システムＬＳＩ設計技術　97－8
（２０００．９．８）

3 次元座標変換のビットシリアル FPGA への実装

太田 章久† 一色 剛‡ 國枝 博昭‡

†東京工業大学 理工学研究科 電気・電子工学専攻
‡東京工業大学 理工学研究科 集積システム専攻

〒 152-8552 東京都目黒区大岡山 2-12-1

Tel : 03-5734-2574 Fax : 03-5734-2574

E-mail : {aohta,isshiki,kunieda}@ss.titech.ac.jp

あらまし 本稿では 3 次元座標変換のビットシリアル FPGA への実装について述べる。3 次元座標変換では扱う空間が広いため、データの処理に高い精度が要求される。一般的に、浮動小数点での処理が不可欠であるが、その処理の複雑さがハードウエアへの実装を困難にしている。特に FPGA の場合はその資源が限られているため十分な性能を得るのは難しい。我々はこれらの問題を踏まえて、ビットシリアル FPGA に適した浮動小数点の加算器と乗算器のアルゴリズムとアーキテクチャを設計し、それらを基に 3 次元座標変換の実装をおこなった。浮動小数点加算で約 89.3 MFLOPS、3 次元座標変換で約 7.2 M vertex/sec という結果が得られた。

キーワード 3 次元座標変換, 浮動小数点演算, ビットシリアルデータパス, FPGA

Implementation of 3D Geometry Engine on Bit-Serial FPGA

Akihisa Ohta†, Tsuyoshi Isshiki‡ ,and Hiroaki Kunieda‡

†Department of Electrical and Electronic Engineering, Tokyo Institute of Technology

‡Department of Communications and Integrated Systems, Tokyo Institute of Technology

2-12-1, O-okayama, Meguro-ku, Tokyo, 152-8552 Japan

Tel : 03-5734-2574 Fax : 03-5734-2574

E-mail : {aohta,isshiki,kunieda}@ss.titech.ac.jp

Abstract In this paper, we present our work on the implementation of 3D geometry engine to bit-serial FPGA. In 3D geometry transformation, the data processing is required the wide dynamic range and high accuracy because the object space is very large. Thus, it is necessary to process the floating-point arithmetic. However, the implementation is very difficult due to its complicated algorithm and structure. In case of FPGAs, we usually do not get high performance for the limit of their hardware resources. To solve these problems, we develop the floating-point operation algorithm and architecture with bit-serial pipelined datapath and the implementation on bit-serial FPGA. As a result, the floating-point adder is performing at about 89.3 MFLOPS and the 3D geometry transformation is about 7.2 M vertex/sec.

key words 3D geometry transformation, floating-point operation, bit-serial datapath, FPGA

1 Introduction

Current 3D computer graphics systems require high speed computing for a large amount of data and various algorithms. Usually, these processing are hardware–accelerated. The performance of 3D computer graphics hardware is depended on its operation speed and how algorithms it supports. In order to achieve faster and higher quality of graphics, the algorithm of 3D computer graphics is progressing at a rapid pace. However, as algorithm grows, dedicated hardwares are out of date due to their low flexibility. In addition, the design of custom chip takes a long time and large costs. Our solution of these problems is the implementation of 3D computer graphics rendering engine on reconfigurable system.

One of the feature of reconfigurable computing is more application specific adaptation and greater computational density than general purpose processors. However, the capability of FPGAs used as reconfigurable device is not sufficient for the requirement of current 3D computer graphics. In 3D computer graphics, floating–point operation is necessary for its dynamic range and accuracy in geometry representation and transformation. Effective implementation of floating–point operations on reconfigurable system is very hard and expensive because these operations require large area and complicated structure. There has been several studies about processing floating–point numbers in reconfigurable device [1]–[4]. They discussed the implementation of floating–point units on FPGA, the algorithm that need the acceleration capabilities of reconfigurable computing and the alternative formats which retain the benefit of a large dynamic range. Unfortunately, their results are not sufficient. In order to archive the high speed computing on the reconfigurable device, we develop a new FPGA for bit–serial pipeline datapath. We have successfully shown that our bit-serial FPGA architecture can achieve near 100% logic utilization on a number of bit-serial applications and guarantees a high clock frequency operation [5]. Our proposed approach is the floating–point operations on this bit–serial FPGA. We develop the floating–point operation algorithm and architecture suited for bit–serial FPGA.

This paper discusses the implementation of 3D graphics geometry engine on reconfigurable system. Especially, we focus on high speed geometry transformation architecture on our bit–serial FPGA. It begins with an introduction to geometry transformation algorithm and 3D geometry engine architecture. Next, we propose the bit–serial floating–point format and the the implementation of floating–point units. Finally, the performance of 3D geometry engine and conclusion are presented.

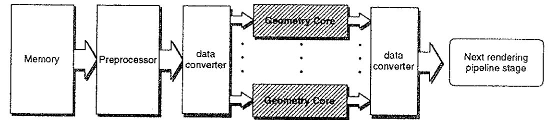

Figure 1: Geometry Engine System

2 3D Graphics

Generally, the process of 3D graphics is implemented as the graphics rendering pipeline [6]. There are three stages: application, geometry and rasterizer. The application stage is software-based implementation. The rasterizer stage is hardware-accelerated. The geometry stage is located on general purpose CPUs, custom chip, or both. In case of the implementation on CPUs, they have supported geometry acceleration on the chip.

The major function of application stage is to generate the 3D object model and to decide the locations of objects, light sources and viewpoint. The geometry stage generates the transformed and projected vertices, colors and texture coordinates. This stage is further divided into the functional stages: geometry transformation (model & view transformation), lighting, projection, clipping and screen mapping. The goal of rasterizer stage is generating the pixels, which have appropriate color to render image correctly, from a color, depth value, texture coordinates with each vertex (from geometry stage). This stage handles per–pixel operations (unlikely the geometry stage is per–vertex operations).

The next section discusses the implementation of 3D geometry engine.

3 Bit–Serial 3D Geometry Engine

3.1 Geometry Transformation

In order to display 3D objects on the screen, objects are transformed into several different coordinate system. These transformation is called "*geometry transformation*". Geometry transformation takes the following two steps.

1. translation "*object coordinate system*" into "*world coordinate system*"

2. translation "*world coordinate system*" into "*screen coordinate system*"

The object is composed of several polygons. A vertex of polygon placed on object coordinate system

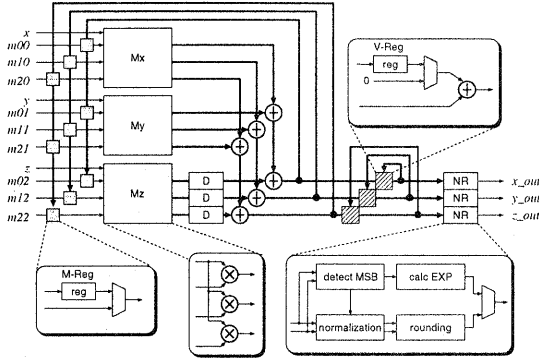

Figure 2: Geometry core architecture

is given by a vector $(x_o, y_o, z_o)^T$. A vector on world coordinate system $(x_w, y_w, z_w)^T$ is given below by

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = M_{ow} \cdot \begin{pmatrix} x_o \\ y_o \\ z_o \end{pmatrix} + \begin{pmatrix} x_{o0} \\ y_{o0} \\ z_{o0} \end{pmatrix} \quad (1)$$

,where M_{ow} is transform matrix and $(x_{o0}, y_{o0}, z_{o0})^T$ is 0–point on world coordinate system.

A vector on screen coordinate system $(x_s, y_s, z_s)^T$ is given below as

$$\begin{pmatrix} x_s \\ y_s \\ z_s \end{pmatrix} = M_{ws} \cdot \begin{pmatrix} x_w - x_v \\ y_w - y_v \\ z_w - z_v \end{pmatrix} \quad (2)$$

,where M_{ws} is transform matrix and $(x_v, y_v, z_v)^T$ is position of view point on world coordinate system.

Thus, the transformation object to screen coordinate system derived by equation (1) and (2) is represented as follows:

$$\begin{pmatrix} x_s \\ y_s \\ z_s \end{pmatrix} = M_{ws} \cdot M_{ow} \cdot \begin{pmatrix} x_o \\ y_o \\ z_o \end{pmatrix} + M_{ws} \cdot \begin{pmatrix} x_w - x_v \\ y_w - y_v \\ z_w - z_v \end{pmatrix} \quad (3)$$

This transformation is handled by the floating–point operations.

3.2 Architecture

Figure 1 shows the architecture of geometry engine system. The geometry engine contains preprocessor, two data converter and geometry core. The feature of each component is discussed as following:

- *"Preprocessor"* is responsible for the calculation of the transform matrix (M_{ws}, M_{ow}).
- Each coordinate vertex is converted from standard floating–point format to bit–serial floating–point format by *"data converter"*.
- Three type of operations, multiplication of matrix, multiplication of matrix and vector and addition of vector are efficiently implemented on *"geometry core"*.

The outputs of *"geometry core"* converted to standard floating–point format, if necessary, are transfered to next rendering pipeline stage.

3.3 Geometry Core

The algorithm of geometry transformation indicated by equation(3) is indicated as the following:

```
For every object do {
  A = M_ws · M_ow
  B = M_ws · (x_w − x_v, y_w − y_v, z_w − z_v)^T

  For every vertex do {
      F = A · (x_o, y_o, z_o) + B
  }
}
```

A and B is calculated per object. At nest loop, F is calculated per vertex composed polygon. This algorithm is operated with three sequential stages At first stage to calculate A, the multiplication of matrix is operated with three passes through the geometry core. These results are buffered at *"M-Regs"* through feedback loop. At next stage, we divide the calculation of B into following two phases:

$$\begin{aligned} B_t &= M_{ws} \cdot (x_w, y_w, z_w)^T \\ B &= B_w + M_{ws} \cdot (x_v, y_v, z_v)^T \end{aligned}$$

The value of B_t is temporarily stored at *"V-Regs"*, and then next operation overwrite the result B at *"V-Regs"*. At final stage, the geometry core calculate F with the vertices $(x_o, y_o, z_o)^T$ and buffered data A and B. Generally, one object is composed of several hundred vertices. Therefore, the majority of operation time for geometry transformation spends the calculation of F, so the preprocess dedicated the calculation A and B is a modicum overhead.

4 Bit–Serial Reconfigurable Architecture

One of the distinctive characteristics of bit-serial circuits is that the connectivity inside the cell is dense, while the connectivity between bit-serial cells

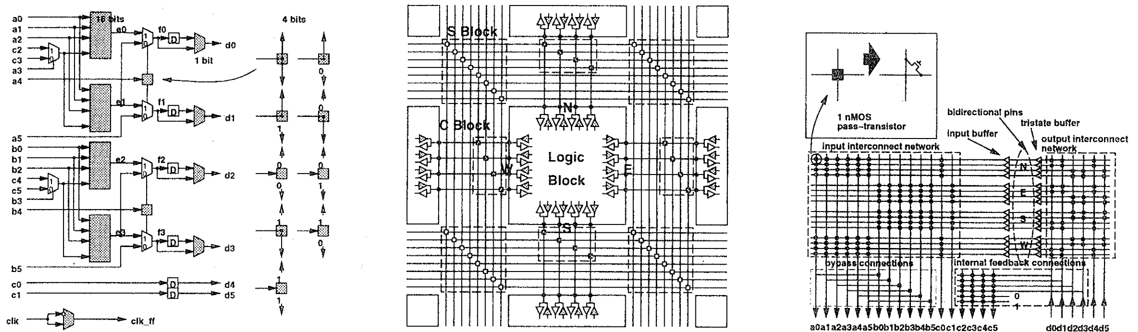

Figure 3: Bit-serial FPGA architecture: left is logic block architecture, middle is external block routing architecture and right is internal routing architecture.

is sparse. Figure 3 shows the bit-serial FPGA architecture. This FPGA is SRAM based architecture and constructed from logic block, routing block (S, C block), and I/O block. Our strategy here is to increase the logic capacity of the logic block and absorb the dense interconnection inside the logic block to reduce the inter-block routing resource.

The performance of bit-serial FPGA is summarized by Table1.

Table 1: Estimated performance of our bit-serial FPGA chip.

area (LB, SB, CB×2)	$385 \times 407\mu^2$
area (total)	$3,500 \times 3,500\mu^2$
transistor count	200k transistors
max. gate/block	~70 gates
max. gate/chip	~4500 gates
clock frequency	156 MHz
(assume 4 manhattan distance routing)	
16-bit multiplication (×2)	19.5 MOPS
8-bit multiplication (×4)	78 MOPS
16-bit addition (×64)	624.64 MOPS
8-bit addition (×64)	1.25 GOPS

4.1 Logic block

Figure 3(left) shows the logic block architecture. The two multiplexers in front of the LUTs are targeted mainly for carry-save operations which are frequently used in bit-serial computations. Programmable switches connected to inputs a_4 and b_4 control the functionality of the four multiplexers at the output of LUTs. As a result, 2 LUTs can implement any 5-input functions. The final outputs d_0, d_1, d_2, d_3 can either be the direct outputs from the multiplexers or the outputs from flip-flops. Two flip-flops are added (inputs c_0 and c_1) to implement shift registers which are frequently used in bit-serial operations to synchronize pipeline. Moreover, one of the major feature of logic block is that we can implement 16-bit barrel shifter on one LUT. This advantage indicates the high performance due to the reduction of the external routing for large synchronization register.

4.2 Routing block

Large portion of the output signals are only used inside the same logic block in bit-serial designs. Since our aim in making the logic block large is to absorb the feedback routing inside the logic block, we provide a rich feedback routing resource inside the logic block. The routing is divided into two-levels : *external block routing* and *internal block routing* shown in Figure 3(middle and right). Advantages of our two-level routing architectures can be summarized as follows : The large number of input and output signals of the logic block would create a significant capacitive load due to the drain capacitance of the pass-transistors on the routing segments. By our routing scheme, the intermediate routing resource inside the logic block (hence the *two-level routing*) enables us to insert buffers at the logic block pins which effectively isolates the capacitive load of the drain capacitance of pass-transistors from the routing segments. The routing delay through the single-length routing segment is greatly reduced. This fact also leads to power reduction.

All logic block outputs can be routed back to itself without consuming any external routing resource. Also, connections between adjacent logic blocks which frequently occurs in bit-serial circuits is implemented via C-blocks without consuming any external routing resource as well. These features are also not seen in Xilinx 4000 FPGA.

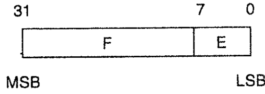

Figure 4: Bit-serial floating-point format: F is fraction (24 bits) and E is exponent (8 bits)

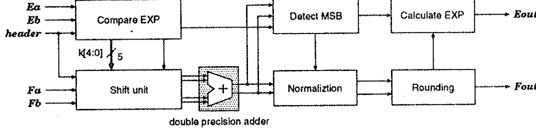

Figure 5: Bit-serial floating-point adder

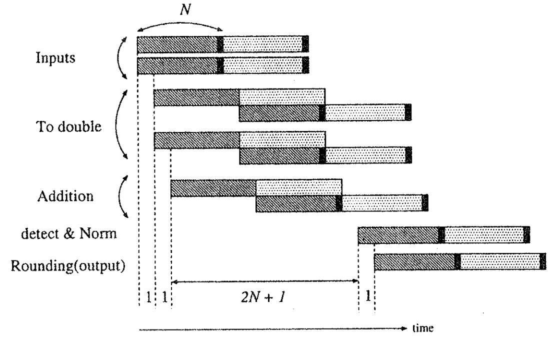

Figure 6: Timing chart of bit-serial floating-point adder

5 Floating Operation on Bit-Serial FPGA

The implementation of floating-point operations requires large area and complicated structure because of bit alignment of different exponent data and normalization. Especially, in case of FPGAs, this problem occurs to drastically decrease the performance because of their physical limitations, low device utilization and low operation frequency. Therefore, it is difficult to implement floating-point units into real-time 3D computer graphics rendering pipeline. As solution of these problems, we propose the approach to implement bit-serial floating-point unit on bit-serial FPGA.

5.1 Bit-Serial Floating-Point Format

The bit-serial single precision floating-point format is shown in Figure 4, where F is the fraction (24 bits) and E is the exponent (8 bits). This format is slightly different from standard floating-point format (IEEE 754). The fraction is represented by 2's complement number. The decimal point is implicitly positioned at the right of MSB. The fraction value is within the range $[-1, 1)$. The exponent value is biased to $E_0 = 127$. Therefore, the actual value is calculated as $F \times 2^{E-E_0}$.

5.2 Bit-Serial Floating Adder

5.2.1 Algorithm

The algorithm of bit-serial floating adder is shown in Figure 7. The first step is to choose the number with the greater magnitude of exponent. If B is greater than A, we swap A with B. The difference number m $(= E_A - E_B)$ is calculated. This number is used as the input of bit-serial shifter. The next step is the alignment of fraction. In order to to keep the data accuracy, the fraction is extended to double precision and shifted right one too many to avoid the overflow. In the result, the fraction of A, F_A, is shifted left $N-1$ and the fraction of B, F_B, is shifted left $N-1-m$. Then, we calculate addition by using 2's complement bit-serial double precision adder. The next step is normalization and rounding. Normalization is to shift the result fraction to the left until at least the value is within range $[-1, -0.5)$ or $[0.5, 1)$. However, this normalization is not necessity to influence of increasing the data accuracy. We normalize the data with this strict way for easy converting bit-serial floating format into standard floating format. If bit-serial floating point data is not strict normalized, it is not able to define the data as unique. For example, $(0.0100)_2 \times 2^{-1}$ equals to $(0.100)_2 \times 2^{-2}$. This fact causes of the complicated implementation of converter. Rounding double precision into single precision variable is implemented with round-to-nearest-even as same as standard floating-point format [7]. The final exponent value is calculated by the result of normalization and rounding steps.

5.2.2 Implementation

A block diagram of bit-serial floating adder is shown in Figure 5. First of all, "compare EXP" evaluate the two exponents and calculate the value "m". "Shift unit" makes the fraction to double precision. Bit-serial double precision data is represented by two wires. Therefore, the cycle time to calculate double precision data equals to that of single precision data. At the same time, the data is aligned with the difference in the exponent values "m". These shifted data is inputted to bit-serial double precision adder.

A 16-bit bit-serial barrel shifter is implemented

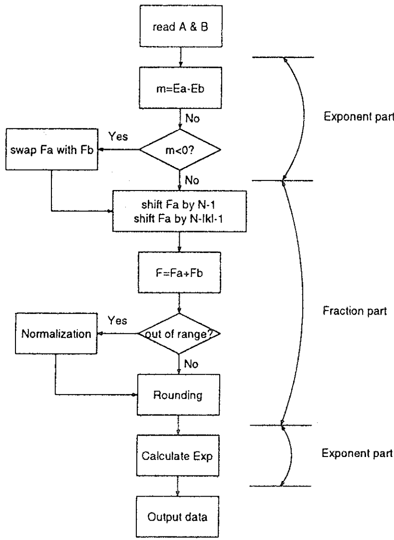

Figure 7: Algorithm of bit–serial floating–point adder

on one LUT (1/4 CLB) on bit–serial FPGA. We can avoid to route the signal with external routing. This feature allow the FPGA to operate with high frequency without routing penalty.

Figure 6 shows the timing chart of bit–serial floating–point adder, where N is data length (24 bits). Data is processed with bit–serial pipeline fashion. The sampling period equals to the data length N and the latency is $2N + 4$. The detection of MSB used as normalization spend long latency.

Table 2 describes the comparison between Xilinx XC4085XL FPGA (speed grade "09" which is the fastest device) [8] and our bit–serial FPGA. Significant points in this comparison is in the logic and routing delays. While our FPGA uses 0.5μ process (0.6μ gate length), XC4000XL uses 0.35μ process. These results are due to the optimized logic block and routing architectures of our bit–serial FPGA. Throughput is calculated for whole chip area. XC4085XL has 56×56 CLBs and our bit–serial FPGA die size is assumed by $1cm^2$.

5.3 Bit–Serial Floating Multiplier

A block diagram of bit–serial floating multiplier is shown in Figure 8. Bit–serial floating multiplication is simpler than addition. It is not necessary to align the data before multiplication. The fractions is calculated by 2's complement multiplier. 2's complement multiplier is consist of 24 processing elements cascaded into a one dimensional array. Each bit of multiplier is transmitted through the linear array, and captured

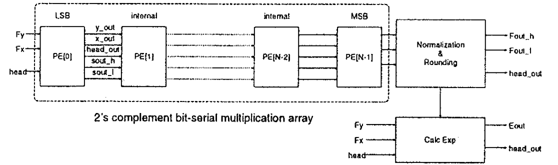

Figure 8: Bit–serial floating–point multiplier

Table 3: Mapping on bit–serial FPGA

Function	Multiplier		Adder		Normalization	
	Exp.	Frac.	Exp.	Frac.	Exp.	Frac.
#CLB	2	24	7	8	2	18
Total	26		15		20	

at the corresponding cell. Each bit of the multiplicand is transmitted through the linear array, and each cell calculates the partial product. The output exponent is only calculated by the addition of two inputs.

6 Performance Comparison of 3D Gemetry Engine

The result of mapping on bit–serial FPGA is shown in Table 3. Total number of CLB is calculated as following:

$$\begin{aligned} Total &= \underbrace{26 \times 3 \times 3}_{multiplier} + \underbrace{15 \times 3 \times 3}_{adder} + \underbrace{20 \times 3 \times 3 + 6}_{normalization\®} \\ &= 435. \end{aligned}$$

Table 4 describes the performance of *"geometry core"*. Here, we compare the performance of the implementation on bit–serial FPGA with the standard cell based ASIC as same process technology. We can see from these results that the high operation frequency of bit–serial implementation is of great advantage to parallel implementation. In the comparison of FPGA and ASIC, the difference of logic density affects the throughput since the operation frequency is about the same.

7 Conclusion

In this paper, we propose the implementation of 3D geometry engine on bit–serial FPGA. We have shown the bit–serial floating–point adder and multiplier fitting in bit–serial FPGA handled our original bit–serial floating–format. We focus on adder implementation and get the good results as compared to Xilinx FPGA. Moreover, we propose the algorithm and

Table 2: Estimated performance of bit-serial floating–point adder

Device	Process	CLB count	Logic delay	Routing delay	Frequency	Throughput
Bit–serial FPGA	0.5 μm	35	2.21 ns	4.17 ns	156.7 MHz	89.29 MFLOPS
XC4085XL	0.35 μm	207	6.93 ns	61.2 ns	14.68 MHz	7.06 MFLOPS

Table 4: Performance of geometry core: Chip size is $1cm^2$. The process technology for implementation ASIC is $0.5\mu m$, 2–metal layer CMOS as same as bit-serial FPGA.

	Bit-serial FPGA	ASIC (bit-serial)	ASIC (bit-parallel)
Sampling period [clock]	32	24	1
Latency [clock]	99	196	4
Frequency [MHz]	156	141.1	21.7
Throughput [vertex/sec]	7.15	16.6	10.86

architecture of 3D geometry engine. Actually, the geometry core is mapped on bit–serial FPGA manually and the estimation of performance is shown.

Although we do not get a significant performance over current commercial rendering engine. However, we have successfully shown good results as implementation on reconfigurable device with the comparison of other FPGA.

ACKNOWLEDGMENT

Authors would like to thank the members of CAD21 Research Body of Tokyo Institute of Technology, VDEC which provides us an opportunity to implement VLSI, and members of Kunieda Laboratory for their suggestion and cooperations.

References

[1] L.Louca, T.A.Cook, and W.H.Johonson, "Implementaion of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs," *Proc. IEEE Symp. FPGAs on FPGAs for Custom Computing Machines*, pp.107–116, 1996.

[2] B.Fagin and C.Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic," *IEEE Transaction on VLSI*, vol.2, no.3, pp.365–367, 1994.

[3] N.Shirazi, A.Walters, and P.Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machine," *Proc. IEEE Symp. FPGAs on FPGAs for Custom Computing Machines*, pp.155–162, 1995.

[4] W.B. Ligon III, S.McMillan, G.Monn, and et al. "A Re-evaluation of the Practicality of Floating–Point Operations on FPGAs," *Proc. IEEE Symp. FPGAs on FPGAs for Custom Computing Machines*, pp.206–215, 1998.

[5] A.Ohta, T.Isshiki, and H.Kunieda "New FPGA Architecture for Bit–Serial Pipeline Datapath," *Proc. IEEE Symp. FPGAs on FPGAs for Custom Computing Machines*, pp.58–67, 1998.

[6] T.Möller and E.Haines "Real–Time Rendering," A K Peters, Ltd., 1996.

[7] D.A.Patterson and J.L.Hennesy, "Computer Architecture: A Quantitative Approach," 1990.

[8] Xilinx Inc., "Programmable Gate Array Databook," 1996.