University of Toronto

Faculty of Applied Science and Engineering

Edward S. Rogers Sr. Department of Electrical & Computer Engineering

Animate **W**ith **A**ction

ECE 532 | Final Report

April 13, 2017

Team Members

Homagni Ghosh

Aditya Nair

Tirthak Patel

TABLE OF CONTENTS

1.0 PROJECT OVERVIEW

1.1 Background and Motivation

Animated feature films often employ the technique of motion capture in order to animate the actions of the character. Motion capture is a recently developed method which is used for recording movements and actions of objects and people [1]. In this technique, the actor is equipped with sensors that capture their motion, the data is stored and later, this motion is mimicked by a character on screen. **Figure 1** provides a visual reference for this process. The following methods dominate the current market for capturing movement :

1. Optical-Passive: Motion is tracked using retro-reflective markers which are captured by an infrared camera [1].
2. Optical-Active: The actor is equipped with LED/bright markers which have a wired connection to a motion capture suit [1].
3. Video/Markerless: Requires video processing to track the motion of the actor [1].

However, a major drawback of these methods is that the processing is done using software technology and subsequently, is a slow process. This makes it difficult to perform tracking and animation rendering in real-time.

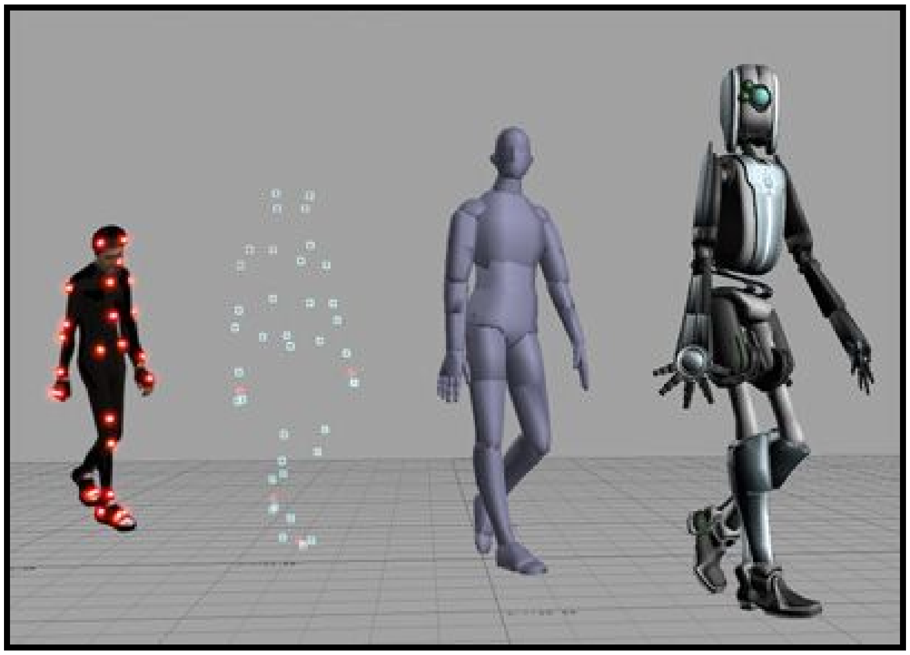

Figure 1: The process of converting LED markers to an animated character [2].

1.2 Goals of Our Project

“Animate With Action” is an application which provides a live motion capture experience. We added a real-time component to the process of recording actions such that the actions of the actor are mimicked by the character on screen in real time. Our project requires the actor to be equipped with red marker tapes which are captured by a PMOD camera. The live video processing unit then would detect the coordinates of the markers as fast as possible and therefore, was implemented in hardware. The graphics component of the project, which includes animating and controlling the character on screen, was achieved using software.

1.3 System-Level Block Diagram

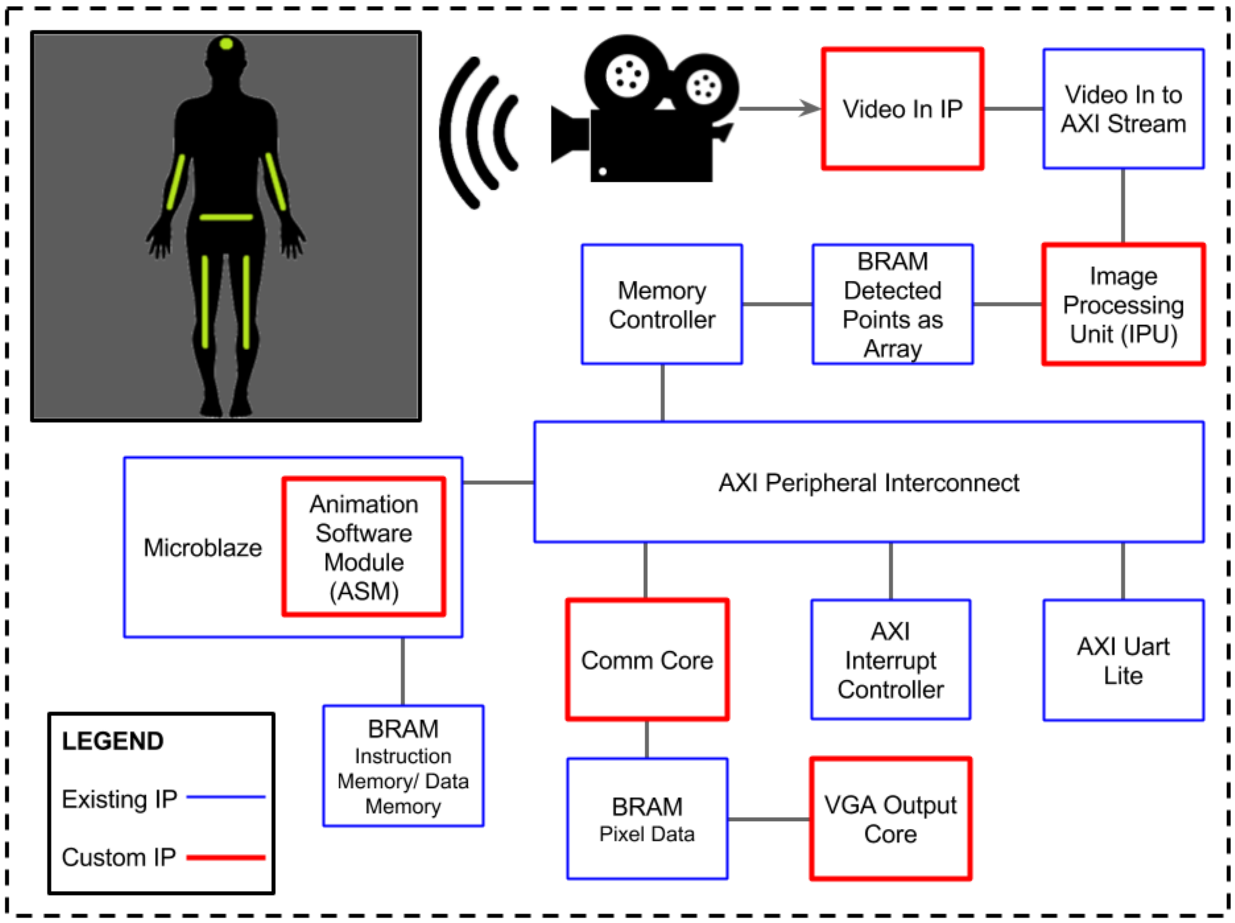

Figure 2: System-level overview of our final design.

1.4 Brief Summary of the Blocks

This section provides an introduction to all the custom components. Further details will be provided later on in this document.

1. **Video In IP:** This IP records the frames sent by the PMOD camera and converts the camera signals into signals compatible with the AXI stream interface. These signals are then streamed directly to the IPU using AXI stream protocol. The IP also configures the camera to send RGB pixels in QVGA format. It also provides other settings such as contrast, saturation and edge enhancement.
2. **Image Processing Unit:** This IP processes the frames streamed by the above module. It looks at each pixel to determine if the R component is greater than, and G and B components are lesser than the required thresholds. If the condition is met, the pixel metadata is stored in order to group all the pixels belonging to the same general area (block) of the frame. Then these blocks are stored in a BRAM to be read by the code running on Microblaze.
3. **Animation Software Module:** This module reads the block information stored by the IPU. It draws and animates a character on screen by decoding the block and finding the corresponding joint and its position. The code for this module is written in C and runs on the Microblaze processor.
4. **Comm Core:** This core was written to provide an interface for the ASM to write animation pixels to BRAM using 12 bit wide data format and 19 bit wide address bus. The ASM code writes to the slave registers in this IP and these signals then write the data to the connecting BRAM.
5. **VGA Output Core:** This core reads the pixels written to the BRAM by the above IP and converts the data into signals compatible with VGA format in order to write a 640x480 frame to the VGA peripheral. It renders the animated scene on the screen.

2.0 PROJECT OUTCOME

2.1 Initial Design

Table 1 lists the functional requirements that we proposed at the beginning of the project, and the current status, including deviations that are present in the final design.

Table 1: Functional requirements and current status.

S. No.	**Initial Feature** (What we intend to achieve)	**Current Status** (in Final design)	**Comments** (Method/Reasoning)
1	Actions performed by the actor should be imitated by character on screen within 5s.	**Present**	The action appears on the screen in 3s.
2	The camera needs to transmit frames to the IPU at the rate of at least 1 frame per second.	**Present**	This process has been verified using the PMOD camera live capture module.
3	Detection of body markers in the image stream should take less than 1 second.	**Present**	Animation rendering was required to be run on Microblaze (requirement). As this is a slow process (implemented in software), we ensure that sufficient time is available to meet the 5s target.
4	The animated result will be rendered on a screen using VGA. The output resolution will be 512x512 pixels.	**Modified** to 640x480 pixels	The VGA display was configured to 640x480 rendering.
5	The input capture will have a maximum resolution of 512x512 pixels in order to enable faster processing.	**Modified** to 320x240 pixels	This was done to minimize the amount of pixels processed and because the QVGA resolution was deemed high enough for detection purposes.

Table 1 continued.			
6	Input format needs to be YUV in order to detect LED brightness.	**Modified** to RGB444 format	The test and demonstration environment (lab) was an area with varying brightness, and this method was not successful. Thus, we used red tape for demonstration purposes, which can be easily processed using RGB format.

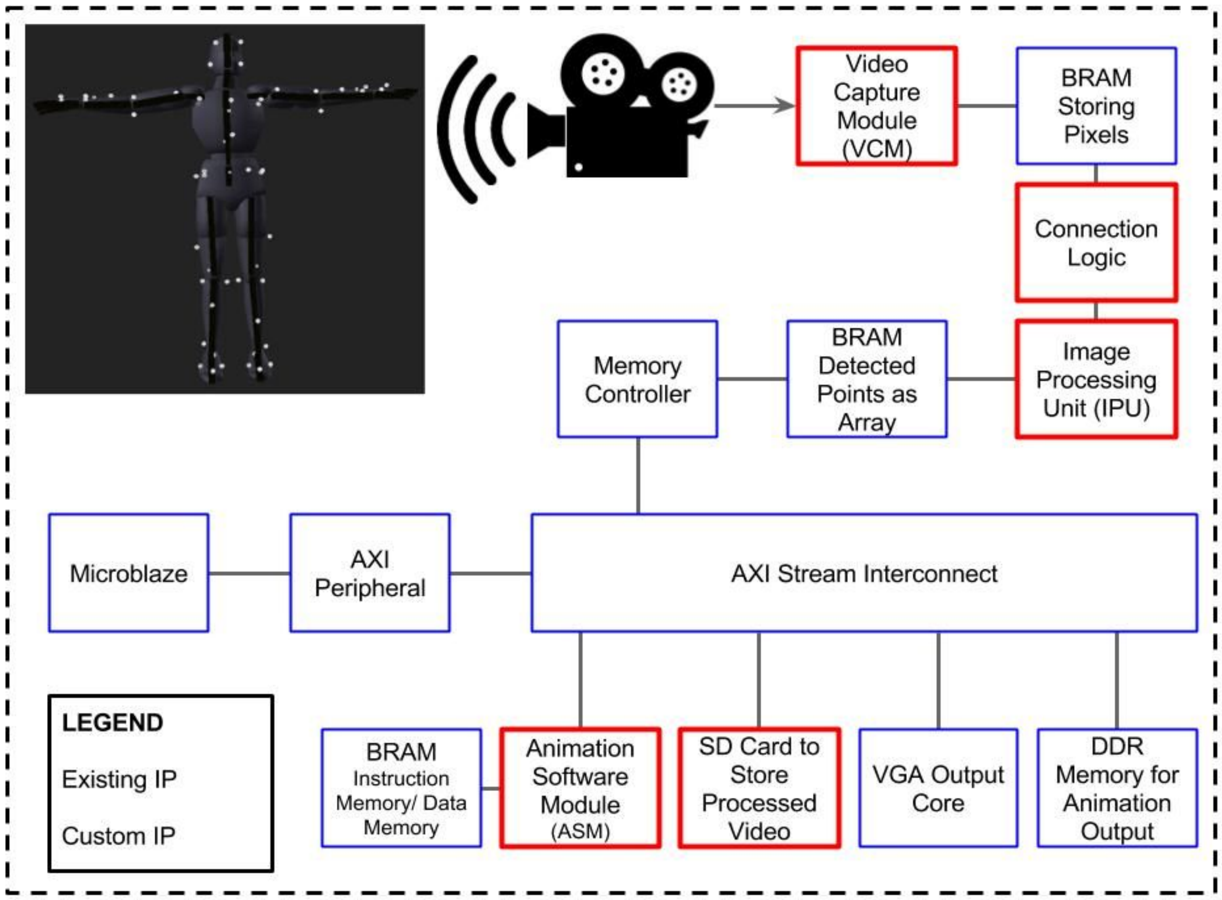
Video Capture Module (VCM)
BRAM Storing Pixels
Connection Logic
Image Processing Unit (IPU)
BRAM Detected Points as Array
Memory Controller
Microblaze
AXI Peripheral
AXI Stream Interconnect
LEGEND
Existing IP
Custom IP
BRAM Instruction Memory/ Data Memory
Animation Software Module (ASM)
SD Card to Store Processed Video
VGA Output Core
DDR Memory for Animation Output

Figure 3: System-level overview of proposed design.

The final and proposed design have been shown in **Figure 2** and **Figure 3** respectively. In the proposed design, we planned to have an SD-card to store recorded video and DDR memory to store animation output. This was intended with the purpose of extending the use of the design for non-live applications. As we primarily intended to have the basic features shown in **Table 1** working, we could not devote time to have the additional peripherals in our design. Moreover, ASM was originally an independent software module.

However, the use of MicroBlaze in the design was a requirement. Thus, the ASM has been implemented using Microblaze, and reflected in the final design.

2.2 Final Achievements

As presented in **Table 1**, we satisfied our proposed functional requirements. However, we could not have a "correct" animation which is reliably consistent with the actual motion on the VGA screen.

Owing to the limited documentation available on the PMOD camera, the team had issues on figuring out the correctness of input video data, as there were pixels being reported from the IPU, which were clearly incorrect. Closer analysis from the live capture module (provided by the TA) revealed that the camera frames had a lot of noise in them. This was all resolved in the last week of project timeline. The new camera had an active reset pin (unlike the other camera that we had in the past), which took us some time to figure out. As we never had the opportunity to test the IPU with the animation code using real data, there were deviations observed in the animation in the final presentation.

If we were to do this project again, we could follow some measures listed below in order to achieve a complete product. These would also be guidelines for anyone taking over our project.

- **IPU**: It is imperative to analyse individual frames of data captured using the PMOD camera (rather than fabricated sketch data) for various scenarios, such as different background and different lighting sources, using the reference software emulation of the IPU algorithm and the Verilog module, in order to calibrate the IPU for improved results.
- **ASM:** The ASM code had issues with jittery animation, which was fixed at the end of the project by selectively clearing the screen. The animation module can be tested with different sets of data loaded into the BRAM (detected points), and observing the results on screen.

The following features can be improved in our code to handle a wide range of scenarios:

- **Noise filtering:** This feature was present in the software reference code and the initial IPU module and worked correctly in simulations. However, after the final integration, we faced issues (mentioned above) owing to reasons which we could not debug in a limited time frame. To get a basic working product, we removed it in the final design. Based on real data testing, one can estimate the noise threshold parameter and reduce the instances of “random” animation actions that show up on the screen sometimes.

- **Smart detection:** Currently, the animation algorithm identifies parts based on coordinate positions in frame. Therefore, animation is limited only to simple, straight body motion. We can improve this by creating a mathematical model that establishes relationships between points based on proximity and patterns, which would help us animate complex movements and positions.
- **Speed:** Currently, we have achieved live capture in less than 5s, there is room for improvement in terms of higher clock speed, further optimised animation, reduced data exchange between modules and multi-threaded animation.

3.0 PROJECT SCHEDULE

3.1 Initial Milestones

The following list of milestones has been taken from the proposal. Each milestone is achieved over one week.

Milestone 1: Basic test bench with the LED capture working for a single point (Components covered include VCM, pixel BRAM, connection logic and IPU)

Milestone 2: Testbench Demo (Identification of location of LEDs on the body)

Milestone 3: Rendering simple animation on VGA (This will be pre-existing animated data stored in image format)

Milestone 4: Rendering animation from given input point coordinates from LEDs

Milestone 5: Mid-Project Demo (Rendering video from stored data)

Milestone 6: Integration and testing of modules

Milestone 7: Final Demo

3.2 Actual Weekly Accomplishments

Table 2 discusses the actual weekly accomplishments and explanations for any observed deviations from the original plan.

Table 2: Accomplishments of each milestone and explanation for deviations.

Mile stone#	**Accomplishment**	**Explanation**
1	1. Formulation of algorithm in MATLAB to detect different body parts based on brightness of "white marked" regions.	1. We had initially planned to detect a single point, but were able to extend to multiple points.
2	1. Generated a Verilog module for IPU based on the software reference, but was not successful (system ran out of memory in synthesis).	1. Software reference code processes image data stored in memory. However, it is impossible to process video data by storing a frame in BRAM. Translation of loops and other variable constructs led to synthesis issues (no memory available error).

Table 2 continued.		
3	1. Successful formulation of IPU in verilog based on streaming data (Improved algorithm time complexity to O(n), from O(n^2) as performed in MATLAB code) 2. Testbench verification of IPU. 3. Finished Video In and AXI Stream IPs.	1. This milestone took time as we have attempted to apply various concepts (pipelining and superscalar) to process AXI-stream data, cluster pixels into body parts and remove noise pixels based on size threshold. The work was verified using a simulation of sample human image data. 2. We could not focus on animation as originally planned, as IPU is the custom core which has to be fully functional for animation to work correctly.
4	1. Creating IP wrapper for the IPU. 2. Interfacing IPU with output BRAM and input AXI-4 stream module (for pixel data).	1. Simulated video data was being correctly streamed and data was written to BRAM, which indicated successful integration. 2. Owing to the complexity of integration, animation was still in progress during this week.
5	1. Rendering blocks (based on coordinates from IPU) on VGA monitor. 2. Completed Comm IP and VGA Output core.	1. We have been consistent with the midterm milestone of rendering video from stored data. 2. It took a lot of experimentation to figure out a memory efficient interface.
6	1. Integrating animation into design.	1. Animation was very jittery. The randomness in IPU coordinates was still the issue, owing to noisy data from video camera.
7	1. Obtained cleaner video data using a replacement camera. 2. Final testing of components. 3. Final demonstration	1. Video camera was replaced, which had an active reset pin. It took some time to figure this out. 2. IPU was not processing data correctly because of different brightness levels in lab. We switched to RGB format, and removed noise filtering to make the design functional.

4.0 MODULE-LEVEL DESCRIPTIONS

This section provides an in-depth description of the custom cores. The cores not described here were all taken from the Vivado IP library and run on a 100 MHz clock.

4.1 Video In IP

This module takes in the href, vsync and the 8-bit data signal from the PMOD camera. The 8-bit data signal transmits a complete packet of RGB values over 2 cycles. The function of this IP is to combine these packets into 16-bit data bus and send this to the AXI stream module. It also generates a plethora of output signals required by the AXI stream IP such as vid_active_video (when the data is valid), vsync (when a new frame starts), hsync (when a pixel row ends) and vid_io_in_clk (the clock rate at which the data is sent: 12.5 MHz). This IP runs on a 25 MHz clock as required by the PMOD camera.

Another function of this IP is to configure the camera. We did a lot of experimentation with the code provided on Piazza in order to get correct settings to detect the red value in the pixels. Accordingly, this IP sets values for contrast, frame format, saturation, edge enhancement, pixel redness level, pixel noise level and pixel format.

4.2 Image Processing Unit

The IPU receives as input four main signals: m_axis_video_tdata (16-bit wide pixel data bus), m_axis_video_tuser (indicates start of a new frame), m_axis_video_tlast (indicates end of a frame row) and m_axis_video_tvalid (indicates when the data is valid). The IPU takes these signals and extracts the R, G and B values from the data signal and pixel position from the other signals. It compares the colour values against the thresholds and saves metadata about the red pixels. It then uses this metadata to label pixels belonging to the same general area as belonging to the same block in order to avoid detecting the same joint or muscle multiple times. The IPU finds all such blocks and stores their position coordinates in the BRAM by specifying the address and memory write enable. These coordinates are then read by the ASM as described below.

4.3 Animation Software Module

The ASM module implements the software code to write the pixels on screen. It reads four blocks from the BRAM written to by the IPU, each corresponding to a joint on the body: face, left arm, right arm and waist. Then based on these positions, the algorithm uses rasterization techniques to interpolate the lines to animate a character on screen. First, it writes black pixels to the positions of the previous pixels in order to erase the previous lines of the character. The writing is done through Comm Core IP as is described in the following section. After, erasing the previous pixels, the algorithm writes white pixels to the positions of the new character. By using this technique, we ensure that the ASM is not writing all the pixels on the screen every time it wants to draw the character in a new position. This gave us a large performance improvement as we were able to write only a few pixels per iteration of the algorithm, as opposed to all of the 307200 pixels. Once, it is done writing the white pixels, it goes back to reading the blocks from the BRAM and repeats the process all over again.

4.4 Comm Core

This is an AXI slave core which was especially written in order to enable the code running on Microblaze to write to the BRAM in a 12-bit data format. Considering that our design has three BRAMs, it was essential to use the limited BRAM space efficiently. We did not want to use DDR memory to store the pixels because it would have resulted in much slower transactions, which in turn would hinder our ability to produce real time results. The existing AXI BRAM controller IP requires the data to be stored in 32-bit formats, which would have wasted 8 bits per two pixels. We did not have enough memory for this; therefore, a custom solution was required.

The IP has 2 used registers. The ASM code running on microblaze can write directly to these registers. The register signals are converted to wires and directly connected to the BRAM write port. The ASM can write a pixel to BRAM by writing the address to slv_reg0[18:0], data to slv_reg0[30:19] and write enable to slv_reg0[31]. All three of these signals were combined into one register in order to avoid three separate writes from the ASM which would take up a lot of extra cycles. This format of writing gave a 3X performance boost as determined empirically. The ASM can also instruct the VGA Output Core to start drawing the pixels from the BRAM by writing 1 to register slv_reg3.

4.5 VGA Output Core

The VGA Output Core constantly reads the pixels from the BRAM as long as the draw signal is asserted by the ASM. It separates the data into 4-bit R, G and B values to send to the VGA pins. It also generates the hsync and vsync signals based on the frame size of 640x480. The parameters required for hsync and vsync were taken from code on Github [3].

The IP also generates tready signal indicating when it is ready to start reading and fsync signal indicating when it is done writing the frame. However, these two signals, while initially thought to be useful, were not required in the integrated design. The IP also outputs the address of the pixel to read from the BRAM and takes 12-bit data as input. This IP runs on a 25 MHz clock as required by the VGA peripheral.

5.0 DESCRIPTION OF OUR DESIGN TREE

The github link for the project is: https://github.com/nairadi/G13_AnimateWithAction

The src folder consists of the entire project. This includes:

1. ip_repo/comm_1.0 - This folder contains all communication interface IPs
2. ipu - This folder contains the image processing code in Verilog
3. vga_ip – This folder contains the VGA ip to write to the screen
4. video_in - This folder contains the PMOD camera code
5. vivado - This folder contains the actual project and the animation code

The docs folder consists of supporting documents. This includes:

1. Report.pdf – The group report for the project
2. Presentation.pdf – The presentation slides for the final demo
3. Detection.mp4 – Video demo of red pixel detection
4. Animation.mp4 – Video demo of animation

To run the program:

1. Go to *src -> project_1 -> vivado* and launch vivado.xpr
2. Launch the SDK from vivado to run the animation code

6.0 TIPS AND TRICKS FOR FUTURE TEAMS

Using approaches in software programming to approach hardware problems can cause major issues especially with resource allocation such as LUTs. We tested the functionality of our algorithm in software and then translated it to hardware. In doing so, our initial Verilog code also contained "for" loops. However, synthesising a loop results in treating each instance as a separate circuit. For this reason, the design would take considerable space on the chip. In some scenarios where we had large loop counters, the project would take significantly longer to synthesise or fail synthesis altogether as a result of the lack or limitation of available resources.

It is also essential to consider the efficiency of the software code running on Microblaze. In our design, drawing to the VGA from the SDK code would result in the animated character flashing on screen instead of transitioning smoothly. This was because initially we were clearing the whole screen. Writing each pixel to the BRAM from the SDK code takes multiple cycles and consequently, adds significant delays. We changed this to erase only the previously drawn pixels for each new frame to make the IO operations more efficient.

7.0 REFERENCES

[1] "What is motion capture", VICON, 2017. [Online]. Available: https://www.vicon.com/what-is-motion-capture. [Accessed: 03- Feb-2017].

[2] "Motion capture", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Motion_capture. [Accessed: 03- Feb-2017].

[3] “SuperHexagonFPGA”, Github, 2015. [Online]. Available: https://github.com/samp20/SuperHexagonFPGA/blob/master/vga_driver.v. [Accessed: 11-Apr-2017].