Reconfigurable HDL Library Development Platform for Arithmetic and Matrix Operations

Semih Aslan
Ingram School of Engineering
Texas State University
S. Marcos, Texas, 78666, USA

ABSTRACT
Embedded systems used in real-time applications require design tools that could be costly and may have long verification cycles. Many design tools use predefined libraries and costly IPs during these design and verification cycles, and most of these libraries and IPs are static and difficult to modify. Many design requirements are changed during or after design and verification cycle, and designers need to address these changes and modify the system. This could be more time consuming due to verification cycle and static libraries. It is important to have dynamic libraries that could be modified and reconfigured based on the applications. This work creates reconfigurable arithmetic design blocks that could be used for arithmetic and matrix operations. The reconfigured library development system modifies the required library elements using Perl scripting language and verifies them on-the-fly using MATLAB. The development tool improves design time and reduces the verification process, but the key point is to use a unified design that combines some of the basic operations with more complex operations to reduce area and power consumption. The results indicate that using the reconfigurable development tool reduces verification time and increases the productivity. These libraries include structural Verilog HDL codes, testbench files, and MATLAB script files for local customization. Even though the reconfigurable HDL library is used for FPGA design flow, it could be easily modified for VLSI design flow.

General Terms
FPGA, Verilog HDL, Perl, MATLAB

Keywords
Hardware optimized, HDL, High Level Synthesis, MATLAB, Optimized Hardware, Perl, Power Efficient, Reconfigurable, RTL, Verilog HDL.

1. INTRODUCTION
Designing complex systems such as image and video processing, compression, face recognition, object tracking, multi-standard CODECs, and HD decoding schemes requires many basic and complex arithmetic blocks and a long verification process [1]. These complex designs are based on many Input/Output, processors, bus interfaces, memories and sensors. Many times, these systems can be designed as a single chip that is known as System-On-Chip (SoC) [2]. Many designers use RTL design flow when SoC are designed and verified. These design tools use design libraries that are mostly static and difficult to configure and verify. When a new or modified library element is needed, designers try to create new library elements from scratch or order new libraries from the vendor. These could be time consuming and costly for the designers. Classic design flow that uses static library and RTL design flow [1] [2] for both FPGA and ASIC is shown in Figure 1.

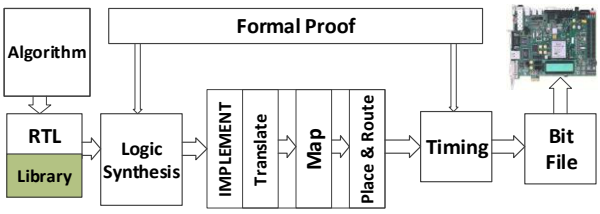

Fig 1: FPGA RTL level synthesis flow

An algorithm can be converted to RTL using the behavioral description model, predefined libraries and IP cores. After completing this RTL code, formal verification must be done before implementation. After implementation of the RTL code, timing verification needs to be done for proper operation. When a required change or future modification is required by the costumer, the design needs to go through same synthesis flow. This drastically increases time-to-market (TTM) if new libraries need to be used, because it will require every part of the design to be verified. This will cause longer verification period of the design and it will increase design cost. The design and verification of the libraries and overall design shown in Figure 1 can take up 40-50% of the "Time to Market" (TTM). The RTL design that is shown in Figure 1 becomes costly and impractical for larger system change and updates. For example, a base system design team that is working on 3G wants to move to 4G design using FPGA can have shorter TTM if the team uses reconfigurable pre-verified libraries [3].

One method to overcome this problem is to introduce high level languages to the design cycle. Because of the extensive work done in Electronics System Level Design (ESLD), HW/SW co-design of a system and High Level Synthesis (HLS) [3][4] are integrated into FPGA and ASIC design flow. RTL description of a system can be implemented from a behavioral description of the system in Perl, C, Python and MATLAB [5]. This will result in a faster verification process and shorter TTM. This HLS idea focuses on design as whole. Some designers want to control the design and make certain changes in HDL code, but this could have a low possibility where HLS is used. The proposed design uses the scripting language Perl to create and MATLAB to verify each required library file. This integration is shown in Figure 2, and it introduces MATLAB and Perl into synthesis flow and verifies all the library design files. Each library element can be created and tested independently without interrupting the design and verification flow. This enables software designers to join the design process during hardware design and verification. After the verification process, the design can be implemented using FPGA synthesis tools.

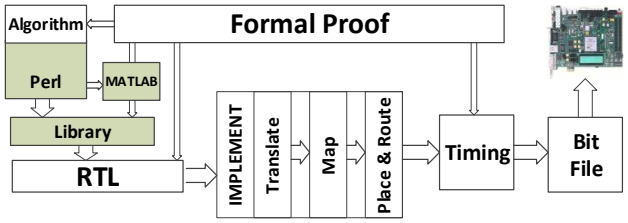

Fig. 2: Proposed FPGA high level synthesis flow

The next section will describe the proposed design and library generation process. Section 3 will focus on the description of each block, and the conclusion will describe future work and improvements.

2. PROPOSED DESIGN

The proposed design focuses on designing a reconfigurable library that could be used for a new or updated design. The TTM will be shorter with a design principle similar to HLS, as explained above. The main work focuses on designing library blocks that could be used and modified based on customer need. The library design will be done using Perl command line interface. The current library development platform supports;

- OS platforms such as Windows, Linux and Mac OS X
- Customized range and accuracy
- FPGA or ASIC support
- Vendor based IP core integration
- Area and power optimized
- User defined module and file names
- Verilog HDL support
- n-bit Fixed point number system (1<n<129)
- Signed or unsigned number systems
- Testbench generation
    - Automated testbench with MATLAB
    - Modelsim .do file for fast automation
    - Error comparison with MATLAB
    - User defined test data option

The development platform creates the library blocks for desired operations such as addition, multiplication, etc. based on user constraints. The proposed design is explained and synthesized and implemented with Xilinx FPGAs [6] for operational verification. Library generation and verification flow is shown in Figure 3.

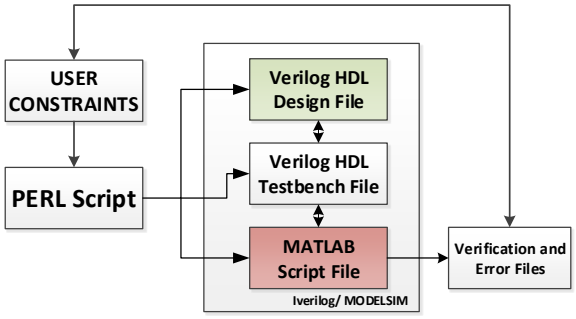

Fig. 3: Library generation and verification flow

The proposed development platform is written in Perl scripting language. The platform accepts user constraints such as file name, number of input bits, test vector number, etc. It generates the Verilog HDL, testbench, MATLAB and batch files, and executes the batch file. The batch file opens MATLAB to generate testvectors and Modelsim or Iverilog for functional verification. After running Modelsim or Iverilog, the output files are generated for verification. These files are transferred into MATLAB to compare with the expected results, and MATLAB displays the error plots. This method can design and verify the design much shorter time, and allows the user to perform a full verification (verification time can increase based on platform and computer specs) or specify his/her testvectors for custom verification.

3. BUILDING BLOCKS

Basic arithmetic and logic operations such as addition, subtraction, multiplication, memory elements, registers, multiplexers, demultiplexers, 1's and 2's complements systems are building blocks of most of the systems in DSP and communication systems. By using these basic building blocks, more complex and useful blocks can be designed. Some of these arithmetic building blocks are division, square root, inverse square root and CORDIC (used to design trigonometric, hyperbolic and exponential functions) and some matrix operations such as addition, and subtraction and multiplication. These library blocks that are generated by the reconfigurable development tool are shown in Figure 4 below.

In this section, these library blocks and their construction by the development tool is discussed in detail. Due to the simplicity and extensive research done on memory elements, multiplexers, demultiplexers, shifters, complement systems, adders and multipliers are not discussed in detail here.

Hardware implementation of complex arithmetic operations and elementary functions is more difficult than hardware implementation of basic arithmetic operations. There are a few algorithms used on today's computers to implement these functions. This chapter includes a brief description of these algorithms.The design platform uses some of these algorithms to implement these elementary functions, as well as some of the basic arithmetic operations.

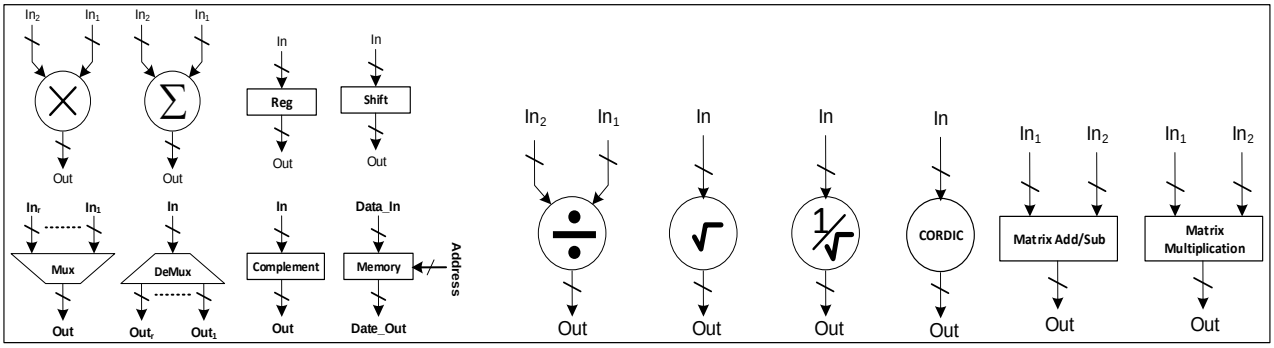

Fig. 4: The reconfigurable library blocks

These algorithms can perform differently based on the required sub functions. As a further example, division using a CORDIC algorithm may require less hardware [7] but it may be slow due to linear convergence; on the other hand, division by the Newton-Raphson [8] method may be faster but it may require more hardware.

3.1 Addition/Subtraction

There are two adder systems designed by the development system. Depending on speed or area, the development tool chooses Carry-Lookahead Adder (CLA) or Ripple-Carry Adder, respectively. Based on the selection, the subtraction system is designed with the adder.

3.2 Multiplication

The development tool can design sign and unsigned type multipliers based on Array and Booth multipliers. Based on the design platform, users can choose their multipliers from an IP core provided by the manufacturers. This may increase the overall performance and suggested method where IPs are available. The Perl code that generates any bit signed multiplier [9][10] Verilog is given in Listing 1 below.

Listing 1. Multiplier Verilog file generation.

```
#!/usr/bin/perl
# Signed Multiplier
use warnings;
use strict;
my $multiplier;
print "Please Enter Module Name for Signed Multiplier :";
chomp ($multiplier = <STDIN>);
my $bit;
print "Please Enter number of bits for Multiplier(x) :";
chomp ($bit = <STDIN>);
my $bit1;
print "Please Enter number of bits for Multiplicand (y) :";
chomp ($bit1 = <STDIN>);
my $i;
my $target;

while (1) {
  my $target = $multiplier;

  chomp $target;
  if (-d $target) {
    print "$target is a directory. This might create problem\n";
    next;
  }
  if (-e $target.".v") {
    print "$target.v already exists. \n";
    print "Enter 'r' to write to a different name : ";
    print "\nEnter 'o' to overwrite \n";
    print "Enter 'b' to back up to $target.old\n";
    my $choice = <STDIN>;
    chomp $choice;
    if ($choice eq "r") {
      next;
    } elsif ($choice eq "o") {
      unless (-o $target.".v") {
        print "Can't overwrite $target.v, it's not yours.\n";
        next;
      }
      unless (-w $target.".v") {
        print "Can't overwrite $target: $!\n";
        next;
      }
    } elsif ($choice eq "b") {
      if ( rename($target.".v",$target.".old") ) {
        print "OK, moved $target.v to $target.old\n";
      } else {
        print "Couldn't rename file: $!\n";
        next;
      }
    } else {
      print "I didn't understand that answer.\n";
      next;
    }
  }
  last if open OUTPUT, "> $target.v";
  print "I couldn't write on $target: $!\n";
  # cannot write.
}
print OUTPUT "/*\n";
print OUTPUT "Name:Signed Multiplier\n";
print OUTPUT "Designer:Semih Aslan\n";
print OUTPUT "*/\n";
print OUTPUT "module $multiplier (x, y, product);\n";
print OUTPUT "parameter M =  ",$bit-1,"; \n";
print OUTPUT "parameter N =  ",$bit1-1,"; \n";
print OUTPUT "input [M-1:0] x;\n";
print OUTPUT "input [N-1:0] y;\n";
print OUTPUT "output [M+N-1:0] product;\n";
print OUTPUT "wire sum [M-1:0][N-1:0];\n";
print OUTPUT "wire carry [M-1:0][N-1:0];\n";
print OUTPUT "genvar i, j;\n";
print OUTPUT "generate for (i=0; i<N; i = i +1) begin:\n";
print OUTPUT "  signed_multiplier\n";
print OUTPUT "  if (i==0)\n";
print OUTPUT "   for (j=0;j<M;j=j+1) begin: first_row\n";
print OUTPUT "    if (j==M-1)\n";
print OUTPUT "   assign sum[j][i] =!(x[i]&y[N-1]),\n";
print OUTPUT "   carry[j][i]=1;\n";
print OUTPUT "  else\n";
print OUTPUT "    assign sum[j][i]=x[j]&y[i],\n";
print OUTPUT "    carry[j][i] = 0; end\n";
print OUTPUT "  else if (i==N-1)\n";
print OUTPUT "    for (j=0;j<M;j=j+1) begin: last_row\n";
print OUTPUT "    if (j==M-1)\n";
print OUTPUT "    assign {carry[j][i],sum[j][i]} =(x[M-1]&y[N-
1])+carry[j][i-1]+carry[j-1][i];\n";
 print OUTPUT " else if (j==0)\n";
print OUTPUT "    assign {carry[j][i],sum[j][i]} =!(x[M-
1]&y[j])+sum[j+1][i-1];\n";
print OUTPUT "else \n";
print OUTPUT "    assign {carry[j][i],sum[j][i]} =!(x[M-
1]&y[j])+sum[j+1][i-1]+carry[j-1][i];end\n";
print OUTPUT "else \n";
print OUTPUT "    for (j=0;j<M;j=j+1) begin : rest_rows\n";
print OUTPUT "    if (j==0)\n";
print OUTPUT "    assign {carry[j][i],sum[j][i]} =(x[j]&y[i])+sum[j+1][i-
1];\n";
```

```
print OUTPUT " else if (j==M-1)\n";
print OUTPUT "    assign {carry[j][i],sum[j][i]} =!(x[i]&y[N-
1])+carry[M-1][i-1]+carry[j-1][i];\n";
print OUTPUT "else \n";
print OUTPUT "    assign {carry[j][i],sum[j][i]} =(x[j]&y[i])+sum[j+1][i-
1]+carry[j-1][i];end\n";
print OUTPUT "end endgenerate\n";

print OUTPUT "generate for (i=0;i<N;i=i+1)\n";
print OUTPUT "  begin: product_lower_part\n";
print OUTPUT "    assign product[i] = sum[0][i];\n";
print OUTPUT "  end endgenerate\n";

print OUTPUT "  generate for (i=1;i<M;i=i+1)\n";
print OUTPUT "  begin: product_upper_part\n";
print OUTPUT "    assign product[N-1+i] = sum[i][N-1]; \n";
print OUTPUT "  end endgenerate\n";
print OUTPUT "  assign product[M+N-1] = carry[M-1][N-1] + 1'b1; \n";
print OUTPUT " endmodule \n";
```

3.3 Division/Square Root/Inverse Square Root

The development tool can design division, square root, and inverse square root hardware and their testbenches individually or all together using the hardware reuse principle. These building blocks are designed using Newton-Raphson [9][10][11][12] and CORDIC algorithms [13][14][15]. The CORDIC algorithm is used in many different arithmetic and elementary functions except for division, square root and inverse square root. The division operation can be written as:

$$N = D \cdot Q + R \quad (1)$$

where N is dividend, D is divisor, Q is quotient, R is remainder and | R | < | D |.ulp and sign (R) = sign(N). The unit in the last position (ulp) represents the lowest term where ulp = 1 for integer numbers and ulp = (radix)-n for n-bit fractional numbers [16].

3.3.1 Newton-Raphson Division

The Newton-Raphson method shown in Equation (1) is a well-known technique to find the root of nonlinear functions. This root can be calculated using an initial value by approaching the root quadratically [8][9][17][18]. The accuracy of the division operation doubles in each iteration. The initial value estimation can reduce the iteration number and increase accuracy.

$$X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)} \quad (2)$$

The function $f(X) = D - \frac{1}{X}$ can be defined to calculate $X_{i+1} = \frac{1}{D}$ where D is the divisor. After applying $f(X)$ and $f'(X)$ to Equation (2), $X_{i+1} = \frac{1}{D}$ can be calculated as

$$X_{i+1} = X_i(2 - DX_i) \quad (3)$$

After n[th] iteration, the value of X_{i+1} converges to 1/D and the quotient can be calculated as Q = N (1/D). Design accuracy and error can be improved by increasing the number of iterations and better estimate of initial value [19]. The hardware implementation of Newton-Raphson division and its iteration steps are described in Figure 5 and Table 1, respectively.

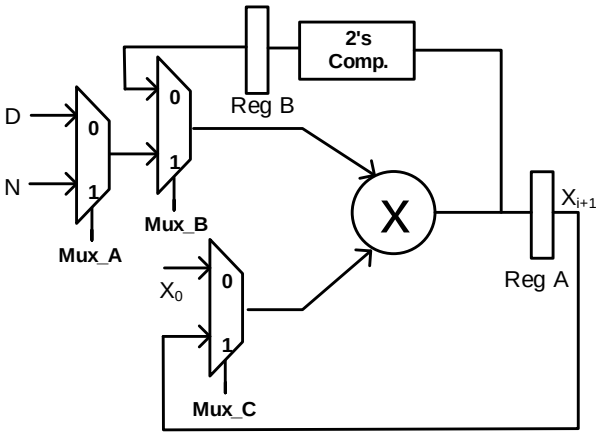

Fig. 5: Newton-Raphson division method

Table 1 below shows the operation of the division block. In cycle 1, pre-determined values of initial approximation X_0 and D are fetched into the multiplier. The product term is calculated and its twos complement is stored in Reg B. In cycle 2, this complemented value is multiplied by X_0 and the result is stored in Reg A as X_1. After three iterations or six clock cycles, the value of X_3 converges to 1/D. After calculation of 1/D, the quotient can be calculated as shown in cycle 7 by multiplying X_3 by N.

This algorithm and its performance depend on the number of iterations and initial approximation. Calculation of 0.85/1.25 when X_0=0.5 and X_0=0.75 for different iterations are shown in Figures 6 and 7, respectively.

Table 1. Newton-Raphson division cycles.

Operation Cycle	Mux A	Mux B	Mux C	Reg A	Reg B
1	0	1	0	---	2-D. X_0
2	0	0	0	X_1	---
3	0	1	1	---	2-D. X_1
4	0	0	1	X_2	---
5	0	1	1	---	2-D. X_2
6	0	0	1	X_3	---
7	1	1	1	N.X_3	---

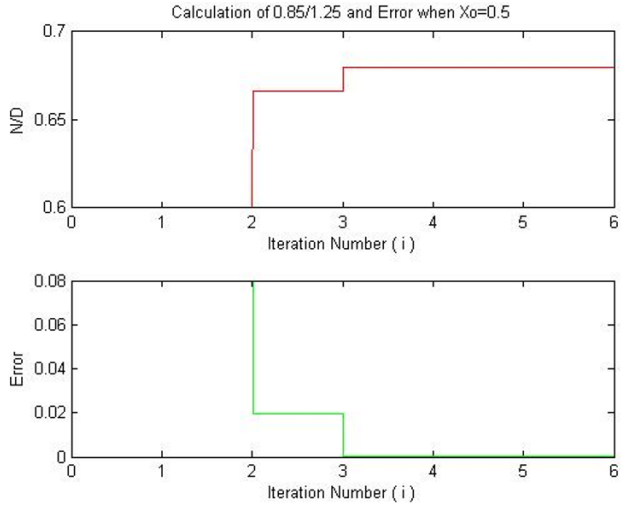

Fig. 6: Calculation of 0.85/1.25 when X_0=0.5

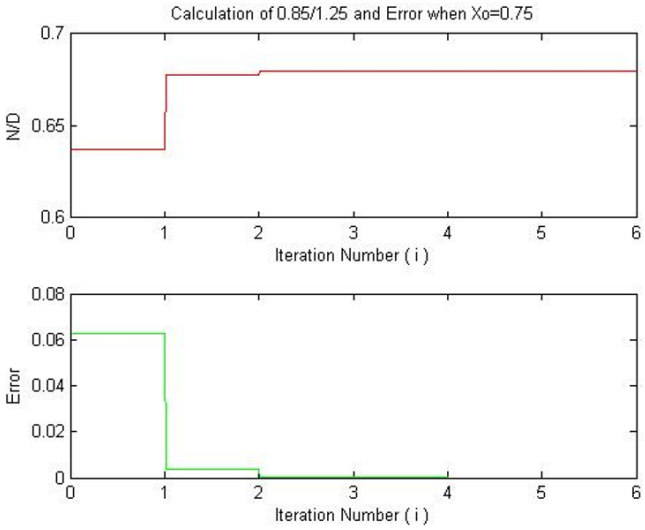

Fig. 7: Calculation of 0.85/1.25 when X_0=0.75

The development system determines the number of iterations and calculates the optimal initial value based on that iteration as shown in Figure 8 below.

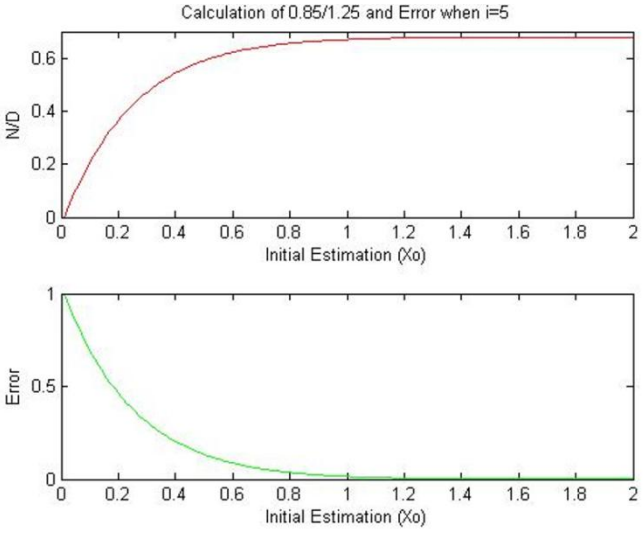

Fig. 8: Calculation of 0.85/1.25 for different initial values.

3.3.2 Newton-Raphson Square Root and Inverse Square Root

Hardware implementation of square root and inverse square root are possible using Newton-Raphson algorithms that are similar to division operations. This makes it possible to have unified division, square root and inverse square root operations. Inverse square root operations can be generated using the Newton-Raphson algorithm. The function $f(X) = X^2 - \frac{1}{D}$ can be defined to calculate $X_{i+1} = \frac{1}{\sqrt{D}}$. After applying $f(X)$ and $f'(X)$ to Equation (2);

$$X_{i+1} = 2^{-1}X_i(3 - DX_i^2) \quad (4)$$

After nth iterations, the value of X_{i+1} will converge at the square root of D. Calculations of the square root can be done by multiplying the final value of Equation (4) by D. The hardware implementation of Newton-Raphson square root and inverse square root methods and their iteration steps are described in Figure 9 and Table2, respectively.

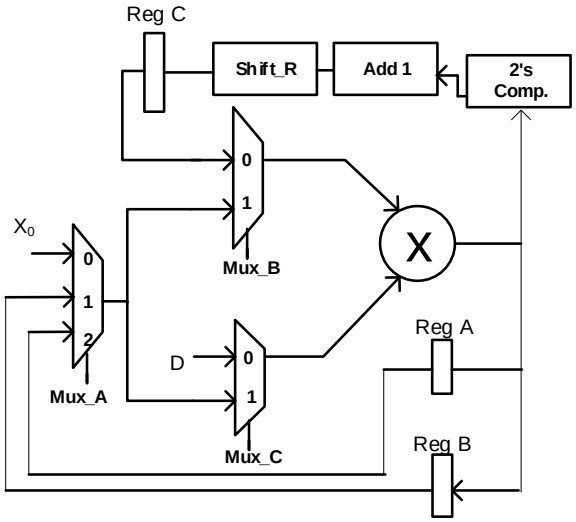

Fig. 9: Newton-Raphson inverse-sqrt and sqrt methods.

Table 2. Newton Raphson inverse sqrt and sqrt cycles.

OC	Reg A	Reg B	Reg C
1	X_0^2	---	---
2	$D.X_0^2$	---	$2^{-1}.(3 - D.X_0^2)$
3	$2^{-1}X_0.(3 - D.X_0^2)$	$2^{-1}X_0.(3 - D.X_0^2)$	---
4	X_1^2	X_1	---
5	$D.X_1^2$	---	$2^{-1}.(3 - D.X_1^2)$
6	$2^{-1}X_1.(3 - D.X_1^2)$	$2^{-1}X_1.(3 - D.X_1^2)$	---
7	X_2^2	X_2	---
8	$D.X_2^2$	---	$2^{-1}.(3 - D.X_2^2)$
9	$2^{-1}X_2.(3 - D.X_2^2)$	$2^{-1}X_2.(3- D.X_1^2)$	---
10	$D.X_3$	---	---

Table 2 above shows the operation of the inverse square root and square root blocks. In cycle 1, X_0^2 is calculated. This value is stored in Reg A. In cycle 2, X_0^2 is multiplied by D and the result $D X_0^2$ is stored in Reg A and $2^{-1}(3 - D X_0^2)$ is stored in Reg C. In cycle 3, $2^{-1}(3 - D X_0^2)$ is multiplied by X_0 and result $2^{-1} X_0(3 - D X_0^2)$ is stored in Reg A and Reg B. This is the end of the first iteration. Registers A and B are holding the value of X_1 that will be used in the second iteration. Iterations 2 and 3 are done in cycles 4 through 9, and

the final result X_3 will converge into $\frac{1}{\sqrt{D}}$. In cycle 10, $\sqrt{D}$ is calculated by multiplying $\frac{1}{\sqrt{D}}D$.

3.4 Sine and Cosine Implementations

Sine and cosine functions can be implemented using the CORDIC algorithm. The CORDIC (COordinate Rotation DIgital Computer) algorithm was first described in 1959 by Jack E. Volder [12][13] to replace the analog resolver in the B-58 bomber's navigation system. Since then, quite a bit of research has been done on the topic. This algorithm is used today in digital filters, FFT, DFT, Kalman filters, adaptive lattice structure, linear algebra applications, singular value decomposition (SVD) calculations, Given's rotation and QRD-RLS filtering [20].

The CORDIC algorithm is based on two modes; vectoring and rotation. This algorithm and its derivation over linear, circular and hyperbolic coordinate systems can compute many elementary functions described above.

Let us assume that point A rotates to B by an angle of rotation angle θ, as shown in Figure 10. This will create a new point B(X',Y'), and the relation of the new point B based on previous point A and the rotation angle θ [7][14][15][20].

$$x = R.\cos(\beta) \tag{5}$$

$$y = R.\sin(\beta) \tag{6}$$

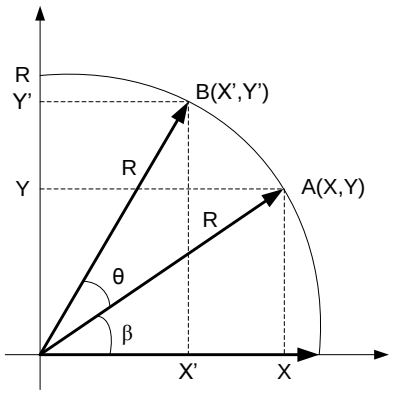

Fig. 10: Rotation of vector A(x,y) to B(x',y')

$$x' = R.\cos(\theta + \beta) \tag{7}$$

$$y' = R.\sin(\theta + \beta) \tag{8}$$

Using the trigonometric properties;

$$sin(\theta + \beta) = sin(\theta)\, cos(\beta) + cos(\theta)\, sin(\beta) \tag{9}$$

$$cos(\theta + \beta) = cos(\theta)\, cos(\beta) - sin(\theta)\, sin(\beta) \tag{10}$$

Putting (9) and (10) in (7) and (8)

$$x' = R.cos(\theta)\, cos(\beta) - R.sin(\theta)\, sin(\beta) \tag{11}$$

$$y' = R.sin(\theta)\, cos(\beta) + R.cos(\theta)\, sin(\beta) \tag{12}$$

Putting (5) and (6) in (11) and (12)

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = cos(\theta) \begin{bmatrix} 1 & -tan(\theta) \\ tan(\theta) & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{13}$$

$$x' = cos(\theta)\,[x - y.tan(\theta)] \tag{14.a}$$

$$y' = cos(\theta)\,[x.tan(\theta) + y] \tag{14.b}$$

In this iteration, the value of $tan(\theta)$ can be chosen in terms of power of 2. This will make it easy to implement as hardware because $\mp 2^{-i}$ is simply an L-R shift. This transformation can be done by a sequence of smaller angle rotations (θ_i) in which,

$$\theta = \sum_{i}^{n} \theta_i \tag{15}$$

And by using this Equation [13] can be written as;

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} cos(\theta_i) & -sin(\theta_i) \\ sin(\theta_i) & cos(\theta_i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{16}$$

$$tan(\theta_i) = \mp 2^{-i} \tag{17}$$

$$\theta_i = \mp\, tan^{-1}(2^{-i}) \tag{18}$$

In addition, the value of the product of the $cos(\theta)$ will be constant. This value is known as K_i (scaling factor).

$$K_i = \prod_{i=0}^{n} cos(\theta_i) \tag{19}$$

$$K_i = \prod_{i=0}^{n} cos(\mp\, tan^{-1}(2^{-i})) \tag{20}$$

Using the trigonometric property;

$$cos(tan^{-1}(\alpha)) = \frac{1}{\sqrt{1+\alpha^2}} \tag{21}$$

Equation (20) scaling factor would be

$$K_i = \prod_{i=0}^{n} \frac{1}{\sqrt{1+2^{-2i}}} \tag{22}$$

This scaling factor value will be constant for large n (iteration number). Using the Equations (18) and (22) in Equation (16) the following may occur.

$$x_{i+1} = K_i\,.\,[x_i - y_i(\mp 2^{-i})] \tag{23}$$

$$y_{i+1} = K_i\,.\,[y_i + x_i(\mp 2^{-i})] \tag{24}$$

and

$$x_{i+1} = K_i\,.\,[x_i - d_i.y_i(2^{-i})] \tag{25}$$

$$y_{i+1} = K_i\,.\,[x_i + d_i.x_i(2^{-i})] \tag{26}$$

The rotation of the angle must be updated. This will introduce a new set of equations:

$$z_{i+1} = z_i - d_i.tan^{-1}(2^{-i}) \tag{27}$$

Values for $tan^{-1}(2^{-i})$ can be stored in the memory. Variable d_i is the direction of the rotation and its value depends on z_i. This may be represented as shown below:

$$d_i \begin{cases} 1 & z_i \geq 0 \\ -1 & z_i < 0 \end{cases} \tag{28}$$

After the description of the vectoring mode operation, a general description of CORDIC algorithm for linear, circular and hyperbolic coordinates using vectoring and rotation modes can be given as shown below [14] [20].

$$m = \begin{cases} 1 & circular\ coordinates \\ 0 & linear\ coordinates \\ -1 & hyperbolic\ coordinates \end{cases} \tag{29}$$

$$x_{i+1} = x_i - m \propto_i y_i 2^{-i} \tag{30}$$

$$y_{i+1} = y_i + \propto_i x_i 2^{-i} \tag{31}$$

$$z_{i+1} = \begin{cases} z_i - \propto_i tan^{-1}(2^{-i}) & if\ m = 1 \\ z_i - \propto_i tanh^{-1}(2^{-i}) & if\ m = 0 \\ z_i - \propto_i 2^{-i} & if\ m = -1 \end{cases} \tag{32}$$

Table 3. CORDIC shift sequences and scaling factor

Coor. System M	Shift Sequence $S_{m,i}$	Convergence α_{max}	Scale Factor K_m $(n\rightarrow\infty)$
1	0,1,2,……,i,….	1.74	1.16676
0	1,2,……,i+1,….	1.0	1.0
-1	1,2,3,4,…….	1.13	0.83816

Table 4. CORDIC processor for three coordinate systems

Coor Sys	Rot. / Vec.	Initializing	Result Vectors
1	Rot.	$X_0 = X_S$ $Y_0 = Y_S$ $Z_0 = \beta$ $X_0=1/K_{1,n}$ $Y_0 = 0$ $Z_0 = \beta$	$X_n = K_{1,n}(X_S \cos(\beta) - Y_S \sin(\beta))$ $Y_n = K_{1,n}(Y_S \cos(\beta) + X_S \sin(\beta))$ $Z_n = \beta$ $X_n = \cos(\beta)$ $Y_n = \sin(\beta)$ $Z_n = 0$
1	Vec.	$X_0 = X_S$ $Y_0 = Y_S$ $Z_0 = \beta$	$X_n = K_{1,n}(\text{sgn}(X_0)(\text{sqrt}(x^2+y^2)$ $Y_n = 0$ $Z_n = \beta+\tan^{-1}(Y_S/X_S)$
0	Rot.	$X_0 = X_S$ $Y_0 = Y_S$ $Z_0 = Z_S$	$X_n = X_S$ $Y_n = Y_S+ X_S Y_S$ $Z_n = 0$
0	Vec.	$X_0 = X_S$ $Y_0 = Y_S$ $Z_0 = Z_S$	$X_n = X_S$ $Y_n = 0$ $Z_n = Z_S+ Y_S / X_S$
-1	Rot.	$X_0 = X_S$ $Y_0 = Y_S$ $Z_0 = \beta$ $X_0=1/K_{-1,n}$ $Y_0 = 0$ $Z_0 = \beta$	$X_n = K_{-1,n}(X_S \cosh(\beta) + Y_S \sinh(\beta))$ $Y_n = K_{-1,n}(Y_S \cosh(\beta) + X_S \sinh(\beta))$ $Z_n = 0$ $X_n = \cosh(\beta)$ $Y_n = \sinh(\beta)$ $Z_n = 0$
-1	Vec.	$X_0 = X_S$ $Y_0 = Y_S$ $Z_0 = \beta$	$X_n = K_{1,n}(\text{sgn}(X_0)(\text{sqrt}(x^2+y^2)$ $Y_n = 0$ $Z_n = \beta+\tan^{-1}(Y_S/X_S)$

Using Table 3 and Table 4, sin(θ) and cos(θ) values can be calculated. The circular coordinate rotation mode CORDIC can be used [14][20].

$$x_{i+1} = K.(x_i - \propto_i y_i 2^{-i}) \quad (33)$$

$$y_{i+1} = K.(y_i + \propto_i x_i 2^{-i}) \quad (34)$$

$$z_{i+1} = K.\left(z_i - \propto_i tan^{-1}(2^{-i})\right) \quad (35)$$

To start the iteration, the following initial values need to be assigned.

$$K = 1.16676 \quad (36)$$

$$x_0 = \frac{1}{K} = \frac{1}{1.16676} \quad (37)$$

$$y_0 = 0, \quad (38)$$

$$z_0 = \theta \quad (39)$$

Using these Equations MATLAB code to for design and verification are shown in Figures 11, 12 and 13, respectively.

```
function [s,c,error_sin,error_cos]=cordic(tetha)
    X = 1;
    Y = 0;
    Z = tetha;
    i = 0;
    Kc = 1.646760255;
    X = (1/Kc)*X;
while i < 30
     if Z >=0
        n=1;
    else
        n=-1;
     end

    Xi=X;
    X=Xi-(Y*n*(2^(-i)));
    Y=Y+(Xi*n*(2^(-i)));
    Z=Z-(n*(atan(2^(-i))));
    i=i+1;
end
```

Fig. 11: MATLAB sine(θ) and cos(θ) calculations

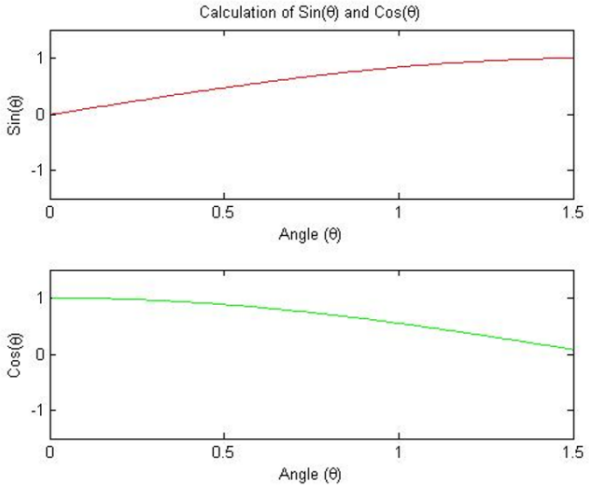

Fig. 12: Sin(θ) and cos(θ) values between 0 < θ < 1.5

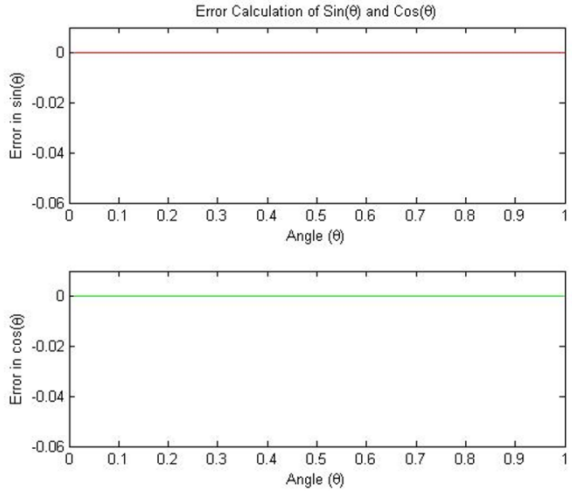

Fig. 13: Sin(θ) and cos(θ) values between 0 < θ < 1.5

3.5 Sinh, Cosh and $e^{(x)}$ Implementations

Sinh and Cosh functions can be implemented using the CORDIC provided above in Table 3 and 4. To calculate sinh(θ) and cosh(θ) values, the hyperbolic coordinate rotation mode CORDIC can be used [14][20].

$$x_{i+1} = K.(x_i + \propto_i y_i 2^{-i}) \quad (40)$$

$$y_{i+1} = K.(y_i + \propto_i x_i 2^{-i}) \quad (41)$$

$$z_{i+1} = K.\left(z_i - \propto_i tanh^{-1}(2^{-i})\right) \quad (42)$$

To start the iteration, the following initial values need to be assigned.

$$K = 0.83816 \quad (43)$$

$$x_0 = \frac{1}{K} = \frac{1}{0.83816} \quad (44)$$

After n^{th} iteration, and using the Equations (40), (41) and (42).

$$x_n = sinh(\theta) \quad (45)$$

$$y_0 = cosh(\theta) \quad (46)$$

$$z_0 = 0 \quad (47)$$

An exponential function can be calculated using (45) and (46).

$$e^x = cosh(\theta) + sinh(\theta) \quad (48)$$

MATLAB representation of sinh, cosh, and exponential functions and error analysis are shown in Figures 14, 15, and 16, respectively.

```
function [sinh_c,cosh_c,exp_c,error_sinh,error_cosh,error_exp]=cordic2(tetha)
    X = 1;
    Y = 0;
    Z = tetha;
    i = 0;
    Kc = 1.205136358446461;
    X = Kc;
    i = 1;
while i < 64
     if Z >=0
        n=1;
    else
        n=-1;
     end

    Xi=X;
    X=Xi+(Y*n*(2^(-i)));
    Y=Y+(Xi*n*(2^(-i)));
    Z=Z-(n*atanh(2^(-i)));
    i=i+1;
end
```

Fig. 14: MATLAB code for sinh(θ), cosh(θ) and exponential function calculations

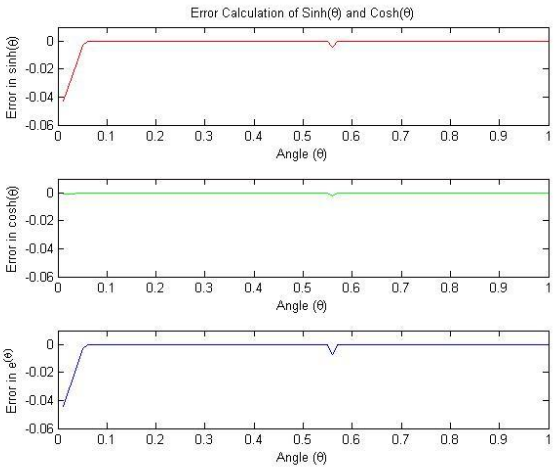

Fig. 15: Error analysis of sin(θ) and cos(θ)

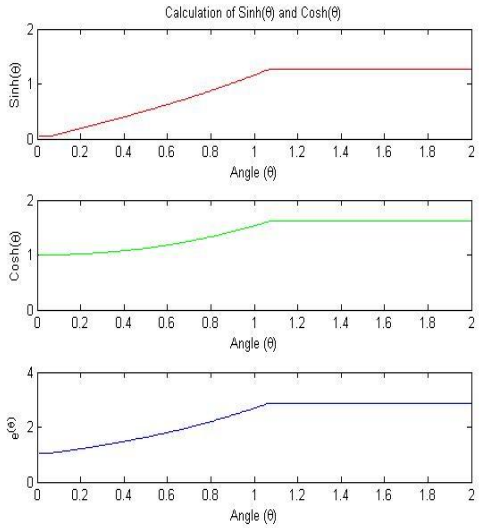

Fig. 16: Sinh(θ), cosh(θ) and e^{θ} values between 0 < θ < 2

3.6 Matrix Operations

The library design platform can create library elements for some basic matrix operations such as addition, subtraction and multiplication. The library design matrix hardware and verification as same as explained above in Figure 3.

3.6.1 Matrix Addition/Subtraction

Hardware implementation of matrix addition is done as shown in Equations (49) and (50), respectively.

$$C = A + B \quad (49)$$

$$c_{i,j} = a_{i,j} + b_{i,j} \qquad where\ i,j = 1,2,\dots,n \quad (50)$$

The conventional design is easy to understand and implement. The design platform uses the equation that is given in Equation (50) to calculate all elements of matrix C. The matrix addition/subtraction block is shown in Figure 17 below.

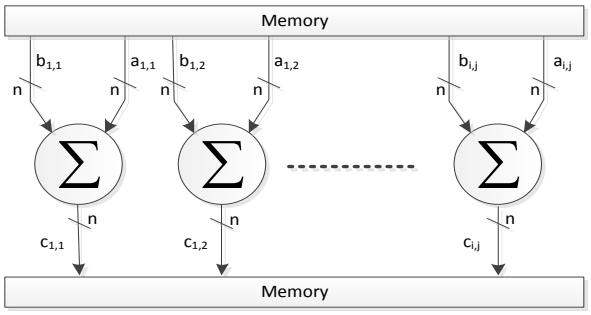

Fig. 17: Matrix addition/ subtraction block

3.6.2 Matrix Multiplication

Matrix multiplications [8][20] are heavily used in many signal and image processing applications, such as adaptive beamforming [16][18] and multiple-input-multiple-output (MIMO) systems [18], and factorizations [21][22][23][24] such as QR factorization and DCT. Matrix multiplication requires operation elements (OE) such as addition and multiplication. In a matrix multiplication, the number of OEs depends on the matrix size. The relation between matrix size and the number of OEs is quadratic. This made it difficult to implement real time matrix multiplication libraries for larger

matrices. The design platform uses traditional matrix multiplication to generate library element. Matrix multiplication of an m×r matrix A and r×n matrix B produces a m×n matrix C [25].

$$A_{m,r} \times B_{r,n} = C_{m,n} \tag{51}$$

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,r} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,r} \end{bmatrix} \begin{bmatrix} b_{1,1} & \cdots & b_{1,n} \\ \vdots & \ddots & \vdots \\ b_{r,1} & \cdots & b_{r,n} \end{bmatrix} = \begin{bmatrix} c_{1,1} & \cdots & c_{1,n} \\ \vdots & \ddots & \vdots \\ c_{m,1} & \cdots & c_{m,n} \end{bmatrix} \tag{52}$$

where;

$$c_{i,j} = \sum_{k=1}^{r} a_{i,k} \times b_{k,j} \tag{53}$$

This matrix multiplication requires m×n×[rM+(r-1)A] arithmetic operations, where M is for multiplier and A is for adder. For example; multiplication of $A_{4,3}$ and $B_{3,6}$ results in $C_{4,6}$. This operation requires 72 multiplication and 48 addition operations. This shows that a $n \times n$ matrix multiplication requires n^3 multiplications and $n^2(n-1)$ additions. The number of multiplications in the matrix multiplication operation increases by an exponent of three (n^3) with the matrix size.

The hardware realization of a matrix multiplication focuses on designing $c_{i,j}$ due to the parallel nature of the system. In general, calculation of $c_{i,j}$ is done using a bottom-up approach. The term $c_{i,j}$ in Equation (53) can be written as;

$$c_{i,j} = a_{i,1} \times b_{1,j} + a_{i,2} \times b_{2,j} + \cdots + a_{i,n} \times b_{n,j} \tag{54}$$

Due to the matrix size choice of $n = 2^k$, the Equation 54 can be written as

$$c_{i,j} = d_1 + d_2 + \cdots + d_t \tag{55}$$

where;

$$d_t = a_{i,2t-1} \times b_{2t-1,j} + a_{i,2t} \times b_{2t,j} \qquad 1 \le t \le \frac{n}{2} \tag{56}$$

The hardware realization of Equation (56) is shown in Figure 18.

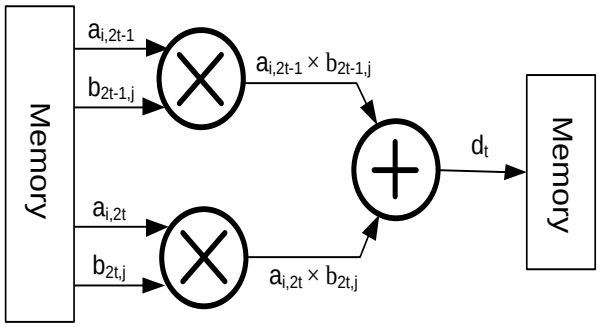

Fig. 18: Hardware realization of d_t

This d_t calculation block shown in Figure 18**Fig** is one of the most basic and important blocks in matrix multiplication. Any matrix multiplication can be done using only one of these blocks, an adder and rounding scheme. However, this is impractical for large matrices due to slow overall operation and increase in error boundaries. There are many different designs [26] that use different numbers of d_t calculation blocks to improve speed. The platform generates the libraries based on the smaller block that is shown in Figure 18 above and using this block larger blocks can be created as shown in Figure 19 below.

Larger matrix multiplications can be realized by using these blocks. A 32x32 matrix multiplication design block is shown in Figure 20 below.

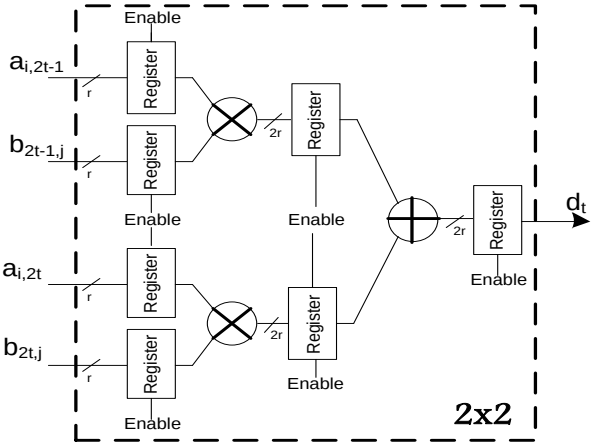

Fig. 19: Realization of 2x2 block

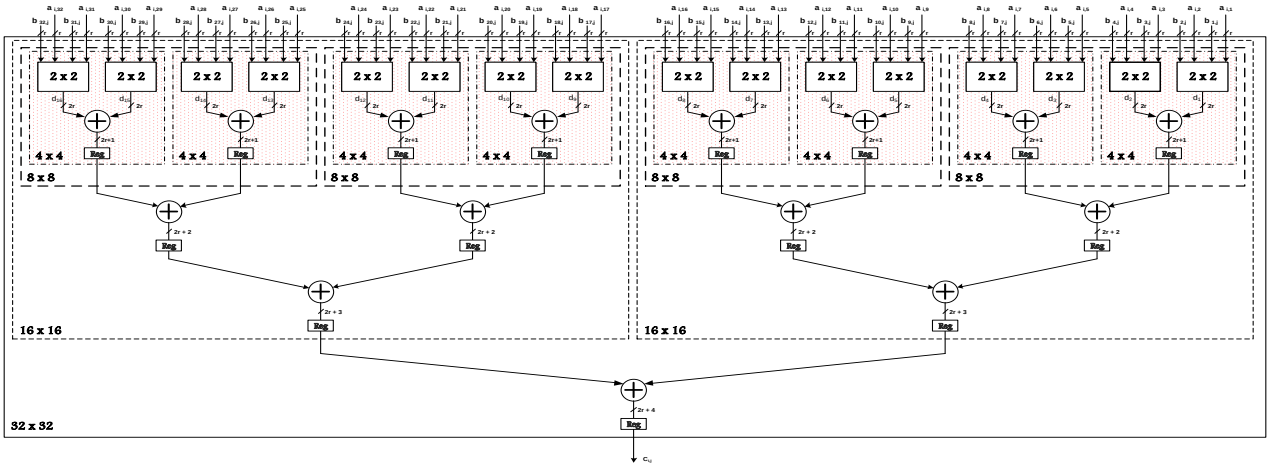

Fig. 20: Realization of 32x32 block

4. CONCLUSION

An area efficient, reconfigurable HDL library development platform for Arithmetic and Matrix Operations is designed for digital design and verification. The platform generates dynamic design libraries and verification files to improve design and verification of overall systems. Based on design complexity and required changes, TTM can be improved by up to 60%. Even though this platform is not true HLS, it could be considered as hybrid HLS. Any designed system can be reconfigured at any time in any way using the dynamic libraries without going through the same design and verification hassle. MATLAB-based verification makes it possible to use all the features of MATLAB for faster and more efficient verification. The Perl-based design makes it possible to use all important text manipulations in Perl language. Even though command line user interface is user friendly, it creates some hassle for users when the library size increases.

The future development platform will integrate a user friendly GUI using Perl/Tk. Another future goal is to make this platform totally open source by using only Iverilog and replacing MATLAB with Octave. In addition, current matrix libraries are limited to addition, subtraction and matrix multiplication. The future development platform will include some important matrix factorizations such as QR and LU factorizations, and Strassen based matrix multiplication for area optimization.

5. ACKNOWLEDGMENTS

Author thanks to Xilinx [6] and MATLAB for their support for this research.

6. REFERENCES

[1] Andrieux, J., M. Feix, G. Mourgues, P. Bertrand, B. Izrar, and V. Nguyen. "Optimum Smoothing of the Wigner Ville Distribution." IEEE Transactions on Acoustics, Speech, and Signal Processing 36.5(1987): 764-769.

[2] Adler, J., K. Delgado, and B. Rao. "Comparison of Basis Selection Methods." Proceeding of IEEE Asilomar Conference on Signals, Systems, and Computer Frequency and Time Scale Analysis (1996): 252-257

[3] Aslan, S., Oruklu, E., and Saniie J., "A high-level synthesis and verification tool for fixed to floating point conversion", IEEE International Midwest Symposium on Circuits and Systems, 2012, Pages, 908-911.

[4] Desmouliers, C., Aslan, S., Oruklu E., Saniie, J., Vallina, F.M., "HW/SW co-design platform for image and video processing applications on Virtex-5 FPGA using PICO" IEEE International Conference on Electro/Information Technology, 2010, Pages, 1-6.

[5] Chen, W. The VLSI Handbook. Boca Raton: CRC Publisher, 2007.

[6] Xilinx. (2016), http://www.xilinx.com/

[7] Kilts, S. Advanced FPGA Design Architecture, Implementation, and Optimizations. New York: Wiley Inter-Science, 2007.

[8] Stine, J. E. "Digital Computer Arithmetic Datapath Design using Verilog HDL", Norwell, Massachusetts, Kluwer Academic Publishing, 2004.

[9] Lin, Ming-Bo, "Digital System Design and Practices Using Verilog HDL and FPGAs", Singapore, Wiley Publishing, 2008.

[10] Joseph Cavanagh, " Computer Arithmetic and Verilog HDL Fundamentals ", Boca Raton, FL, CRC Press, Taylor & Francis Group, 2010.

[11] Flynn, M. J., and S. F. Oberman. "Division Algorithms and Implementations." IEEE Transactions on Computers 46.8 (1997): 833-854.

[12] Schulte, M. J., and L. K. Wang. "Decimal floating-point square root using Newton-Raphson iteration." Application-Specific Systems, Architectures and Processors (2005): 309 -315.

[13] Volder, J. "The CORDIC Trigonometric Computing Technique." IEEE Transactions Electronic Computers 8.3 (1959): 330-334.

[14] Striling, W. C., and T. K. Moon. Mathematical Methods and algorithms for Signal Processing. New Jersey: Prentice Hall, 2000.

[15] Andraka, R. "A survey of CORDIC algorithms for FPGAs." Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays (1998): 191-200.

[16] Lang, T. M., and D. Ercegovac. Digital Arithmetic. San Francisco: Morgan Kaufmann, 2004.

[17] Flynn, M. J., and S. F. Oberman. "Division Algorithms and Implementations." IEEE Transactions on Computers 46.8 (1997): 833-854.

[18] Swartzlander, E.E. Jr., and W.L.Gallagher. "Fault-Tolerant Newton-Raphson and Goldschmidt Dividers Using Time Shared TMR." IEEE Transactions on Computers 49.6 (2000): 588-595.

[19] Omar, J., E. E. Swartzlander Jr., and M. J. Schulte. "Optimal Initial Approximations for the Newton-Raphson Division Algorithm." Springer-Verlag Journal of Computing 53.3-4 (1994): 233-242.

[20] Dehon, A., and S. Hauck. Reconfigurable Computing The Theory and Practice of FPGA-Based Computing. Burlington, Massachusetts: Elsevier, 2008.

[21] Teukolsky, S. A., W. T. Vetterling, B. P. Flannery, and W. H. Press. "Numerical Recipes: The Art of Scientific Computing." Numerical Recipes: The Art of Scientific Computing, 3rd ed. New York, New York: Cambridge University Press, 2007.

[22] Erisman, A. M., I. S. Duff, and J. K. Reid. Direct Methods for Sparse Matrices. New York, United States of America: Oxford University Press, 2003.

[23] Fujii, A, R Suda, and A Nishida. "Parallel Matrix Distribution Library for Sparse Matrix Solvers." Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region (2005): 219-226.

[24] Aslan, S., E. Oruklu, and J. Saniie. "Realization of area efficient QR factorization using unified division, square root, and inverse square root hardware." IEEE International Conference on Electro/Information Technology (2009): 245-250.

[25] Watkins, David S. Fundamentals of Matrix Computations, 2nd ed. New York, United States of America: John Wiley & Sons, 2002.

[26] Hendry, D.C., and A.A. Duncan. "Area Efficient DSP Datapath Synthesis." Design Automation Conference (1995): 130-135.