Hardware Architecture Design and Mapping of ‘Fast Inverse Square Root’ Algorithm

Saad Zafar, Numair Zulfiqar
College of Electrical and Mechanical Engineering
National University of Science and Technology
Pakistan
saadzafar31@ce.ceme.edu.pk

***ABSTRACT*: *The Fast Inverse Square Root algorithm has been used in 3D games of past for lighting and reflection calculations, because it offers up to four times performance gains. This paper presents a hardware implementation of the same algorithm on an FPGA board by designing the complete architecture and successfully mapping it on Xilinx Spartan 3E after thorough functional verification. The results show that this implementation provides a very efficient single-precision floating point inverse square root calculator with a practically accurate result being made available after just 12 short clock cycles. This performance measure is far superior to the software counterpart of the algorithm, and is not processor dependent like rsqrtss of x86 SSE instruction set. Results of this work can aid FPGA based vector processors or graphic processing units with 3D rendering. The hardware design can also form part of a larger floating point arithmetic unit for dedicated reciprocal square root calculations.*

Keywords: X5F3759DF, Digital Design, FPGA, Reciprocal Root, Reconfigurable, Verilog

Received: 24 November 2013, Revised 18 January 2014, Accepted 25 January 2014

1. Introduction

The ‘*Fast Inverse Square Root*’ algorithm or often referred to simply as 0x5f3759df is a method for computing the reciprocal square root, $y = 1/\mathrm{sqrt}(x)$. This algorithms has a mysterious origin with considerable confusion about where it was developed and by whom [1]. Due to no formal derivation and lack of conventional mathematical development, there has been little to none research on why this particular algorithm works as well as it does – achieving approximately four times the performance of previous approaches [2].

This unusual algorithm was first spotted in *Quake III* source code [3] and later found use in titles such as *Crystal Space* and *Titan Engine*. The working of this algorithm was studied by David Eberly in [4] where an attempt was made to give mathematical explanation for the working, especially about the part involving “*magic number*”. A slight flaw in Eberley’s explanation is pointed out in [2], and in this report a more formal derivation is suggested. Proposals for better magic numbers and how this technique can be extrapolated to other computational problems is also encouraged.

The Quake III game appeared at a time when reciprocal square roots were still calculated by means of software. There was no specialized machine-level instruction to carry out the task. However, due to the growing importance and usage of $x^{-1/2}$ operations, modern *x*86 processors have incorporated the *rsqrtss* in their SSE instruction sets. All inverse square root directives in the compiler are assembled into *rsqrtss* machine level instruction, and hence the speed of operation is considerably improved [5]. AMD K7 processor's implementations of this instruction is discussed in [6]. The paper shows that the initial seed for Newton-Raphson method is obtained by means of a lookup table, and this value is then further improved by successive iterations.

Calculation of fixed point square root calculation is discussed in [7]; but the subject algorithm of this paper handles floating point inputs allowing a larger range. A detailed FPGA/ASIC implementation of square root calculation with emphasis on low-cost and low-power is presented in [8].

This paper will attempt to port the original Fast Inverse Square Root algorithm to an FPGA based hardware implementation. Unlike the K7 SSE implementation, no lookup ROM will be stored and used for fetching the initial guess, but instead a pure arithmetic transformation will be carried out in keeping with the essence of original algorithm. Attempt will be made to utilize the parallel architecture of FPGA board to further optimize the performance of intermediate calculations. The reconfigurable architecture of FPGA can also be used to try and experiment with different magic values, to achieve an optimum between speed and accuracy. The main objective of this paper is to present a working hardware equivalent of this algorithm, which has not been tried yet other than some microprocessor based designs.

The paper is organized into different sections: II.A starts the design by exploring algorithm from software viewpoint. Section II.B proposes the architecture design by employing top-down approach. Section II.C implements this design on FPGA and discusses modules that make up the design. Then, section III presents synthesis and timing results. Some simulation waveforms are also presented for performance analysis. Lastly, section IV concludes the paper by highlighting some possible application domains and how the work can be further improved.

2. Architectural Details

2.1 Algorithm overview

Before delving into hardware mapping of this algorithm, it's important to understand the working in the original software form. The algorithmic details of Fast Inverse Square Root is contained in the following steps:

• *Step* 1: Store the input real number *x* in single-precision floating point format.

• *Step* 2: Calculate and store half the input number in xhalf, also as floating point.

• *Step* 3: Cast *x* to integer and store in a new variable *i* which is in int32 format.

• *Step* 4: Shift the variable *i* to the right by one and then perform conventional arithmetic subtract from 0x5f3759df.

• *Step* 5: Restore the floating-point equivalent of the number.

• *Step* 6: Perform an iteration of Newton-Raphson to get the result. This step can be repeated to achieve better accuracy, but generally a good value is reached after just the first iteration.

The flowchart depicting this scheme is provided in Figure 1.

2.2 Hardware design

The hardware architecture follows directly from the software implementation. However, unlike the software flowchart of Figure 1 , the hardware mapping need not be sequential. FPGA implementations allows great degree of operational parallelism and this feature needs to be exploited in the architecture to improve performance.

Steps 1, 2, 3, 4 and 5 can all be carried out concurrently. As soon as a new value of *x* is obtained, two independent sub-modules can start preparing the operands xhalf and *y*0. Note that *y*0 denotes the initial guess for Newton-Raphson method. The *x* value can be fed into a scalar floating point multiplier to obtain one-half the original value. Meanwhile, the same *x* value is passed through binary shifter and then subtractor.

The last partition of this design takes in as input xhalf and *y*0 to carry out Netwon-Raphson iteration. A high-level view of this scheme is given in Figure 2. From this scheme it is possible to break down the chain of steps into two stage process.

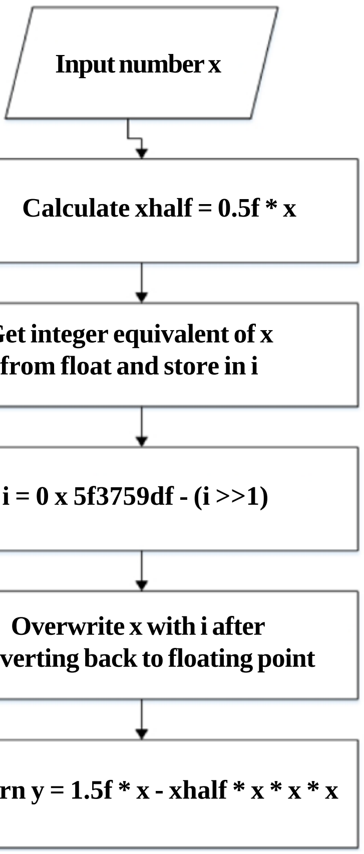
Input number x
Calculate xhalf = 0.5f * x
Get integer equivalent of x from float and store in i
i = 0 x 5f3759df - (i >>1)
Overwrite x with i after converting back to floating point
Return y = 1.5f * x - xhalf * x * x * x

Figure 1. Flowchart representation of Fast Inverse Square Root algorithm

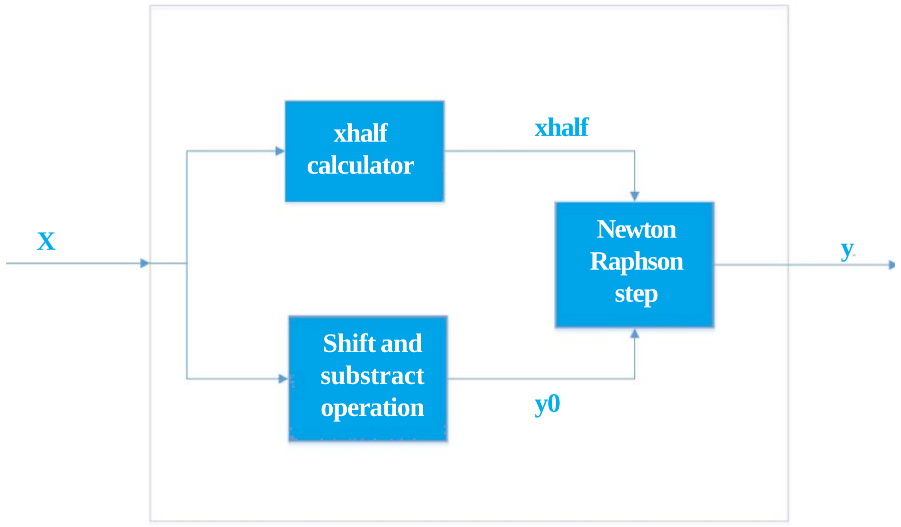

Figure 2. High level design partition

The architecture of xhalf calculator unit depends on two registers – one for latching the value on x input wire, and other to latch the xhalf wire. The xhalf will be calculated using a floating point multiplier that stores its result in the xhalf_reg register. The multiplier also takes the scalar value 0.5 as input for multiplication operation. This scheme is shown in Figure 3 . The bottom branch of Figure 2 corresponds to generation of initial guess, *y*0, as seed for Newton-Raphson iteration. This branch will first perform a shift right by one operation and then subtract the result from 0x5f3759df. A shift register is not used for this purpose

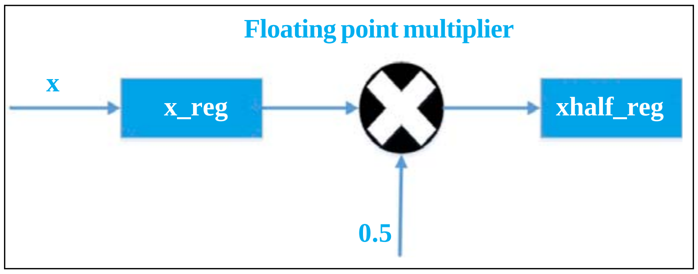

Figure 3. Calculating and storing xhalf

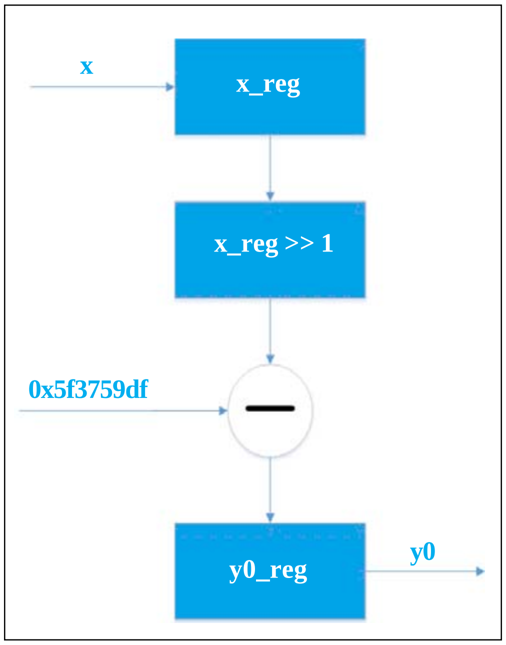

Figure 4. Hardware design of shift and subtract operation

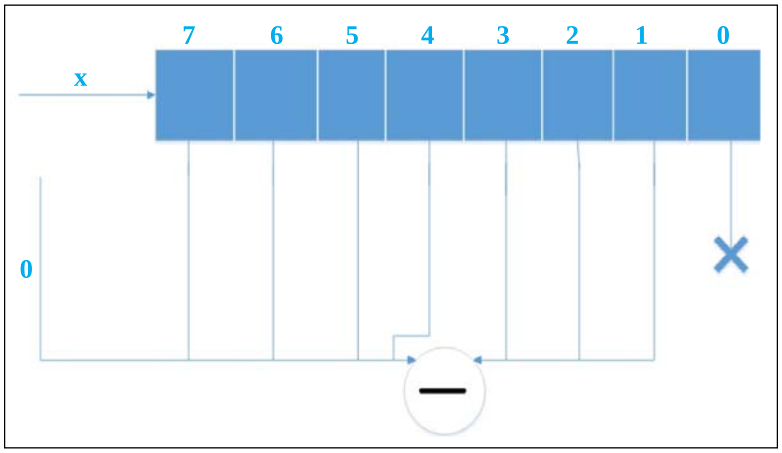

Figure 5. Static shift right and input to subtractor

in the design because it will waste a clock cycle just for performing a single shift. Instead, a hard-wired approach where a bus originating from *x* register, and suitably concatenated with zero-padding is fed to the subtractor. This particular design choice will ensure that *y*0 is available almost instantaneously whenever a change in *x* occurs. The design methodology is captured in the diagram of Figure 4. An expanded view of *x*_reg >>1 block to demonstrate how hard-wired shift right is carried out is given in Figure 5. The last design step is to map the Newton-Raphson process in hardware. The equation for this process is:

$$y = y0 * 1.5f - y0 * y0 * y0 * \text{xhalf}$$

Unfortunately, it is the slowest and most cycle consuming operation of the entire algorithm. Although xhalf and $y0$ are made available almost instantaneously from previous blocks but they have to be fed into cascaded floating point multipliers which lead to a final floating point subtractor. Each of these multiplication/subtraction operation taxes clock cycles and equation has to be implemented in its original form without much optimization possible. An illustration of this mapping is given in Figure 6.

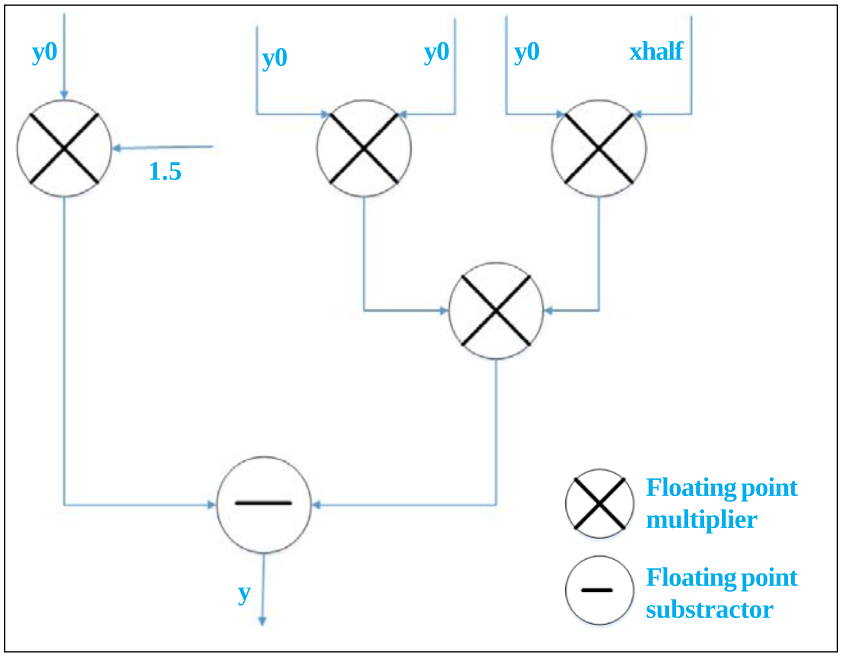

Figure 6. Newton raphson equation hardware mapping

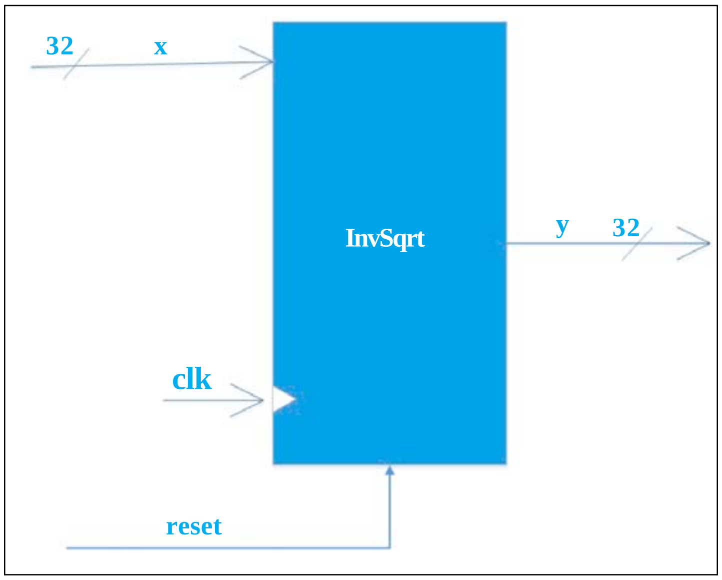

Figure 7. Block diagram of top-module InvSqrt

2.3 Hardware Description and Implementation

To realize the design presented previously, it has to be translated into a hardware description. For this purpose, Verilog HDL is utilized to write the hardware and data flow paths. The top-level module, called InvSqrt receives the number *x* and also returns the answer *y* in float. A block diagram of this module is given in Figure 7.

Name	Direction	Width (bits)	Description
x	In	32	Input number to undergo inverse square root
clk	In	1	Clock signal to synchronize operations and transfers
reset	In	1	Asynchronous reset to restore every register to default value
y	Out	32	Result of performing inverse square root operation on *x*

Table 1. Port structure of InvSqrt module

A summary of ports for InvSqrt module is given in Table 1.

Next step is to include a floating point unit (FPU) in the hierarchy which will carry out the required multiplications and subtraction. There are two implementation choices – first, to write a floating-point unit module from scratch and instantiate it within the top-level module. However, this effort is not aligned with the paper goals and time-consuming in terms of description and verification. The second choice is to incorporate an already created floating point unit IP core and use it in this setting. These cores are thoroughly verified by their developers and only minor port adjustment/customization is required to meet our requirements. Xilinx CORE Generator System [9] is selected for creating and customizing the floating point modules. The reason being that it is already included with Xilinx ISE and provides architecture-specific core generation which helps to maximize performance. While customizing the floating-point multiplier core, full usage of all four MULTI18X18S blocks (embedded in XC3S500E FPGA) is made to enhance the performance of resulting module. The block diagram of generated multiplier core is given in Figure 11 . There are five instantiations of this module inside InvSqrt module – four are consumed in implementing Newton-Raphson equation multiplications and the fifth is used to multiply the value of *x* by 0.5. Note, that an alternative framework for highest performance would have been to convert all the floating point values to their fixed point Int32 representations and then implement all these arithmetic operations using this form. This approach would have required a float-to-fixed converter before arithmetic steps and a fixed-to-float converter to return result in IEEE754 format. In this architecture, the algorithm could be represented in its entirety using behavioral description and then synthesizer can help in performance and area optimizations. But, this approach is not adopted in this paper because of the crucial factor of range. The fixed point representation is handicapped by smaller range of values that can be stored and this limits the usefulness of the module.

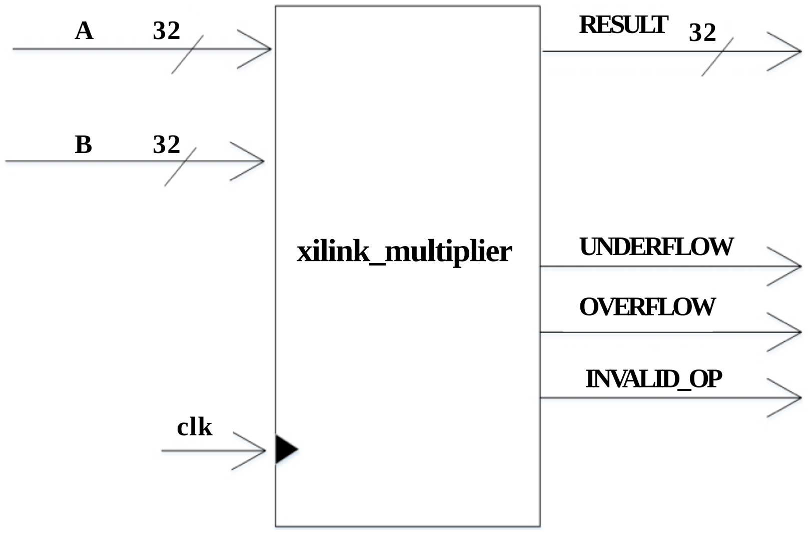

Figure 8. Block diagram of generated floating point multiplier

A description of the ports and their properties is summarized in Table 2Port structure of xilinx_multiplier module.

Second important component of FPU is the subtractor which is generated as xilinx_subtractor. There is only going to be one instantiation of this module inside InvSqrt, and this module will perform the final subtraction of NR iteration as shown by Figure 6. Again the module is generated using Xilinx CORE generator wizard, and its block diagram is given in Figure 9. Most port pins carry the same meaning as the one listed in Table 2; however, the RESULT port now gives answer of subtracting B operand from A operand.

Name	***Direction***	***Width (bits)***	***Description***
clk	In	1	Clock signal to synchronize operations and transfers
A	In	32	First single precision operand for multiplication
B	In	32	Second single precision operand for multiplication
RESULT	OUT	32	The result of performing multiplication $A * B$
UNDERFLOW	OUT	1	Flag raised to indicate when result is too small to be represented in 32 bits
OVERFLOW	OUT	1	Flag raised to indicate when result is too large to be represented in 32 bits
INVALID_OP	OUT	1	Flag to indicate request for an invalid operation

Table 2. Port structure of xilinx_multiplier module

3. Results

The Verilog description of module was successfully synthesized using XST tool and mapped on Digilent's Spartan 3E (XC3S500E) Starter board. The utilization summary is given in Table 3.

Device utilization summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	748	9, 312	8%
Number of 4 input LUTs	1, 815	9, 312	19%
Number of occupied Slices	1, 108	4, 656	23%
Number of Slices containing only related logic	1, 108	1, 108	100%
Number of Slices containing unrelated logic	0	1, 108	0%
Total Number of 4 input LUTs	1, 939	9, 312	20%
Number used as logic	1, 814		
Number used as a route-thru	124		
Number used as a Shift registers	1		
Number of borded IOBs	16	232	6%
IOB Flip Flops	1		
Number of RAMB 16s	16	20	80%
Number of BUFGMUXs	1	20	4%
Number of MULT18x18SIOs	2	20	10%
Average Fanout of Non-Clock Nets	3.23		

Table 3. Device utilization summary on XC3S500E

Also, the resulting clock period is 6.180ns allowing a Maximum Frequency of 161.812 MHz.

A self-checking and time agnostic test-bench was used for carrying out thorough functional verification of the InvSqrt module.

Intensive testing over a range of random values did not produce any unexpected results and every answer conformed to correct outputs.

A representative clock by clock simulation showing the evolution of waveform when the input *x* is forced to be 16 is given in Figure 10 at the bottom of this page. The output y equals 0x3e7f910e in floating point which equals 0.24957678 in fixed point.

The evolution of xhalf and *y*0 signals is depicted in Figure 11 overleaf. Note that *y*0 signal is made available almost instantly after input port *x* changes to a new value.

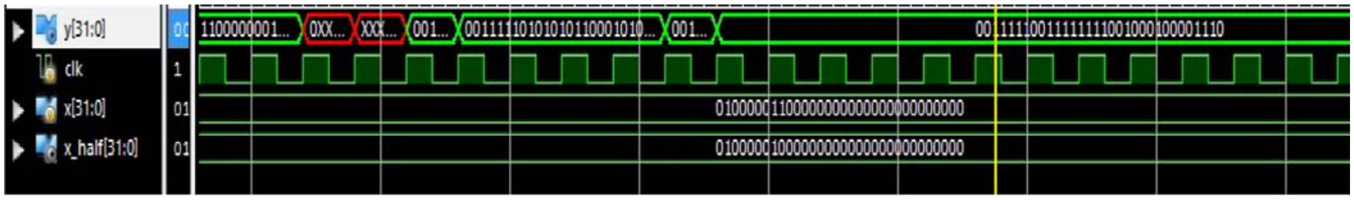

Figure 10. Signal waveform when input at *x* is 16 (fixed point)

4. Conclusion and Suggested Improvements

The Inverse Square Root calculator algorithm was successfully designed, described in Verilog, synthesized and finally mapped on the FPGA board. The final result is made available after only 12 clock cycles and offers good accuracy for most practical uses. This performance measure is far superior to the one obtained by pure software-based calculation not utilizing SSE support. This module can be readily included as part of bigger Arithmetic Units for supporting fast inverse square roots. The designed module can be used in FPGA based graphic processing units [10] [11], which require intensive reciprocal square root calculations for lighting and shading of 3D operations especially video games. FPGA based graphic processors are still in infancy and this module can greatly improve their throughput for rendering graphics by offloading other modules. Vertex shader implementation on FPGA as discussed in [12] can also utilize the module. This module can also find use in microprocessors not making use of fast *rsqrtss* of SSE instruction set, especially some RISC implementations on FPGA architectures.

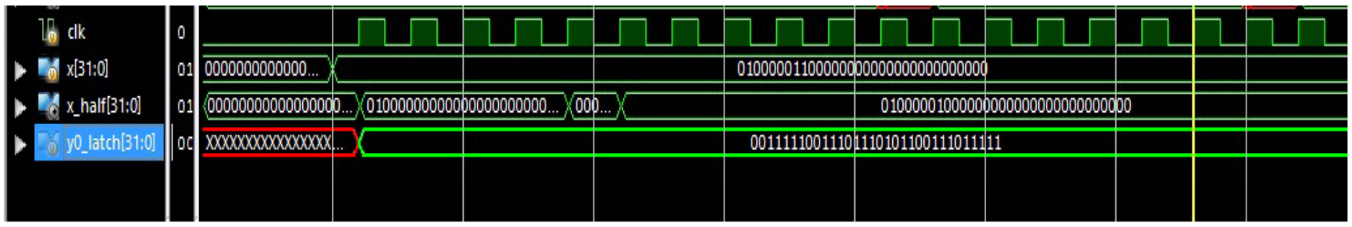

Figure 11. Clock by clock waveform testing of xhalf and *y*0 signals

There is much room for improvement in this work. First of all, the cycle count can be drastically reduced by working in fixed point. If the range of operational values is well within that covered by Int32, then this algorithm can be efficiently implemented by utilizing float to fixed converters. This will save at least 8 cycles during Newton Raphson block, and also aid in conserving FPGA area. But care should be taken to only use this approach if there is absolutely no hazard of overflowing/underflowing.

Attempt is made to write RTL as generic as possible. Meaning, with little modification it should not be too difficult to synthesize this for a different board. The implementation can be sped up by using advanced target devices. By using a more modern board, the Floating Point Unit can better make use of embedded DSP blocks and increase the overall performance of this module.

Also as was suggested in Lomont's report [2] on this algorithm, a better initial guess can be obtained by using a refined magic number instead of 0x5f3759df. His work suggests possible value and by adjusting the values of a few register, a more accurate result is possible.

References

[1] Sommefeldt, R. (2013). Origin of Quake3's Fast InvSqrt(), Beyond3D, 29 November. [Online]. Available: http://www.beyond3d.com/content/articles/8/. [Accessed 15 June 2013].

[2] Lomont, C. (2003). FAST INVERSE SQUARE ROOT, Department of Mathematics, Purdue University, West Lafayette.

[3] Quake3-1.32b/code/game/q_math.c, Id Software, [Online]. Available: ftp://ftp.idsoftware.com/idstuff/source/quake3-1.32b-source.zip. [Accessed 20 June 2013].

[4] Eberly, D. (2002). Fast Inverse Square Root (Revisited), Geometric Tools, LLC.

[5] Elan. (2013). Timing square root, 16 October 2009. [Online]. Available: http://assemblyrequired.crashworks.org/2009/10/16/timing-square-root/. [Accessed 21 June 2013].

[6] Oberman, S. F. Floating Point Division and Square Root Algorithms and Implementation, California Microprocessor Division, Advanced Micro Devices, Sunnyvale.

[7] Sajid, I., Ahmed, M. M., Ziavras, S. G. (2010). Pipelined implementation of fixed point square root in FPGA using modified non-restoring algorithm, *In*: The 2nd International Conference on Computer and Automation Engineering (ICCAE), Singapore.

[8] Suresh, S., Beldianu, S. F., Ziavras, S. G. (2013). FPGA and ASIC square root designs for high performance and power efficiency, in *IEEE* 24th International Conference on Application-Specific Systems Architectures and Processors (ASAP), Washington DC.

[9] Xilinx CORE Generator System, Xilinx Inc., [Online]. Available: http://www.xilinx.com/tools/coregen.htm. [Accessed 25 June 2013].

[10] Kasik, V. (2008). FPGA-Powered Embedded Vector Graphics, in The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologie UBICOMM, Valencia.

[11] Kim, K., Cho, H.-L. S., Park, S. (2008). Implementation of 3D graphics accelerator using full pipeline scheme on FPGA, in *International SoC Design Conference ISOCC*, Busan.

[12] Middendorf, L., Mühlbauer, F., Umlauf, U., Bobda, C. (2007). Embedded Vertex Shader in FPGA, *The International Federation for Information Processing*, 231, p. 155-164.