VEHICLE SPEED DETECTION WITH IMAGE PROCESSING USING FPGA

A graduate project submitted in partial fulfillment of the requirements
for the degree of Masters of Science
in Electrical Engineering

By:
Bahram Pourtaherian

May 2013
The graduate project of Bahram Pourtaherian is approved:

_________________________________________  date

Ali Amini, Ph.D.

_________________________________________  date

Shahnam Mirzaei, Ph.D.

_________________________________________  date

Ramin Roosta, Ph.D, Chair

California State University, Northridge
ACKNOWLEDGEMENT

I would like to thank Professor Dr. Ramin Roosta for all the motivation he provided for learning new technologies and his continuous support.
I would also like to thank Dr. Shahnam Mirzaei for support and the time and energy he spent on my countless questions.
I would also like to thank Dr. Ali Amini for his guidance, without his patient help, I would never be able to enter the program.
# Table of Content

Signature Page .......................................................................................................................... ii
Dedication .................................................................................................................................... iii
List of Figures ............................................................................................................................. vi
Abstract ......................................................................................................................................... viii

1. **CHAPTER 1 INTRODUCTION** .............................................................................................. 1
   
   Introduction .................................................................................................................................. 1

2. **CHAPTER 2 FIELD PROGRAMMABLE GATE ARRAY** ...................................................... 2
   
   Technology ................................................................................................................................... 2
   Xilinx Virtex 5 Family Architecture .......................................................................................... 3
   Xilinx FPGA Design Flow .......................................................................................................... 5

3. **CHAPTER 3 METHODOLOGY** ............................................................................................. 7
   
   3.1. Frame Read ......................................................................................................................... 9
   3.2. Segment Memory .................................................................................................................. 10
   3.3. Background .......................................................................................................................... 11
   3.4. Pre Background memory ...................................................................................................... 14
   3.5. Background memory ............................................................................................................ 15
   3.6. Motion .................................................................................................................................... 16
   3.7. Motion Memory .................................................................................................................... 18
   3.8. Cars Block ........................................................................................................................... 19
   3.9. Car Memory .......................................................................................................................... 24
   3.10. Speed ................................................................................................................................... 25
   3.11. Speed Memory ................................................................................................................... 29

4. **CHAPTER 4 CONCLUSION** ................................................................................................ 30

   4.1. Results .................................................................................................................................... 30
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>Typical FPGA architecture</td>
<td>2</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>FPGA configurable logic block</td>
<td>3</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Slice detailed structure</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Dual port cascadable block RAM</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>FPGA design flow</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Five Major Blocks and Memories Block Diagram</td>
<td>8</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Segment Memory</td>
<td>10</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Pre Background segment to memory allocation</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Background result after 4 frame</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Background result after 10 frame</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Background result after 56 frame</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>Background result after 98 frame</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>Pre Background Memory</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>Background Memory</td>
<td>15</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>Motion result after 3 frame</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>Motion result after 10 frame</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.12</td>
<td>Motion result after 56 frame</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.13</td>
<td>Motion result after 96 frame</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.14</td>
<td>Motion Memory</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3.15</td>
<td>Part of Motion memory</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3.16</td>
<td>each pixel neighbors</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3.17</td>
<td>FIFO configuration with common clock and distributed ram</td>
<td>20</td>
</tr>
<tr>
<td>Figure 3.18</td>
<td>Divider Generator</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.19</td>
<td>Cars Block result after 6 frame</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.20</td>
<td>Cars Block result after 15 frame</td>
<td>23</td>
</tr>
<tr>
<td>Figure 3.21</td>
<td>Cars Block result after 56 frame</td>
<td>23</td>
</tr>
<tr>
<td>Figure 3.22</td>
<td>Cars Block result after 96 frame</td>
<td>23</td>
</tr>
<tr>
<td>Figure 3.23</td>
<td>Car Memory</td>
<td>24</td>
</tr>
<tr>
<td>Figure 3.24</td>
<td>real color image in a frame with a car in the bottom of picture</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3.25</td>
<td>real color image in next frame with a car in the bottom of picture</td>
<td>26</td>
</tr>
</tbody>
</table>
Figure 3.26: Real color image in a frame with a car in upper side of picture ...................... 27
Figure 3.27: Real color in next frame with a car in upper side of picture ...................... 27
Figure 3.28: Speed Memory ....................................................................................................... 29
Figure 4.1: Speed result after 4 frame ...................................................................................... 30
Figure 4.2: Speed result after 12 frame .................................................................................... 31
Figure 4.3: Speed result after 49 frame .................................................................................... 31
Figure 4.4: Speed result after 56 frame .................................................................................... 31
Figure 4.5: Speed result after 57 frame .................................................................................... 31
Figure 4.6: Speed result after 73 frame .................................................................................... 32
Figure 4.7: Part of simulation for one frame to show performance ......................................... 34
ABSTRACT
VEHICLE SPEED DETECTION WITH IMAGE PROCESSING USING FPGA
By
Bahram Pourtaherian
Master of Science in Electrical Engineering

By more technology advance in the past few years there has been a huge increase in automated speed limitation enforcement equipments, such as speed cameras. Most of these devices work with a sensor. Therefore, there is a need to install new expensive equipment to include new cameras and sensors. Still there are many traffic cameras installed all over cities that only used for observing traffic. The main idea of this project is to install an FPGA chip near traffic cameras and together they perform as speed camera.
The FPGA connected to traffic camera do not need additional sensor and by using Image Processing and video processing algorithms, system can detect speed of the cars. Using FPGA gives the advantage of real time processing.
CHAPTER 1: INTRODUCTION

Introduction

In the past few years, there has been an enormous growth in the field of embedded system. The growing need for image processing applications forced researcher across the globe to come up with innovative design techniques that can meet the need for these types of applications. Most of these designs use high level programming languages, a lot of them use Matlab functions, which make researchers able to create an application with a couple lines of coding and in some cases, and they use C/C++ languages for creating the image processing applications. Although this applications works in Matlab environment, they are not yet advanced enough to generate a system that can run on a hardware, because Matlab use matrix algorithms and this algorithms cannot be implemented on a hardware that do bit by bits processing. It is a same story with applications wrote with higher level programming languages like C/C++. Some of this programs still can be run on a CPU inside a hardware like microprocessors or CPU inside a FPGA chip, but most of them violate one of essentials of real image processing applications, this applications need to be Real Time, because software executes in a sequential manner where hardware can execute completely parallel. One of best way of implementing this kind of applications is using pure hardware chips, we can use Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA) for this purpose, but ASIC are not suitable for image processing purpose, ASSIC take longer time to create and it is not changeable. The main reason is, for most of Image Processing applications designer need to change the design for any change in the environment like change of location, angel or any other physical changes happens in the environment. This is where FPGA can come in, because it has a high performance of ASIC and flexibility of Microprocessors. FPGAs chip can be reconfigured easily if any modification needed, but this is not the case for ASIC designs.

The purpose of this project is to implementing a Vehicle Speed Detection that use output of Traffic Cameras that installed in many highways across the globe as input and with some algorithms that I designed myself will detects locations of cars in each frames. By having car locations in each frames and knowing the time between each frames it is possible to calculate speed of cars, the methodology will be fully discussed in chapter 3.
CHAPTER 2: FIELD PROGRAMMABLE GATE ARRAY

Technology

FPGAs (Field programmable gate arrays) are programmable ICs and digital circuits can be designed using them. The FPGAs are programmed with hardware description languages like Verilog or VHDL. On lot of applications, the reconfigurability feature of FPGAs and low non-recurring engineering (NRE) cost of the FPGAs offers huge advantages. Unlike application specific integrated circuits (ASICs) where after the chip has been made, designers cannot do any reconfiguration on their design.

Logic circuits on FPGAs have been made of matrix of configurable logic blocks (CLBs), designers have ability to reprogram CLBs, and reconfigure interconnections to connect the CLBs. FPGAs have on-chip blocks of memory, which can be connected to CLBs. Figure 2.1 shows a typical FPGA architecture that has basic building blocks.

Figure 2.1: Typical FPGA architecture
Xilinx Virtex 5 Family Architecture

The Virtex 5 family is not the most recent family within Xilinx FPGA families, but it is one of most powerful families within Xilinx families. Virtex 5 FPGAs have many hard-IP system level blocks built in, such as powerful 36-Kbit block RAMs, second-generation 25x18 DSP slices and advanced configuration options. The CLBs are the primary logic resources for implementing combinational and sequential circuits. All of CLB elements connect to a switch matrix for access to general routing matrix. General CLB shown in Figure 2.2. A CLB have two slices. These slices are not connected to each other. Each slice has its own carry chain.

![Figure 2.2: FPGA configurable logic block](image)

All the slices contains four look up tables (LUTs), four Flip Flop, a carry chain for each slice, and multiplexer. Configuring these elements can create arithmetic, logic, and ROM functions.

Figure 2.3 show the architecture of each slice in CLBs. LUTs can implement any functionality that has 4 inputs or less. Several multiplexers provided to connect LUTs to the neighboring logic resources. Output of LUTs could be connected to the Flip Flop or bypass the Flip Flop.
Virtex 5 family provides many 36 Kb block RAMs. Each 36 Kb block RAM contains two separately controlled 18 Kb RAMs. Number of block RAMs in a FPGA depends on the Virtex 5 device. Figure 2.4 shows a cascadable block RAM with two separate read and write ports.
Figure 2.5 shows the Xilinx FPGA design flow that includes following steps: Functional specification of the system, design entry in hardware description language Such as VHDL or Verilog, design synthesis, design implementation (place and route), Device programming, and finally in circuit verification. Design verification, which Includes both functional verification and timing verification, takes places at different Points during the design flow.
Figure 2.5: FPGA design flow
CHAPTER 3: METHODOLOGY

For implementation of Vehicle Speed Detection on FPGA, based on research I used some known algorithm for some parts and developed some suitable algorithm myself. There are five major blocks in the design, that each do specific task, each of this blocks write produced data to a memory, and next block uses previous blocks memory to do what it is supposed to do. The Figure 3.1, shown the Five major blocks and Memories.

1. Frame Read, that get raw grayscale image for each frame, and segment the image and write it to segment memory.
2. Background, reads segmented data for each frame, by reading and writing to Pre Background Memory, creates the background image and write the final background value to Background Memory.
3. Motion, reads data from segment and background, compare the 2 sets of data together and write a one bit data, 1 for places motion detected and 0 for places the segment data were close to background to Motion Memory.
4. Cars, reads the Motion Memory and calculate the center of mass, writes the detected values to Cars Memory.
5. Speed, calculate speed of cars and write it to Speed Memory.
Figure 3.1: Five Major Blocks and Memories Block Diagram

For this project, for this design four frames per second video rate have been used, size of image is 120*160, and it is black and white to reduce the memory usage and processing power needed. The Images are in an external memory outside FPGA on the board, and Frame Read block manage the read addressing. For simulation, a text file with images values have connected to test bench and give the top module data it asked.
3.1. Frame Read

The first stage would be reading images frame by frame, because I do not have hardware and ability to connect my FPGA to real camera above highway. I put images in an external memory and connected my design to this external memory. Like all the memories, the output of my design would be an address to this memory an input would be a data. This data is value of a pixel in certain frame, because of FPGA limitations that I mentioned earlier, my each frame size is 120x160, and there is 100 frame of image in external memory, the data in the external memory is one-dimensional, and address 0 is first pixel of first row and first column of first frame. Address 1 is in first frame and first row and second column, and so on. After 160 data, we get to second row data and so on. Understanding of data order is essential for reading this data from external memory and managing addressing, to assure reading correct pixel value.

For reducing the noise in Image Processing algorithms, researchers suggest image segmentation. Image Segmentation also help to get data that are more meaningful from image and it has always used in object detection algorithms. The idea is pixel values in each neighborhood is similar and by averaging the certain segments of image, the algorithm can still work, with less data to process and getting noise reduction. Image Segmentation simply means averaging the pixel values of segment size. For this project the segment size of 4x4 pixel have chosen, for 2 reason, the first reason was based on the image size (120x160) if the segment size be too big we will lose a lot of meaningful data, and if the segment be too small, it don’t have that much effect on noises. The other reason was simplicity for calculating the average, that in FPGAs division only works for numbers that are power of 2. In addition, 4x4=16 make the design easier and smaller, and do the job that supposes to do.

The first block of design does two main functions, its first job is to manage addressing for reading the data from external memory and its second job is to calculate the segment value, average of 4x4 pixels. The main challenge here was managing addressing. For averaging 4x4 segment, 16 data need to be read, add together and then divide by 16, but this 16 data are not in a row in external memory as mentioned earlier, therefore there is a need to manage the read addressing, to manage reading a correct data of a segment and calculating it segment value when reading the data. This process has done in 18 clock cycle, 16 clock cycle for reading 16 data, one data each clock, and 2 clock cycle for calculating the average and write the average data in segment memory so next block can use it. With help of 4 counter, sxcntr,syncntr this 2 are segment counters that keep track of how many value in each segment have been read, and xcntr, ycntr that keep track of position of each pixel in x and y axis. we need to know what is the position of current pixel in x and y axis in each frame so for next cycle we be able to read correct value with correct address, because of special cases like when we reached end of row or end of image the xcntr and ycntr are essential in setting next address. All the codes are available at the end for reader that is more curious.
3.2. Segment Memory

Figure 3.2: Segment Memory

As shown in Figure 3.2 Segment memory is dual port RAM, created with Distributed Memory Generator, the read and write data width is 8 bit, and memory depth is 1200, with 11 bit address. Outputs are connected to segment block and background block, and segment block writes to it.
3.3. Background

The main purpose of background block is background estimation, as mentioned earlier, in the design, the algorithm used to detect vehicles is comparison between current frame and background. Other algorithms are also able to detect objects in image, such as edge detection, and Mask, with comparing efficiency and resource each algorithm needs comparison between current frame and background have been chosen. The background estimation is one of most critical functions that need to be done. The goal is to have an image without any car in it, just the highway and the environment. The easiest way to get this image is simply take a photo when there is no car in the road. However, this is not possible; there are always some cars in the road, in most of the highways. If this approach chosen it could cause another problem too, the background is static, if any changes happen to background and environment there is no way to correct the background image rather than taking a new picture. Therefore, there is a need for an algorithm to estimate background, by removing the motions from the image and it need to be dynamic, so if there were any changes in background itself, the algorithm should be able to adjust the background to that changes.

Another issue that needs to be considered is time, what is the fastest way to create a background? In addition, in a highway with a high traffic, we may end up do not getting the asphalt a lot, but different cars with different color and shape.

The algorithm used in the design, looks for most repeated data and assumes that is the background. Because the background value going to repeats, more than other values this algorithm works. For making this algorithm to work, I used a big memory to store pervious values for each segment and for each data, there is a value in next address that shows how many times this data is repeated. The size of this memory is 8x24000(30x40x20), there is 20, 8 bit memory cell for each segment, 10 for values and 10 for number they repeated. When new segment value arrives for new frame, the background block reads all 10 values for each segment and if it’s the value in memory matches the segment value + threshold, increase the number it get repeated (the count value in next address) and write it back to memory in same location.

The block needs to read count values too, so it can determine which one is repeated most and write it to background memory. By repeating to write to background memory for each frame, it guaranties the dynamic update for background. So if some changed happened for actual background this change adopt to memory background after some frames.

Background memory also needs to keep track of which count has less value than others do, because the number of memory cells that we put previous frames values in it is limited. There is a need to replace some old values with new value so if something changed in background, the change in background value can raise and become the background. For this purpose the background block keep track of min count and in case current segment value was not found in previous stored data, the new value can be replaced in the address of least repeated value and count value of 1.
Figure 3.3 shows how the data have been written in pre background memory for each segment, keep in mind that each segment has its own 20 memory cell in pre back memory.

Figure 3.3: Pre Background segment to memory allocation

In figure, 3.4 to figure 3.7 you can see the input image in left hand side and produces background by FPGA in right hand side.

Figure 3.4: Background result after 4 frame
As it shown in figure 3.4 through figure 3.7, by processing more frames, the background image have data that are more meaningful and are more similar to real background, have less noise and are more suitable for comparing background to each frame data to detect motions. Less noise we have in background system will be more able to detect vehicles in next stage. In Video Processing algorithms, we may get some noises at first, but after sometimes system reaches to a stable stage that can produce reliable and accurate results.
3.4. Pre Background memory

Figure 3.8: Pre Background Memory

As shown in figure 3.8, this memory is the largest memory used for design, read, and write. The width is 8 bits, and the memory depth is 24000 addresses. The addresses are 15 bits. It is a simple dual port RAM, where only the background block has access to it, and it can read and write to it with the same address.
3.5. Background memory

![Distributed Memory Generator](image)

**Figure 3.9: Background Memory**

As shown in figure 3.9 Background Memory is distributed memory, it is simple dual port RAM, with read and write width of 8 bit and depth of 1200, with 11 bit addresses. Background Memory writes to it, and motion Blocks reads from it.
3.6. Motion

In this block, we want to compare each frame with background and detect motions. The block going to read every address of Segment memory and background memory, if the change between the values for same location of image were greater than threshold, block going to write 1 in Motion Memory for that address. If the change is less than threshold, block going to write 0 in Motion Memory for that address. At the end, what we get in Motion Memory is 1 for places we have motions and 0 for places the image is same as memory.

One of challenges for doing so is finding suitable threshold; if threshold were small number we get many pixels detected as motion, but they can be noises in current frame or in background. If threshold were big number, we do not detect pixels that have motion. Like cars with darker color, because the asphalt is gray, difference between gray or black cars with asphalt color is not significant in 8 bit gray scale. Based on experience and changing threshold to different values and comparing the results, 16 were giving the best result. Although we still detect some pixels in image that do not have any motion, we still can filter them out in next step.

For seeing the result and present them in this report, and for choosing the best threshold, I wrote all the motion values to the file and transferred them to Matlab, to see exactly what I detect. In Figure 3.10 through Figure 3.13, the right hand side is segment image and left hand side is motion values (0 or 1) multiplied by segment image. Therefore, in left hand side, where ever there is no motion detected, image is black and for where we detect motion the image shows the segment value.

Figure 3.10: Motion result after 3 frame
As you can see by processing more frames, the result got better, because the background is getting more reliable. We still have some noises at the end because of threshold, but this noises going to be filtered in next phase.
3.7. Motion Memory

As shown in Figure 3.14, Motion memory is true dual port ram created with Block Memory Generator, it has 1 bit read/write write width, and with depth of 1200, that makes the address 11 bit. True dual port RAMs could not be created by distributed RAM. Therefore, there is a need to use Block Memory. Motion Block only writes to this memory, and Cars block Read and write to it.
3.8. Cars Block

In last phase, system detects all the motions in a current frame, but they are bunch of 1s and 0s. However, for calculating the speed of vehicles we need to group the 1s together so we have each car 1s group together and calculate the center of mass for each vehicle so we know the place of cars. Then in next phase, we could be able to compare the center of mass together and calculate the speed. This is not an easy task in low level languages such as Verilog or VHDL. The RTL languages are so limited. There is no queue, or array supported in this languages. The Figure 3.15 can give you better idea. There are 3 groups of 1s, one group of 10 in left, a group of 7 in lower right and single 1 in upper right.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.15: Part of Motion memory

The ideas is when we have a 1, in a pixel, we should search the neighbors and go through all the pixels that their value is 1. We should keep track of where we started, which neighborhood we searched and which neighborhood has left. Each pixel has 8 neighborhoods, and we need to search thorough all of this neighborhoods, for possible 1s. Moreover, Figure 3.16 can give reader better understanding of each pixel neighborhood.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 3.16: each pixel neighbors
To do this task, we search through the image and when we got our first 1, we put all the neighborhood addresses in a queue, and then go to one of neighbors, and if it were 1, we add it neighbors to queue too. We continue until, the neighbor we chose have 0 value. Then we start reading addresses from queue and examine them, to see, if they are 1 or not. By continuing to do this, the block can detect all the 1s connected together. However, there is no queue or array in Verilog (Queues are added to System Verilog, but they are not synthesizable. Arrays are added to System Verilog and they are synthesizable, but System Verilog is not supported in FPGAs) so I used a common clock FIFO, it can act like a queue. For implementing an FIFO, I used a Xilinx Core IP. The settings and it overall information is available in Figure 3.17.

![Fifo Generator](image)

**Figure 3.17: FIFO configuration with common clock and distributed ram**

This FIFO have been made by distributed RAM, it have 1024 write depth, and write width is 23 bit.

For calculating center of mass, I used general central mass equation:

\[ x_{cm} = \frac{\Sigma (m_i)(x_i)}{\Sigma (m_i)} \]

because all of pixel have same weight, we can ignore m in this equation. However, we need to have exact x and y location of each address. So in whole process, I kept track of x and y position. Keeping track of x and y is also useful for detecting the image boundaries, and does their special treatments. When I start a frame, it’s easy to keep track of x and y, but when load the data from FIFO, calculating the X and Y is time consuming and needs a lot of process. So I chose to write the x and y coordinate of each address in the queue, so I could have less process and clock latency when I read from FIFO, that is why the FIFO write width is 23 bit, 11 bit address, and 6 bit each X and Y coordinate.
When we write neighbors of a pixel A for example in FIFO, if each one of this neighbors lets say B, have 1 in its address. We are writing A as B’s neighbor in FIFO, and we are in an infinite loop. For solving this problem, when I found a 1 in A address. After adding all of its neighbors to FIFO, I am writing 0 to A address, in motion memory. Therefore, when A address is read as B’s neighbor, its value is 0, and system going to read another address from FIFO, and continue process. For calculating center of mass, we need to know how many 1s we found in our search. There is a counter that each time we find a 1 its increment. In addition, when there is no more 1 in neighborhood, it is going back to 0.

For calculating center of mass of each car, there is 2 sum value, one for X and one for Y, each time we find a 1, we add X coordinate of that pixel to Xsum and Y coordinate of that pixel with Ysum. At the end when there was no 1 in neighborhood, we should divide Xsum and Ysum with counter value that counts numbers of 1s. The division to non power of 2 is not supported in FPGAs. The synthesizer does not do that automatically. So I used Xilinx IP Core for division, data about it is available in Figure 3.18.

At the end, when there we detect all 1s in a neighborhood, we write X, Y coordinate of center of mass, and size of this vehicle in Car Memory. In addition, go back to address when we found our first 1. In addition, search for another 1 to detect another neighborhood. Groups with less than 5 pixels in neighborhood are ignored to reduce the noise and not meaningful data. In next phase, we need to have 2 frames data, to compare the current frame vehicles center of mass coordinate with previous frame, and detect speed. Therefore, I divided Cars Memory to half. In addition, in each frame I write to half of frame. Next frame write to other half. Address management get a little bit more complicated. However, in each frame, the previous frame data are available in the memory.
Figure 3.18: Divider Generator

For showing the result here and for simulation, data have been written to a file during simulation in Xilinx Ise and have been read in Matlab, rectangle with size counter/2 and center of calculated X and Y coordinate have been added to segment picture. The Figure 3.19 trough Figure3.22 shows the result.

Figure 3.19: Cars Block result after 6 frame
Figure 3.20: Cars Block result after 15 frame

Figure 3.21: Cars Block result after 56 frame

Figure 3.22: Cars Block result after 96 frame
3.9. Car Memory

![Distributed Memory Generator](image.png)

Figure 3.23: Car Memory

Figure 3.23 shows Car Memory is Distributed Memory, with 8 bit read/write data width, it has depth of 512, and addresses of 9 bit, Cars block writes to it, and Speed block reads from it.
3.10 Speed

The main purpose of this project is to calculate speed, time between 2 frames is a constant, and by calculating the distance, a vehicle moved in 2 frames, we could easily calculate speed. In last block, we detected cars positions and registered it to Car Memory. In Car Memory, data for 2 frames is restored and for each frame, there is bunch of center of mass for different cars. If we could know, which car is which, calculating distance is not going to be hard. For detecting same car in previous frame, their X, Y coordinate data should satisfy some conditions, in highway cars go straight, so Y coordinate in current frame cannot be less than Y in previous frame. Another condition is X coordinate cannot have many changes between two frames, because our sample rate is 4 frames per second, so time between 2 frames is 25 ms. with this speed, even if the drivers were in middle of changing lanes. In 25 ms, they are still in the same lane, and change in X coordinate not going to be a big number. Therefore, for finding a car in previous frame, each candidate (each car in previous frame) should satisfy two conditions, it should have closest X coordinate, to current X coordinate. In addition, its Y coordinate should be smaller than current Y coordinate.

For doing so, the block reads the first car coordinate in current frame, and look for a match in previous frame, if the match was found, the speed going to be calculated and X, Y coordinate and size of the car in pixels, and speed of the car going to be written in Speed Memory. Then it goes back, read next car data in current frame, and searches the previous frame for match. It repeats this algorithm until there is no more car data in current frame, then it exit the block.

One of issues for calculating speed is, the image is 2 dimensions and in reality, the highway is 3 dimensions. It means cars near camera are bigger than cars that are future and are in upper part of image, same thing happens for asphalt too. For example, if we assume 5 pixel in upper part of image is equal to 100 meter, in the bottom of image 5 pixels is equal to 20 meter. The difference between figure 3.24 and figure 3.25 is 25 ms, and you can see car position changes dramatically. In figure 3.26 and figure 3.27, the time difference between two images is also 25 ms and the move in Y axial is so little. If picture would 3 dimensional, the change in Y would be linear, and with same speed, the change of position between in any frame would be same.
Figure 3.24: real color image in a frame with a car in the bottom of picture

Figure 3.25: real color image in next frame with a car in the bottom of picture
Figure 3.26: Real color image in a frame with a car in upper side of picture

Figure 3.27: Real color in next frame with a car in upper side of picture
To solve this problem, I assumed in lower half of image the image act linearly. It is not a true assumption and it has consequences in calculating speed. It lowers the accuracy of algorithm by 1-2%. However, this is the best solution to avoid very complicated algorithms that can solve inaccuracy of system.

For calculating speed, as discussed above, the car should be in lower half of image. Original video have 24 frames per second, the time between two frames is known. By having the position change, we can calculate speed. However, it is not possible to measure distance between beginning and end of image by hand in highway! However, if a car by known steady speed passes the image, we can calculate asphalt size in image. My brother passed the highway by steady speed of 112 KMH (70MPH). By counting frames that my brother car is in lower half of video, and counting number of frames I was able to calculate time it take the car to enter image and reach half of image, and by knowing speed of car, calculating the distance car moved (lower half of picture) is easy. I have count 16 frame, from when car entered image, until it reached half of picture. By knowing the video rate of 24 frames per second, we easily calculate that it took 0.66s for the car to pass the image. My picture have 120 vertical pixels, half of it would be 60 pixels. Therefore, I can calculate what the difference between two pixels in highway is.

\[
\text{Time}=16/24=0.66s \\
70\text{MPH} = \frac{10\times1.6\times1000}{3600} = 31.11\frac{m}{s} \\
31.11\frac{m}{s} \times 0.66s = 20.55m \\
\frac{\text{eachpixel}}{20.55m} = \frac{60\text{pixel}}{60\text{pixel}} = 0.34\frac{m}{\text{pixel}}
\]

So now, because I reduce the video rate to 4 frame per second, time between 2 frames are 0.25 s, and because we segmented the pixels at the beginning, each pixel speed module detect is 4 frame. By knowing this we can calculate the speed of each vehicle by knowing how many pixel it moved.

\[
\frac{\#\text{pixel} \times 0.34 - \frac{m}{\text{pixel}}}{0.25s} = \frac{4 \times 3600}{1000} = \text{speed} \frac{km}{h}
\]

Everything in above equation is constant except number of pixels. So

\[
\frac{\text{speed} \ km}{h} = \#\text{pixels} \times 19.58
\]

Because the floating point is not supported in FPGAs, I use the below equation.

\[
\frac{\text{speed} \ km}{h} = \#\text{pixels} \times 19
\]
3.11. Speed Memory

As shown in Figure 3.28 Speed memory is distributed Memory with 8 bit data Width and depth of 256, and 8 bit of addresses. Speed Block writes the position of cars and their speed to this memory.
CHAPTER 4 : CONCLUSION

4.1. Results

In this project, I developed and implemented a new algorithm to detect speed of cars with Image Processing, for the car I knew its speed is 112 KMH; I have a 114 KMH speed. In addition, this is impressive result in my idea, but it is not the case all the times, I have a speed for some cars as high as 256 KMH. the problem is the image resolution is so low. Because I left out half of the image, I had to overcome the difference between 2D image and 3D highway. In addition, the resolution of video I used from beginning was low and only had 120 pixels in Y axial. In addition, by ignoring half of picture, I only left with 60 frames, and because I segment the data at the beginning by size of 4*4, I ended up with only 15 pixels, and difference between each pixel is 19KMH. But by increasing the resolution of video, there would be much better results, the original video size was 720*1280, by using that image, pixels count going to increase 5 time, and move in each segment would be equal to 4 KMH. but, the size of image would be 25 time of what I used. The process time going to increase at least 25 times, and it still would be ok, and works real time. However, the main downside would be increase in resources needed in FPGA. For my biggest memory I used 8*24000 memory block, multiply it by 25 and it would be 8*600000. In addition, this is just one memory block needed, the addresses would dramatically increase, and address managing logics would increase dramatically. It may be possible to implement algorithm with high resolutions image, with expensive and huge Vertex 7 FPGAs, and the only thing needed is changing some parameters and some constants in RTL code. However, in FPGAs smaller than Vertex 7, high resolution image and high accuracy is not possible, even for simulation. Simulation with low resolutions video, took at least 1 hour at the end of design, and by high resolution video it’s going to increase at least 40-50 time (logic going to be bigger too) so without very expensive equipment it’s not possible to get accurate data. However, I developed and implemented new algorithm and proved that it works. I get some accurate data, and it proves that algorithm is correct and it is implantable. In addition, the only thing between the design and accurate result is expensive equipment.

![Figure 4.1: Speed result after 4 frame](image-url)
Figure 4.2: Speed result after 12 frame

Figure 4.3: Speed result after 49 frame

Figure 4.4: Speed result after 56 frame

Figure 4.5: Speed result after 57 frame
Figure 4.6: Speed result after 73 frame
4.2. Resources

Xilinx ISE software has an ability to determine used resources, after synthesizing Design and run place and routing. Xilinx ISE shows below results for used resources on Virtex5 XC5VLX30 device. This chip is reasonably big chip with lot of Slices available, the design did not used a lot of resources for 120*160 video resolution, but as discussed earlier the higher resolution video going to take a lot more resource than what is available in this chip.

<table>
<thead>
<tr>
<th>Slice Logic Utilization</th>
<th>Used</th>
<th>Available</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Registers</td>
<td>907</td>
<td>19,200</td>
<td>4%</td>
</tr>
<tr>
<td>Number used as Flip Flops</td>
<td>907</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of Slice LUTs</td>
<td>2,669</td>
<td>19,200</td>
<td>13%</td>
</tr>
<tr>
<td>Number used as logic</td>
<td>1,471</td>
<td>19,200</td>
<td>7%</td>
</tr>
<tr>
<td>Number using O6 output only</td>
<td>1,214</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number using O5 output only</td>
<td>89</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number using O5 and O6</td>
<td>168</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used as Memory</td>
<td>1,188</td>
<td>5,120</td>
<td>23%</td>
</tr>
<tr>
<td>Number used as Dual Port RAM</td>
<td>1,188</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number using O6 output only</td>
<td>1,188</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number used as exclusive route-thru</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of route-thrus</td>
<td>116</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number using O6 output only</td>
<td>98</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number using O5 output only</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of occupied Slices</td>
<td>962</td>
<td>4,800</td>
<td>20%</td>
</tr>
<tr>
<td>Number of LUT Flip Flop pairs used</td>
<td>2,967</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number with an unused Flip Flop</td>
<td>2,060</td>
<td>2,967</td>
<td>69%</td>
</tr>
<tr>
<td>Number with an unused LUT</td>
<td>298</td>
<td>2,967</td>
<td>10%</td>
</tr>
<tr>
<td>Number of fully used LUT-FF pairs</td>
<td>609</td>
<td>2,967</td>
<td>20%</td>
</tr>
<tr>
<td>Number of unique control sets</td>
<td>139</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of slice register sites lost</td>
<td>145</td>
<td>19,200</td>
<td>1%</td>
</tr>
<tr>
<td>to control set restrictions</td>
<td>49</td>
<td>220</td>
<td>22%</td>
</tr>
<tr>
<td>Number of BlockRAM/FIFO</td>
<td>7</td>
<td>32</td>
<td>21%</td>
</tr>
<tr>
<td>Number using BlockRAM only</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 36k BlockRAM used</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 18k BlockRAM used</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Memory used (KB)</td>
<td>234</td>
<td>1,152</td>
<td>20%</td>
</tr>
<tr>
<td>Number of BUFG/BUFGCTRLs</td>
<td>1</td>
<td>32</td>
<td>3%</td>
</tr>
<tr>
<td>Number used as BUFGs</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average Fanout of Non-Clock Nets</td>
<td>6.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.3. Performance

For this design to work real time, based on 4 frames per second input video rate, processing each frame should not take more than 25ms. Examine simulation timing that shown in figure 4.7, shows that 3rd frame starts in time 36000 micro second simulation time, and it ends in 46000 micro second simulation times. So processing each frame takes around 12000 micro second, this is 20 time faster than what was needed for real time processing for this resolution and this design can handle at least 20 time growth in image size.

![Figure 4.7: Part of simulation for one frame to show performance](image-url)
4.2. Timing

Timing analysis is one of the most important stages of any digital design, it determines how fast a clock cycle can run, and faster clock cycles mean better performance. For this design, the 100MHZ was aimed for clock speed so system would be able to work real time. After synthesizing and place and routing the timing report, become available. Timing report shows that all constraints were met and critical path has 8.967 ns delay that 1.982ns is logic delay and 6.985ns is routing delay. Therefore, FPGA chip can be clocked up to 110MHZ. The part of timing report is available below.

=====================================================================================================
Timing constraint: TS_clk = PERIOD TIMEGRP "clk" 10 ns HIGH 50%
INPUT_JITTER 0.1 ns;
3520888 paths analyzed, 14553 endpoints analyzed, 0 failing endpoints
0 timing errors detected. (0 setup errors, 0 hold errors, 0 component switching limit errors)
Minimum period is 9.063ns.
=====================================================================================================

Paths for end point back/addressread_0 (SLICE_X44Y21.C2), 15319 paths
=====================================================================================================
Slack (setup path): 0.937ns (requirement - (data path - clock path skew + uncertainty))
Source: back/addressin_1 (FF)
Destination: back/addressread_0 (FF)
Requirement: 10.000ns
Data Path Delay: 8.967ns (Levels of Logic = 8)
Clock Path Skew: -0.035ns (0.414 - 0.449)
Source Clock: clk_BUFGP rising at 0.000ns
Destination Clock: clk_BUFGP rising at 10.000ns
Clock Uncertainty: 0.061ns

Clock Uncertainty: 0.061ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
Total System Jitter (TSJ): 0.070ns
Total Input Jitter (TIJ): 0.100ns
Discrete Jitter (DJ): 0.000ns
Phase Error (PE): 0.000ns

Maximum Data Path: back/addressin_1 to back/addressread_0
Location Delay type Delay(ns) Physical Resource
Logical Resource(s)
----------------------------------------------------------------------------------------------
SLICE_X48Y39.BQ  Tcko  0.346  back/addressin<3>
back/addressin_1
SLICE_X36Y42.A2  net (fanout=155)  1.776  back/addressin<1>
SLICE_X36Y42.AMUX  Tilo
seg_mem/BU2/U0/gen_dp_ram.dpram_inst/N86

seg_mem/BU2/U0/gen_dp_ram.dpram_inst/Mram_ram38/DP.HIGH

seg_mem/BU2/U0/gen_dp_ram.dpram_inst/Mram_ram38/F7.DP
SLICE_X37Y44.A2  net (fanout=1)  0.892
seg_mem/BU2/U0/gen_dp_ram.dpram_inst/N87
SLICE_X37Y44.AMUX  Tilo  0.288
seg_mem/BU2/U0/gen_dp_ram.dpram_inst/inst_LPM_MUX12_5_f7

seg_mem/BU2/U0/gen_dp_ram.dpram_inst/inst_LPM_MUX12_6

seg_mem/BU2/U0/gen_dp_ram.dpram_inst/inst_LPM_MUX12_5_f7
SLICE_X45Y36.A3  net (fanout=1)  0.917
seg_mem/BU2/U0/gen_dp_ram.dpram_inst/inst_LPM_MUX12_5_f7
SLICE_X45Y36.A  Tilo  0.080  seg_data<4>

seg_mem/BU2/U0/gen_dp_ram.dpram_inst/dpra<10>41
SLICE_X46Y27.A4  net (fanout=4)  0.754  seg_data<4>
SLICE_X46Y27.CMUX  Topac  0.513  back/mincnt_sub0000<7>
back/Msub_mincnt_sub0000_lut<4>
back/Msub_mincnt_sub0000_xor<7>
SLICE_X46Y23.B1  net (fanout=3)  0.914  back/mincnt_sub0000<6>
SLICE_X46Y23.B  Tilo  0.080  back/N60
back/bwe_not000121
SLICE_X46Y23.A5  net (fanout=21)  0.202  back/N60
SLICE_X46Y23.A  Tilo  0.080  back/N60
back/Mcount_addressread_lut<0>
SLICE_X51Y17.A1  net (fanout=1)  0.786  back/Mcount_addressread_lut<0>
SLICE_X51Y17.AMUX  Topaa  0.298  back/addressread<3>
back/Mcount_addressread_lut<0>_rt
back/Mcount_addressread_cy<3>
SLICE_X44Y21.C2  net (fanout=1)  0.744  back/Mcount_addressread
SLICE_X44Y21.CLK  Tas  0.006  back/addressread<0>
back/addressread_0_rstpot
back/addressread_0

------------------------------------
Total  8.967ns (1.982ns logic, 6.985ns route)
(22.1% logic, 77.9% route)

------------------------------------
Hold Paths: TS_clk = PERIOD TIMEGRP "clk" 10 ns HIGH 50% INPUT_JITTER 0.1 ns;

<table>
<thead>
<tr>
<th>Paths</th>
<th>for</th>
<th>end point</th>
</tr>
</thead>
<tbody>
<tr>
<td>motion_mem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v5_noinit.ram/TR UUE_DP.SINGLE_PRIM18.TDP (RAMB36_X0Y7.ADDRAL12), 1 path</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Slack (hold path): 0.191 ns (requirement - (clock path skew + uncertainty - data path))
Source: motion/writeadd_8 (FF)
Destination: motion_mem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v5_noinit.ram/TR UUE_DP.SINGLE_PRIM18.TDP (RAM)
Requirement: 0.000 ns
Data Path Delay: 0.356 ns (Levels of Logic = 0)
Clock Path Skew: 0.165 ns (0.626 - 0.461)
Source Clock: clk_BUFGP rising at 10.000 ns
Destination Clock: clk_BUFGP rising at 10.000 ns
Clock Uncertainty: 0.000 ns

Minimum Data Path: motion/writeadd_8 to motion_mem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v5_noinit.ram/TR UUE_DP.SINGLE_PRIM18.TDP

<table>
<thead>
<tr>
<th>Location</th>
<th>Delay type</th>
<th>Delay(ns)</th>
<th>Physical Resource</th>
<th>Logical Resource(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLICE_X9Y38.AQ</td>
<td>Tcko</td>
<td>0.300</td>
<td>motion/writeadd&lt;10&gt;</td>
<td>motion/writeadd_8</td>
</tr>
<tr>
<td>RAMB36_X0Y7.ADDRAL12</td>
<td>net (fanout=1)</td>
<td>0.296</td>
<td>motion/writeadd&lt;8&gt;</td>
<td></td>
</tr>
<tr>
<td>RAMB36_X0Y7.CLKARDCLKL</td>
<td>Trckc_ADDRA (-Th)</td>
<td>0.240</td>
<td>motion_mem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v5_noinit.ram/TR UUE_DP.SINGLE_PRIM18.TDP</td>
<td></td>
</tr>
</tbody>
</table>

motion_mem/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v5_noinit.ram/TR UUE_DP.SINGLE_PRIM18.TDP

Total 0.356 ns (0.060 ns logic, 0.296 ns route)
(16.9% logic, 83.1% route)
All constraints were met.

Data Sheet report:
-------------------
All values displayed in nanoseconds (ns)

Clock to Setup on destination clock clk
---------------------------------------
| Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------------------------------
clk | 9.063 | | | |
---------------------------------------

Timing summary:
-------------------
Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)
Constraints cover 3520888 paths, 0 nets, and 16104 connections
Design statistics:
  Minimum period: 9.063ns\{1\}  (Maximum frequency: 110.339MHz)
CHAPTER 5 : RTL CODES

5.1. Top

`timescale 1ns / 1ps
module memtop(
    input clk,
    input [7:0] frame_data,
    output [20:0] frame_add,
    input reset,
    input [7:0] outadd,
    output [7:0] outdata,
    input start,
    output done
);

//wire [15:0] frame_add;
wire [7:0] seg_write_data, seg_data, preback_write_data, preback_read_data, back_write_data, back_read_data, seg_to_motion_data;
wire seg_write_enable, pwe, bwe, car2diffdata, car2diffwe, diff2cardata, dina, dout;
wire [10:0] seg_read_add, seg_write_add, back_read_add, seg_read_write_add;
wire [10:0] backadd;
wire [10:0] motion_write_add;
wire [14:0] pbaddwrite, pbaddrread;
wire [10:0] car2diffadd;
wire motion_we;
wire [8:0] carwriteadd;
wire [7:0] carwritedata, car_to_speed_data, speed_wr_add, speed_wr_data;
wire speed_wr_en;
wire [8:0] car_to_speed_add;
wire start_read, start_back, start_motion, start_car, start_speed;
//wire [10:0] seg_to_motion_add;

//frame read
read_data read_frame(  
    .datain(frame_data), // read frame  
    .clk(clk),  
    .reset(reset),  
    .addressread(frame_add), // [15:0]  
    .addresswrite(seg_write_add), // first seg write [13:0]  
    .dataout(seg_write_data),  
    .wren(seg_write_enable),  
    .start(start),  
    .frame_end_flag(start_back)
);
//2
//seg mem
assign seg_read_write_add = seg_write_enable? seg_write_add : back_read_add;
segmem seg_mem (  
  .a(seg_read_write_add), // Bus [10 : 0]  
  .d(seg_write_data), // Bus [7 : 0]  
  .dpr(a(seg_read_add)), // Bus [10 : 0]  
  .clk(clk),  
  .we(seg_write_enable),  
  .spo(seg_to_motion_data), // Bus [7 : 0]  
  .dpo(seg_data)); // Bus [7 : 0]

//3
//background
background back(  
  .clk(clk),  
  .seg_data(seg_data), // data from segment for this frame  
  .data_out(preback_write_data),  
  .addressin(seg_read_add), // segment read address  
  .addresswrite(pbaddrwrite),  
  .addressread(pbaddrread),  
  //input [7:0]  
  .data_read(preback_read_data), // data from memory with background info  
  .reset(reset),  
  .we(pwe),  
  .backaddress(backadd),  
  .backdata(back_write_data),  
  .bwe(bwe),  
  .start(start_back),  
  .frame_done(start_motion)
);

//4
//preback mem
prebackmem preback_mem (  
  .clka(clk),  
  .wea(pwe), // Bus [0 : 0]  
  .addra(pbaddrwrite), // Bus [14 : 0]  
  .dina(preback_write_data), // Bus [7 : 0]  
  .clkb(clk),  
  .addrb(pbaddrread), // Bus [14 : 0]  
  .doutb(preback_read_data)); // Bus [7 : 0]

//5
//back mem
background_mem backgroundmem (  
  .a(backadd), // Bus [9 : 0]
.d(back_write_data), // Bus [7 : 0]
dpra(back_read_add), // Bus [9 : 0]
.clk(clk),
.we(bwe),
.dpo(back_read_data)); // Bus [7 : 0]

//6
//motion

motion motion(
.clk(clk),
.reset(reset),
.backdata(back_read_data),
.segdata(seg_to_motion_data)//seg mem
.backreadadd(back_read_add),//11 # need to be conected to segmem too
//.enable(1'b1),//??# need enable
.dataout(motion_write_data),//1
.we(motion_we),
.writeadd(motion_write_add),
.start(start_motion),
.frame_done_motion(start_car)));//11

//7
//motion mem

motionmem motion_mem(
.clka(clk),
.wea(motion_we), // Bus [0 : 0]
.addra(motion_write_add), // Bus [10 : 0]
.dina(motion_write_data), // Bus [0 : 0]
.douta(open), // Bus [0 : 0] #open
.clkb(clk),
.web(car2diffwe), // Bus [0 : 0]
.addrb(car2diffadd), // Bus [10 : 0]
.dinb(car2diffdata), // Bus [0 : 0]
.doutb(diff2cardata));

//8
//cars

car cars(
.clk(clk),
.reset(reset),
.diffrwadd(car2diffadd),

diff
5.2. Read Frame

module read_data(
    input [7:0] datain, // read frame
input clk,
    input reset,
output reg [20:0] addressread,
    output reg [10:0] addresswrite,
    output reg [7:0] dataout,
    output reg wren,
    input  start,
    output reg frame_end_flag
);

parameter xpix=160, ypix=120, xs=3, ys=3, xubpix=0, xuepix=160, yubpix=0,
yuepix=120; //segment x width , segment y width
// x used begin pix , x used end pix
reg [2:0] sxcntr; // segment cntr
reg [2:0] sycntr;
reg [9:0] xcntr, ycntr;
reg segment_end_flag, sum_flag;
reg [5:0] framecnt;
reg [7:0] ave;
reg [7:0] l='b0;
reg [12:0] sum;
reg enable, start1, startpe;

always @ ( posedge clk)
begin
    if(!reset)
    begin
        frame_end_flag<=1'b0;
        addressread<=xubpix*yubpix;
        framecnt<='b0;
        xcntr<=xubpix;
        ycntr<=yubpix;
        //addresswrite<='b0;
        sxcntr<='b0;
        sycntr<='b0;
        segment_end_flag<=1'b0;
        framecnt<='b0;
        enable<=1'b0;

        //sum<='b0;

        addresswrite<=1'b0;
        wren<=1'b0;
        sum_flag<=1'b1; // so with first address restore a datain in sum
        ave<='b0;
        sum<='b0;
    end

end
/*if( frame_end_flag==1'b1) 
begin
    addressread<=(xubpix*yubpix)*(framecnt*xpix*ypix);//begining address of the frame
    xcntr<=xubpix;
    ycntr<=yubpix;
    frame_end_flag<='0';
end*/
else if(enable)
begin
    if(sum_flag)
        begin
            sum<=datain;
            sum_flag<=1'b0;
            wren<=1'b0;
        end
    else if(segment_end_flag)
        begin
            k<=1'b1;
            segment_end_flag<=1'b0;
            dataout<=(sum+datain)/16;
            //dataout<=ave;
            sum_flag<=1'b1;// cause this datain belong to previous
            //sum<=datain;
            if(addresswrite>=11'b10010101111)//1200
                addresswrite<='b0;
            else
                addresswrite<=addresswrite+1'b1;
            wren<=1'b1;
        end
    else
        begin
            wren<=1'b0;
            l<=l+1'b1;
            sum<=sum+datain;
        end
    if(sycntr==ys && sxcntr==xs)
        begin
            segment_end_flag<=1'b1;
            sxcntr<='b0;
            sycntr<='b0;
            //ave<=sum/20;
if(ycntr>=yuepix-ys && xcntr>= xuepix-xs)//when frame finish
begin
  d<=1'b1;
  frame_end_flag<=1'b1;
  enable <=1'b0;
  framecnt<=framecnt+1'b1;
  addressread<=(xubpix*yubpix)*(framecnt*xpix*ypix);//begining address of the next frame
  xcntr<=xubpix;
  ycntr<=yubpix;
  //frame_end_flag<=1'b0;
  //sxcntr<='b0;
  //sycntr<='b0;
  //segment_end_flag<=1'b1;
end
else if(xcntr >= (xuepix-xs) && sycntr==ys)// at the end of x line
begin
  addressread<=addressread+1'b1+xpix-(xuepix-xubpix);
  xcntr<=xubpix;
  ycntr<=ycntr+1'b1;
  //sxcntr<='b0;
  //sycntr<='b0;
  //segment_end_flag<=1'b1;
end
else begin
  f<=1'b1;
  frame_end_flag<=1'b0;
  addressread<=addressread-xpix*(ys)+1'b1;
  xcntr<=xcntr+1'b1;
  ycntr<=ycntr-ys;
  //sxcntr<='b0;
  //sycntr<='b0;
end
else begin
  g<=1'b1;
  segment_end_flag<=1'b0;
end
if(sxcntr==xs)
begin

    //
    h<=1'b1;
    addressread<=addressread+xpix-xs;
    ycntr<=ycntr+1'b1;
    xcntr<=xcntr-xs;
    sxcntr<='b0;
    sycntr<=sycntr+1'b1;

end
else
begin

    //
    i<=1'b1;
    addressread<=addressread+1'b1;
    ycntr<=ycntr;
    xcntr<=xcntr+1'b1;
    sxcntr<=sxcntr+1'b1;
    sycntr<=sycntr;

end
end
else//not enable
begin
if(startpe)
begin
frame_end_flag<=1'b0;
enable<=1'b1;
end
end
end

always @(posedge clk)
begin
if(!reset)
begin
startpe<=1'b0;
start1<=1'b0;
end
else
begin
start1<=start;
startpe<=!start1 & start;
end
end
endmodule
5.3. Background

module background(  
  input clk,  
  input [7:0] seg_data, // data from segment for this frame  
  output reg [7:0] data_out,  
  output reg [10:0] addressin, // segment read address  
  output reg [14:0] addresswrite,  
  output reg [14:0] addressread,  
  //input [7:0]  
  input [7:0] data_read, // data from memory with background info  
  input reset,  
  output reg we,  
  output reg [10:0] backaddress,  
  output reg [7:0] backdata,  
  output reg bwe,  
  input start,  
  output reg frame_done  
);  
parameter segmemsize=10 , pbmemsize=14;  
//reg [7:0] buff [memsize-1:0];  
reg [4:0] cntr;  
reg [7:0] cnt,value,mincnt,maxcnt,maxvalue;  
//reg even;  
reg found,seg_done,w2flag,enable,efound;  
reg [pbmemsize:0] valueaddress, minaddress,maxaddress,cntaddress;  
reg [4:0] a;  
//reg frame_done;  
reg start1 , startpe;  
//wire[8:0] seg_d;  
always @(posedge clk or negedge reset)  
begin  
  if(!reset)  
    begin  
      mincnt<='b0;  
      maxcnt<='b0;  
      maxvalue<='b0;  
      found<='b0;  
      seg_done<='b1;  
      w2flag<='b0;  
      minaddress<='b0;  
      maxaddress<='b0;  
      cntaddress<='b0;  
      data_out<='b0;  
      addressin<='b0;  
    end  
  end  
end

47
addresswrite<='b0;
addressread<='b0;
backaddress<='b0;
backdata<='b0;
cntr<='b0;
we<='1'b0;
bwe<='1'b0;
value<='b0;
valueaddress<='b0;
cnt<='b0;
a<='b0;
frame_done<='b0;
efound<='b0;
//even<='1'b0;

//buff<='8'b0;// ??

end
else if(enable)
begin

//if(cntr>=5'b10011)
//begin

if(addressread[0]== 1'b1)
begin
value<=data_read;
valueaddress<=addressread;
cntr<=cntr+1'b1;
if(efound)
begin
we<='1'b1;
addresswrite<=addressread;
efound<=1'b0;
end

else  if((seg_data-data_read<=8'b00001010 || data_read-
seg_data<=8'b00001010)  && seg_done)//don't need to update value , just increment the
cnt

  //if((seg_data>=data_read  || seg_data<=data_read)  &&
seg_done)//don't need to update value , just increment the cnt
begin

begin

end

end

end

end
if(cntr>=5'b10100)// if I find data in last cell
it takes 2 more clk to write and addressread going to be messed up
efound<=1'b1;
else
begin
found<=1'b1;
addressread<=addressread+1'b1;// need to be reset at the end of frame
end
seg_done<=1'b0;
bwe<=1'b0;
else
begin
if(cntr>=5'b10011 )
//if(cntr>=5'b01111 )//because I need to
write 2 data, I'm gonna start here
//19 need to do the whole thing in 22 clk this
is for last 2 clk
begin
if(seg_done)
begin
we<=1'b1;
w2flag<=1'b1;
seg_done<=1'b0;// if dont
reset, w2flag don't enter
addresswrite<=minaddress-1'b1;//min address should be data address
data_out<=seg_data;
end
else if(w2flag)//
begin
we<=1'b1;
//cntr<='b0;
addresswrite<=minaddress;
data_out<=8'b00000001;
w2flag<=1'b0;
seg_done<=1'b1;
mincnt<=8'b11111111;
maxcnt<='b0;
cntr<='b0;
backaddress<=addressin;
bwe<=1'b1;
backdata<=maxvalue;
end
else
begin
we<=1'b1;
addresswrite<=minaddress;
data_out<=8'b00000001;
w2flag<=1'b0;
seg_done<=1'b1;
mincnt<=8'b11111111;
maxcnt<='b0;
cntr<='b0;
backaddress<=addressin;
bwe<=1'b1;
backdata<=maxvalue;
end
else
begin
we<=1'b1;
addresswrite<=minaddress;
data_out<=8'b00000001;
w2flag<=1'b0;
seg_done<=1'b1;
mincnt<=8'b11111111;
maxcnt<='b0;
cntr<='b0;
backaddress<=addressin;
bwe<=1'b1;
backdata<=maxvalue;
end
else
begin
we<=1'b1;
addresswrite<=minaddress;
data_out<=8'b00000001;
w2flag<=1'b0;
seg_done<=1'b1;
mincnt<=8'b11111111;
maxcnt<='b0;
cntr<='b0;
backaddress<=addressin;
bwe<=1'b1;
backdata<=maxvalue;
if(addressin==11'b10010101111)//1199 1001010111
addressin<='b0;
else
    addressin<=addressin+1'b1;
end
else
    begin
        we<=1'b0;
        seg_done<=1'b1;
        cntr<='b0;
        backaddress<=addressin;
        maxcnt<='b0;
        mincnt<=8'b11111111;
        bwe<=1'b1;
        backdata<=maxvalue;
    end

if(addressin==11'b10010101111)//1199 10010101111
    addressin<='b0;
else
    addressin<=addressin+1'b1;
end

if(addressread>=15'b101110110111111)//24000(20*1200)-1
    begin
        addressread<='b0;
        cntr<='b0;
        frame_done<=1'b1;
        enable<=1'b0;
    end
    else
        begin
            addressread<=addressread+1'b1; // need to be reset at the end of frame
        end
end
end  
else//odd  
begin  
cnt<=data_read;  
//even<=1'b1;  
cntaddress<=addressread;  

bwe<=1'b0;  
cntr<=cntr+1'b1;  
if(found)  
begin  
//mincnt<='b0;  
//maxcnt<='b0;  
found<=1'b0;  
//addressread<=addressread+5'b10100-  
(2*cntr);//address of next read  

if(data_read!=8'b11111111)  
begin  
addresswrite<=addressread-  
1'b1;//cntr==255  
// already got incremented, don't  
need to increment again  
we<=1'b1;  
data_out<=data_read+1'b1;  
//addressread<=addressread+1'b1;  
end  
else  
we<=1'b0;  
addressread<=addressread+1'b1;  
//else  
//addressread<=addressread+1'b1;  
end  
else// not found  
begin  
we<=1'b0;  
end  
if(data_read>=maxcnt)  
begin

51
maxcnt<=data_read;
maxaddress<=addressread-1'b1;
maxvalue<=value;

end
if(data_read<=mincnt)
begin
    a<=a+1'b1;
    mincnt<=data_read;
    minaddress<=addressread-1'b1;//#
end
addressread<=addressread+1'b1;

end

//we
// max and min =256
// update the background

end
else//not enbale
begin
    if(startpe)
        begin
            enable<=1'b1;
            frame_done<=1'b0;

end
always @(posedge clk)
begin
    start1 <= start;
    startpe <= !start1 & start;
end

endmodule

5.4. Motion

module motion(
    input clk,
    input reset,
    input [7:0] backdata,
    input [7:0] segdata,
    output reg [10:0] backreadadd,
    output reg dataout,
    output reg we,
    output reg [10:0] writeadd,
    input start,
    output reg frame_done_motion
);

//reg second_flag;
//reg [7:0] pdata;
reg enable, start1, startpe;

always @(posedge clk or negedge reset)
begin
    if(!reset)
    begin
        backreadadd <= 'b0;
        writeadd <= 'b0;
    end

    end

end
else if( enable)
begin
    we<=1'b1;
    if(backdata<=8'b00010000)
        if(segdata<=8'b00010100)
            begin
                dataout<=1'b0;
            end
        else
            dataout<=1'b1;
    else if(backdata>=8'b11110000)
        if(segdata>=8'b11110100)
            dataout<=1'b0;
        else
            dataout<=1'b1;
    else if(backdata+5'b10000>=segdata && backdata-5'b10000<=segdata)
        begin
            dataout<=1'b0;
        end
    else
        dataout<=1'b1;
end
if(backreadadd>=11'b10010110000)
begin
    enable<=1'b0;
    backreadadd<='b0;
    //writeadd<='b0;
    we<=1'b0;
    frame_done_motion<=1'b1;
end
else
begin
    backreadadd<=backreadadd+1'b1;
    writeadd<=backreadadd;
end
end
else
begin
    we<=1'b0;
    if(startpe)//start positive edge
        begin
            enable<=1'b1;
            frame_done_motion<=1'b0;
        end
    end
end

always @(posedge clk)
begin
    start1<=start;
    startpe<=!start1 & start;
end
endmodule

5.5.Cars

module car(
    input clk,
    input reset,
    output reg [10:0] diffrwadd,
    output reg diffwritedata,
    output reg [8:0] carwriteadd,
    output reg [7:0] carwritedata,
    output reg carwren,
    input diffdataread,
    input start,
    output reg frame_done
);

parameter xpix=40 , ypix=30;

reg [22:0] fifoin;
wire [22:0] fifoout;
reg [10:0] lastadd,curadd;
reg fifowren,fiforden,fiforead,first,fiforead2,frame_complete,write_done,fiforead3;
wire fifofull,fifoempty;
reg waitflag;
reg [1:0] w2,w3;
reg [3:0] w,a,latencycntx,latencycnty;
reg [5:0] x,y,xmax,ymax,ymin,xmin,lx,ly;
reg [6:0] cnt,divis;
wire [6:0] frac;
reg enable,startpe,start1;
//reg a,b,c,d,e,f,g,h,i,j,k;
reg [11:0] xsum,ysum,divid;
wire [11:0] quo;
wire rfd;


fif fifo (  
  .clk(clk),  
  .rst(~reset),  
  .din(fifoin), // Bus [22 : 0]  
  .wr_en(fifowren),  
  .rd_en(fiforden),  
  .dout(fifoout), // Bus [22 : 0]  
  .full(fifofull),  
  .empty(fifoeempty));

div division (  
  .clk(clk),  
  .rfd(rfd),  
  .dividend(divid), // Bus [11 : 0]  
  .divisor(divis), // Bus [6 : 0]  
  .quotient(quo), // Bus [11 : 0]  
  .fractional(frac)); // Bus [6 : 0]

always @(posedge clk or negedge reset) begin
  if(!reset) begin
    a<='b0;
    xsum<='b0;
    divid<='b0;
    divis<='b0;
    ysum<='b0;
    waitflag<='b0;
    xmax<='b0;
    latencycntx<='b0;
    latencycnty<='b0;
    ymax<='b0;
    ymin<='b0;
    xmin<='b0;
    first<='b0;
    frame_done<='b0;
    fifowren<='b0;
    //ififorden<='b0;
    write_done<='b0;
    //ififowren<='b0;
    fifoin<='b0;
    frame_complete<='b1'b0;
    frame_done<='b1'b0;
    write_done<='b1'b0;
    w<='b0;
  end
w2<='b0;
w3<='b0;
fiforead<='b0;
fiforead2<='b0;
fiforead3<='b0;
y<='b0;
x<='b0;
xmax<='b0;
cnt<='b0;
fiforden<='b0;
diffrwadd<='b0;
diffwritedata<='b0;
diffwren<='b0;
carwriteadd<='b0;
carwritedata<='b0;
carwren<='b0;
end
else if(enable)
begin
if(!frame_done)
begin

if(first)
begin

if(diffdataread)
begin

waitflag<=1'b0;
case (w)//write 3 address to fifo
4'b0000: begin
diffwren<=1'b0;
if(y!=6'b000000)
begin
w<=4'b0001;
fifoin[10:0]<=(curadd-xpix);//above
1'b1;
fifoin[22:17]<=y-

1'b1;
fifoin[16:11]<=x;
fifowren<=1'b1;
end
else
begin
w<=4'b0011;
end
end
4'b0001: begin
  w<=4'b0010;
  if(x!=6'b000000)//Y already checked in state 0
  then
  begin
    fifoin[10:0]<=curadd-xpix-1'b1;//above left
    fifoin[22:17]<=y-1'b1;
    fifoin[16:11]<=x-1'b1;
    fifowren<=1'b1;
  end
  else
  fifowren<=1'b0;
end

4'b0010: begin
  w<=4'b0011;
  if(x!=xpix-1)
  begin
    fifoin[10:0]<=curadd-xpix+1'b1;//below
    fifoin[22:17]<=y;
    fifoin[16:11]<=x+1'b1;
    fifowren<=1'b1;
  end
  else
  fifowren<=1'b0;
end

4'b0011: begin
  //fifowren<=1'b1;
  //diffrwadd<=curadd;
  if(y!=ypix-1)
  begin
    w<=4'b0100;
    fifoin[10:0]<=curadd+xpix;//below
  end
end
fifoin[22:17] <= y+1'b1;
fifoin[16:11] <= x;
fifowren <= 1'b1;
end

else
begin
fifowren <= 1'b0;
w <= 4'b0110;
end
end

4'b0100: begin
w <= 4'b0101;
if(x != 6'b000000)
begin
fifoin[10:0] <= curadd+xpix-1'b1://below right
fifoin[22:17] <= y+1'b1;
fifoin[16:11] <= x-1'b1;
fifowren <= 1'b1;
end
else
fifowren <= 1'b0;
end

4'b0101: begin
w <= 4'b0110;
diffwren <= 1'b0;
if(x != xpix-1'b1)
begin
fifoin[10:0] <= curadd+xpix+1'b1://below left
fifoin[22:17] <= y+1'b1;
fifoin[16:11] <= x+1'b1;
fifowren <= 1'b1;
end
else
fifowren <= 1'b0;
end

4'b0110: begin

59
w<=4'b0111;

//replace the 1 in motion pic with 0 so after state machine
done
again
    //we don't repeat all of this
diffwritedata<=1'b0;
diffwren<=1'b1;  //#read first
    //diffwritedata<=1'b0;
    //diffwren<=1'b0;
    //fifowren<=1'b1;
    if(x!=6'b000000)
        begin
            fifoin[10:0]<=curadd-1'b1;//left
            fifoin[22:17]<=y;
            fifoin[16:11]<=x-1'b1;
            fifowren<=1'b1;
            end
        else
            fifowren<=1'b0;
            end

4'b0111: begin
    cnt<=cnt+1'b1;
    w<=4'b1000;
    fifowren<=1'b0;
    diffwren<=1'b0;
    
    if(x!=xpix-1)//right
        begin
            diffrwadd<=curadd+1'b1;//right
            curadd<=curadd+1'b1;
            x<=x+1'b1;
            end
        else if(x!=6'b000000)
            begin
                diffrwadd<=curadd-1'b1;//left
                curadd<=curadd-1'b1;
                x<=x-1;
                end
        end
    xsum=xsum+x;
ysum<=ysum+y;
end

4'b1000: begin // diffrwadd will be set in this clock cycle and in next // clock cycle its data gonna appear so I need // one clock delay
w<=4'b0000;
end

default :begin
w<=4'b0000;
fifowren<=1'b0;
fifoin<=fifoin;
endcase
end

end
else//diffdataread==0
begin
if(fifoempty)
begin
if(cnt>=7'b0000100)// to reduce the noise, I assumed if there is // less than 5 pixel in a area it's not a noise begin
case(w2)
2'b00: begin
divid<=xsum;
divis<=cnt;
latencycntx<=latencycntx+1'b1;
if(&latencycntx)
begin
carwritedata <= quo;//x

carwriteadd <= carwriteadd + 1'b1;

carwren <= 1'b1;

w2 <= 2'b01;

end

else

carwren <= 1'b0;

divid <= ysum;

dis <= cnt;

latencycnty <= latencycnty + 1'b1;

if (&latencycnty)

begin

carwritedata <= quo;//y

carwriteadd <= carwriteadd + 1'b1;

carwren <= 1'b1;

w2 <= 2'b10;

end

else

carwren <= 1'b0;

end

2'b01: begin

divid <= ysum;

dis <= cnt;

latencycnty <= latencycnty + 1'b1;

end

2'b10: begin

2'b10: begin

carwritedata <= cnt;//cnt

carwriteadd <= carwriteadd + 1'b1;

end
carwren<=1'b1;

w2<=2'b11;

2'b11: begin

carwren<=1'b0;
w2<=2'b00;
first<=1'b0;
cnt<='b0;

x<=lx;
y<=ly;

diffrwadd<=lastadd;
xsum<='b0;
ysum<='b0;

end

default :begin

w2<=2'b00;
carwren<=1'b0;

end

endcase
end//if cnt<5
else if(waitflag)//cnt<5

begin

carwren<=1'b0;
first<=1'b0;
cnt<='b0;
x<=lx;
y<=ly;
diffrwadd<=lastadd;
xsum<='b0;
ysum<='b0;
waitflag<=1'b0;
end

else

begin

end
waitflag <= 1'b1;
end

end//empty

else//not empty
begin
																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																															}

else//first
	if(diffdataread)
begin
first<=1'b1;
xsum<=x;
ysum<=y;
if(diffrwadd!=11'b00000000000)
begin
  curadd<=diffrwadd-1'b1;
diffrwadd<=diffrwadd-1'b1;//because the data have
  one cycle latency
  //address already incremented
  lastadd<=diffrwadd-1'b1;
end
else
begin
  curadd<=diffrwadd;
diffrwadd<=diffrwadd;
lastadd<=diffrwadd;
end
if(x==6'b000000 && y==6'b000000)
begin
  x<='b0;
y<='b0;
xsum<='b0;
ysum<='b0;
ly<='b0;
end
else if( x==6'b000000)
begin
  lx<=xpix-1'b1;
x<=xpix-1'b1;
xsum<=xpix-1'b1;
y<=y-1'b1;
ly<=y-1'b1;
ysum<=y-1'b1;
end
else
begin
  ysum<=y;
  lx<=x-1'b1;
x<=x-1'b1;
  ly<y;
  ysum<y;
  xsum=x-1'b1;
end
else
begin
if(diffrwadd>=11'b1001010111)
begin
frame_done<=1'b1;
diffrwadd<='b0;
x<='b0;
y<='b0;
end
else
diffrwadd<=diffrwadd+1'b1;
if(x>=xpix-1)
begin
x<='b0;
if(y>=ypix-1)
begin
frame_done<=1'b1;
diffrwadd<='b0;
x<='b0;
y<='b0;
end
else
y<=y+1'b1;
end
else
x<=x+1'b1;
end
end
else//frame_done

if(write_done)
begin
carwriteadd<=carwriteadd;
carwren<='b0;
end
else
begin
if(carwriteadd==9'b011111111)
carwriteadd==9'b111111111)
carwriteadd==9'b0111111111
begin
  write_done<=1'b1;
carwren<=1'b0;
  enable<=1'b0;
  frame_complete<=1'b1;
end
else
begin
  carwren<=1'b1;
carwritedata<='b0;
carwriteadd<=carwriteadd+1'b1;
  endadd<=endadd+1'b1;
end
end
else
if(startpe)
begin
  frame_done<=1'b0;
  frame_complete<=1'b0;
  write_done<=1'b0;
  enable<=1'b1;
end
end
always @(posedge clk or negedge reset)
begin
  if(!reset)
    begin
      start1<=1'b0;
    end
  else
    begin
      start1<=start;
      startpe<=!start1 & start;
    end
end
endmodule

5.6. Speed
module speed( // for half a picture there is 16 frame with 70mph speed and 24 frame per second
    input clk,
    input reset,
    input [7:0] cardata,
    output reg [8:0] caradd,
    output reg [7:0] speedadd,
    output reg [7:0] speeddata,
    output reg speeden,
    input start,
    output reg all_done
);

    reg [7:0] cx,cy,ccnt,speed,ypix,px,py;

    reg searchdone,cur,frame_done,fclr ,start1,startpe,enable ;
    reg [1:0] rd   ;
    reg [2:0] rd2;
    reg [8:0] lastrd,sadd,minadd   ;
    reg [5:0] a,b,c,d,e,g,h,f='b0;
    reg [5:0] xdis,ydis,min;

    always @(posedge clk or negedge reset)
    begin
        if(!reset)
            begin
                px<='b0;
                sadd<='b0;
                py<='b0;
                xdis<='b0;
                ydis<='b0;
                min<=6'b111111;
                caradd<='b0;
                speedadd<=b0;
                speeddata<=b0;
                rd<=2'b00;
                cur<='b0;
                fclr<='b0;
                speed<='b0;
                rd2<=3'b0;
                cx<='b0;
                cy<='b0;
                ccnt<='b0;
                lastrd<='b0;
                enable<=1'b0;
                frame_done<=1'b0;
            
        end

        // remainder of the code
searchdone<='b0;
a<='b0;//
minadd<='b0;

b<='b0;//@
c<='b0;//@
d<='b0;//@
e<='b0;//@
f<='b0;//@
g<='b0;
h<='b0;

end
else if(enable)
begin
if(!frame_done)
begin
//b<='b+1'b1;
if(!searchdone)
begin

if(!cur)
begin

case (rd)

2'b00 : begin
caradd<='lastrd;
rd<='b01;
end

2'b01 : begin
caradd<='caradd+1'b1;
cx<='cardata;
rd<='b10;
end

2'b10 : begin
caradd<='caradd+1'b1;
cy<='cardata;
rd<='b11;
end

2'b11 : begin
//caradd<='caradd+1'b1;
if(caradd[8])
begin
caradd<=b0;
if(!cardata)
begin
  lastrd<=b0;
  frame_done<=1'b1;
end
else
  lastrd<=caradd+1'b1;
end
else//9=0
begin
  caradd<=9'b100000000;
  if(!cardata)
    begin
      lastrd<=9'b100000000;
      frame_done<=1'b1;
    end
  else
    lastrd<=caradd+1'b1;
  end
ccnt<=cardata;
rd<=2'b00;
cur<=1'b1;
//lastrd<=caradd+1'b1;
endcase
//end
end//current data has been read
else//cur
begin
  // I should start reading the next frames address
  //read x, y, cnt and compare them to current values
  //
  case(rd2)
  3'b000: begin//x
    speeden<=1'b0;
speeddata<=speeddata;//##
    if(!cardata)
      begin
        rd2<=3'b001;
    end
endcase
caradd <= caradd + 1'b1;
end
else if((cardata-cx <= 8'b00001000) ||
(cx-cardata <= 8'b00001000))
  // (cardata <= 8'b00010111 &&
cardata > cx) || (cardata >= 8'b00010111 && cardata < cx))
  // (cardata >= cx-8'b00001000 &&
cardata <= 8'b00001000) || (!cardata) ||
begin
rd2 <= 3'b001;
caradd <= caradd + 1'b1;
sadd <= caradd;
a <= a + 1'b1;
if(cx-cardata <= 8'b0001000)
  begin
    xdis <= cx-cardata;
caradd: // seg23 is middle of road, in th
  end
else if(cardata-
  cx <= 8'b00001000)
    begin
    xdis <= cardata-cx;
  end
else
  begin
  xdis <= xdis;
  end
else
  begin
  rd2 <= 3'b000;
if(cardadd[7:0] >= 8'b11111100)
  searchdone <= 1'b1;
else
  caradd <= caradd + 2'b11;
end
3'b001: begin
  speeden <= 1'b0;
speeddata <= speeddata://
##
if(!cardata)
    begin
        rd2<=3'b010;
        caradd<=caradd+1'b1;
        end
    else if((cardata>cy &&
        (cardata<='8'b00001000 &&
        cy<='8'b00001000))
        begin
            rd2<=3'b010;
            caradd<=caradd+1'b1;
        end
    else
        begin
            rd2<=3'b000;
        end
end
3'b010: begin
    if(!cardata)
        begin
            if(min!=6'b111111)//didn't find anything min is still the same
                begin
                    caradd<=minadd;
                    min<='6'b111111;
                    rd2<='3'b011;
                    f<='f+1'b1;
                end
            else
                begin
                    rd2<='3'b000;
                    searchdone<=1'b1;
                    e<='e+1'b1;
                end
        //searchdone<=1'b1;
        //rd2<='3'b011;
    end
end
//speeddata<='b0;

//speeden<=1'b1;//what we are writing ?

/*if(caradd[8])
begin
    caradd<='b0;
    end
else
    begin

    caradd<=9'b100000000;
    end
*/

end
else
begin
    if(xdis<min)
    begin
        min<=xdis;
        minadd<=sadd;
        g<=g+1'b1;
        end
    h<=h+1'b1;
    rd2<=3'b000;
    caradd<=caradd+1'b1;
    //if((cardata>=ccnt-8'b00011000 && cardata<=ccnt+8'b00011000)||(cardata>=8'b00100000 &&
    ccnt>=8'b00100000)||(cardata<=8'b00011000 && ccnt<=8'b00011000))
    end
3'b011: begin
    speedadd<=speedadd+1'b1;
    speeden<='b0;
    speeddata<=cx://x
caradd<=caradd+1'b1;
    rd2<=3'b100;
    speeden<='b1;
    end
3'b100:begin
    rd2<=3'b101;
    speeddata<=cy://y
    //caradd<=caradd+1'b1;
speed<=(cardata-cy)*5'b10011;//#cardata is previous frame

//need to be justified;#@?
speedadd<=speedadd+1'b1;
speeden<=1'b1;
end

3'b101:begin
rd2<=3'b110;
speeddata<=ccnt;
speedadd<=speedadd+1'b1;
speeden<=1'b1;
end

3'b110:begin
rd2<=3'b111;
speeddata<=speed;
speedadd<=speedadd+1'b1;
speeden<=1'b1;
end

3'b111:begin
rd2<=3'b000;
speeddata<=speeddata;
speeden<=1'b0;
searchdone<=1'b1;
end
case
end
else//searchdone
begin
caradd<=lastrd;
cur<='b0;
searchdone<='b0;
speeddata<=speeddata;//##
end
else//frame_Done
begin
if(speedadd==8'b11111111)
begin
speedadd<='b0;
speeden<='b0;
end
end
searchdone<=b0;
frame_done<=1'b0;
cur<=1'b0;
all_done<=1'b1;
enable<=1'b0;
//lastrd<=##
//speeddata<=speeddata:///##
end
else
begin
speedadd<=speedadd+1'b1;
speeden<=1'b1;
speeddata<=b0;
end
end
end
else if(startpe)
begin
enable<=1'b1;
all_done<=1'b0;
end
end
always @(posedge clk)
begin
start1<=start;
startpe<=!start1 & start;
end
endmodule

5.7.Test Bench

module ext_mem4;

    // Inputs
    reg clk;
    reg reset;
    reg [7:0] mem [1939200:0];
    wire [7:0] frame_data;
    //reg [7:0] a;
    wire [20:0] frame_add;

reg [7:0] outadd;
wire [7:0] outdata;
reg start,swrite,first;
wire done;
reg [7:0] x,y,cnt,bx,ex,by,ey,filewrd;
reg modata;
reg [9:0] add3;
reg beg,moti;
integer frame_cnt,add,file,xcnt,ycnt,file2,a,b,c,file3,file4,file5,file6,file7;

// Instantiate the Unit Under Test (UUT)
/*memtop uut (  
  .clk(clk),  
  .reset(reset)  
);*/

memtop uut(
  .clk(clk),
  .frame_data(frame_data),
  .frame_add(frame_add),
  .reset(reset),
  .outadd(outadd),
  .outdata(outdata),
  .start(start),
  .done(done)
);

initial begin
clk<=1'b1;
forever #100 clk<=~clk;
end

initial begin
  // Initialize Inputs
  a<=0;
b<=0;
c<=0;
clk = 0;

  moti<=0;
  first<=0;
  reset = 0;
  frame_cnt<=b0;
beg<=1'b0;
add<=0;
swrite<=0;
xcnt<=0;
ycnt<=0;
add3<=0;
// Wait 100 ns for global reset to finish
#1000;
reset=1'b1;

#100
start<=1'b0;
#1000
start<=1'b1;
#1000
start<=1'b0;
/*#130000
start<=1'b1;
#1000
start<=1'b0;
#130000
start<=1'b1;
#1000
start<=1'b0;*/
// Add stimulus here

end
initial $readmemh("romhexnew.txt",mem);
initial file3=$fopen("motionpicall.txt");
initial file4=$fopen("backpicall.txt");
initial file5=$fopen("segpicall.txt");
initial file6=$fopen("carpicall.txt");
initial file7=$fopen("speeds.txt");

//always @(*)
assign frame_data=mem[(frame_cnt*15'b100101100000000)+frame_add];
/*
initial begin
    $display("Contents of Mem after reading data file:");
    for (k=0; k<6; k=k+1)
        begin
            #300;
            a<=mem[k];
        end
end */
always @(posedge clk)
begin
    if(uut.cars.carwren)
        $fdisplay(file6,uut.cars.carwriteadd","uut.cars.carwritedata","frame_cnt);
always @(posedge clk)
begin
  if(uut.speed.speeden)
    $fdisplay(file7,uut.speed.speedadd","",uut.speed.speeddata","",frame_cnt);
end

always @(posedge clk)
begin
  if(uut.motion.we)
    $fdisplay(file3,uut.motion.writeadd","",uut.motion.dataout);
end

always @(posedge clk)
begin
  if(uut.back.bwe)
    $fdisplay(file4,uut.back.backaddress","",uut.back.backdata);
end

always @(posedge clk)
begin
  if(uut.read_frame.wren)
    $fdisplay(file5,uut.read_frame.addresswrite","",uut.read_frame.dataout);
end

always @(moti)
begin
  a<=a+1'b1;
  if(moti)
    begin
      b<=b+1'b1;
      file2 = $fopen("motionpic3.txt");
    end
    repeat(1200)
    begin
      //force uut.car2diffadd=add3;
      force uut.motion_write_add=add3;
      #1000// modata=uut.diff2cardata;
      modata<=uut.open;
      c<=c+1'b1;
      $fdisplay(file2,modata);
      #100 add3<=add3+1'b1;
    end
end
always @ (done)
begin
if (done)
begin
if(!beg)
begin
if(frame_cnt==99)
begin
moti<=1'b1;
beg<=1'b1;
frame_cnt<=0;
end
else
begin
#1000
start<=1'b1;
end
frame_cnt<=frame_cnt+1'b1;
end
#500
start<=1'b0;
end
end
endmodule
REFERENCES