E90 - Embedded Video and Image Processing using ARM based FPGA

Danielle Sullivan
Swarthmore College
dsulliv2

Eliza Bailey
Swarthmore College
ebailey1

June 18, 2014

Abstract

To meet the demands of the real-time video and image processing (VIP) applications, Field Programmable Gate Array (FPGA) is considered as one of the ideal system architectures for implementation [10]. It is because the customizable hardware of FPGA offers much higher performance than the software-only approaches using microprocessor or DSP. There is a new breed of System-on-Chip (SoC) FPGA which embeds a dual-core ARM processor with FPGA hardware. Combining the benefits of FPGA hardware and CPU programmability, such an SoC FPGA can not only provide parallelization and program speedup that the traditional CPUs computational ability typically prevents, but also offer software intelligence to increase the ease and flexibility to implement complex computational algorithms.

The objectives of this project include the understanding of the multitude of the development system for SoC FPGA and how to utilize it for VIP applications. The culmination of our project is a runtime comparison of a picture-in-picture video image application on the SoC FPGA versus the software-only implementation on CPU via OpenCV's Python platform. The experimental results show that SoC FPGA implementation run about 3000 times faster in moving a picture-in-picture application than a software-only implementation.
# Contents

1 Introduction ............................................. 3  
  1.1 Objectives and Accomplishments .......................... 3  
    1.1.1 Embedded System Design Methodology for SoC FPGA ........ 3  
    1.1.2 Altera VIP Design Suite .................................. 3  
    1.1.3 VIP System Architecture Comparison ....................... 3  

2 Background .............................................. 4  
  2.1 FPGAs ................................................... 4  
  2.2 Image Processing ........................................ 4  
  2.3 SoC Embedded Core Architectures ............................ 4  
  2.4 SoC FPGA Architectures ..................................... 4  

3 Development Process and Tools ............................. 5  
  3.1 Software-only Development .................................. 6  
    3.1.1 OpenCV (Open Source Computer Vision) ................. 6  
  3.2 Development Tools ......................................... 6  
    3.2.1 Quartus II .............................................. 6  
    3.2.2 Quartus Programmer ....................................... 6  
    3.2.3 QSYS .................................................. 6  
    3.2.4 PuTTY and Embedded Shell ................................. 6  

4 Demos ..................................................... 6  
  4.1 Blinking LED ............................................. 6  
  4.2 Video Mixer ............................................... 7  

5 Our Idea .................................................. 7  
  5.1 SoC Picture-in-Picture Implantation ......................... 8  
  5.2 OpenCV - Picture-in-Picture Implementation ............... 8  

6 Results ................................................... 8  
  6.1 OpenCV Timing Analysis ..................................... 8  
  6.2 SoC Timing Analysis ........................................ 9  

7 Discussion ............................................... 10  
  7.1 Analysis ................................................. 10  

8 Conclusion .............................................. 10  

9 Future Work ............................................ 10  

10 Appendix ............................................... 12  
  10.1 Raw Data ................................................. 12  
  10.2 Matlab Code .............................................. 12  
    10.2.1 MIF File Generation .................................... 12  
    10.2.2 Timing Analysis ....................................... 13
1 Introduction

This project is for the fulfillment of our senior project requirements. Topics in Video and Image Processing (VIP) are among our interests. With the introduction of the new generation FPGA, the System-on-Chip (SoC) FPGA, we decided to study how to apply hardware and software co-design techniques to improve the performance of VIP applications.

The SoC FPGA embeds a dual-core ARM processor with FPGA hardware. It presents tremendous opportunities for the evolution of VIP system architecture. However, the design methodology for SoC FPGA development also presents significant challenges.

1.1 Objectives and Accomplishments

Through the completion of this project, we have accomplished three major objectives:

1. Gained familiarity with the design methodology for SoC development platform [5], including the tools for FPGA hardware design, embedded software design, and embedded system integration.

2. Learned how to use the libraries of FPGA design intellectual properties, the Altera VIP package, to facilitate the development of VIP application in SoC FPGA system architecture.

3. Analyzed the tradeoff of the SoC FPGA architecture in comparison to a software-only implementation using CPU.

In the following, we will describe in detail about each of the three key objectives that we have accomplished.

1.1.1 Embedded System Design Methodology for SoC FPGA

It is a common practice for hardware and software development processes of a system to be done concurrently and independently. The Altera software-to-hardware handoff utilities allow separate hardware and software teams to work independently, and follow their own design flows [3]. As stated earlier, one of the goals for the project was to gain experience in the elements of the hardware and software development flow. To achieve this, we worked together to program both sides of the system. This ensured that we gained experiences in all parts of the development process on both the hardware and software sides. The system development flow and the tools [1] used at each stage are discussed in more detail below in the Background section of our paper.

1.1.2 Altera VIP Design Suite

The Altera Video and Image Processing (VIP) Suite [2] is a set of MegaCore functions we used in our project to develop our custom VIP design. We used VIP cores to support our process of building a custom video processing system via Quartus software with Qsys tool. The VIP MegaCore functions use standard interfaces for data input and output, control input and external memory access. These standard interfaces enable video systems to be quickly and easily assembled by connecting the VIP MegaCore functions in the Qsys tool.

1.1.3 VIP System Architecture Comparison

It’s a common adage in the tech world that, "software gets slower faster than hardware gets faster." This colloquial saying coined by Swiss computer scientist, and Turing Award winner Nicholas Wirth, and again brought to light recently by Google's Larry Page, explains the common trend of software outgrowing hardware in size and resulting in added sluggishness to software processes. Knowing this, and experiencing the lag of software programs ourselves, we were motivated to further explore software/hardware (SW/HW) tradeoff. After initial research, we chose to use a popular FPGA architecture Altera's Cyclone V System-on-Chip architecture to support our exploration of the SW/HW Performance tradeoff.

The SoC FPGA capabilities take advantage of the performance of hardware, with an ease and flexibility of Field Programmable Implementation. To explore the SW/HW tradeoff, we implemented the VIP application on an SoC FPGA to compare with a software-only implementation using CPU. The choice of the SoC FPGA is Altera Cyclone V architecture. It combines an embedded on-chip processor with the programmable hardware of an FPGA. The FPGA portion offers user to design low-cost and low-power hardware applications. The embedded ARM-based hard processor system (HPS) is equipped with peripherals and external memory interface. HPS also interfaces the FPGA portion with high-bandwidth interconnects. The development board that we choose is the SoCKit development board [7], which is a good candidate for VIP application because it has Altera Cyclone V SoC FPGA, high-speed DDR3 memory for HPS and FPGA, and video output adapter.
2 Background

2.1 FPGAs

FPGAs offer at least the following four major benefits:

1. Performance: As with nonprogrammable hardware, certain sections of the hardware can be designated for specific tasks. This enables more computing to be done per cycle compared to software implementation on the CPU. By designating portions of the hardware to specific tasks, less overhead is required to allocate resources between processes enabling more efficient computing. As stated above, one of the goals of our project is to measure the performance difference between a software-only implementation and an implementation which combines both SW and HW. Thus, this benefit is of high interest to us.

2. Time to Market: FPGAs enable rapid prototyping, meaning you can test an idea and verify it in hardware without the need for a custom circuit design, such as Application-Specific Integrated Circuit (ASIC). This capability enables modification to hardware design to be made quickly. This characteristic was imperative to the timely completion of our project. The development of our hardware’s design, including the SW/HW interconnects, required many iterations of trial and error.

3. Cost: The expense of making incremental changes to ASIC far exceeds that of changes to incremental changes in FPGA designs. While the initial cost is high (still lower than initial cost of ASIC), updates are free whereas ASIC chips modifications are costly each time.

4. Long-Term Maintenance: Given their programmability, FPGAs provide the capability to make quick modifications to hardware. As a system gets older, functional changes can quickly be made to meet to requirements or protocols. Due to quickly growing nature of the image processing field, this flexibility is imperative for any application to meet modern demands.

2.2 Image Processing

Real-time processing of images is computationally expensive, and growing more so every day. As image technology progresses, pixel number and quality are growing quickly. This progress results in image processing requiring an increasingly large amount of processing power. The quickly developing field of image processing requires a system which can support quickly changing demands of the field. An ideal architecture to support these developments would at the very least have the following capabilities: high performance, design flexibility, and low development cost. We believe these computationally intensive tasks need closely aligned with the capabilities of FPGAs mentioned above, and performance could be greatly improved with FPGAs as mention above.

2.3 SoC Embedded Core Architectures

Single core sequential architectures provide increasing computational power, but their limited capabilities constrain the range of applications they can be used for. The complexity of VIP applications requires more processing power to meet reasonable program runtime constraints. These performance demands make multi-core designs, built from a combination of different types of intellectual property (IP) cores, such as CPU, DSP processors, FPGAs and dedicated functional components. All the IP cores are connected via high speed bus interconnect. Such an SoC embedded core system is a prime candidate for complex applications which require parallelization. Bringing the benefits of each component together and to take advantage of the each of the performance strengths of given hardware, results in a system which can meet the increasing demands on computational power of real time applications. Our choice of the Altera Cyclone V SoC FPGA with the use of VIP cores is an example of the SoC embedded core architectures. It an ideal candidate to meet the demands on image processing programs.

2.4 SoC FPGA Architectures

As mentioned earlier, hardware offers speed-up that is not achievable by most software implementations. One such piece of hardware used to achieve this speedup is the FPGA, which allows us to program and reprogram the hardware directly.

In addition to (and as a result of) the customizable nature of the FPGA, it is incredibly conducive to parallel programming; the programmer can simply program different parts of the board to do different operations, and the result of the process is that it runs in parallel. Clearly, the FPGA has a lot of great applications, and is very useful to speeding up runtime for most programs. The applications we are
specifically interested in for this project are image processing programs, FPGAs parallelizable architecture make them a good candidate to meet the demands of such a computationally intensive problem.

![Figure 1: Detailed Layout of the SoC FPGA](image)

Due to their highly customizable nature, FPGAs can be designed to fit the needs of their designer. The version used in our project is the Embedded ARM based SoC FPGA. This model combines an Altera FPGA with a Hard Processor System (HPS) microcontroller. The HPS in turn comprises of a dual ARM core microprocessor, peripheral elements and memory (See Figure 1).

![Figure 2: Overview of the Layout of the SoC FPGA](image)

The resulting ARM based SoC FPGA (See Figure 2) has a highly configurable interface between its hardware component, i.e. the FPGA, and its software component, i.e. the HPS. The SoC FPGA improves system performance through the use of the high-bandwidth interconnect between its components and reduces power by having a simpler power distribution layout.

3 Development Process and Tools

Figure 3 is a high level depiction of the HW and SW development processes that we followed for our project. On the left side is a representation of the hardware development. It starts with the writing of the Verilog code. Once written, this code is used to program the FPGA portion of the Cyclone V board. Programming is completed via Quartus II tool. After the hardware portion of the board had been programmed, Qsys system integration tool was used to design the interconnection among various IP cores, including the dual-core ARM processor and many VIP cores for image processing.

The simulation and debug phases of hardware development process are crucial to the verification of any successful hardware design. Each of these phases allow for the testing of the system, without the programming the system. The majority of our project closely followed tutorials; so we ourselves did not enter the simulation or debug phases.

Once the Verilog code had been written and verified, and the SoC embedded core system interconnect had been finalized via Qsys; the Verilog files are compiled to create a single .sof file for the project which is used to program the FPGA.

The Software development process, shown on the right side of Figure 3, follows a similar flow with different tools. The C code for our project was developed in the DS-5 tool, which was later also used for debugging the final HW/SW SoC FPGA design. Once the final C code was completely developed on PC, compiled binaries were file transferred to the embedded Linux [6] through an Ethernet connection using the terminal tools PuTTY and the Embedded Shell [9]. This process of compiling code on one system, and transferring compiled binaries to a target machine is known as cross-compilation. The simulation stage of the process is typically done by test-
ing the code on a virtual target. We simply tested our code on the physical device. It is important to note that the hardware and software design flows can happen concurrently - this is known as the HW/SW handoff.

3.1 Software-only Development

3.1.1 OpenCV (Open Source Computer Vision)

The software-only implementation portion of our project was done via OpenCV’s Python interface. OpenCV is a library of programming functions optimized for VIP performance. OpenCV provides a common infrastructure for developers to build upon. This makes code more easily transferable and readable. This common infrastructure with interfaces in widely used languages made our development process much easier through the availability instructional resources.

The process for developing our software code involved setting up the OpenCV environment (downloading the platform), and then simply coding in the text editor of our choice. It is important to note the ease of development in comparison to the SW/HW development. We did not need to familiarize ourselves with new tools for testing and code development. We were able to write high level code, and have our final project within a 5 day period. This is a very short time in comparison to the ongoing SW/HW development which took months to complete. It is because the SoC FPGA development process is fairly new and very complicated. However, we shall see the significant performance benefit in using Soc FPGA.

3.2 Development Tools

3.2.1 Quartus II

The hardware development process started by using Quartus II design software to develop the Verilog code we would use to program our FPGA. For the demos, the Verilog code was provided. For our picture-in-picture extension of mixer demo, we modified the provided demo design codes to fit our needs.

3.2.2 Quartus Programmer

Quartus Programmer was used to program the board’s hardware to our desired configuration. The files used to program were pre-compiled Verilog project .sof files.

3.2.3 QSYS

Qsys system integration tool was used to design the interconnection between hardware and software.

3.2.4 PuTTY and Embedded Shell

Once the C code had been written on the PC, we file transferred compiled binaries to the embedded Linux through an Ethernet connection using the terminal tools PuTTY and the Embedded Shell.

4 Demos

As a way of gaining knowledge of, and testing our understanding of the SoC FPGA, we performed two different demo labs. These two experiments were chosen because they helped ensure we understood the tools needed to complete our project. These tools included but are not limited to the following: Quartus II, Quartus Programmer, QSYS, and PuTTY.

4.1 Blinking LED

We completed two versions of the blinking LED demo labs. The first lab consisted of a purely hardware approach (no software was involved), known as bare metal development. Bare metal development is simpler in nature, because it does not have the associated overhead of an OS.

The second version of the lab was completed using the Golden Reference Hardware Design (GRHD) [8]. This is a preconfigured and verified HPS configuration. This set up the peripheral function and the FPGA configuration. The preconfigured Qsys system created the interconnection between modules and enables the SW/HW handoff. GRHD is a very specific design, which controls limited peripherals. This design is what enabled the control of the LED output. The green portion of Figure 4) shows the setup of the LED outputs. Through the completion of this lab, we gained experience in the software and hardware design flow, and knowledge about the system integration within SoC designs via Qsys integration between modules.
The result of the demo was a row of blinking LEDs whose blinking rate could easily be manipulated through simple changes in the CPU code.

4.2 Video Mixer

The second demo consisted of taking two video streams as input data and through the use of a mixer displayed two moving video frames on a background screen display with logos that are generated by hardware. To do this required using the VIP Reference Design (See Figure 5) [4]. This particular reference design takes advantage of the dual core HPS and the AXI busses, which is faster since it is on same silicon and not connected via PCB wires. In particular, the HPS is able to split up the work between the two ARM CPU cores allowing each input video stream to be handled by one thread on one core. This means that the frame by frame reading of the videos can be done in parallel decreasing the runtime of the program. It is accomplished by reading and decoding the input MPEG stream via software (C program threads) running on the dual core ARM CPU. The input MPEG input videos are stored in the SD Card.

Each ARM CPU core is then responsible for saving two frames of each video stream at a time to the external DDR3 memory that is controlled by FPGA. This is when the FPGA hardware component’s functionality begins. The FPGA hardware design contains two frame readers which read the decoded video frames that were previously stored in memory by software. The mixer in turn is responsible for moving the actual addresses that result in the output frames moving. The desired frame location is determined by a C program that is running on the HPS ARM CPU. Such a software control is illustrated as the VIP Control block in Figure 5. The final output is then sent directly to the display screen through the VGA output without having to go through the HPS again.

Besides the two decoded video frames, the mixer has two additional inputs: one is the background screen display (which we set it to black background) and the other is a logo generator. The logo generator is a hardware design that is written in Verilog and integrated in our design via Qsys tool. The logo generator is to create logos at a desired location on the screen display. The logo is a bitmap (which is in Altera Memory Initialization File format) that is stored in a ROM device in the FPGA.

The result of this demo was a projected image of two frames, which each played one of the input videos, moving around a black background, which contained a logo that is generated via hardware design. The two video streams bounced across the screen, but never overlapped.

5 Our Idea

The end goal of our project is a runtime comparison between a picture-in-picture video image processing application running on the Embedded ARM FPGA architecture versus the software-only implementation on CPU via OpenCV’s Python platform. Comparing the runtimes of the two implementations will give qualitative proof of the assumed benefits or
disprove the benefits of using programmable hardware.

5.1 SoC Picture-in-Picture Implementation

The Altera VIP Suite facilitated the development of our picture-in-picture design. VIP is a collection of MegaCore functions. One of the functions was the Alpha Blending Mixer. This function enables the user to mix multiple streams of video data and display them as output on the screen.

5.2 OpenCV - Picture-in-Picture Implementation

In addition to the SoC version, we implemented the picture-in-picture application in a software-only method. OpenCV is a library of programming functions which enable user to develop real time image processing projects. We developed our project in the Python interface. We mimicked the C code from our SoC development to ensure a fairer comparison between versions. The full code can be found in the appendix.

In summary, our program initially assigns the pixels of the window to be the value of the larger image. Subsequently a portion of the initial image is assigned the pixel values of the smaller second image. The location of these values changes by a set delta x and y value. If the picture hits the boundary of the window, the delta x or y values are negated appropriately. The code for this is shown (Figure 6).

6 Results

Our observations of the frame transition clearly showed that the SW implementation visibly performed much better than the SW/JHW version. When watching the result of both versions of our application, the FPGA HW and ARM Core SW implementation had a clearly evident lag when watching the background video. The lag of the background video occurred while the video in the foreground transitioned between frames at a rate visually equivalent to the SW implementation. This resulted in a total program runtime of 3 minutes 33 second for the SW version, and an extreme 15 minutes and 46 second runtime of the FPGA HW/SW implementation. This was contrary to our initial hypothesis which we believed that hardware would show distinct performance advantage.

6.1 OpenCV Timing Analysis

We decided to investigate this result further, by looking into the source of the real delay when using FPGA HW/SW. To find which function was causing the lag in the FPGA SW/JHW version, we recorded the times to decode, read, and move the images.

<table>
<thead>
<tr>
<th>Frame</th>
<th>Reading Layer 0 (ns)</th>
<th>Reading Layer 1 (ns)</th>
<th>Moving Layer 1 (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>3443200</td>
<td>912499</td>
<td>11000000</td>
</tr>
</tbody>
</table>

Table 1: Average SW Implementation Function Runtimes

Reviewing the trends shown in Figure 7 we see...
that a majority of the program runtime is due to the time required to read layer 0 (larger background video). This lag is not detrimental to the programs successful implementation. A significant amount of delay is associated with the software MPEG-2 decoder which is implemented using libmpeg2 library that is included in the reference design.

The main interest of our project was not to learn about the speed and capabilities of the two platforms image/video reading and decoding speeds, but to learn more about the speedups they offer in respect to video and image processing techniques. Thus, we chose to perform a closer analysis of the time associated with move function of our implementations. Figure 8 shows the runtime of ten trials of our frame move function. The average time of the trials is also plotted on the figure.

### 6.2 SoC Timing Analysis

For us to make a fair comparison between the two platforms, we performed the same timing analysis done on the OpenCV version of our picture-in-picture application, with our SoC implementation. SoC platform requires the images to not only be read, but also decoded so they can then be output via the VGA cable, we recorded ten trials of this timing data in addition to the metrics recorded for our OpenCV implementation.

<table>
<thead>
<tr>
<th>Frame</th>
<th>Reading Layer 0 (ns)</th>
<th>Reading Layer 1 (ns)</th>
<th>Decoding Layer 0 (ns)</th>
<th>Decoding Layer 1 (ns)</th>
<th>Moving Layer 1 (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>5383300000</td>
<td>1097700000</td>
<td>1316700000</td>
<td>1010630000</td>
<td>313597000</td>
</tr>
</tbody>
</table>

Table 2: Average SW/HW Implementation Function Runtimes

A quick visual inspection reveals that a majority of the computation time in the SoC implementation is spent reading the larger background frame (as with the OpenCV version) Figure 9. The time required for reading the background frame is much higher in comparison to the SW implementation. This major difference in reading times would worsen the delay effect noticed in the SW/HW version of our project.

After performing the runtime analysis, it is evident that the decode function and reading of layer 0 (the large background video) result in major delays of the SW/HW implementation.
7 Discussion

7.1 Analysis

Reviewing the time required to read and decode each layer; it becomes evident what the major lag in the background of our SW/HW implementation is a result of reading of layer 0, and the decoding process cannot be separated in the OpenCV reading operation but overall OpenCV reading operation takes much lesser time than the decoding operation in our SW/HW implementation.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Reading Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 0</td>
<td>74.4</td>
</tr>
<tr>
<td>Layer 1</td>
<td>1.13</td>
</tr>
<tr>
<td>Moving</td>
<td>0.0003</td>
</tr>
</tbody>
</table>

Table 3: Runtime Ratio Comparisons

Analyzing these numbers reveals important characteristics about the runtime of the operations most frequently used in our picture-in-picture application, and a few of the major contributors to the lag seen in our SoC implementation. We see the lag resultant in the background frame of our SW/HW implementation, is due to the slow reading time and decoding times. The reading time required in OpenCV is on average 74.4 times quicker than the SoCKit implementation. While the reading of layer 1 is also slower, it is not as large a magnitude as seen with layer 0, resulting in no delay visible to the naked eye. The decode operation necessary with the SoC program is not necessary in OpenCV. Thus, the decoding process adds even more associated delay that is not present in the OpenCV implementation.

One of the most noteworthy results of our project was the timing analysis performed on the mixer move function of each implementation. While the reading and decoding processes were much slower via SoC, the actual video and image processing, the move function, was 3 million times faster in HW/SW compared to the time required with OpenCV. This leads us to believe that combining the 2 systems on the SoC (running OpenCV on the SoC board), would enable users to take advantage of OpenCV’s reading function and SoC’s faster VIP capabilities. We also believe that while the slower speed of OpenCV’s image processing is undetectable to the eye with our simple move function, as image processing functions increase in complexity a lag would result. SoC may help avoid this predicted delay with its quick image processing capabilities.

8 Conclusion

With the completion of our project, we were able contribute to certain components of the SoC development process, and our own implementations of a picture-in-picture application through the SoC and Altera VIP Cores Foundation and then again via OpenCV.

To aid in the SoC development process, we wrote a MatLab script which given an input image with .jpg extension, generates a .mif (Altera Memory Initialization File) file of the image in black and white. The dimensions of this file fit the dimensions of the SoC logo generator, enabling quick logo generation for the SoC VIP platform.

While the components of the VIP Core remain unchanged, we modified existing intellectual property to design the outcome specific to our project goals. We implemented our picture-in-picture application, through two platforms: HW/SW co-design using embedded SoC FPGA and software-only OpenCV.

9 Future Work

Seeing that the OpenCV platform has been highly optimized for quick reading and decoding video frames, we think incorporating this into our SoC implementation could result in further runtime benefits. We could combine the image reading benefits of OpenCV with the speed benefits offered in hardware’s VIP function, to create an even further optimized video and image processing application.

References


10 Appendix

10.1 Raw Data

<table>
<thead>
<tr>
<th>Frame</th>
<th>Reading Layer 0 (ns)</th>
<th>Reading Layer 1 (ns)</th>
<th>Decoding Layer 0 (ns)</th>
<th>Decoding Layer 1 (ns)</th>
<th>Moving Layer 1 (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>270160852</td>
<td>2061312</td>
<td>107562760</td>
<td>90747560</td>
<td>3440</td>
</tr>
<tr>
<td>2</td>
<td>250125852</td>
<td>872686</td>
<td>107562760</td>
<td>90747560</td>
<td>3220</td>
</tr>
<tr>
<td>3</td>
<td>240166840</td>
<td>940800</td>
<td>108472100</td>
<td>9407420</td>
<td>3760</td>
</tr>
<tr>
<td>4</td>
<td>270016008</td>
<td>768250</td>
<td>111843520</td>
<td>92347520</td>
<td>4880</td>
</tr>
<tr>
<td>5</td>
<td>250125852</td>
<td>851004</td>
<td>107562760</td>
<td>90747560</td>
<td>3200</td>
</tr>
<tr>
<td>6</td>
<td>240166840</td>
<td>699742</td>
<td>106833520</td>
<td>95325280</td>
<td>3600</td>
</tr>
<tr>
<td>7</td>
<td>240166840</td>
<td>1050224</td>
<td>111843520</td>
<td>92133760</td>
<td>3220</td>
</tr>
<tr>
<td>8</td>
<td>270016008</td>
<td>877120</td>
<td>107489240</td>
<td>94875600</td>
<td>3480</td>
</tr>
<tr>
<td>9</td>
<td>250125852</td>
<td>821504</td>
<td>107359080</td>
<td>93929480</td>
<td>3200</td>
</tr>
<tr>
<td>10</td>
<td>250125852</td>
<td>821504</td>
<td>107359080</td>
<td>93929480</td>
<td>3200</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>25626000</td>
<td>10917960</td>
<td>91230768</td>
<td>3372</td>
</tr>
</tbody>
</table>

Table 4: SW/IHW Implementation Function Runtimes

<table>
<thead>
<tr>
<th>Frame</th>
<th>Reading Layer 0 (ns)</th>
<th>Reading Layer 1 (ns)</th>
<th>Moving Layer 1 (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3348827</td>
<td>852108</td>
<td>12861013</td>
</tr>
<tr>
<td>2</td>
<td>3683010</td>
<td>877141</td>
<td>10774135</td>
</tr>
<tr>
<td>3</td>
<td>3505945</td>
<td>1106977</td>
<td>10619163</td>
</tr>
<tr>
<td>4</td>
<td>3375861</td>
<td>854969</td>
<td>10900020</td>
</tr>
<tr>
<td>5</td>
<td>3293991</td>
<td>851154</td>
<td>11017084</td>
</tr>
<tr>
<td>6</td>
<td>3365953</td>
<td>993013</td>
<td>11269092</td>
</tr>
<tr>
<td>7</td>
<td>3297090</td>
<td>849002</td>
<td>11169910</td>
</tr>
<tr>
<td>8</td>
<td>3435850</td>
<td>907897</td>
<td>11307954</td>
</tr>
<tr>
<td>9</td>
<td>3770397</td>
<td>967025</td>
<td>11083841</td>
</tr>
<tr>
<td>10</td>
<td>3353118</td>
<td>873804</td>
<td>11084794</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>3443200</td>
<td>912499</td>
</tr>
</tbody>
</table>

Table 5: SW Implementation Function Runtimes

10.2 MatLab Code

10.2.1 MIF File Generation

```matlab
1 function [outfname, rows, cols] = miffilegen(infile, outname)
2 % read file and resize
3 img = imread(infile);
4 imgresized = imresize(img, [114 1024]);
5 [rows, cols, rgb] = size(imgresized);
6 imgscaled = imresized/16 - 1;
7 imshow(imgscaled*16);
8 fid = fopen(outfname, 'w');
9 fprintf(fid, 'WIDTH = 1;\r\n');
10 fprintf(fid, 'DEPTH = 116736;\r\n');
11 fprintf(fid, 'ADDRESS_RADIX = DEC;\r\n');
12 fprintf(fid, 'DATA_RADIX = DEC;\r\n');
13 fprintf(fid, 'CONTENT BEGIN\r\n');
14```

12
count = 0;
% for every pixel
for r = 1:rows
  for c = 1:cols
    % get existing pixel color values
    red = uint16(imgscaled(r,c,1));
    green = uint16(imgscaled(r,c,2));
    blue = uint16(imgscaled(r,c,3));
    color = red*256 + green*16 + blue;
  end
end

% determine if pixel should be black
% or white
if color == 4095
  color = 0;
  fprintf(fid,'%d %x
',count,color);
else
  color = 1;
  fprintf(fid,'%d %x
',count,color);
end

end

fprintf(fid,'END;
');
fclose(fid);

10.2.2 Timing Analysis

clear
close all

%% Results from Software

SW = [1 3348827 852108 10861013; 2 3683090 877141 10774135; 3 3505945 1106977 10619163; 4 3373861 85 4969 10900020; 5 3293991 8511 4 11017084; 6 3365993 993013 11269092; 7 3297090 84 0902 11169910; 8 3435850 907897 1145934; 9 3773927 967025 11083841; 10 3353118 873804 1108794];

%% Results from SoCKit Board

SoC = [1 270160832 2061312 107359080 3200; 2 250125824 872996 10755160 29202480 3220; 3 24803840 940900 108472160 29404720 3760; 4 270019508 798208 111643520 32537920 4800; 5 250125824 872996 10755160 33925480 3200; 6 249186304 969742 106835360 35324280 3600; 7 248015488 1020224 111643520 32537920 4800; 8 270063816 897280 107492040 28687800 3480; 9 249249600 1046784 120282680 29245600 3120; 10 257464448 949120 11257280 31294080 3400];

%% Calculations

% mean values for SW functions
sw.reading.0 = SW(:,2);
mean_sw_reading.0 = mean(sw.reading.0);
sw.reading.1 = SW(:,3);
mean_sw_reading.1 = mean(sw.reading.1);
% mean values for SoC functions
soc_reading_0 = SoC(:,2);
mean_soc_reading_0 = mean(soc_reading_0);
soc_reading_1 = SoC(:,3);
mean_soc_reading_1 = mean(soc_reading_1);
soc_decoding_0 = SoC(:,4);
mean_soc_decoding_0 = mean(soc_decoding_0);
soc_decoding_1 = SoC(:,5);
mean_soc_decoding_1 = mean(soc_decoding_1);
soc_moving = SoC(:,6);
mean_soc_moving = mean(soc_moving);

% ratios
lay0_read = mean_soc_reading_0/mean_sw_reading_0
lay1_read = mean_soc_reading_1/mean_sw_reading_1
move = mean_soc_moving/mean_sw_moving

% plots
n = 1:1:10;
l = ones(length(n),1).*mean_sw_moving;
m = ones(length(n),1).*mean_soc_moving;

% SW function runtimes
figure(1)
plot(n,soc_moving, 'o-', n, soc_reading_0, 'o-', n, soc_reading_1, 'o-', 'LineWidth',2, 'LineWeight',2)
legend('Reading Layer 0', 'Reading Layer 1', 'Moving', 'Location', 'Best')
title('Runtime of Frequently Used SW Functions', 'FontSize',14, 'FontWeight', 'bold')
xlabel('Trial', 'FontSize',14)
ylabel('Time (ns)', 'FontSize',14)

% SoC function runtimes
figure(2)
plot(n, soc_decoding_0, 'o-', n, soc_decoding_1, 'o-', 'LineWidth',2, 'LineWeight',2)
title('Runtime of Frequently Used SoC Functions', 'FontSize',14, 'FontWeight', 'bold')
xlabel('Trial', 'FontSize',14)
ylabel('Time (ns)', 'FontSize',14)

% SW moving function and mean
figure(3)
plot(n,soc_moving, n, m, 'LineWidth',2, 'LineWeight',2)
xlabel('Trial', 'FontSize',14)
ylabel('Time (ns)', 'FontSize',14)
legend('Moving Time - Trials', 'Mean Moving Time of Trials')

% SoC moving function and mean
figure(4)
plot(n,soc_moving, n, l, 'LineWidth',2, 'LineWeight',2)
title('Moving Time of SW Implementations', 'FontSize',14, 'FontWeight', 'bold')
xlabel('Trial', 'FontSize',14)
ylabel('Time (ns)', 'FontSize',14)
legend('Moving Time - Trials', 'Mean Moving Time of Trials')