Chapter 7: Exploration of Hardware Algorithms and Architectures of the Functional Blocks and their FPGA Implementation and Testing

7.1 Introduction

Exploration of hardware algorithms and architectural choices corresponding to operators and functions (that were identified for hardware implementation during application analysis in chapter 4) has been carried out in a bid to achieve execution within the allocated clock cycles at 25 MHz or higher clock rates [20] [21]. During these explorations the constraints on clock cycle counts and clock speed have been the primary concern, followed by gate count. All the functional blocks have been coded at RT-level in VHDL and have been extensively simulated for functional correctness before synthesis.

The next section describes the algorithms and hardware architectures finally selected for different functional blocks (functional units).

7.2 IGEN (Noise Generator)

In the source-filter model of human speech production, two of the three sound sources (namely aspiration source and frication source) are modeled as random noise sources. In our design the IGEN block acts as a random noise generator. As per corresponding Klatt’s ‘C’ code it generates 12 random numbers in the range from 0 to 32767 and then computes their average value to give the noise sample. The algorithm used for generating random numbers [51] (D. Lehman in 1951) is the so-called Linear Congruential method i.e.

\[ S_i = (a \times S_{i-1} + c) \mod m \]  

\[ \text{equation 7.1} \]

Where, \( m = \) modulus,

\( a = \) positive integer called the multiplier and

\( c = \) positive integer called the increment
The Linear Congruential method has the advantage of being very fast and requires only a few operations per call; hence it is used almost universally.

The recurrence shown in equation 7.1 will eventually repeat itself, with a period that is obviously no greater than m. If m, a, and c are properly chosen, then the period will be of maximal length, i.e. of length m.

The random number generator is initialized by providing some arbitrary positive seed value. We have taken it one. Each initializing value will typically result in a different random sequence. Architecture for the noise source computation implementing the above algorithm is shown in figure 7.1. The controller is a dedicated FSM with 14 states. IGEN requires 14 clock cycles to generate one noise sample. RT-level VHDL code of IGEN is provided at Appendix-A.

The same initializing seed value will always return the same random sequence. The sequence will repeat after m-1 and the random numbers lies in between 0 to m-1. To generate the random number sequences, we have chosen the values:

\[ a = 13821 \]
\[ c = 0 \]
\[ m = 2^{15} \]
\[ S_0 = 1 \]

Its repeat length is 8192 i.e. \(2^{13}\) and generates random numbers between 0 to 32767. We know that multiplication in hardware is generally a slow and costly process. It involves the generation of partial products and their accumulation. The recurrence as shown in equation 7.1 requires the multiplication i.e. \(a \times S_{i-1}\) where both \(a\) and \(S\) are 16-bit integers.

A 16 x 16 multiplication generates 16 partial products. The idea here is to somehow reduce the number of partial products taking advantage of the specific value of \(a\) chosen. We have, therefore, broken the constant \(a\) i.e. 13821 as \(2^{14} - (2^{11} + 2^9 + 2^1 + 2^0)\). Therefore, it generates only 5 partial products as compared to 16 partial products generated in a general 16 x16 multiplication. This application specific fact is taken advantage of to reduce the cost and increase the speed of the multiplier used. To further speedup multiplication operation, we have used Wallace tree implementation as shown in
Figure 7.2 for the summation of the 5 partial products generated. The IGEN design also serves to illustrate how application specific choices and optimizations help to realize a lower cost higher performance hardware function block (that eventually goes into a highly optimized ASIP micro-architecture).

Figure 7.1: Architecture for IGEN functional Block
7.2.1 Testing of IGEN

Steps used in the functional verification of the IGEN functional block (functional unit) are:

- Generation of test vectors in the input file.
- Application of test vectors to the functional block/unit.
- Taking out the outputs of the functional block/unit in the output file.
- Comparing these outputs with the reference outputs generated by a ‘C’ program (functional model).

Testing of each individual functional block is important before the final implementation of the ASIP on the FPGA. As the design of each functional block has been done in VHDL, functional verification of each functional block has been done through VHDL simulation. Each functional block designed at RT-level has been tested extensively through the application of test vectors and comparing its outputs with the outputs of a reference functional model, which is a ‘C’ program. Extensive test vectors have been generated using a ‘C’ program and applied to both the RT-level VHDL code and the reference functional model ‘C’ code and results of both are taken out in output.
files. For the same set of operands, the RT-level VHDL code generated results and the reference functional model ‘C’ code generated results are compared as shown in figure 7.3 for any differences to check whether the functional block is working properly or not. Separate test benches are developed for the simulation of each functional block. After a number of simulations on RT-level code of the functional block, gate-level simulations are also carried out after synthesizing the functional block to check whether the generated gate-level netlist is correct or not using the same approach.

Same set of test vectors are applied both at RT-level and gate-level simulations.

---

**Figure 7.3: Approach Used for Testing Individual Functional Blocks**

7.3 FASC (Floating-point Add-Subtract-Compare)

Let a₁ and a₂ be two IEEE-754 single-precision floating-point numbers to be added. The notations eᵢ and sᵢ are used for the exponent and significand (which is 1 plus the fractional part) of the addends aᵢ. When performing arithmetic on IEEE floating-point numbers, the fraction part is usually *unpacked*, which is to say the implicit 1 is made explicit and that sᵢ has an explicit leading bit i.e. '1' (one). To add or subtract a₁ and a₂ [10], these nine steps are performed:

1. If e₁ < e₂, swap the operands. This ensures that the difference of the exponents satisfies d = e₁ - e₂ ≥ 0. Tentatively, set the exponent of the result to e₁.
(2) If the signs of $a_1$ and $a_2$ differ, replace $s_2$ by its two's complement.

(3) Place $s_2$ in a shift register and shift it $d = e_1 - e_2$ places to the right (shifting in 1's if the $s_2$ was complemented in previous step). From the bits shifted out, set $g$ (guard bit) to the most-significant-bit, $r$ (round bit) to the next most-significant-bit, and set sticky bit $s$ (sticky bit) to the OR (Boolean function) of rest of the bits.

(4) Compute a preliminary significand, $S = s_1 + s_2$ by adding $s_1$ to the shift register containing $s_2$. If the signs of $a_1$ and $a_2$ are different, the most-significand-bit of $S$ is '1', and there was no carry out, then $S$ is negative. In this case, replace $S$ with its two's complement. This can only happen when $d = 0$.

(5) Shift $S$ as follows. If the signs of $a_1$ and $a_2$ are the same and there was a carry out in step 4, shift $S$ right by one, filling the high order position with 1 (the carry out). Otherwise shift it left until it is normalized. When left shifting, on the first shift fill-in the low order position with the $g$ bit. After that, shift in zeros. Adjust the exponent of the result accordingly.

(6) Adjust $r$ and $s$. If $S$ was shifted right in step 5, set $r = \text{low order bit of } S$ before shifting, and $s = g \lor r \lor s$. If there was no shift, set $r = g$, $s = r$. If there was a single left shift, do not change $r$ and $s$. If there were two or more left shifts, set $r = 0$, $s = 0$. (In the last case, two or more shifts can only happen when $a_1$ and $a_2$ have opposite signs and the same exponent, in which case the computation $s_1 + s_2$ in step 4 will be exact).

(7) Round $S$ using rounding rules as shown in table 7.1.

<table>
<thead>
<tr>
<th>Rounding Mode</th>
<th>Sign of Result $\geq 0$</th>
<th>Sign of Result $&lt; 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nearest</td>
<td>(+1\text{ if } r \land p_0 \text{ or } r \land s)</td>
<td>(+1\text{ if } r \land p_0 \text{ or } r \land s)</td>
</tr>
</tbody>
</table>

**Table 7.1: Rounding Modes Used in Floating Point Add/Sub**

(8) If a table entry is non-empty, add 1 to the low order bit of $S$. If rounding causes carry out, shift $S$ right and adjust the exponent. This is the significand of the result.
(9) Compute the sign of the result as shown in table 7.2. If $a_1$ and $a_2$ have the same sign, this is the sign of the result. If $a_1$ and $a_2$ have different signs, then the sign of the result depends on which of $a_1$, $a_2$ is negative, whether there was a swap in the step 1 and whether S was replaced by its two’s complement in step 4.

<table>
<thead>
<tr>
<th>swap</th>
<th>complement</th>
<th>Sign($a_1$)</th>
<th>Sign($a_2$)</th>
<th>Sign(result)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td></td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Yes</td>
<td></td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

**Table 7.2: Sign Calculation Used in Floating Point Add/Sub**

Architecture of floating-point addition-subtraction block (FASC) is shown in figure 7.4. The same block is also used for comparison of two floating-point numbers. A flag named FASC_flag is used for comparison of two floating-point numbers. It is set to ‘1’ when first operand is greater than the second operand otherwise it resets to ‘0’. The controller is a dedicated FSM with 4 states. The output of the FASC functional block is available after 4 clock cycles of loading the operands. Operations on special numbers (zero) are also handled by this block. RT-level VHDL code of FASC is provided at Appendix-A.

Functional verification of the FASC functional block is also done using the same approach as described in section 7.2.1. RT-level VHDL code and the generated gate-level netlist of the FASC functional block are tested extensively through the application of test vectors and by comparing their outputs with the outputs of a reference 'C' program functional model.
7.4 FMPY (Floating-point Multiplier)

Let $a_1$ and $a_2$ be two IEEE-754 floating-point numbers to be multiplied. The notations $e_i$ and $s_i$ are used for the exponent and significand of $a_i$.

$$a_i = s_i \times 2^{e_i}$$

$s_i$ has an explicit leading bit i.e. '1'. The floating-point multiplication of $a_1$ and $a_2$ [10] has several parts as shown in equation 7.1:

$$\left(s_1 \times 2^{e_1}\right) \times \left(s_2 \times 2^{e_2}\right) = \left(s_1 \times s_2\right) \times 2^{e_1 + e_2} \quad \text{equation 7.1}$$

1) The first part multiplies the significands using integer multiplication i.e. $s_1 \times s_2$.

In our case, significands are unsigned 24-bit numbers, so the generated product is of 48 bits.

2) The second part rounds the generated product to a 24-bit number.

3) The third part computes the new exponent by taking sum of two exponents ($e_1$ and $e_2$) and then subtracting the bias i.e. 127 from the sum.

4) Finally, the sign of the result is calculated by doing the XOR operation (on the sign1 and sign2).

For the multiplication of significands, we used a 24*6 multiplier block in a sequential manner to get a final 24*24 product (48 bits). Then rounding is performed on
the 48 bits product to get the final 24 bits product. The two cases of the rounding modes used are given in figure 7.5.

![Product](Figure 7.5: Rounding Modes)

The top line in figure 7.5 shows the contents of the P and A registers after multiplying the significands (here, p = 6). The sticky bit (s) is the logical OR of the s bits, g (for guard) the most-significant bit of A, and r (for round) the second most-significant bit of A. There are two cases:

1) The high-order bit of P is 0. Shift P left 1 bit, shifting in the g bit from A. Shifting the rest of A is not necessary.

2) The high-order bit of P is 1. Set \( s = s \lor r \) and \( r := g \), and add 1 to the exponent.

Now if \( r = 0 \), P is the correctly rounded product. If \( r = 1 \) and \( s = 1 \), then \( P + 1 \) is the product (where by \( P + 1 \) we mean adding 1 to the least-significant bit of P). If \( r = 1 \) and \( s = 0 \), we are in a halfway case, and round up according to the least significant bit of P.

Architecture of the floating-point multiplication is shown in figure 7.6. A dedicated controller is designed to generate the necessary controls. The Controller is a dedicated FSM with 6 states. The output of the FMPY functional bock is available after 6 clock cycles of loading the operands. This block also handles the special operation i.e. multiplication by zero. RT-level VHDL code of FMPY functional bock is provided at Appendix-A.

Functional verification of the FMPY functional block is also done using the same
approach as described in section 7.2.1. RT-level VHDL code and the generated gate-level netlist of the FMPY functional block are tested extensively through the application of test vectors and by comparing their outputs with the outputs of a reference 'C' program functional model.

Figure 7.6: Architecture of FMPY Functional Block
7.5 FDIV (Floating-point DIVision)

Let \( a_1 \) and \( a_2 \) be two IEEE-754 floating-point numbers to be divided. The notations \( e_i \) and \( s_i \) are used for the exponent and significand of \( a_i \) [10]. \( s_i \) has an explicit leading bit i.e. '1'. Division of \( a_1 \) by \( a_2 \) [10] has several parts as shown in equation 7.2:

\[
\left( s_1 \times 2^{e_1} \right) / \left( s_2 \times 2^{e_2} \right) = \left( s_1 / s_2 \right) \times 2^{e_1 - e_2}
\]

\[
\text{equation 7.2}
\]

1) The first part divides the significands using ordinary integer division i.e. \( s_1 / s_2 \). In our case, significands are unsigned 24-bit numbers (single precision). The divider operates on two significands and produces quotient bit one at a time.

2) The second part rounds the result.

3) The third part computes the new exponent by taking difference of the two exponents (\( e_1 \) and \( e_2 \)) and then adding the bias i.e. 127 from the difference.

4) The sign of the result is calculated by doing the XOR operation on the sign1 and sign2.

For the division of significands, we used the restoring division algorithm [33] in a sequential manner (one quotient bit generated in one cycle) to get the final quotient of 24 bits and then rounding is performed on this final quotient to get the final normalized result of 24 bits. Architecture of the floating-point division is shown in figure 7.7. A dedicated controller is designed to generate the necessary controls. The Controller is a dedicated FSM with 26 states. The output of the FDIV functional block is available after 26 clock cycles of loading the operands. This block also handles the special operation i.e. zero divide by a non-zero number. RT-level VHDL code of FDIV functional block is provided at Appendix-A.

Functional verification of the FDIV functional block is also done using the same
approach as described in section 7.2.1.

![Architecture of FDIV Functional Block](image)

*Figure 7.7: Architecture of FDIV Functional Block*
7.6 SEXP (Exponential Function)

This functional block computes the exponential value (w.r.t. base e) for a given argument. In the source-filter model of human speech production, the vocal tract has been modeled as a set of filters, connected either in cascade or in parallel mode. The filters that are used are second order IIR filters represented by

\[ y(nT) = Ax(nT) + By(nT - 1) + Cy(nt - 2) \]

The constants A, B and C are related to the resonant frequency F and bandwidth BW of the filters as:

\[ C = -e^{-2\pi BW} \]
\[ B = 2e^{-\pi F} \cos(2\pi FT) \]
\[ C = 1 - B - C \]

These expressions involve computation of exponential and cosine functions. The steps for the calculation of exponential function [33] are as follows:

(1) Store 1 in the output register Y and the argument of the exponential function is stored in the input register X. The input is in 25-bit special format in which the first two most-significant bits specify the integer part and the rest 23 bits specify the fraction part. The binary point is fixed after the two most significant bits in this format.

(2) Using recursive equation:

\[ x_{i+1} = x_i - \ln b_i \]
\[ y_{i+1} = y_i b_i \]

where \( b_i \) is taken as \( b_i = 1 + s_i 2^{-i} \)
Here $s_i$ takes values from the set \{-1, 0, 1\}. In our case we have used the reduced set \{-1, 1\} for $s_i$. After $m$ iterations, the contents of $X$ register converge to zero. When $X$ becomes zero, $Y$ register will contain the exponential of the input as given by

$$y_m = y_0 \prod_{i=0}^{m-1} s_i 2^{-i}$$

In our case, number of iterations $m$ is 24.

(3) To find $s_i$ values, the difference $D = x_i - \ln(1 + 2^{-i})$ is calculated. Now a two-sided selection rule is used in the following way to get $s_i$ values:

$$s_i = \begin{cases} -1 & \text{if } D < 0 \\ 1 & \text{if } D \geq 0 \end{cases}$$

(4) The above computation is valid for the input argument range $-1.24 \leq x_0 \leq 1.56$ for $m \geq 20$. To handle inputs beyond this range, the following equivalent representation $x = x \log_2 e \ln 2$ is used. Now $x \log_2 e$ can be expressed as $I + f$, where $I$ is the integer portion and $f$ is the fractional portion. We can now write

$$y = e^x = e^{(I + f) \ln 2} = e^{I \ln 2} \cdot e^{f \ln 2} = 2^I \cdot e^{f \ln 2}$$

Therefore to compute $e^x$, we need to compute $e^{f \ln 2}$ and shift the result to the left by $I$ bits.

Architecture of the exponential functional block is shown in figure 7.8. A dedicated controller is designed to generate the necessary controls. The Controller is a dedicated FSM with 54 states. The output of the SEXP functional block is available after 54 clock cycles of loading the operands. RT-level VHDL code of SEXP functional block is provided at Appendix-A.
Functional verification of the SEXP functional block is also done using the same approach as described in section 7.2.1.

Figure 7.8: Architecture of SEXP Functional Block

preset_Wi_Yi_reg, en_Wi_Yi_reg, en_Xint_reg, reset_cc_reg, en_cc_reg, preset_count, reset_count, en_count, selector_control, add_sub_control1, cc_control, reset_start_reg, add_sub_control2
7.7 SCOS (Sine-COSine)

This block computes the sine and cosine function simultaneously. Algorithm for computation of cosine function is similar to the exponential function \[33\] as

\[ e^{jx} = \cos(x) + j \sin(x), \quad j = \sqrt{-1} \]

The recursive relations are given by

\[ x_{i+1} = x_i - g(b_i) \]
\[ y_{i+1} = y_i \cdot b_i \]

Where \( b_i \) is selected as

\[ b_i = (1 + js_i 2^{-i}) \]
\[ = e^{i\theta_i} \sqrt{1^2 + (s_i 2^{-i})^2} \]

If the recursive relation converges in the \( m^{th} \) iteration, then we can write

\[ y_m = y_0 \prod_{i=0}^{m-1} b_i = y_0 \prod_{i=0}^{m-1} e^{j\theta_i} \cdot \sqrt{(1 + s_i 2^{-i})^2} \]
\[ = y_0 \left( \exp \left( j \sum_{i=0}^{m-2} \theta_i \right) \cdot \prod_{i=0}^{m-1} \sqrt{(1 + s_i 2^{-i})^2} \right) \]
\[ = y_0 \cdot K \cdot \exp \left( j \sum_{i=0}^{m-1} \theta_i \right) \]

Let \( g(b_i) = \theta_i = \tan^{-1}(s_i 2^{-i}) \).
The input recursive relation can be written as
\[ x_{i+1} = x_i - \tan^{-1}(s, 2^{-i}) \]

When \( x_m \to 0 \), we obtain
\[ \sum_{i=0}^{m-1} \tan^{-1}(s, 2^{-i}) = \sum_{i=0}^{m-1} \theta_i = x_0 \]

Substituting this in \( y_m \) we get
\[ y_m = y_0 \cdot K \cdot e^{j \theta_0} = y_0 \cdot K \cdot (\cos(x_0) + j \sin(x_0)) \]

\( x_i \)'s real numbers but \( y_i \)'s are complex numbers.

Let, \( y_i = Z_i + jW_i \),
where \( Z_i \) is the real part and \( W_i \) is the imaginary part of \( y_i \). The expression for \( y_{i+1} \) is
\[ y_{i+1} = Z_{i+1} + jW_{i+1} = (Z_i + jW_i)(1 + js_i 2^{-i}) \]

The recursive relation for \( Z_{i+1} \) and \( W_{i+1} \) are
\[
Z_{i+1} = Z_i - s_i 2^{-i} W_i \\
W_{i+1} = W_i + s_i 2^{-i} Z_i
\]

……..equation 7.3

The initial value of \( y_0 \) can also be a complex number, \( y_0 = Z_0 + jW_0 \). If we set \( W_0 \) equal to zero, making \( y_0 \) equal to \( Z_0 \), we get the desired result
\[ Z_{i+1} \rightarrow y_0 \cdot K \cdot \cos(x_0) \]
\[ W_{i+1} \rightarrow y_0 \cdot K \cdot \sin(x_0) \]

The convergence domain for this algorithm for large values of \( m \geq 20 \) is 
\(-1.743 \leq x_0 \leq 1.743 \) which covers the useful domain i.e. \( 0 \leq x_0 \leq \pi / 2 \).

\( s_i \) take values from the set \{-1, 0, 1\}. In our case we have used the restricted set \{-1, 1\} for \( s_i \) values. For this restricted set, 
\[ K = \prod_{i=0}^{m-1} \sqrt{1+(s_i \cdot 2^{-i})^2} \]

is computed to be \( K = 1.6468 \) for \( m > 16 \). The selection rule for \( s_i \) now becomes

\[ s_i = \begin{cases} 
1 & \text{if } x_i \geq 0 \\
-1 & \text{if } x_i < 0 
\end{cases} \]

The steps used in the implementation of SCOS functional block are as follows:

1. Store \((1/K) = 0.60723\) in Y register, zero in W register and the input (same as in SEXP block in 25-bit special format) in X register.

2. Make the contents of X register converge to zero after \( m \) (in this case \( m \) is equal to 24) iterations. Correspondingly, the contents of Z and W registers are changed according to equation 7.3 for \( m \) iterations.

3. After \( m \) iterations when X register becomes zero, Z register will contain cosine of the input and W register will contain the sine of the input.

Architecture of the cosine functional block is shown in figure 7.8. A dedicated controller is designed to generate the necessary controls. The Controller is a dedicated FSM with 78 states. The output of the SCOS functional block is available after 78 clock cycles of loading the operands. RT-level VHDL code of SCOS functional block is provided at Appendix-A.
Functional verification of the SCOS functional block is also done using the same approach as described in section 7.2.1.
7.8 Synthesis Results

All the functional blocks are individually implemented on the Xilinx FPGA. Synthesis results of all the functional blocks after mapping on the FPGA are given in table 7.3 along with the vendor, device family and device details.

Vendor: Xilinx, Device family: Spartan2E, Device: Xc2s300e

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Functional Block</th>
<th>Equivalent Gate Count</th>
<th>Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>IASC</td>
<td>811</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>FASC</td>
<td>8,266</td>
<td>4</td>
</tr>
<tr>
<td>3.</td>
<td>FMPY</td>
<td>4,322</td>
<td>6</td>
</tr>
<tr>
<td>4.</td>
<td>FDIV</td>
<td>2,949</td>
<td>26</td>
</tr>
<tr>
<td>5.</td>
<td>SCOS</td>
<td>3,768</td>
<td>78</td>
</tr>
<tr>
<td>6.</td>
<td>SEXP</td>
<td>3,387</td>
<td>54</td>
</tr>
<tr>
<td>7.</td>
<td>IGEN</td>
<td>1,302</td>
<td>14</td>
</tr>
<tr>
<td>8.</td>
<td>ITOF</td>
<td>2,010</td>
<td>1</td>
</tr>
<tr>
<td>9.</td>
<td>F2IS</td>
<td>7,409</td>
<td>1</td>
</tr>
<tr>
<td>10.</td>
<td>STOF</td>
<td>2,258</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 7.3: Synthesis Results of Functional blocks