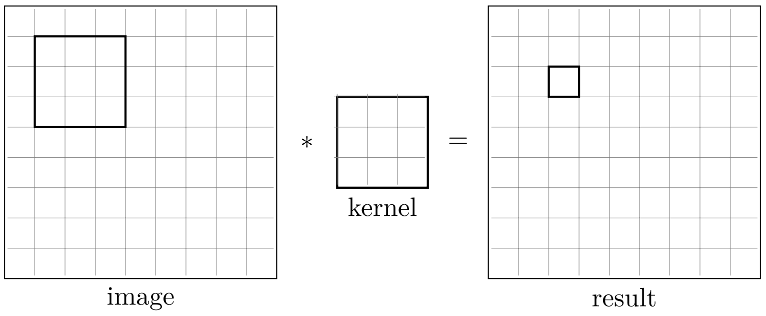


Figure 1.1: Image kernel convolution.

The Sobel operator is an image filter designed to extract edges by computing the derivative at each pixel. Gradients correlate well with edges because an even surface contains pixels that are very similar to each other whereas pixels along edges show a high level of difference, or gradient, along the direction of the edge.

$$G_y = \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ +3 & +10 & +3 \end{bmatrix} * A \tag{1.1a}$$

$$G_x = \begin{bmatrix} -3 & 0 & +3 \\ -10 & 0 & +10 \\ -3 & 0 & +3 \end{bmatrix} * A \tag{1.1b}$$

$$G = |G_x| + |G_y| \tag{1.1c}$$

$$q = \begin{cases} 1 & G \geq \text{threshold} \\ 0 & G < \text{threshold} \end{cases} \tag{1.1d}$$

Performing the computation requires access to a 3 x 3 neighbourhood of pixels. This requires two line buffers to access the pixels from the previous two rows, shown in Figure 1.2. Once two rows have been filled up, the output can be computed at one clock cycle per pixel.

Flooding

When the edge detection was tested in hardware, two main issues arose. First, there were many small speckles of noise. Second, the outline appeared grainy, like in a sketch where some of the paper still shows through a penciled outline. A filter was applied that takes a population count of the window, by applying an image kernel of all one's. Then, if this count

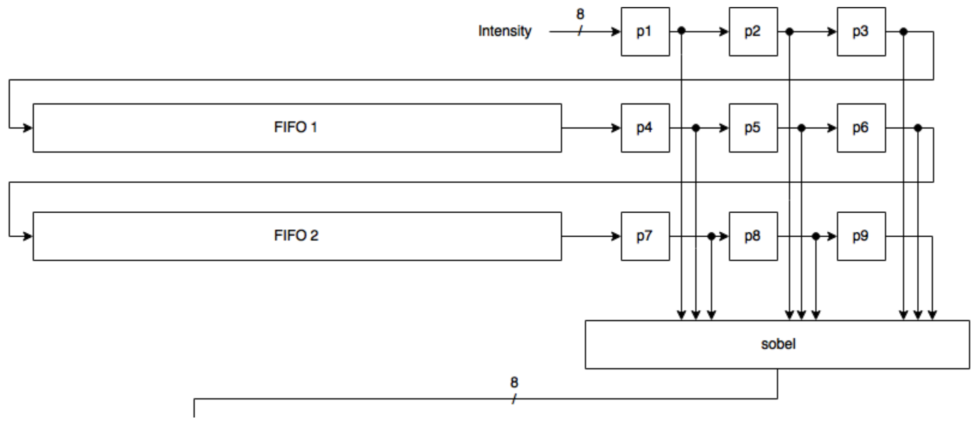


Figure 1.2: Generating the pixel array for Sobel computation.

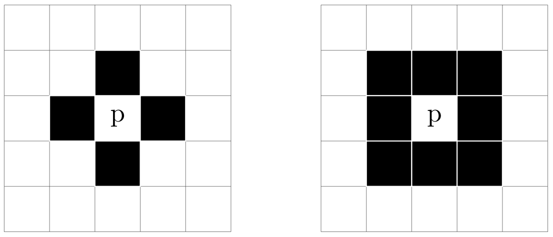
meets a threshold the input pixel is accepted as part of the outline, otherwise it is rejected and marked part of the background.

$$\text{count} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} * A \tag{1.2a}$$

$$q = \begin{cases} 1 & \text{count} \geq \text{threshold} \\ 0 & \text{count} < \text{threshold} \end{cases} \tag{1.2b}$$

Connected Components Analysis

Connected components analysis is a method to extract information about objects in a *binarized* image. *Connectivity*, illustrated in Figure 1.3, is defined as pixels that (1) share edges or (2) share edges or vertices. A *binarized* image is an image where there are only two pixel values, which represent background or non-background. This method was used to locate the different objects in view.



(a) 4-connectivity (b) 8-connectivity

Figure 1.3: 4- and 8- connectivity

The analysis works by scanning the image in raster order, scanning each row from left right from the top to the bottom of the image. At the time that pixel p is processed, the pixels before it have already been labeled, as shown in Figure 1.4. The label for p is selected based on the labels A through D.

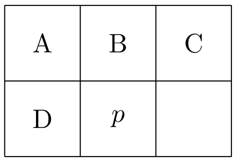


Figure 1.4: Neighbourhood of input pixel p. Labels A through D are the labels of the pixels that have already been processed through connected components analysis and p is an input pixel from the binarized image.

There are several possible scenarios:

1. p is a background pixel.

   It is not given a label.

2. p is not a background pixel and A through D are background pixels.

   It is the start of a new object, so it gets a new label.

3. p is not a background pixel, and among A through D there is a single label.

   It is part of this object, it copies that label.

4. p is not a background pixel, and among A through D there are two or more labels.

   These labels need to be merged. p is assigned the smallest of these labels.

Scanning in this fashion, there comes a point where two regions may need to be merged. Consider a 'U' shape. The two arms of the 'U' will be labeled separately, and it is not until the bottom right of the 'U' that the two labels appear in the same neighbourhood. At this point, the data for the two objects can be merged. This is valid since the entries are just a summation of data points for each pixel considered to be part of the object.

Bailey and Johnston [1] illustrate a single pass connected components analysis implementation on an FPGA. This approach was followed with some simplifications. The main components are shown in Figure 1.5. There is a label selection module, a merge table, and a data table.

The first two stages select the label for the current pixel. The first stage is label selection. This block is all combinational logic. The next stage reads from the merge table. It looks up the selected label in the merge table which is in BRAM to see if this pixel really belongs to another label. At the end of this stage, we know the correct label for this pixel.

The next two stages update the data for this label. The features being extracting from the objects are its moments, up to the third order. In the third stage, the data for the resolved label is read out of the table, updated and in the fourth stage written back.

Feature Extraction

In the one pass method described by Bailey and Johnston [1], data is extracted on the fly as the objects are being labeled.

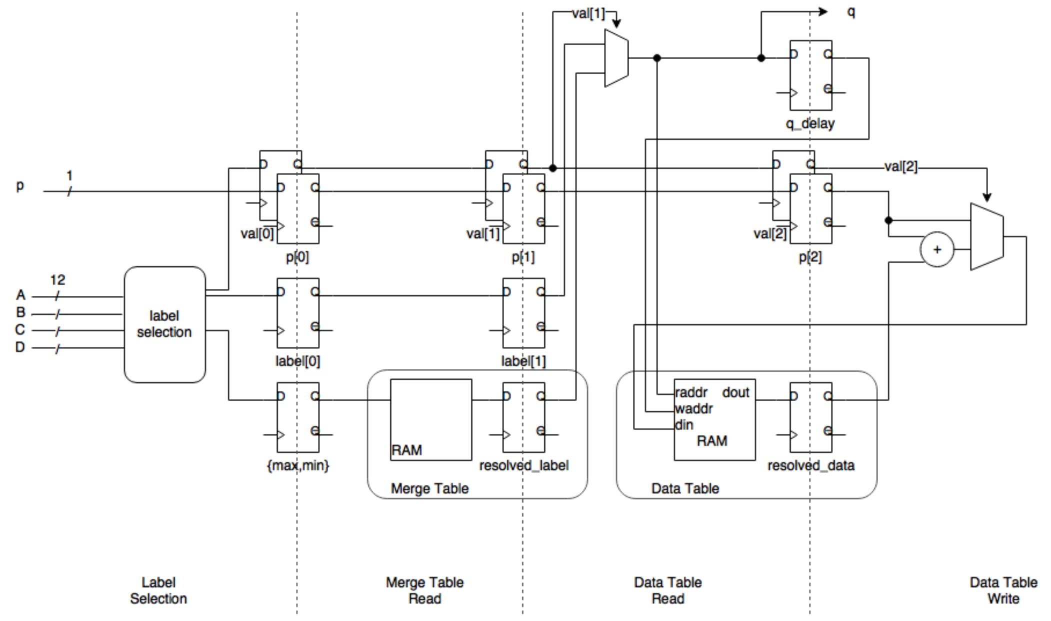


Figure 1.5: Connected components pipeline.

Location

Locations of objects were found by finding the centre of mass of the outlines. As the objects were being labeled, their moments up to the third order were accumulated in a data table.

The locations were then extracted using Equations (1.3). Due to the nature of division in hardware, this was left to the Microblaze. When it was attempted in hardware, it resulted in an astounding 75 ns delay on a single path.

$$area = m_{00} \tag{1.3a}$$

$$\bar{x} = \frac{m_{10}}{m_{00}} \tag{1.3b}$$

$$\bar{y} = \frac{m_{01}}{m_{00}} \tag{1.3c}$$

Higher order moments

Higher order moments were extracted for use to characterize each object. The characterization method was only prototyped in software. Unfortunately, delays in the project schedule did not allow for this to be fully implemented in hardware. For a discussion of the theory and software results, see Appendix C.

The multiplications themselves also required effort to close timing. The highest order moment m_{30}, involves raising a number to the power of three, where the maximum input value is the frame width, 1280. That means, at worse case, the computation is $1280^3 \approx 2x10^9$. In order to break it down into smaller computations, it was pipelined so that the the next order moment was calculated at each stage using the result from the previous stage. For a block diagram, see Figure B.1 in Appendix B. It takes three cycles, which conveniently matches the number of cycles until the data write stage in the connected components pipeline.

Debug Features

In order to facilitate debugging, each stage of image processing can be displayed to video output. In particular, the connected components labels are coloured in a colour-by-numbers fashion so the different outlines can be easily observed. Internally, they are labeled $1, 2, 3, \ldots$, which would appear as various shades of grey if displayed directly.

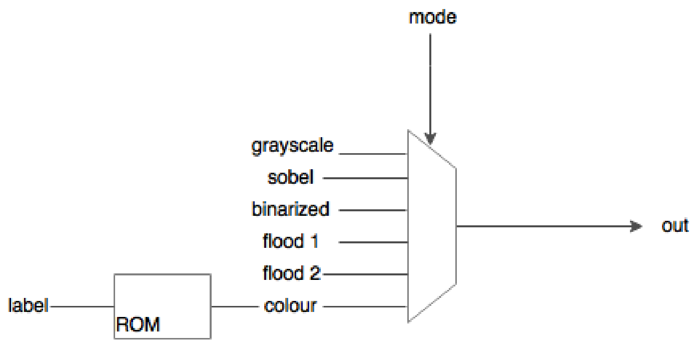


Figure 1.6: Output modes

Tools

Pen and Paper

My most invaluable possession relating to this project is my notebook. In this notebook I kept my milestone reports, meeting notes, notes on digital logic, notes on image processing, debugging notes, block diagrams, schematics and thoughts. I frequently referred to and annotated older notes. Overall it helped me organize and develop my thoughts. Select pages from my notebook can be found in Appendix E.

Verilog

Throughout this project, I learned to use more features of Verilog. This is by far the largest Verilog project I have taken on and I need to keep it modular and easy to work with. I used parameters and modules to organize my design hierarchically. I used macros in a header file to easily adjust bus widths and access debug features. Eventually, I wanted my design to compile differently depending on if I was in simulation or compiling for hardware. Here I used Modelsim's interface for defining macros at compile time.

I hit a limitation with Modelsim ASE. It doesn't support the use of SystemVerilog asserts, which was unfortunate as it seemed like a good way to introduce unit testing in my modules.

Vivado

RTL Viewer

The first challenge in Verilog is to remember that it is a hardware description language. It is very easy to fall into procedural coding habits. I often drew out what I thought my design looked like and compared it to what I saw in the RTL viewer.

Timing reports

I frequently checked the timing reports to get a sense of where the critical paths were. This allowed me to insert registers and incorporate pipelining stages as required.

Modelsim

Testing

After each change I used Modelsim to verify that my design was working. Overall, this testing gave me a high degree of confidence that my design would work in hardware. My tests ran very quickly on small (550 x 300) bitmaps so my main design iteration time was actually comparable to working with software.

Debugging

Modelsim was my main debug tool. I used it extensively when things weren't working. Learning to work with the various features, such as saving wave views or using the jump to next change buttons really sped debugging along.

Memory dump

Some components of my design in involved keeping data in tables. Sometimes the contents of these tables were the most useful information for debugging. I used Modelsim to dump the contents to text files to browse through the contents easily. This was especially helpful at times when the waveforms were unnecessary.

Autossh

Using autossh, I never dropped an ssh tunnel connection (as opposed to my team mates using Putty). The only times I lost a connection was when the remote went down. Dropped connections became a great source of annoyance to my teammates. This was important to me because it was one more source of workflow friction that I avoided easily.

Design Flow

Testbench

I designed my testbench to read an image from disk into memory, feed it as a pixel stream to the IP core, write the output stream to memory and finally write that to disk, shown in

Figure 1.7. At the time I wasn't aware of this, but it's quite similar to how we ended up working with frame buffers and VDMAs. To do this I had to choose a file format and be able to read and write it. I chose bitmap for its simplicity. I wanted to do this so I could look at what the IP was doing without needing to run in hardware. This was crucial to being able to work with confidence before we had the hardware setup with HDMI and also to being able to work with a reasonable design cycle time.

As the core evolved to do more things beyond edge detection, I enhanced my testbench to follow. For connected components, I had the testbench color each label a colour rather than just the label, which would have been a subtle grayscale. When I had centroids of objects, I got my testbench to draw dots in the image by manipulating the "frame buffer" contents.

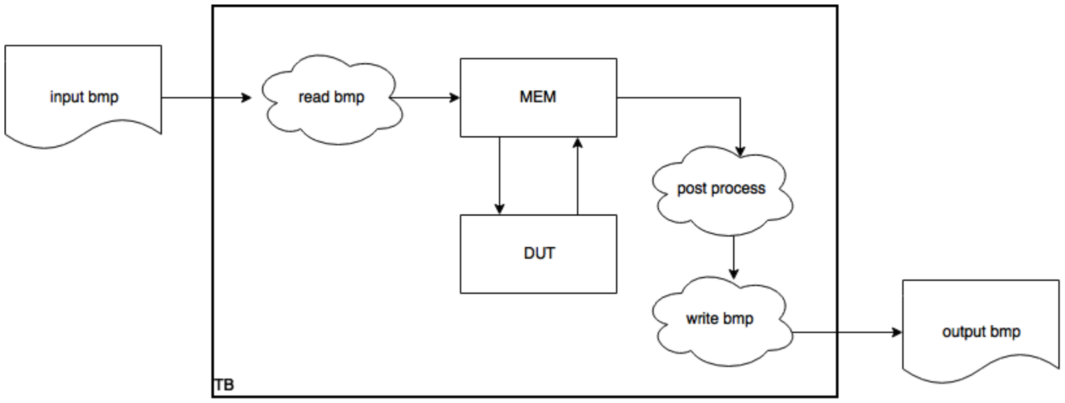


Figure 1.7: Testbench that feeds a bitmap to the core and writes the output to a bitmap.

In my day-to-day flow, I first make some changes to my Verilog. Then I simulate it using Modelsim RTL sim to verify its correctness. I then compile it through Vivado to ensure there are no errors. This scripts I used for this flow is available in Appendix G.

Although I didn't use later stages of simulation extensively, I used it at times when our design wasn't working in hardware. A description of how to set this up is available in Appendix G.

Version Control

I kept my Verilog sources in version control and with compile scripts for generating and compiling my project.

Challenges

Software/Hardware Paradigms

The biggest challenge I faced was the shift in mindset from software to hardware. This is the largest hardware project I have ever taken on, and although I would not consider myself to be in expert in either, I have had much more time to develop a mental model for writing imperative, procedural software.

I've come to realize that there are two assumptions that I make every day that I don't even think about when writing software:

1. Sequential semantics

2. Uniform latency across all operations

That is, each line that I write is executed in order and each result is available on the next line. As I write my Verilog, I need to always think instead about what circuit I am describing. I need to consider that combinational logic happens all at once, in parallel. I need to consider that results that pass through registers are not available until the following clock cycle, and I may need to line things up.

In reality CPUs have all the same limitations except that they are all hidden behind the ISA. This contract allows the programmer to safely write software under these assumptions while the CPU architecture is free to finely schedule these operations of various latencies.

The best tool that I had here was drawing out diagrams before writing my Verilog and checking the schematic in Vivado after writing my Verilog. By consciously making this effort, it became not hard in not too long to figure out how to implement most of my software in hardware.

Timing Closure and Pipelining

As I started designing our video processing IP, I soon ran into timing violations, to which the answer was always to register and pipeline. It took a considerable amount of time that I didn't account for in the original project timeline to deal with timing closure.

I initially wrestled with how to insert systematically pipeline my design and how to think about the data flow and dependencies in my design. I quickly developed a habit of drawing pipeline diagrams. This experience further developed my understanding of sequential and combinational logic, as well as timing logic to line up in a given cycle.

Noise

The real environment proved to be much more challenging than I had anticipated. As a team, we scoped out our project such that we would control the environment by having a plain monochrome background with some crisp images projected onto it. Unfortunately, lighting and camera quality made what we had hoped to be an "easy" environment quite noisy.

It was especially hard because as we tried to deal with the noise, we would break up the outlines of our objects. Our design depends on the objects that we are detecting to show up as a connected outline. If it broke up into two, it would be detected as two objects. However, if there was too much noise it would end up connected two objects that are in fact separate. This lead to a constant trade-off between rejecting noise and not being so aggressive as to break up the outlines.

Section 2

Reflection

Team

Our team is fairly diverse within ECE. Together we have significant background (for undergrad students) in circuit design, embedded software, FPGA CAD flow and good project workflow practices. Most of us had also worked together closely before, so we already had a good sense of each others strengths and weaknesses. Overall, we were able to play to each of our strengths in this project.

Michael has a strong background in circuits and power electronics. He also has experience working with PLLs at the physical design level.

Patrick has a strong background in embedded software. In addition, he has experience working on timing modeling in FPGAs.

Roberto also has a strong background in embedded software and is overall very well rounded in computer engineering.

We naturally divided up into two groups: Michael and I on the custom IP and circuitry, and Roberto and Patrick on system level design and embedded software. Across these two groups, we correspondingly split into two pairs for design and control of the custom IP. Patrick worked with Michael and being able to control the laser and Roberto worked with me to interface with the object detection IP. This division allowed us to work pretty well individually for a few weeks and perform integration smoothly.

Testing in the real environment

My greatest oversight in this project was testing both the software prototype and hardware simulation on clean images only, and not camera images.

Regression testing

One thing I would have done differently is set up regression tests. Ideally, I would have a set of tests run nightly and automatic pass/fail reporting with logs. This set of tests should cover RTL, post-syn, post-impl functional tests, timing closure and utilization tests. The way

I would implement the automatic functional tests is with a golden test vector, where I store the correct output and perform a binary file comparison on the test run output. Timing and utilization could be easily reported automatically by grepping the Vivado reports. Since I already had everything running from the command line and a remote server running Vivado and Modelsim, it would not have been hard to extend to automatic testing.

One of the reasons I didn't do this is because I didn't think of the result as a pass/fail. I always looked at the output image and never considered a simple file comparison. I felt like I had enough testing by running tests when I made a change, but in retrospect there were situations where nightly tests would have reported to me discrepancies that I didn't notice until later. In particular, I always tested on smaller images because a full 720p image takes almost ten minutes to run in RTL sim. Had I ran tests on full resolution images nightly, I would have caught that before going to hardware.

Approach to image processing

At the beginning of this project, I had no idea about anything related to computer vision. Since the focus of the course is hardware design, I didn't want to spend too long researching and prototyping. I learned most of what I needed from OpenCV tutorials and following the topics introduced in further depth. I was able to prototype ideas very quickly using the OpenCV library [3]. While I was doing my research, I tried to stick with algorithms that were easy to understand and that were hardware friendly.

Charles introduced Viola Jones object detection in a little more detail than discussed in lecture. I dismissed it as being too complicated because of the artificial intelligence component. I did not think I would be able to understand and implement and then train a Viola Jones detection module in the timeframe of this project, primarily due to my lack of experience with training a neural network. In retrospect, the approach I chose was not all that easy to implement in hardware because of the many memory accesses required and I'm sure I would have gotten the help I needed to get Viola Jones. Still, I went with the more classical methods of computer vision where I know exactly what is being calculated for each pixel.

At the end of this process, I learned just as much about computer vision as I did hardware design. I also learned that new and different, possibly better ways of doing things are always on the horizon and it's best to keep an open mind. As Jason Foster would say to us in first year, keep strong opinions, weakly held. Such as it is that neural nets are taking over in computer vision, high level synthesis is on the rise in the digital design world. I would say that I learned a lot here on extracting image features and designing at the RTL level, but I wouldn't be opposed to continuing to pursue these topics in slightly different avenues.

Section 3

Community Contribution

I answered any questions on Piazza where I felt knowledgeable enough to answer.

Section 4

Feedback to Xilinx

Vivado

Coming from Quartus, Vivado was very easy to use. I could easily find all the equivalent buttons in the gui and equivalent tcl script interfaces. Overall, the gui is much better looking. I particularly liked the IP integrator layout which provided an intuitive way of adding IPs to a block design.

One missing feature that I would have enjoyed is virtual IOs. I had some projects instantiate inner cores of our design, which had more ports than available user pins. This is a common use case and I found others who expressed their interest in this feature on Xilinx forums.

SDK

The SDK frequently crashes when trying to re-run our program, ofter requiring us to restart the SDK and/or re-program the board.

Section 5

Course feedback

Being in Engineering Science, I had the choice between Digital Systems Design, Energy Systems capstone, and Electrical and Computing capstone to fulfill my degree requirements. I chose this course because I wanted to gain more experience working with FPGAs and from what I knew about the course, I expected it to be more technical and more structured than the capstone courses.

Now that I've completed the course, I'm really glad I did. I learned a lot. From what I gather, the expectations are much higher here (that we have a certain level of difficulty, certain number of components) and we are expected to always have something working. I think these expectations pushed us to expect more from ourselves and the success of our demo is the most rewarding thing that came out of it. My experience here goes beyond getting a grade. I've taken away skills and confidence.

Pre-project phase

This part of the course really set the expectation for the amount of work required weekly. We especially took it seriously since in total the pre-project phase was worth 35% of the course. The tasks were simple, and the focus was clearly on learning the tools and learning to work as a team. It was pretty helpful to have a non-project related warm up.

Lectures

I attended lecture up until reading week. The best piece of advice was given very early: if you want to learn hardware design, find someone to work under. It was good to become aware of topics that I hadn't heard of before. Many topics require a significant theoretical background to really appreciate (e.g. the background on on-chip-termination is given in a part two electromagnetism course at UofT), so in some cases I didn't have the mental framework to place this information.

Milestone reports

I'm really a fan of the weekly reports and the fact that they're due before the lab session. I like the structure, and how it forces you to consider whether or not you've hit this week's milestone and consider also the aim for the coming week. They also gave Charles a chance to look over our progress and challenges before meeting in the lab which made the lab session more valuable.

My milestone reports are available in Appendix F.

TAs

It was really great to have very knowledgeable and helpful TAs. We (my team) really appreciate the help and advice we got and especially appreciate the extra office hours and correspondences outside lab hours. I've heard in the capstone course that groups have to constantly remind TAs what their project is about. I'm glad that wasn't the case for us. Charles kept up with our goals, progress, and challenges throughout the semester. In this sort of project, it really matters that our supervisor cares about project and our success. It says a lot students.

Appendix A

IP Documentation

Register list

Table A.1: Table of AXI registers. ‘w’ indicates read and write register, ‘r’ indicates read only register.

address	r/w	description
5’h00	r	obj_m11
5’h01	r	obj_m12
5’h02	r	obj_m21
5’h03	r	rx_read_pointer
5’h04	r	tx_write_pointer
5’h05	r	tx_read_pointer
5’h06	r	rx_fifo_track
5’h07	r	tx_fifo_track
5’h08	r	mm2s_tready
5’h09	r	mm2s_tvalid
5’h0A	r	s2mm_tvalid
5’h0B	r	s2mm_tready
5’h0C	w	ctrl_reg1
5’h0D	w	frame_resetn
5’h0E	r	laser_xy
5’h0F	w	ctrl_reg2
5’h10	r	num_labels
5’h11	r	obj_area
5’h12	r	obj_x
5’h13	r	obj_y
5’h14	r	obj_m20
5’h15	r	obj_m02
5’h16	r	obj_m30
5’h17	r	obj_m03

31	23	15	7	0
flood threshold	red threshold	sobel threshold	output mode	

(a) Control register 1.

31	23	15	11	0
flood 2 threshold	input only	unused	object id	

(b) Control register 2.

Figure A.1: Control register layout.

Mode enums

i	mode
0	pass
1	gray
2	sobel
3	thresh
7	flood1
8	flood2
4	cc
5	color
6	laser

Appendix B

Block Diagrams

This appendix contains lower level block diagrams that are while illustrative, too detailed for the body of the document.

Multiplication for Moments

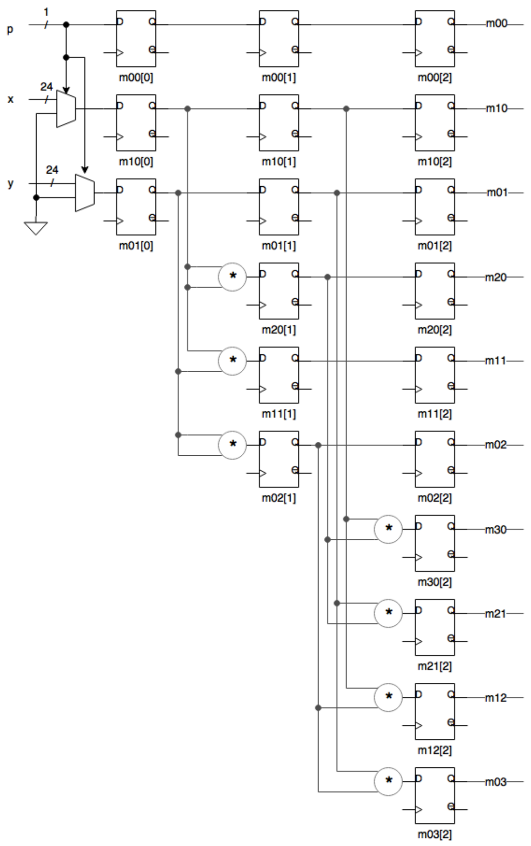


Figure B.1: Pipeline for calculating moment at pixel *p*.

Top

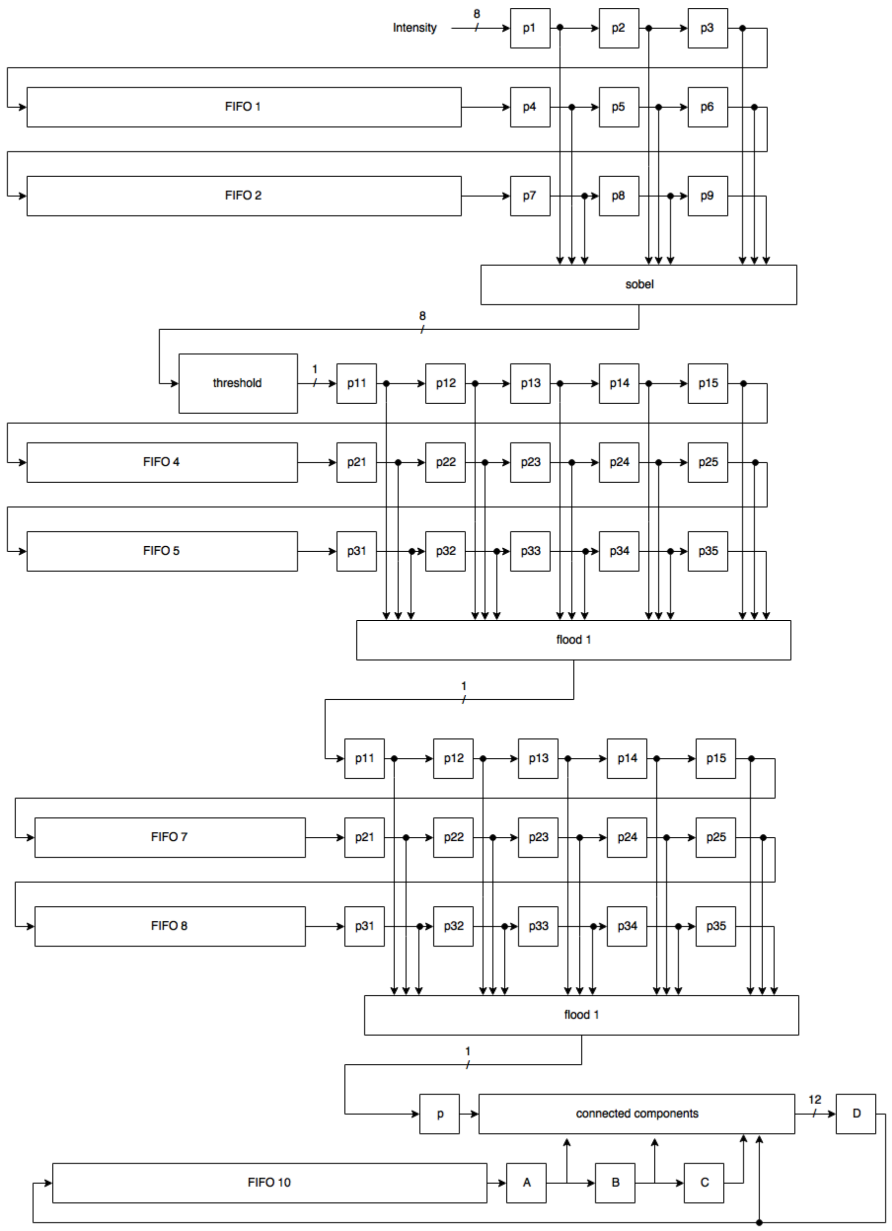


Figure B.2: Top level, highlighting line buffer structure.

Appendix C

Moment Invariants

> This method was intended for the differentiation between targets and non-targets in our design. Due to time constraints, we were unable to complete this. I had originally selected this method because I thought it was elegant to essentially hash an image based on its moments, which are relatively easy to extract from the image in one pass.

Moment invariants were introduced by Hu in 1962 [2] as a means of uniquely identifying an object. By this method, the outlines could be characterized as a unique fingerprint of seven numbers. In a training phase, outlines of friends are committed to memory. Then, the set of moments are calculated for each object detected and if they don't match any of the committed moments, it is deemed a foe.

Invariants

The invariants used are listed below.

Translational invariants

$$u_{ij} = \sum_x \sum_y (x - \bar{x})^i (y - \bar{y})^j p_{ij}$$

Scale invariants

$$n_{ij} = \frac{u_{ij}}{u_{00}^{1+\frac{i+j}{2}}}$$

Rotational invariants

$$\begin{aligned}
I_0 &= n_{20} + n_{02} \\
I_1 &= (n_{20} - n_{02})^2 + 4n_{11}^2 \\
I_2 &= (n_{30} - 3n_{12})^2 + (3n_{21} - n_{03})^2 \\
I_3 &= (n_{30} + n_{12})^2 + (n_{32} + n_{03})^2 \\
I_4 &= (n_{30} - 3n_{12})(n_{30} + n_{12}) \left((n_{30} + n_{12})^2 - 3(n_{21} + n_{03})^2\right) \\
&\quad + (3n_{21} - n_{03})(n_{21} + n_{03}) \left(3(n_{30} + n_{12})^2 - (n_{21} + n_{03})^2\right) \\
I_5 &= (n_{20} - n_{02}) \left((n_{30} + n_{12})^2 - (n_{21} + n_{03})^2\right) \\
&\quad + 4n_{11}(n_{30} + n_{12})(n_{21} + n_{03}) \\
I_6 &= (3n_{21} - n_{03})(n_{30} + n_{12}) \left((n_{30} + n_{12})^2 - 3(n_{21} + n_{03})^2\right) \\
&\quad - (n_{30} - 3n_{12})(n_{21} + n_{03}) \left(3(n_{30} + n_{12})^2 - (n_{21} + n_{03})^2\right)
\end{aligned}$$

Comparison

The sum of square differences was used to calculate a difference score to determine how different a given image is from a known image. The smaller the score, the more similar the images. A score of 0 indicates that the images are identical.

$$x = \sum_{i=0}^{5} \frac{(I_i - I_i')^2}{I_i I_i'}$$

Software Prototyping

This method was reasonably successfully in software. It could correctly identify scaled, shifted and rotated images. It is not designed to match skewed images. A sample image annotated with difference scores is shown in Figure C.1.

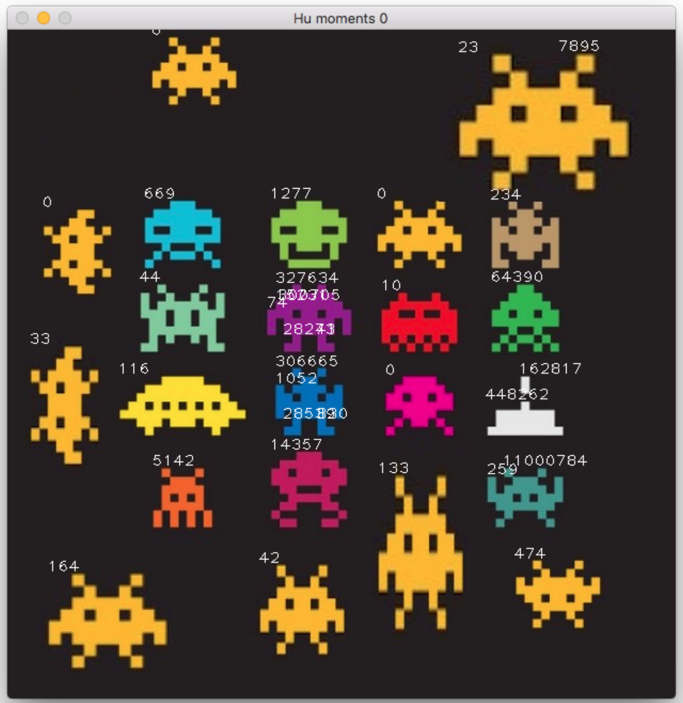
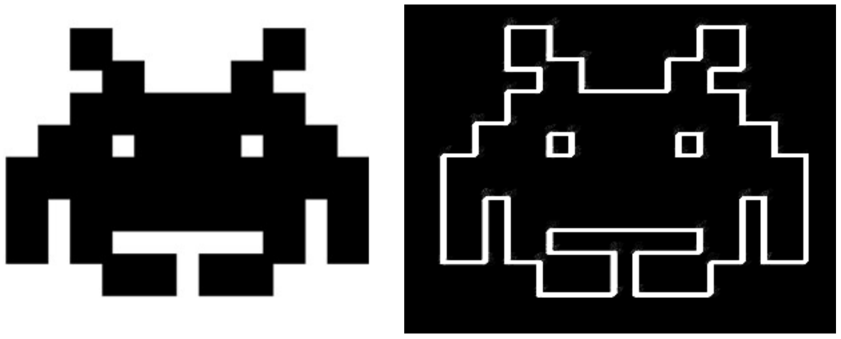


Figure C.1: Moments

Appendix D

Sample image processing output

Simulation results


(a) Input (b) Sobel

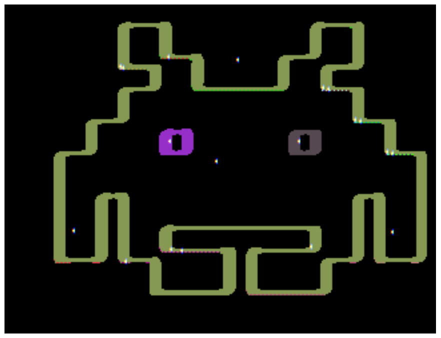
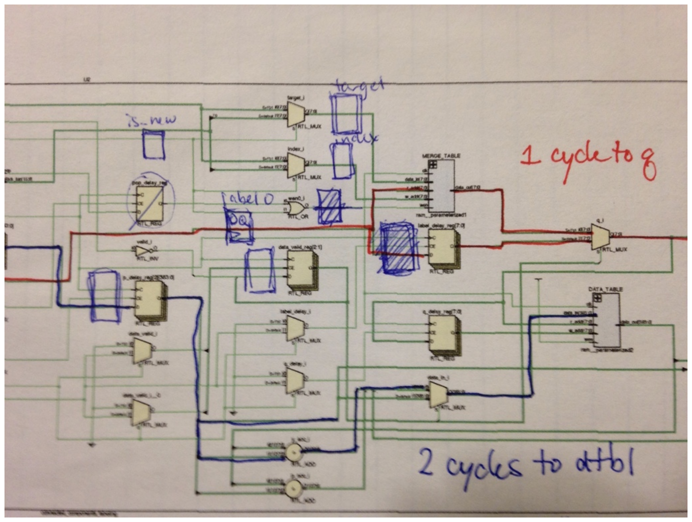

(c) Connected components

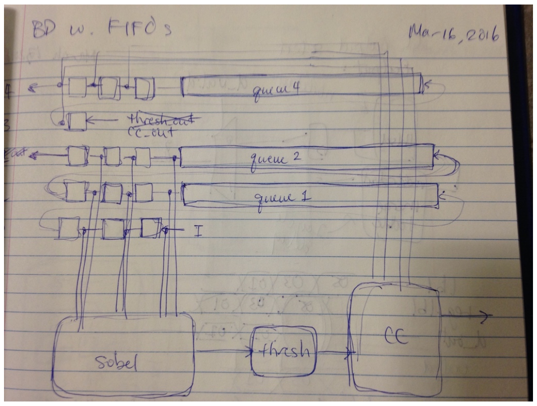
Figure D.1: Simulation results.

Appendix E

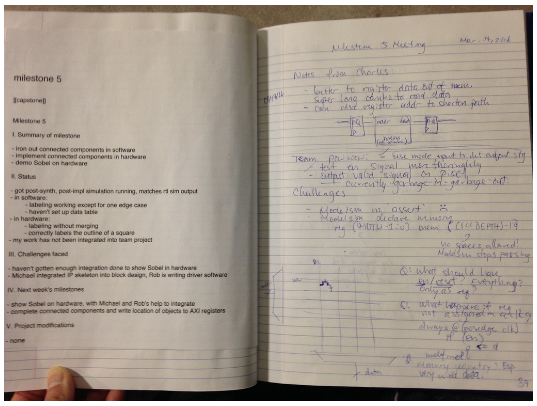
Notebook



(a) Schematic modification.



(b) Block diagram.



milestone 5

[[capstone]]

Milestone 5

I. Summary of milestone

- iron out connected components in software
- implement connected components in hardware
- demo Sobel on hardware

II. Status

- got post-synth, post-impl simulation running, matches rtl sim output
- in software:
    - labeling working except for one edge case
    - haven't set up data table
- in hardware:
    - labeling without merging
    - correctly labels the outline of a square
- my work has not been integrated into team project

III. Challenges faced

- haven't gotten enough integration done to show Sobel in hardware
- Michael integrated IP skeleton into block design, Rob is writing driver software

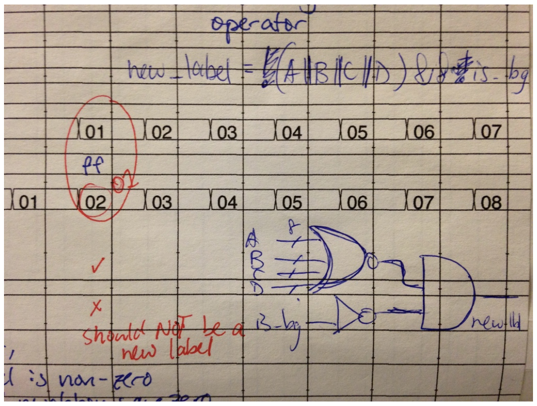
IV. Next week's milestones

- show Sobel on hardware, with Michael and Rob's help to integrate
- complete connected components and write location of objects to AXI registers

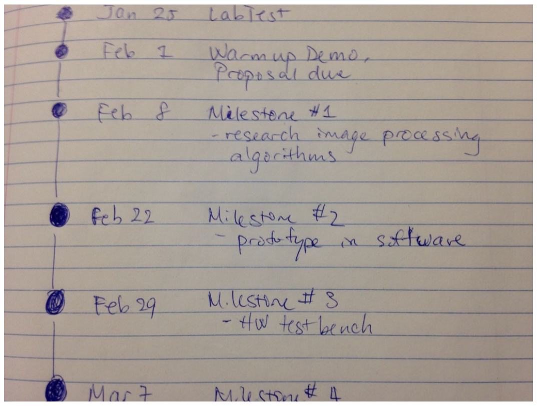
V. Project modifications

- none

(c) Milestone spread.



(d) Debugging simulation.


Jan 25 LabTest
Feb 1 Warmup Demo, Proposal due
Feb 8 Milestone #1 - research image processing algorithms
Feb 22 Milestone #2 - prototype in software
Feb 29 Milestone #3 - HW test bench
Mar 7 Milestone #4

(e) Project timeline.

Figure E.1: Excerpts from notebook.

Appendix F

Milestone Reports

The following milestone reports detail the major progress and challenges on a week to week basis.

Milestone 1

I. This week's milestones:

- find image processing algorithm
- evaluate how hardware friendly it is

II. Current status

- read some background on image processing, including:
  - image kernel convolution
  - image moments
- explored OpenCV
  - collection of computer vision libraries for C and Python
  - widely used (iOS, Android, OpenCL), comprehensive
  - is compatible with Xilinx HLS
  - documentation includes references to papers for algorithms implemented
- found example for finding contours, tried it on sample image

III. Challenges faced

- don't fully understand algorithms behind library methods
- OpenCV is object oriented; won't be easy to write in pure C
- also won't be easy to design in hardware

IV. Next week's milestones

- prototype in software
- reduce OpenCV functions used to simpler computational kernels, boil it down to something simpler V. Project modifications
- none so far

Milestone 2

I. Summary of milestone

- prototype object detection algorithms in software
- demonstrate functionality on some sample images

II. Status

- implemented in C:
  - color to grayscale
  - isolate single colour out of full colour image
  - Sobel operator for edge detection
  - isolate individual objects in image
  - calculate image moments
- successfully identified laser point from image
- successfully identified an object as a scaled version of a reference image, and reject dissimilar objects using image moment invariants

III. Challenges faced

- abandoned shape context approach for evaluating object similarity, due to complexity and because it is computation intensive
- haven't checked for robustness of image moments on real picture from camera

IV. Next week's milestones

- skeleton IP to interface with the rest of the system
- preliminary version of laser detecting IP
- set up tests to compare to software

V. Project modifications

None so far.

Milestone 3

I. Summary of milestone

- skeleton IP to interface with the rest of the system
- preliminary version of laser detecting IP
- set up tests to compare to software

II. Status

- surveyed possibilities for getting input in simulation, considered:

  - custom ASCII format data file ("VHDL test bench for digital image processing systems using a new image format" explores this idea)
  - existing image file format (e.g. bitmap)

- completed test bench that reads bitmap into memory, and writes memory to bitmap https://github.com/ngemily/detectinator

III. Challenges faced

- milestone too ambitious

IV. Next week's milestones

- implement software functions from Milestone 2 as hardware modules that work on pixel stream input

V. Project modifications

- pushed start of Vivado IP project to next milestone, since it's not necessary for starting work on IP and not very useful until we begin integration
- laser detecting moved to Michael's responsibility

Milestone 4

I. Summary of milestone

- implement software functions from Milestone 2 as hardware modules that work on pixel stream input

II. Status

- fixed bug in test bench; previously did not account for row padding
- implemented sobel operator
  - verified functionality in rtl sim
  - checked synthesis and implementation in Vivado
  - passes timing at 100 MHz
- started on connected components in software
- added IP skeleton to team git repo

III. Challenges faced

- it's hard to write hardware
  - thinking in circuits vs sequential semantics
  - less experience
- having trouble getting post-syn, post-impl sim
  - Vivado won't generate libraries for Modelsim

IV. Next week's milestones

- iron out single pass connected components algorithm
- implement connected components in hardware

V. Project modifications

- detection IP milestones have been shifted back about a week
- clarification of integration work, aim for Milestone 6:
  - modify IP to accept input accordingly, after Rob has HDMI feed
  - work with team to integrate/debug our IP with microblaze

Milestone 5

I. Summary of milestone

- iron out connected components in software
- implement connected components in hardware
- demo Sobel on hardware

II. Status

- got post-synth, post-impl simulation running, matches rtl sim output
- in software:
  - labeling working except for one edge case
  - haven't set up data table
- in hardware:
  - labeling without merging
  - correctly labels the outline of a square
- my work has not been integrated into team project

III. Challenges faced

- haven't gotten enough integration done to show Sobel in hardware
- Michael integrated IP skeleton into block design, Rob is writing driver software

IV. Next week's milestones

- show Sobel on hardware, with Michael and Rob's help to integrate
- complete connected components and write location of objects to AXI registers

V. Project modifications

- none

Milestone 6

I. Summary of milestone

- show Sobel on hardware, with Michael and Rob's help to integrate
- complete connected components and write location of objects to AXI registers

II. Status

Integration:

- integrated into group block design
- Sobel is showing up on HDMI output with some glitches

IP development:

- extended test bench to color merged labels, and draw dots to indicated centre of objects
- connected components labeling with merges is done
- connected components data collection is partially done
  - correctly finds centre of simple objects

III. Challenges faced

Integration:

- long compile time
- random hardware behaviour

IP development:

- getting cycles to line up; can't just access memory like in software
- failing setup time on merge table read address, need to pipeline

IV. Next week's milestones

- finalize interface to Microblaze so main software can be completed
- show connected components on HDMI display
- output locations of objects to AXI interface

V. Project modifications

- none so far

Milestone 7

I. Summary of milestone

- finalize interface to Microblaze so main software can be completed
- show connected components on HDMI display
- output locations of objects to AXI interface

II. Status

- showed Sobel and connected components on HDMI display
- can control display mode (various intermediate stages) from UART terminal
- can control IP thresholds from UART terminal, allows us to set Sobel threshold at run time
- output locations of objects upon request from microblaze*

III. Challenges faced

- noisy feed from webcam, rather poor quality outlined images
- trying something similar to median filtering

IV. Next week's (demo!) milestones

- finalize setup as a group, looking at getting data project to project shapes onto a white wall

V. Project modifications

Appendix G

Workflow

Post-syn and Post-impl sim using Modelsim

> The following method for setting up post-synthesis and post-implementation simulation is adapted from Aldec [4].

While RTL sim is usually a good indicator of design correctness, some discrepancies can arise as the design is synthesized and implemented. It is generally a good practice to perform testing at each of the stages show in Figure G.1. To do so, however, requires a little more setup.

As the design is compiled through Vivado, primitives are added to the user's netlist. This new, more low level netlist need to be exported. Then, these primitives need to have their behaviour defined in order for Modelsim to simulate it. The following commands detail how to do this.

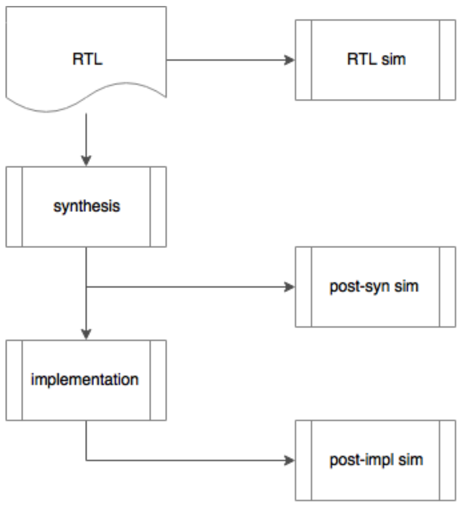


Figure G.1: Stages of compilation and testing.

Run the following command in Vivado's tcl console[1]:

[1]Modelsim ASE's installation must be modified slightly for this to work. In the Modelsim ASE install directory, perform `ln -s linuxaloem linuxpe`

```
compile_simlib -simulator modelsim -family artix7 -library unisim -language verilog
```

After synthesis or implementation, export the Verilog netlist. This can be done from the Vivado console using:

```
reset_run synth_1
launch_runs synth_1
wait_on_run synth_1
open_run synth_1 -name netlist_1
write_verilog -mode funcsim $origin_dir/top_synth.vo -force
```

When running Modelsim, link the libraries.

```
vsim -gui -L secureip -L unisims_ver work.tb work.glbl
```

Vivado compile script

```
set srcDir "src"
set outDir "proj"

create_project detectinator $outDir -part xc7a100tcsg324-1 -force

add_files -fileset sources_1 -norecurse $srcDir/connected_components.v
add_files -fileset sources_1 -norecurse $srcDir/location_generator.v
add_files -fileset sources_1 -norecurse $srcDir/mem.v
add_files -fileset sources_1 -norecurse $srcDir/sobel.v
add_files -fileset sources_1 -norecurse $srcDir/util.v
add_files -fileset sources_1 -norecurse $srcDir/top.v
add_files -fileset constrs_1 -norecurse $srcDir/timing.xdc

update_compile_order -fileset sources_1

reset_run synth_1
launch_runs synth_1
wait_on_run synth_1
open_run synth_1 -name netlist_1
write_verilog -mode funcsim $origin_dir/top_synth.vo -force

reset_run impl_1
launch_runs impl_1
wait_on_run impl_1
open_run impl_1 -name netlist_2
write_verilog -mode funcsim $origin_dir/top_impl.vo -force
```

Modelsim do file

```
file mkdir out

vlib work

vlog src/connected_components.v
vlog src/location_generator.v
vlog src/mem.v
vlog src/sobel.v
vlog src/util.v

vlog +define+RTL_SIM src/top.v
vlog +define+RTL_SIM src/tb.sv

vsim work.tb
run 2 ms

mem save -o out/merge_table.mem -f mti -data hex -addr hex \
    -startaddress 0 -endaddress 255 -wordsperline 1 /tb/dut/U2/MERGE_TABLE/mem

mem save -o out/data_table.mem -f mti -data hex -addr hex \
    -startaddress 0 -endaddress 255 -wordsperline 1 /tb/dut/U2/DATA_TABLE/mem

run -all

exit
```

References

[1] D.G. Bailey and C.T. Johnston. “Single Pass Connected Components Analysis”. In: *Proceedings of Image and Vision Computing* (2007), pp. 282–287.

[2] Ming-Kuei Hu. “Visual pattern recognition by moment invariants”. In: *IEEE Trans. Inform. Theory* 8.2 (1962), pp. 179–187. DOI: 10.1109/tit.1962.1057692.

[3] *OpenCV*. Version 3.1. 2016. URL: http://opencv.org.

[4] *Simulating a Design with Xilinx Libraries (UNISIM, UNIMACRO, XILINXCORELIB, SIMPRIMS, SECUREIP)*. 2016. URL: https://www.aldec.com/en/support/resources/documentation/articles/1674.