Design and Development of Stream Processor and Texture Filtering Unit for Graphics Processor Architecture IP

Krishna Bhushan Vutukuru[1], Sanket Dessai[2]
1- M.Sc. [Engg.] Student, 2- Assistant Professor
Computer Engineering Dept.,
M. S. Ramaiah School of Advanced Studies, Bangalore 560 058.

Abstract

Graphical Processing Units (GPUs) have become an integral part of today's mainstream computing systems. They are also being used as reprogrammable General Purpose GPUs (GP-GPUs) to perform complex scientific computations. Reconfigurability is an attractive approach to embedded systems allowing hardware level modification. Hence, there is a high demand for GPU designs based on reconfigurable hardware.

This paper presents the architectural design, modelling and simulation of reconfigurable stream processor and texture filtering unit of a GPU. Stream processor consists of clusters of functional units which provide a bandwidth hierarchy, supporting hundreds of arithmetic units. The arithmetic cluster units are designed to exploit instruction level parallelism and subword parallelism within a cluster and data parallelism across the clusters. The texture filter unit is designed to process geometric data like vertices and convert these into pixels on the screen. This process involves number of operations, like circle and cube generation, rotator, and scaling. The texture filter unit is designed with all necessary hardware to deal with all the different filtering operations. For decreasing the area and power, a single controller is used to control data flow between clusters and between host processor and GPU. The designed architecture provides a high degree of scalability and flexibility to allow customization for unique applications. The designed stream processor and texture filtering unit are modelled in Verilog on Altera Quartus II and simulated using ModelSim tools. The functionality of the modelled blocks is verified using test inputs in the simulator.

The simulated execution time of 8-bit pipelined multiplier is 60 ps and 100 ns for 8-bit pipelined adder while operating at 90 MHz. Circle and cube coordinates are generated for circle and cube generation. The work can form the basis for designing a complete reconfigurable GPU.

Key Words: GPU, FPGA, Stream Processor, Texture Filtering Unit, Geometrical modelling

Abbreviations

AA	Anti Aliasing
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
APIs	Application-Programming Interfaces
Auto CAD	Auto Computer-Aided Design
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DSPs	Digital Signal Processors
FPGA	Field-Programmable Gate Array
FPMAC	Floating-Point Multiply-Accumulator
GB	Giga Byte
GPU	Graphics Processing Unit
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
JTAG	Joint Test Action Group
LRF	Local Register File
MATLAB	MATrix LABoratory
MHz	Mega Hertz
MPEG	Moving Picture Experts Group
MPSoC	Multi-processor System-on-Chip
MPI	Message Passing Interface
PEs	Processing Elements
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer

1. INTRODUCTION

As 3D has become more determined in our lives, the need for faster processing speeds is increasing. With the introduction of the GPU, computationally intensive transform and lighting calculations were offloaded from the CPU onto the GPU—allowing for faster graphics processing speeds. This means all scenes increase in detail and complexity without sacrificing performance.

A GPU is a single-chip processor mainly used to manage and boost the performance of video and graphics. GPU features include 2D or 3D graphics, digital output to flat panel display monitors, texture mapping, and application support for high intensity graphics software such as Auto CAD, rendering polygons, support for YUV color space, hardware overlays, and MPEG decoding. These features are designed to lessen the work of the CPU and produce faster video and graphics. A GPU is not only used in a PC or a video card or motherboard, it is also used in mobile phones, display adapters, workstations, and game consoles.

Designing a low-cost GPU with real-time rendering is challenging. Creating an image out of binary data is a demanding process. To make a 3D image, the graphics card first creates a wire frame out of straight lines. Then, it **rasterize** the image and also adds lighting, texture and color. For fast-paced games and other signal processing tasks, the computer has to go through this process about tens to hundreds of billions of times per second. Without a graphics card to perform the necessary computations, the workload would be too much for the computer to handle. To achieve these computation rates, current media processors use special purpose architectures customized to one specific application.

Field Programmable Gate Arrays (FPGAs) are basically pieces of programmable logic. FPGAs have become more affordable they have found their way into more and more designs. The appeal of FPGAs is the ability to handle, in hardware, a number of complex functions which previously would have been handled in the software domain. FPGA development is performed in a similar way to software development, except with different languages and tools. Debugging is somewhat more difficult, and building a design generally takes longer for an FPGA than for software.

This paper discusses the advantages of the FPGA and provides an approach for modelling single controller based stream processor and texture filter unit using geometrical methodology.

2. BACKGROUND THEORY

The internal functionality of the GPU has been discussed in this section Initially, the GPU receives vertex data from the host CPU, the vertex-shader stage is the first major stage in GPU. The next step in the classic pipeline is the setup, where vertices are assembled into primitives such as triangles, lines, or points. The primitives are then converted by the rasterization stage into pixel fragments, but are not considered full pixels at this stage. Fragments undergo many other operations such as shading, Z-testing, possible frame buffer blending, and anti-aliasing. Fragments are finally considered pixels when they have been written into the frame buffer.

2.1. Stream Processor

Stream processors are signal and image processors, which offer both efficiency and programmability. Stream processors have efficiency comparable to ASICs, while being programmable in a high-level language. Streaming processors have been widely developed for many applications.

Stream processors in GPUs can process vertices, pixels, geometry or physics—they are effectively general purpose floating-point processors. Most signal-processing applications are naturally expressed in this style. Stream processors are highly efficient computing engines that perform calculations on an input stream, while producing an output stream that can be used by other stream processors. Stream processors can be grouped in close proximity, and in large numbers, to provide immense parallel processing power. The streaming processor can also be implemented as a reconfigurable streaming processor that can be reconfigured for several scientific computing applications (Rajagopal, et al. 2004).

2.2. Texture Filter Unit

Texture filtering is a fundamental feature of modern 3D graphics. Before 3D acceleration was born, 'blocky' point-sampling filter was the only choice for real-time texture mapping. Variety of filters available for texture magnification, mostly based on geometrical mipmapping technique, and bilinear filter for texture magnification.

Texture filtering is the method used to determine the texture color for a texture mapped pixel, using the colors of nearby texels (pixels of the texture). Mathematically, texture filtering is a type of anti-aliasing (AA), but it filters out high frequencies from the texture filter whereas other AA techniques generally focus on visual edges (Heckbert 1986). In a simple it is defined as, it allows a texture to be applied at many different shapes, sizes and angles while minimizing blurriness, shimmering and blocking.

2.3. Advantages of FPGA

FPGAs have the advantage that designer can customize the process fully without wasting resources compared to controllers. FPGAs have ability to carry out complex processes, in other words; designers can design special-purpose processors, general-purpose processors at very high clock speeds than controllers. At the highest level, FPGAs can also bring these important factors to bear

- **Programming Power:** Compared with standard microprocessors, FPGAs provide much greater raw processing power, especially because they give the designer the choice to run applications in hardware vs. software.
- **Reprogrammability/Reconfigurability:** Compared with ASICs and with standard microprocessors, FPGAs can be reprogrammed or reconfigured. Because of the growing market in "intellectual property" (IP), FPGAs are a good choice for purchasing and configuring intellectual property to create highly customized and efficient designs.
- **Intellectual Property Reuse:** Few designs are completed by just one person these days, especially in more complex applications. Buying or obtaining third party IP is increasingly important, and FPGAs allow designers to mix and match third party IP to create a solution customized for respective application but drawing on the knowledge of the design community.
- **Low Cost vs. ASICs:** ASIC designs are very expensive and FPGAs allow ASIC designers to prototype quickly and efficiently, and in more and more cases to avoid doing an ASIC altogether.

3. PROBLEM DEFINITION

The aim of this paper is to design single blocks of two critical sections in the GPU, stream processor and texture filter unit for graphics processor architecture IP in FPGA platform.

4. SYSTEM REQUIREMENTS

Based on the stated problem definition, the requirements for modelling single block of stream processor can be stated as

# RAMs	1
256x16-bit single-port distributed RAM	1
# Adders/Subtractors	3
16-bit add/sub	3
# Multipliers	2
16-bit Multiplier	2
# Counters	1
16-bit up counter	1
# Multiplexers	1
16-bit 5-to-1 multiplexer	1
# ROMs	4
16x7-bit ROM	4
# Registers	109
Flip-Flops	109

Based on the stated problem definition, the requirements for modelling single block of texture filter can be stated as

# RAMs	4
512x16-bit single-port block RAM	4
# Multipliers	9
18x18-bit multiplier	9
# Adders/Subtractors	7
16-bit adder	4
9-bit adder	2
5-bit adder	1
# Registers	278
1-bit register	6
16-bit register	9
5-bit register	1
Flip-Flops	271
# Comparators	2
16-bit comparator greater	1
16-bit comparator not equal	1
Slice LUTs	270

5. METHODOLOGY

To achieve the stated requirements, from the literature survey conducted, feasible stream processor and texture filter unit of a GPU are selected as the basis for being modelling in FPGA. From the literature survey, verilog algorithms and ALTERA QUARTURS II, ModelSim Simulation tools were identified to model the selected blocks.

6. SOLUTION

6.1. Design of Stream Processor

The design is based on Harvard architecture, which defines physically separated memories for program instructions and data. This implies that the widths of data busses may differ per memory type. This is especially useful for VLIW architectures, because to issue very long instruction words from instruction memory.

The important sections in the stream processor are stream controller, streaming register file, arithmetic clusters. The stream controller is designed with following four stages fetch, decode, execute and write back. General-purpose Register (GR) file with 32-bit and a Branch Register (BR) file with 1-bit are used as dedicated registers for each arithmetic cluster.

6.1.1. Stream Register File (SRF)

SRF effectively isolates the arithmetic clusters from the memory system, making streaming processor load/store architecture for streams. Two arithmetic clusters and a stream controller are connected to the SRF. The arithmetic clusters consume data streams from the SRF, process them, and return their output data streams to the SRF. The two clusters operate simultaneously on interleaved data records transferred from the SRF, allowing two elements to be processed in parallel. Each arithmetic cluster contains three adders, two multipliers, and one divide/square root unit. These units are controlled by statically scheduled VLIW instructions issued by the controller.

The flow of data in the SRF is explained in the flow chart shown in Figure 1. The data flows from SRF to C (Mux_3) only when there is a positive edge clock pulse and if the write signal is high which is issued by the B (stream controller). The address register itself stores the address of the next instruction; the present instruction address is stored in SRF that is sent by A (Address_Register).

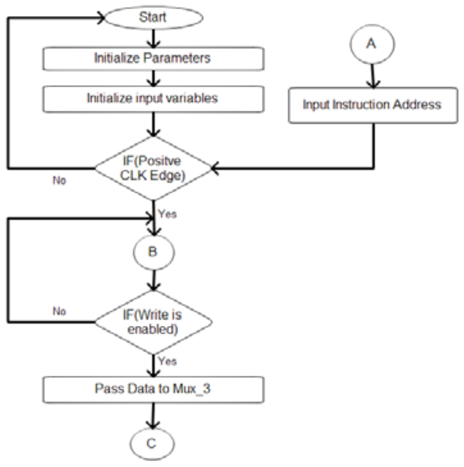


Fig. 14 Flow Chart for Stream Register File

6.1.2. Stream controller

Stream Controller unit is implemented in the streaming processor to control the processor configuration and the flow of streaming computations. The processor configuration and the flow of the streaming computations are controlled by fetching and executing instruction vectors stored in the instruction memory of host processor implemented on the stream controller. Each instruction vector contains micro codes to enable the arithmetic and memory units and to control the streaming computation flows.

The controller has three phases of operation fetch, decode, and execute. Fetching retrieves an instruction from memory, decoding decodes the instruction, manipulates data paths, and loads registers; execution generates the results of the instruction. As shown Figure 2 the fetch phase will require two clock cycles – one to load the address register and one to retrieve the addressed word from memory. The decode phase is accomplished in one cycle. The execution phase may require zero, one, or two more cycles, depending on the instruction.

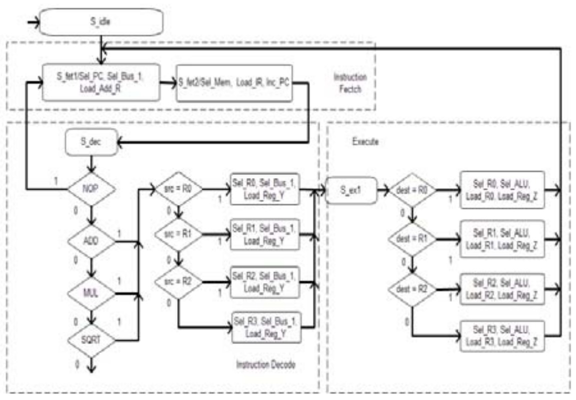

S_idle
S_fet1/Sel_PC, Sel_Bus_1, Load_Add_R
S_fet2/Sel_Mem, Load_IR, Inc_PC
Instruction Fectch
S_dec
Execute
NOP
ADD
MUL
SQRT
src = R0
src = R1
src = R2
Sel_R0, Sel_Bus_1, Load_Reg_Y
Sel_R1, Sel_Bus_1, Load_Reg_Y
Sel_R2, Sel_Bus_1, Load_Reg_Y
Sel_R3, Sel_Bus_1, Load_Reg_Y
Instruction Decode
S_ex1
dest = R0
dest = R1
dest = R2
Sel_R0, Sel_ALU, Load_R0, Load_Reg_Z
Sel_R1, Sel_ALU, Load_R1, Load_Reg_Z
Sel_R2, Sel_ALU, Load_R2, Load_Reg_Z
Sel_R3, Sel_ALU, Load_R3, Load_Reg_Z

Fig. 15 Flow Chart of Stream Controller that Implements the Instruction Set NOP, ADD, MUL, SQRT

6.1.3. Arithmetic cluster

Two arithmetic clusters are controlled by the controller in a SIMD fashion. The arithmetic clusters operate simultaneously on interleaved data records transferred from the SRF, allowing two data records to be processed in parallel. Each arithmetic cluster contains two adders, two multipliers, and one divide/square root unit. These units are controlled by statically scheduled VLIW instructions issued by the stream controller. In addition, the adders and multipliers support 16-bit and 8-bit parallel-subword operations for a subset of the integer operations. The adders and multipliers are fully pipelined, allowing a new operation to issue every cycle. The divide/square root unit has two SRT cores, so no more than two divide or square root operations can be in flight at any given time.

Figure 3 explains pipelined adder. At positive edge clock pulse if the select/reset signal is high the LSB's registers will get cleared or else if any data is available then the LSB's of the available two data's will get added and the result is stored in LSB's register. Similarly for MSB's, first the MSB data will be stored in respective registers and the contents of the MSB registers will get added and stored in another register. Finally, the LSB and MSB get concatenate and result is sent to E (SRF).

6.2 Design of Texture Filter Unit

The goal of this paper was to create a programmable graphics processing unit with as many aspects as possible to

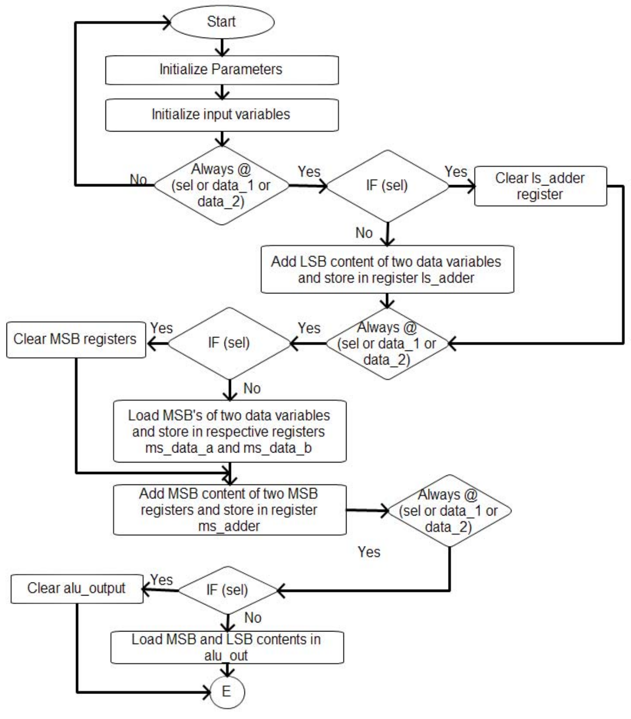

Start
Initialize Parameters
Initialize input variables
No
Always @ (sel or data_1 or data_2)
Yes
IF (sel)
Yes
Clear ls_adder register
No
Add LSB content of two data variables and store in register ls_adder
Clear MSB registers
Yes
IF (sel)
Yes
Always @ (sel or data_1 or data_2)
No
Load MSB's of two data variables and store in respective registers ms_data_a and ms_data_b
Add MSB content of two MSB registers and store in register ms_adder
Always @ (sel or data_1 or data_2)
Yes
Clear alu_output
Yes
IF (sel)
No
Load MSB and LSB contents in alu_out
E

Fig. 16 Flow chart for pipelined adder

be coded in hardware, even with object and edge generation. Texture mapping is a shading technique for image synthesis in which a texture image is mapped onto a surface in a three dimensional scene, much as wallpaper is applied to a wall. If a table is need to be modelled, a rectangular box for the table top, and four cylinders for the legs is used. Unadorned, this table model would look quite dull when rendered. The realism of the rendered image can be enhanced immensely by mapping a wood grain pattern onto the table top, using the values in the texture to define the colour at each point of the surface. The advantage of texture mapping is that it adds much detail to a scene while requiring only a modest increase in rendering time. Texture mapping does not affect hidden surface elimination, but merely adds a small incremental cost to the shading process. The technique generalizes easily to curved surfaces.

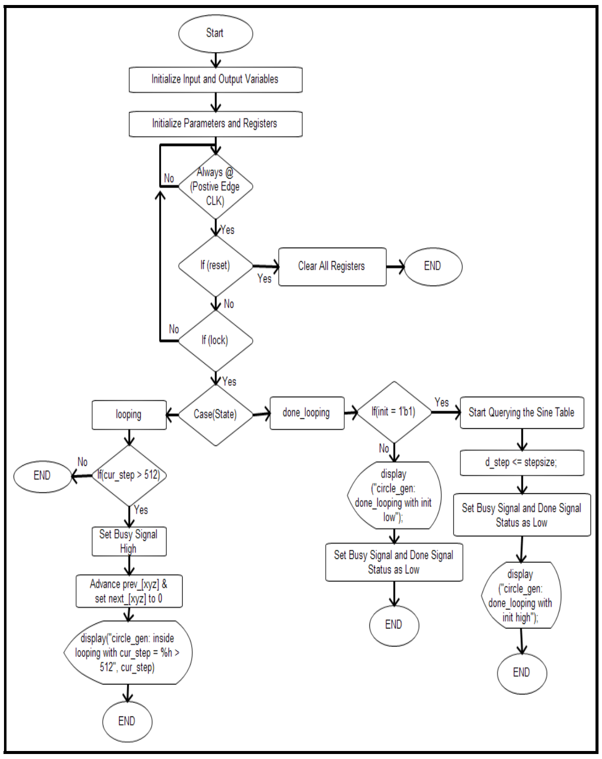
There are three components of the circuit an object generation pipeline to generate edges of the target shape; a transformation pipeline that performed transformations on the unit objects1; and a rasterizing pipeline that generates the points for the VGA controller to display. The design has made certain tradeoffs due to the constraints imposed by the FPGA to synthesize the circuit. First, the transformation pipeline does not employ a generalized 4x4 matrix multiply because the limited number of multipliers on the FPGA. Instead, the transformation pipeline is currently designed as an operate-and-accumulate module, with intermediate data values stored in registers. Alternatively with more available multipliers, by first generating a reduced matrix transformation, one data set can be transformed in one cycle. Second, the available memory on the FPGA is limited to 8.5 megabytes at most, of which about 512 kilobytes are available memory that are designed to be read from within a single cycle of exerting the desired address. Due to the time constraint, circle, and cube generator have been implemented as part of the study.

6.2.1. Circle generation unit

The circle generator unit draws the two-dimensional circles at the specified origin and scale. The flow chart of this unit is shown in Figure 4. This unit will be active when respective arguments are available at positive edge clock cycle. If the reset signal is high then all registers available in the circle generator unit will get cleared. During done_looping case, querying for sine and cosine table will be done, when init signal is high or low. During looping case, the circle will be drawn depending on the sine and cosine values.

6.2.2. Cube generation unit

The cube generator unit draws the cubes at the specified origin and scale. The flow chart of this unit is shown in Figure 6. This unit will be active when respective arguments are available at positive edge clock cycle. If the reset signal is high then all registers available in the cube generator unit will get cleared. For this model eleven possible combinational cases has been written. A basic model of the cube with has been considered initially as shown in Figure 5 (a). The dimensions of the cube depend on three main axis [(Ax, Ay, Az)] and three imaginary axis [(Bx, By, Bz)]. Whenever, Ay axis is high the cube will change accordingly in y-axis direction as shown in Figure 5 (b), Ax axis is high the cube will change its shape accordingly in x-axis direction as shown in Figure 5 (c), and if Az axis is high the cube will change its shape accordingly in z-axis direction as shown in Figure 5 (d) respectively.



Start
Initialize Input and Output Variables
Initialize Parameters and Registers
Always @ (Postive Edge CLK)
No
Yes
If (reset)
Yes
Clear All Registers
END
No
If (lock)
No
Yes
Case(State)
looping
done_looping
If(init = 1'b1)
Yes
Start Querying the Sine Table
d_step <= stepsize;
Set Busy Signal and Done Signal Status as Low
display ("circle_gen: done_looping with init high");
END
No
display ("circle_gen: done_looping with init low");
Set Busy Signal and Done Signal Status as Low
END
If(cur_step > 512)
No
END
Yes
Set Busy Signal High
Advance prev_[xyz] & set next_[xyz] to 0
display("circle_gen: inside looping with cur_step = %h > 512", cur_step)
END

Fig. 17 Flow chart for circle generation unit

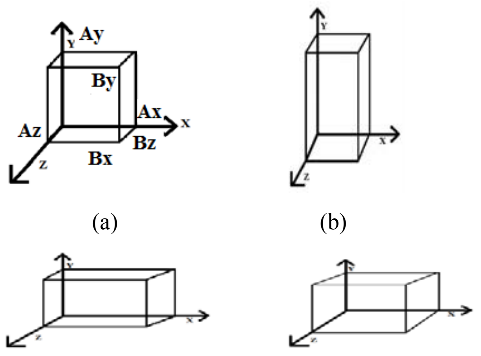
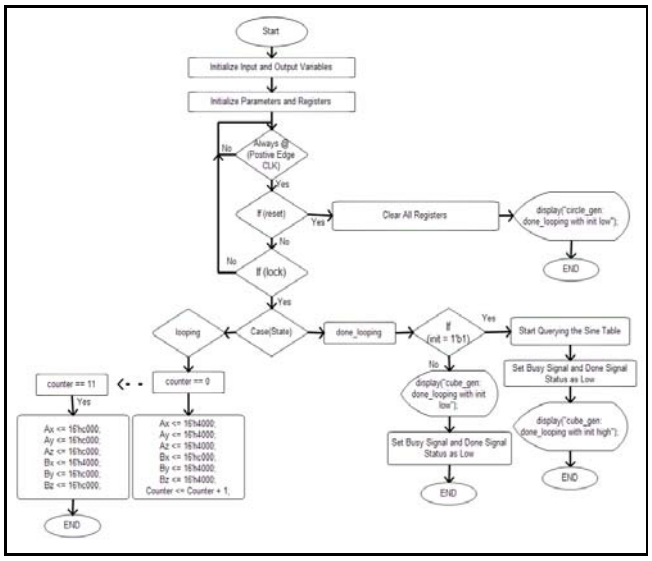


Fig. 18 Stages of cube generation

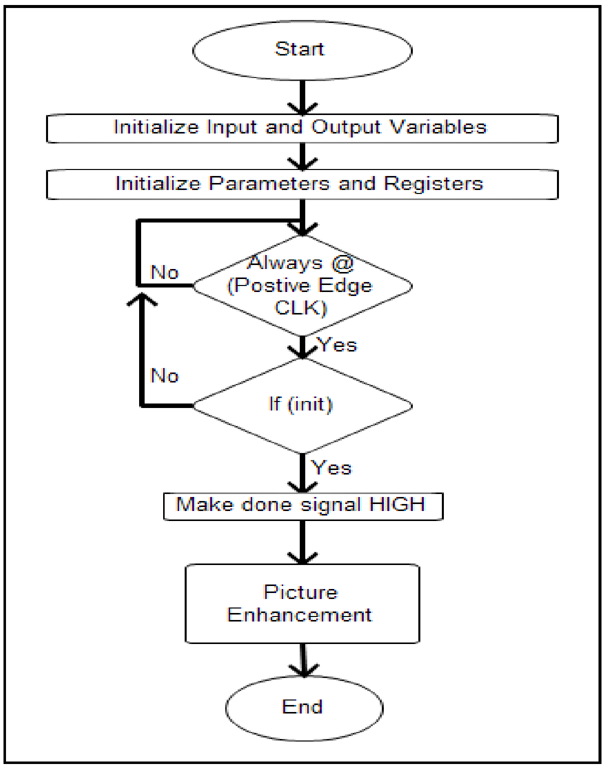
6.2.3. Scaling unit

Scaling is the process of resizing a digital image. Scaling is a non-trivial process that involves a trade-off between efficiency, smoothness and sharpness. As the size of an image is increased, so the pixels which comprise the image become increasingly visible, making the image appears "soft". The flow chart of scaling unit shown in Figure 7 is explains reducing an image will tend to enhance its smoothness and apparent sharpness. Apart from fitting a smaller display area, image size is most commonly decreased in order to produce thumbnails. Enlarging an image is generally common for making smaller imagery fit a bigger screen in full screen mode, for example. In "zooming" an image, it is not possible to discover any more information in the image than already exists, and image quality certainly suffers. However, there are several methods of increasing the number of pixels that an image contains, which evens out the appearance of the original pixels.



Start
Initialize Input and Output Variables
Initialize Parameters and Registers
Always @ (Postive Edge CLK)
No
Yes
If (reset)
Yes
Clear All Registers
display("circle_gen: done_looping with init low");
END
No
If (lock)
No
Yes
Case(State)
looping
done_looping
If (init = 1'b1)
Yes
Start Querying the Sine Table
Set Busy Signal and Done Signal Status as Low
display("cube_gen: done_looping with init high");
END
No
display("cube_gen: done_looping with init low");
Set Busy Signal and Done Signal Status as Low
END
counter == 0
counter == 11
Yes
Ax <= 16'hc000; Ay <= 16'hc000; Az <= 16'hc000; Bx <= 16'h4000; By <= 16'hc000; Bz <= 16'hc000;
Ax <= 16'h4000; Ay <= 16'h4000; Az <= 16'h4000; Bx <= 16'hc000; By <= 16'h4000; Bz <= 16'h4000; Counter <= Counter + 1;
END

Fig. 19 Flow chart for cube generation unit



Start
Initialize Input and Output Variables
Initialize Parameters and Registers
No
Always @ (Postive Edge CLK)
Yes
No
If (init)
Yes
Make done signal HIGH
Picture Enhancement
End

Fig. 20 Flow chart for scaling unit

6.2.4. Rotator unit

The rotation process is used to rotate or turn an object based on the angle of rotation required by the user. A rotation transformation is generated by specifying a rotation axis and rotation angle. Parameters are the rotation angle θ and a position (X_r, Y_r) called the rotation point about which the object is to be rotated as shown in Figure 8.

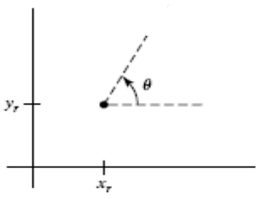


Fig. 21 Rotation of an object through angle θ about the position (X_r,Y_r)

6.3. RESULTS AND DISCUSSIONS

6.3.1. Stream controller

The main function of the stream controller is to issue control signals to the executable unit and to the stream register file. When an instruction of length 8-bit is issued as shown Figure 9, it is observed that the first two bits of the instruction are for destination address and third and fourth bits are for source address. The next four bits are opcode.

The program counter holds the address of the next instruction to be executed. When the external reset is asserted, the program counter is loaded with 0, indicating that the bottom of memory holds the next instruction that will be fetched. Under the action of the clock, for single cycle instructions, the instruction at the address in the program counter is loaded into the instruction register and the program counter is incremented. An instruction decoder determines the resulting action on the data paths and the ALU.

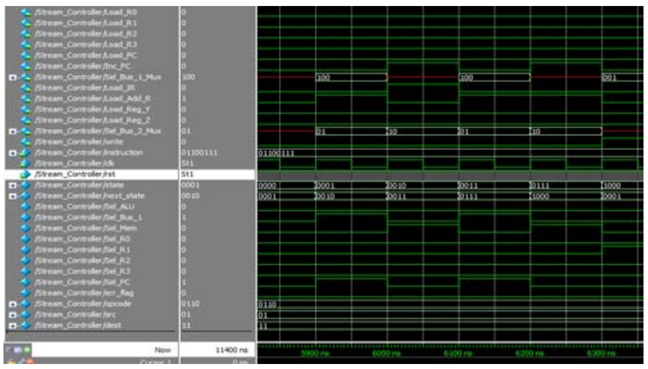


Fig. 22 Simulation result of stream controller

6.3.2. Pipelined adder

The simulation result of the pipelined adder is shown in Figure 10. The time taken to perform the compilation by using pipelined adder is 100 ns. In this model first the LSBs of the two numbers will be added and next the MSBs. The internal carry from the LSBs will be carried to the MSBs

6.3.2. Pipelined multiplier

The simulation result of the pipelined multiplier is shown in Figure 11. The time taken to perform the compilation by using pipelined adder is 60 ps. In this model each bit of the two numbers are multiplied parallel.

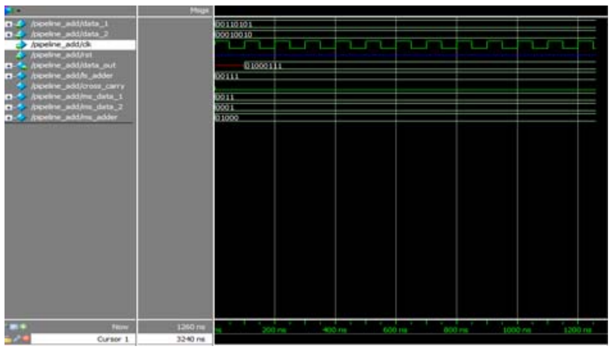


Fig. 23 Simulation Result of Pipeline Adder

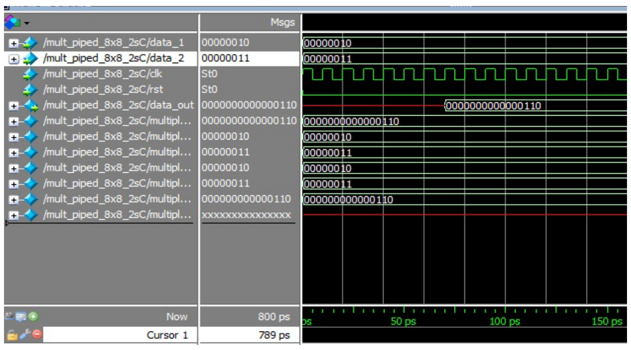

Msgs
/mult_piped_8x8_2sC/data_1 00000010
/mult_piped_8x8_2sC/data_2 00000011
/mult_piped_8x8_2sC/clk St0
/mult_piped_8x8_2sC/rst St0
/mult_piped_8x8_2sC/data_out 0000000000000110
Now 800 ps
Cursor 1 789 ps

Fig. 24 Simulation result of pipelined multiplier

6.3.3. Circle generator

This unit is used to draw the circular objects in an image. Initially, when the reset signal is high, and all other signals are low, registers in the unit will get cleared as shown in Figure 12. When reset signal is low and lock, init signals are high, at this time step size should be mentioned to draw a circle. It has been observed in the Figure 13 that the outputs [(Ax, Ay, Az), (Bx, By, Bz)] of the block carry the coordinates of the circle. Is has also been observed that for the given step size 0000000000000011, the shape of the circle is changed along Ax, Ay, Bx, and By axis.

6.3.4. Cube generator

This unit is used to draw the cubical objects in an image. Initially, when the reset signal is high and all other signals are low, registers in the unit will get cleared. When reset signal is low and lock, init signals are high, at this time step size should be mentioned to draw a cube shape objects. It can be observed in the Figure 13 that the coordinates of the cube changes according to the counter.

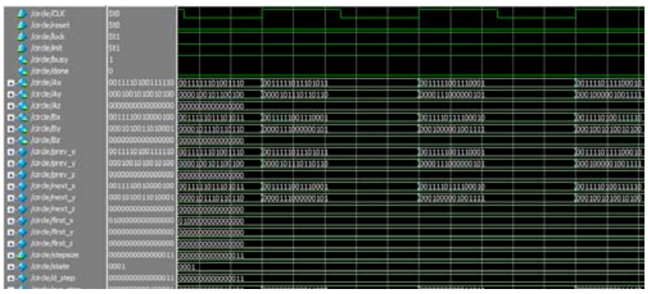


Fig. 25 Simulation result of circle generator

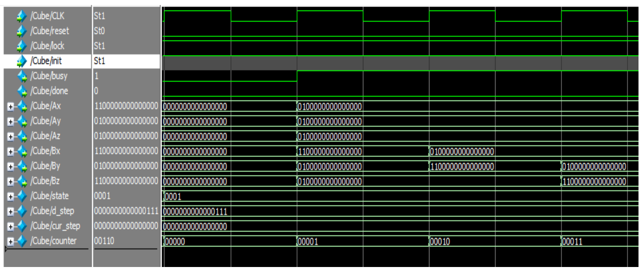

/Cube/CLK St1
/Cube/reset St0
/Cube/lock St1
/Cube/init St1
/Cube/busy 1
/Cube/done 0
/Cube/Ax 1100000000000000
/Cube/Ay 0100000000000000
/Cube/Az 0100000000000000
/Cube/Bx 1100000000000000
/Cube/By 0100000000000000
/Cube/Bz 1100000000000000
/Cube/state 0001
/Cube/d_step 0000000000000111
/Cube/cur_step 0000000000000000
/Cube/counter 00110
00000
00001
00010
00011

Fig. 26 Simulation result of cube generator

7. CONCLUSIONS

GPUs benefit from pipeline programmability. FPGAs provide a unique middle way between software and hardware, as the hardware itself can be configured as a processor where the software can run. One of the aims of this study was to investigate how this can be used for graphic system modelling. After theoretical design and an investigating implementation, the experience is that FPGAs can be used rather effectively for graphic system design.

During the study, it was found that synthesis tools from different vendors other than that of the FPGA can impose problems. Generally, a frustrating part of the problems that did occur could be blamed on the tools. It has also been observed that there is no clear methodology on designing an embedded stream processor that meets performance requirements and provides power efficiency. The number of clusters in the stream processor, the number of arithmetic units and the clock frequency – each can be varied to meet real-time constraints but can have a significant impact on power consumption.

From this study it has been concluded that the introduced single controller controls the data flow between host processor and GPU, within the arithmetic clusters thus reducing area and power requirements. Reconfigurable realisation of two of the principal components of a GPU allows us to conclude that a complete reconfigurable GPU can be realised. We conclude that the work can form the basis for designing a complete reconfigurable GPU.

8. REFERENCES

[1] Altera Corporation (2010) '*Introduction to Simulation of Verilog Designs Using ModelSim Graphical Waveform Editor'* United States of America [online] available from <ftp //ftp.altera.com/up/pub/Altera_Material/9.1/Tutorials/Verilog/ ModelSim_GUI_Introduction.pdf> [13 February 2012]

[2] Banerjia, S., Ozer, E., and Conte, T.M. (1998) Unified Assign and Schedule A *New Approach to Scheduling for Clustered Register File Microarchitectures*, *IEEE International Symposium on Microarchitecture,* 308-315

[3] Chen, Q.K. and Zhang, J.K. (2009), '*A Stream Processor Cluster Architecture Model with the Hybrid Technology of MPI and CUDA'*, IEEE International Conference on Information Science and Engineering, 86 - 89.

[4] Chittamuru, J., Euh, J., and Burleson, W. (2003), *'A Low-Power Content-Adaptive Texture Mapping Architecture for Real-Time 3D Graphics'*, University of Massachusetts Amherst [online] available from <http //www.citeseerx.ist.psu.edu http //viewdoc/summary?doi=10.1.1.132.2127> [12 December 2011].

[5] Doulos Inc., (2008) *OVM Golden Reference Guide,*Version 2.0 United Kingdom, Hamspire Doulos.

[6] Doulos Inc., (1996) *The Verilog Golden Reference Guide* Version 1.0, United Kingdom Doulos.

[7] Foley, J. D., Dam, A. V., Feiner, S. K., and Hughes, J. F., (1990), *Computer Graphics Principles and Practice in C*, 2nd edn., USA Addison-Wesley.

[8] Heckbert, P.S. (1986), *'Survey of Texture Mapping'*, *IEEE Computer Graphics and Applications,* 6, 56-67.

[9] Rajagopal, S., Cavallaro, J.R., and Rixner, S. (2004), *'Design Space Exploration for Real-Time Embedded Stream Processors'*, *IEEE Computer Society.*