Traditionally sinusoidal signal generation has been implemented with purely analog circuits such as phase-locked loops. The alternative of using a digital system to perform this signal generation has previously been unattractive due to limitations in clock frequency and size. However, recent advancements in sub-micron fabrication techniques have made the digital alternative tractable. The advantages of a digitally implemented signal frequency synthesizer include finer control of output frequency, reduced frequency drift due to part degradation over time, and faster response time for frequency change.

Digital frequency synthesis has been previously realized using the Tierney, Rader, and Gold phase accumulator architecture. This method utilizes a variable-increment digital integrator that is input to a read-only memory. This memory then generates a quantized amplitude value. This thesis presents an alternative method for digital frequency synthesis based on circular interpolation and compares it to the performance of a comparable phase-accumulator structure for varying bit-accuracies of phase. The comparison of transistor count and required die-size for each method reveals a lower requirement of both resources in the case of the new circle interpolator. Evaluation of the discrete-time spectral purity of synthesized signals also demonstrates less out of band noise in the new design. Finally, analysis
of energy efficiency shows the new design to be generally optimal compared to the reference design.
Interpolation-Based Digital Quadrature Frequency Synthesizer

by

Ryan John Larson

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 5, 2000
Commencement June 2001
Master of Science thesis of Ryan John Larson presented on June 5, 2000

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical & Computer Engineering

Redacted for Privacy

Chair of the Department of Electrical & Computer Engineering

Redacted for Privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Ryan John Larson, Author
ACKNOWLEDGMENT

This thesis acknowledges the following people for their support and assistance:

Shih-Lien Lu
Roger Traylor
Eric Campbell
Mary French
John T. Stonick
Un Ku Moon
Eva Piccininni
Brandon Greenley
John Bennett
Kent Lusted
Aaron Caffee
Matthew Coe
Georgi Todorov
Junlin Zhou
Carey Larson
TABLE OF CONTENTS

1. INTRODUCTION ................................................................................. 1
   1.1 Problem Definition ................................................................. 1
   1.2 Statement of Purpose .............................................................. 2

2. REVIEW OF LITERATURE ................................................................. 3
   2.1 Phase Calculation ................................................................. 3
   2.2 Magnitude Conversion ............................................................ 4
   2.3 Noise Sources ....................................................................... 6
   2.4 Critical Path Delay ............................................................... 7
   2.5 Power Dissipation ................................................................. 8
       2.5.1 Dynamic Switching ...................................................... 9
       2.5.2 DC Path Power ........................................................... 9
       2.5.3 Static Leakage ............................................................ 10
   2.6 Summary of Review .............................................................. 10

3. DDFS DESIGNS .............................................................................. 11
   3.1 Circle Interpolation Synthesizer ........................................... 11
       3.1.1 Interpolation Calculation ........................................ 11
       3.1.2 Circle Interpolator Modulation .................................. 14
       3.1.3 Output Limitation ..................................................... 15
       3.1.4 Frequency Change Synchronization .......................... 15
       3.1.5 Design Synthesis ...................................................... 16
   3.2 Phase Accumulation Synthesizer ........................................... 16
       3.2.1 ROM Implementation .............................................. 17
       3.2.2 ROM Compression ................................................... 18
       3.2.3 Phase Accumulator Modulation ............................... 20
   3.3 Recursive Doubling Adder .................................................... 20
   3.4 Summary of Designs ............................................................ 21

4. EXPERIMENTAL RESULTS ............................................................ 23
   4.1 Simulation Variations ............................................................ 23
   4.2 Propagation Delay ............................................................... 23
   4.3 Spectral Content ................................................................. 26
<table>
<thead>
<tr>
<th>TABLE OF CONTENTS (Continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4 Spectral Purity at Frequency Modulation</td>
</tr>
<tr>
<td>4.5 Power Dissipation</td>
</tr>
<tr>
<td>4.6 Experimental Results Summary</td>
</tr>
<tr>
<td>5. CONCLUSIONS</td>
</tr>
<tr>
<td>5.1 Summary</td>
</tr>
<tr>
<td>5.2 Future Research</td>
</tr>
<tr>
<td>REFERENCES</td>
</tr>
<tr>
<td>APPENDICES</td>
</tr>
<tr>
<td>A CircleInterpolator VHDL Design</td>
</tr>
<tr>
<td>B Phase Accumulator VHDL Design</td>
</tr>
<tr>
<td>C ROM Netlist Generation Script</td>
</tr>
<tr>
<td>D MATLAB Simulation Files</td>
</tr>
<tr>
<td>Figure</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>2.1 Phase Accumulator Data-path</td>
</tr>
<tr>
<td>2.2 MOSFET Parasitic Circuit Model</td>
</tr>
<tr>
<td>2.3 Delay vs. Feature Size</td>
</tr>
<tr>
<td>3.1 Coordinate Plane Diagram for Algorithm</td>
</tr>
<tr>
<td>3.2 Datapath Diagram for Circle Interpolator</td>
</tr>
<tr>
<td>3.3 ROM Array Architecture</td>
</tr>
<tr>
<td>3.4 ROM Decoder and Array</td>
</tr>
<tr>
<td>3.5 ROM Data Reconstruction Datapath</td>
</tr>
<tr>
<td>3.6 Prefix Structure</td>
</tr>
<tr>
<td>4.1 Design Flow Graph</td>
</tr>
<tr>
<td>4.2 Critical Delay Path Time</td>
</tr>
<tr>
<td>4.3 ROM Critical Delay Path Time</td>
</tr>
<tr>
<td>4.4 Signal to Noise Floor Power</td>
</tr>
<tr>
<td>4.5 Carrier Lobe Spectral Purity</td>
</tr>
<tr>
<td>4.6 Interpolation Phase Error</td>
</tr>
<tr>
<td>4.7 Overshoot Noise Power</td>
</tr>
<tr>
<td>4.8 Carrier to Noise Floor at Modulation</td>
</tr>
<tr>
<td>4.9 Large Increment Overshoot</td>
</tr>
<tr>
<td>4.10 Dynamic Power Dissipation</td>
</tr>
<tr>
<td>4.11 Speed Power Product</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 State Encoding for Interpolation Algorithm</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Transistor Count and Area Variations for the Two Designs</td>
<td>22</td>
</tr>
<tr>
<td>4.1 Carrier Power Relative to Noise Floor Power</td>
<td>27</td>
</tr>
<tr>
<td>4.2 Dynamic Power Consumption as a Function of Design and Size</td>
<td>34</td>
</tr>
</tbody>
</table>
This thesis is dedicated to the memory of Stanley Larson.
INTERPOLATION-BASED DIGITAL QUADRATURE FREQUENCY SYNTHESIZER

1. INTRODUCTION

Carrier-signal generation is one of the common features in modern digital communication systems. Modulation and demodulation from signal to data source or sink requires a reference carrier waveform. While carrier generation has traditionally been carried out in analog circuits, new methods of digital synthesis have been developed with numerous benefits. However, as the need for greater device density increases, space, power, and speed are primary concerns.

1.1 Problem Definition

Previously frequency or signal synthesis saw implementation through analog hardware, most commonly in phase-locked loop (PLL) circuits. While this method has high frequency output capabilities and reasonable spectral purity, it does suffer from some inherent problems. PLL circuits result in a small operating frequency range, long frequency tuning periods, and difficult production [1]. Digital techniques, known as direct digital frequency synthesis (DDFS), benefit from fast switching times, smooth frequency transitions, and fine frequency granularity. Also, given a large frequency control word, a large number of frequencies can be synthesized.

Prior DDFS implementations, such as phase accumulation, have relied upon a memory storage device to provide quantized values of the carrier signal. This memory, usually a ROM, contributes considerable area to the design as well as increasing the power consumption due to the internal capacitances. The capacitances
also result in long read-cycle times for a large amount of data. Finally, generation of sinusoids in quadrature also requires either phase-offset hardware for a second lookup or another ROM to perform magnitude conversion in parallel.

To increase the performance of the synthesizer it would be desirable to remove the ROM and perform phase to magnitude translation in a more efficient manner. Such a method could be purely algorithmic in nature and not rely on any form of look-up memory device. This thesis presents a new algorithm-based method to digitally synthesize quadrature sinusoidal waveforms.

1.2 Statement of Purpose

The purpose of this thesis is to demonstrate that by utilizing an algorithmic approach to DDFS spectral purity can be increased and the required die-size can be decreased. Simulated implementations of the algorithm and phase-accumulator based designs are used to verify the performance benefits. This thesis presents circuit simulation results performed in HSPICE, MATLAB, and the Mentor Graphics application set. The two designs were simulated for the worst-case propagation delay, power, and spectral purity. This thesis intends to show that the new algorithm based design is smaller and more spectrally pure than the existing phase accumulator, as well as being more energy efficient.
2. REVIEW OF LITERATURE

Before describing the designs, simulations, and results, concepts related to DDFS need to be reviewed. To meet the requirement for a variable-frequency high-accuracy digital oscillator, a method known as a phase accumulation is commonly employed. In this process phase increments of a known waveform, in this case a carrier sinusoid, are generated at regular intervals referenced to an input clock source. At each clock period a phase-to-magnitude conversion takes place which is passed to a digital to analog converter (DAC) for signal generation. The following sections describe the specific details of phase calculation, phase to magnitude conversion, and sources of noise in phase accumulation. Limitations in propagation delay and sources of power consumption in digital circuits are also reviewed.

2.1 Phase Calculation

The most popular technique used in DDFS designs is the Tierney, Rader, and Gold architecture [2] [3]. The core of this architecture is a modulo-N accumulator realized from a two-input N-bit adder. The output value of the accumulator is added to the value stored in a frequency-control register once during each period of the system clock [4]. The sum is stored in an accumulation register and fed back to the other input of the adder for the following cycle. To generate a desired frequency the increment value is loaded into the frequency-control register and provides a phase step during accumulation. A simple variation on this phase calculator, known as phase truncation, is to make the accumulator larger (in regard to the number of operand bits) than the number of bits required for the phase-to-magnitude conversion. By using the higher order bits and truncating the bits of lower significance a more frequencies can be synthesized without resizing the memory device.
The size of the utilized accumulator output dictates the phase-step granularity of the output waveform. This is analogous to determining the sampling frequency of the waveform before being stored digitally. For a N-bit phase to magnitude conversion a total of $2^N$ phase steps per sinusoid period are possible without phase truncation when the frequency control word is set at unity. Taking the input clock source into consideration the effective output frequency, $F_o$, can be described as $F_o = \frac{F_i W}{2^N}$ where $F_i$ is the frequency of the clock source and $W$ is the phase increment. An alternate explanation considers the increment in phase to be $\delta \phi(t) = \frac{W \cdot 2\pi}{2^N}$. The change in time is then $\delta T = \frac{1}{F_i}$, and output frequency is calculated from the phase derivative $F(t) = \frac{\delta \phi}{2\pi \delta T} = \frac{F_i W}{2^N}$ [5].

An advantage of having the phase from the accumulator immediately available is that it facilitates simple phase-offset changes. A secondary adder following the accumulator allows instantaneous phase changes relative to the current phase. Phase modulation or phase shift-keying are easily implemented in this manner. All of these features make a digital integrator a very versatile phase generator.

### 2.2 Magnitude Conversion

Implementation of the phase to magnitude conversion has been one of the important areas of research in phase accumulation based DDFS. The computation from phase argument can have high latency and therefore may become a bottleneck in the maximum achievable clock rate. In DDFS designs the computation of the sinusoid has been a fundamental limitation in system throughput, causing the usable output bandwidth of DDFS designs to lag behind DAC designs [6].

Attempts at improving DDFS clock rates have come at the expense of spurious noise performance by approximating the sinusoid calculation in order to reduce the required size and access time of the ROM. As sine or cosine are transcendental
functions, they are difficult to calculate with a high degree of accuracy. Methods such as Taylor Series expansion or CORDIC Algorithms have been considered but lack the speed of a ROM table look-up [6]. Effort has therefore been placed on content compression to minimize the size of the ROM and decrease read time latency. A method proposed by Sunderland et. al replaces a ROM of $2^{A+B+C}$ words by coarse and fine ROMs, $2^A + B$ and $2^A + C$ words in size, respectively [7]. The approximation

$$\sin(\alpha + \beta + \chi) = \sin(\alpha + \beta)\cos(\chi) + \cos(\alpha + \beta)\sin(\chi)$$

$$\approx \sin(\alpha + \beta) + \cos(\alpha)\sin(\chi)$$

is then used to calculate the contents, where $\alpha$, $\beta$, and $\chi$ are three segments of the phase bit-lines normalized by $\frac{2\pi}{2^{A+B+C}}$. Data reconstruction is performed by addition and therefore requires minimal hardware. Similar work by Samueli, Nicholas, and Kim has examined more exact methods of calculating the fine ROM contents [2].
2.3 Noise Sources

Digital sources of noise in phase accumulator DDFS systems are found at two points in the architecture. The first occurs within the magnitude to frequency conversion as magnitude distortion. In the optimal case magnitude distortion doesn’t occur within the ROM contents and appears only as quantization error at the DAC. However, in the case of ROM content compression, the reconstructed data may contain an error determined to be acceptable in the output word. This error is usually caused by an approximation made when generating the ROM contents as described in section 2.2. This concession is often made because of the desire to increase the ROM read time.

The second source of digital noise is the inherent quantization error due to the sampled nature of the waveform data. If the quantization error is assumed to be evenly distributed over half a quantile interval, then the quantization noise and corresponding signal-to-noise ratio (SNR) component can be described as

$$\sigma^2 = \int_{-q/2}^{q/2} e^{-x^2/2} dx = \frac{q^2}{12}, \quad \left( \frac{S}{N} \right)_{q, peak} = \frac{12V_p^2}{q^2}$$

[8]. However, the presence of ROM data compression causes the magnitude conversion to generate more spurious noise than the data quantization.

As the described synthesizer is digital and discrete in nature, analog noise sources such as clock switching aren’t of any concern unless they are directly disrupting logic signals. While magnitude integrity is relatively devoid of noise effects, the phase is not. Jitter associated with the governing clock is the major contributor to phase noise in DDFS systems. Frequency division of the source clock by the synthesizer does alleviate some of this noise, however. The phase noise improvement of the DDFS system over the clock source is expressed as $20 \cdot log_{10}(F_i/F_o)$. As the frequency of the output approaches the input the system phase noise increases.
2.4 Critical Path Delay

In sequential logic systems the cumulative longest propagation delay between sequential devices limits the maximum clock period for a design. The delay consists of combinational gate input to output transition time in conjunction with the interconnect delay from the wiring of the logic. In addition, fan-in, fan-out, and input gate capacitance also add to the signal propagation delay. Within the scope of a MOSFET device a simple RC network consisting of the source and drain capacitance and channel resistance models the delay path \([9]\), and is shown in figure 2.2. MOSFET circuit formulations that realize logic gates therefore result in varying complex RC networks that determine the gate transition delay.

The interconnect delay arises from the stray capacitances and sheet resistance of a metal path. This delay becomes more of a prominent concern as reduction in fabrication feature size continues. While the intrinsic capacitances and resistances associated with MOS devices are minimized, the inter-block connection length in hierarchical designs become longer. Compounding this problem, signal coupling and transmission line effects between nearby interconnecting networks are more

![FIGURE 2.2: MOSFET Parasitic Circuit Model](image-url)
pronounced, causing propagation delay to match rise and fall times. These factors cause interconnect delay to dominate over gate delay as feature size is scaled down, as shown in figure 2.3 [10]. Mask extraction of parasitic capacitances and resistances to facilitate interconnect delay computation has become more of a requirement in current design efforts. Typically this information is back-annotated to a static delay model of the design for timing evaluation.

2.5 Power Dissipation

Power dissipation is another critical characteristic of digital circuits that requires some consideration. Sources of power dissipation are be divided in three primary categories: dynamic switching, DC path current, and static leakage current. Each of these is now described in turn.
2.5.1 Dynamic Switching

The largest contribution of power dissipation comes from dynamic switching. Ignoring leakage and DC paths, a capacitive load in static CMOS may be either charged or discharged at any one logic event in time. Power spent in these two states can be calculated by $P_{\text{dyn}} = C_L V_{dd}^2 f \alpha [10]$. While the load capacitance and power supply voltage have an apparent contribution to the calculation, the frequency at which switching events occur also effects the utilized power. The term $\alpha$ denotes the activity factor of a particular switching node relative to the clock frequency. A value of $\alpha = 1$ indicates that the node is actively switching every period while a value of $\alpha = 0.5$ approximates a node switching every other cycle, and consuming half of the relative power.

2.5.2 DC Path Power

The next most dominant contribution in power comes from DC current paths. Finite fall and rise times in CMOS logic circuits result in a direct path across the pull-up and pull-down components between $V_{dd}$ and ground. In static logic devices both PMOS and NMOS devices induce a carrier channel between the source and drain nodes simultaneously. This is due to the input gate voltage exceeding both threshold voltages. The power lost to these events can be characterized by $P_{\text{dc}} = \frac{1}{2} (t_f + t_r) V_{dd} I_{\text{peak}} f \alpha$. Like dynamic switching, the power loss is proportional to both the clock frequency and the activity factor of the gate node.

Another form of DC path power loss exists for non-static logic circuits, such as fixed-load or ratio-ed logic circuits. A DC current path exists continuously if the pull-down network facilitates a DC path to ground. This type of logic implementation is clearly undesirable if a design effort is power-conscious and can be neglected.
Circuit formulations in this work are static and this form of ratio-ed logic power loss can be ignored.

2.5.3 Static Leakage

Finally, the last expense in power consumption is static leakage current. Reverse-biased PN junctions found in CMOS technology have a small finite amount of current opposing the biased nodes. Thermally generated carriers cause this leakage and is it is obviously magnified by increased junction temperature. Another source is sub-threshold current that occurs in MOSFET devices. This loss is increased due to small threshold voltages and is usually offset by voltage threshold implants. Lastly, leakage current through the gate-oxide also contributes to static power loss. As fabrication technology progressively becomes smaller, so does the silicon-oxide layer that separates the MOSFET gate from the channel. This decrease in thickness results in an exponential increase in gate-substrate leakage current [11]. The total static power from these sources is defined by $P_{st} = I_{leak}V_{dd}$.

2.6 Summary of Review

The Tierney et. al phase accumulator model for DDFS has been reviewed. Both the phase generation and magnitude translation theories were elaborated. Digital noise sources contributed by this architecture were also described. The primary sources are quantization noise, magnitude distortion, and clock jitter. Signal propagation delay was elaborated as well. It was noted that interconnect parasitics contribute a large portion of critical path delay in smaller technologies. Finally, sources of power dissipation were identified and reviewed. Dynamic switching was shown to be the dominant source of power usage in static CMOS designs.
3. DDFS DESIGNS

Now that concepts behind DDFS systems has been reviewed two different implementations can be inspected in detail. The first is a new system based on a purely algorithmic method of value calculation to synthesize sinusoid waveforms. The second is the previously described phase accumulator that uses both a digital integrator and a lookup table to perform frequency synthesis. Both designs presented here have the same output-word size and were scaled for size comparison and simulation results. The Mentor Graphics software set, the application MATLAB, and the circuit simulator HSPICE were used to simulate the two designs.

3.1 Circle Interpolation Synthesizer

The first design presented is the new circle interpolation synthesizer. The name is self-descriptive as the circuit continuously calculates the quadrature sine and cosine outputs as the result of interpolating a circle of a set radius. The method to perform this operation was derived from a graphical algorithm to plot circles without utilizing tables or floating-point arithmetic [12].

3.1.1 Interpolation Calculation

The algebraic determinant function for a circle is described as \( x^2 + y^2 - r^2 = f(x, y) \), where:

all points \( x, y \) for which \( f(x, y) = 0 \) lie on the circle

all points \( x, y \) for which \( f(x, y) > 0 \) lie outside the circle
all points \((x,y)\) for which \(f(x,y) < 0\) lie inside the circle

In discrete units the equation can be represented as \(X^2 + Y^2 - R^2 = F\). In order to derive the new error function \(F'\), consider the difference between an increment in the positive direction of the X-axis, \(X + i\), and the equation for the previous coordinate position.

\[
((X + i)^2 + Y^2 - R^2) - (X^2 + Y^2 - R^2) = F'(X, Y) - F(X, Y)
\]

\[
(X^2 + 2Xi + i^2 + Y^2 - R^2) - (X^2 + Y^2 - R^2) = F'(X, Y) - F(X, Y)
\]

\[
2Xi + i^2 = F'(X, Y) - F(X, Y)
\]

\[
F(X, Y) + i(2X + i) = F'(X, Y)
\]

Similar results are derived when considering decrementing in the X axis as well as any change in the Y axis. Therefore, there a total of four cases to consider. The cumulative change between subsequent error function values can be interpreted as a series summation of \(i(\pm2(X|Y) + i)\) terms. As the desired sum is a value of zero the increment term \(i\) can be factored out resulting in the simplified error calculation \(F'(X, Y) = F(X, Y) + (\pm2(X|Y) + i)\). The coordinate-plane diagram in figure 3.1 suggests a set of decision determinants based on the sign of the error function and the current quadrant to determine the next operation. These decisions are based on counter-clockwise interpolation of the circle.

From table 3.1 a state-based algorithm can be derived. Utilizing a set of adders for the coordinate axis variation and error function, the sign bits in two's complement notation can be used as next-state variables in a sequential state machine.
FIGURE 3.1: Coordinate Plane Diagram for Algorithm

<table>
<thead>
<tr>
<th>$F_{sign}$</th>
<th>$X_{sign}$</th>
<th>$Y_{sign}$</th>
<th>Axis Operation</th>
<th>Error Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$X' = X - i$</td>
<td>$F' = F - 2X + i$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$Y' = Y - i$</td>
<td>$F' = F - 2Y + i$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$X' = X + i$</td>
<td>$F' = F + 2X + i$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$Y' = Y + i$</td>
<td>$F' = F + 2Y + i$</td>
</tr>
</tbody>
</table>

TABLE 3.1: State Encoding for Interpolation Algorithm
3.1.2 Circle Interpolator Modulation

The number of clock cycles required to traverse the circle and complete one period of the quadrature outputs can be determined from the bit accuracy of the implementation and the radius of the interpolation algorithm. If it is assumed that the coordinate position is initialized on the X coordinate axis, then it will require the intercept value in cycles to reach the zero value of the X axis. Similarly, it will take as many cycles to start from the orthogonal Y coordinate zero and reach its peak value. Only one step in one axis occurs in any cycle in the algorithm, therefore it will take twice the number of steps from the X axis-intercept to interpolate one quadrant of the circle. Under the numeric encoding context of Radix Complement (RC), using an increment value of 1, and starting at the peak of the integer range, the number of cycles required to complete one period is \( Cycles_{clk} = 4 \cdot 2^{(2^N - 1)} \). N is the number of bits used per axis register.

For larger increment values, initializing the amplitude at the peak of the RC integer range is impractical. Consider the sine component of the quadrature outputs initialized at zero. A total of \( \left\lceil \frac{2^N - 1}{i} \right\rceil \) steps are required to meet or exceed the amplitude set by the initial cosine value. Clearly if \( \left\lceil \frac{2^N - 1}{i} \right\rceil \cdot i > 2^N - 1 \) then an arithmetic overflow will result causing the controlling state machine to interpret the summation result as a sudden phase change. To avoid this situation while allowing for large increment values the X intercept value is initialized to a smaller value of \( 2^{N-2} - 1 \). To determine the number of cycles to generate a sinusoid period with non-unit increments the number of required incremental steps for each quadrature component is accounted for. This results in \( Cycles_{clk} = 4 \left\lceil \frac{2^{N-2} - 1}{i} \right\rceil + 2 \left\lceil \frac{2^{N-1} - 2}{i} \right\rceil \) cycles to complete a waveform for a given increment \( i \). This calculation also takes into account the reduced initial value to avoid arithmetic overflow.
3.1.3 Output Limitation

For $i > 1$ there is a concern of exceeding the output value range. As mentioned the algorithm will increment each axis value until it meets or exceeds the initial starting value. While this doesn’t affect the internal calculation, it does cause the output word to overflow beyond its intended two’s complement $\pm 2^{N-2} - 1$ range. To control this problem extra logic was introduced that verifies the values are in range by examining the two most significant bits of the entire axis word. Consider the case where the axis variable is in the positive plane, then for $k \leq 2^{N-2} - 1$ bits $k_{N-1}$ and $k_{N-2}$ should not be set. Conversely for $k \geq -2^{N-2} - 1$ bits $k_{N-1}$ and $k_{N-2}$ should be set. All that’s required for detection is a two-input XOR gate tied to the two most significant bits. When the range is exceeded constants $\pm 2^{N-2} - 1$ are multiplexed to the output port instead of the calculated value.

3.1.4 Frequency Change Synchronization

Contrasting the phase accumulator the circle interpolator requires some synchronization to change the synthesized frequency. Correct management of the axis variables requires that the algorithm return to its initial starting point in order to start with a new increment value. Mid-period change of the increment value results in erroneous behavior as the current coordinate value is based on a multiple of the current increment value. If the new increment value is not a divisor of the previous one, return to the initial starting point is not guaranteed, nor may it be in synchronization with the previous error variable calculation. This may cause the interpolated coordinate to spiral toward zero or out toward the arithmetic range limit. These constraints force the frequency change to occur on a new waveform period.
3.1.5 Design Synthesis

As the described design only requires simple state logic and arithmetic structures it can be easily implemented as a standard-cell ASIC. A hardware description language such as VHDL is sufficient source to provide gate synthesis for place-and-route mask layout. The final datapath for the new design is shown in figure 3.2.

3.2 Phase Accumulation Synthesizer

The next design presented is an implementation of the phase accumulator architecture. As described previously the name results from the method utilized to track the phase of the output waveform. This design is elaborated in detail as it is used as a performance reference against the new method in the following chapter.
3.2.1 ROM Implementation

Unlike the interpolation synthesizer this system requires no state-control logic and only uses one adder to perform two operand addition. It does require, as mentioned, a memory device that contributes a large amount of area and delay depending on the scale of the implementation. A NOR-type bit array, shown in figure 3.3, was used in this implementation to encode the data as it was feasible for the modeled technology in the simulations and it is optimal in terms of speed [10]. The same type of array was also used to create the corresponding row decoder. A synchronous bit-line pre-charge and evaluate structure similar to the one used by Samueli and Nicholas in [6] was used to minimize the power consumed by the ROM. Implementation in this manner avoids direct current paths and power loss is limited to static leakage and dynamic power.
3.2.2 ROM Compression

Compression of the ROM contents was performed by the method described by Samueli et al. [2]. Specifically the one quarter of the waveform is stored in coarse and fine ROM portions and thus requires reconstruction logic. The algorithm used to calculate the ROM contents for the sine waveform is given by

\[
F_{e}(\alpha, \beta) = \sin\left(\frac{\pi}{2} \left( \frac{\alpha 2^{B} + \beta}{2^{A+B}} + \frac{1}{2^{A+B+C}} \right) \right)
\]

\[
F_{f}(\alpha, \chi) = \sum_{n=0}^{N-1} \frac{1}{2^{n}} \left[ \sin\left(\frac{\pi}{2} \left( \frac{\alpha 2^{B+C} + \beta 2^{C} + \chi}{2^{A+B+C}} + \frac{1}{2^{A+B+C+1}} \right) \right) - F_{e}(\alpha, \beta) \right]
\]
where $\alpha$, $\beta$, and $\chi$ are the independently interpreted values of three phase-bus segments. Respectively, the length of each segment is denoted $A$, $B$, and $C$. This algorithm determines the fine values to minimize the mean square error of the reconstructed sine wave relative to the expected value. Coarse and fine vectors are then scaled to the appropriate magnitude and quantized for use. The same procedure is used to produce the cosine coarse and fine ROM contents, implying that the architecture requires a total of four storage devices for quadrature outputs.

Reconstruction can be partitioned into three stages: address reversal, coarse-fine addition, and two's complement negation. The first stage is an inverter array that complements the phase bits in order to read the contents of the ROMs backwards in two of the four quadrants of the sinusoid. The coarse-fine value addition is carried out by a recursive doubling adder which is the same type used in the new design. Finally, the last stage performs radix complement inversion and complements the most significant bit to adjust the value for unsigned interpretation by the following DAC.

![FIGURE 3.5: ROM Data Reconstruction Datapath](image-url)
3.2.3 Phase Accumulator Modulation

Contrasting the circle interpolator the number of cycles required to generate a full period is dependent on the width of the accumulator and independent of the size of the output word. If $N$ is the width of the accumulator word and $i = 1$ it requires $2^N$ cycles to complete a full period of the waveform. For larger values of $i$ it requires $\left\lfloor \frac{2^N}{i} \right\rfloor$ cycles to traverse the stored waveform.

To keep comparisons meaningful between the two designs the same recursive doubling adder was also used for the accumulator and reconstruction logic in this design. No modifications were necessary to the adder structure other than the appended frequency control and phase registers for accumulation.

3.3 Recursive Doubling Adder

Both the new and existing designs make use of adders in their respective structures. To optimize both designs and make comparisons relevant, the same adder structure was used for both, specifically the recursive doubling adder. This adder is a carry look-ahead structure taking advantage of parallel prefix computation for carry bit propagation. The composition of the carry prefix system is shown in figure 3.6.

Fanout from each prefix node is limited to two minimizing propagation delays due to capacitive loading and maximizing speed. Research by Shah has affirmed this adder to be optimal in speed compared to other parallel prefix adder structures [13]. In addition, work by Zimmermann has also shown this design to be easily scalable in VHDL and thus was chosen to implement in these two designs [14].
3.4 Summary of Designs

Two different DDFS implementations were presented, as well as the adder structure used in both. The first design was the new circle interpolation synthesizer, based on graphic interpolation and variable axis increment. The phase accumulator was the second design presented, based on variable phase accumulation and magnitude look-up.

Table 3.2 summarizes the two designs with respect to transistor count and area. The circle interpolation structure is shown to be optimal in transistor count, and in some cases area. This information alludes that the circle interpolator should have better performance in some of the simulated implementations.
<table>
<thead>
<tr>
<th>Bits</th>
<th>Circle Interpolator Area ($\mu m^2$)</th>
<th>Transistor Count</th>
<th>Phase Accumulator Area ($\mu m^2$)</th>
<th>Transistor Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>362675</td>
<td>2714</td>
<td>235393</td>
<td>3576</td>
</tr>
<tr>
<td>9</td>
<td>439739</td>
<td>3100</td>
<td>285406</td>
<td>4706</td>
</tr>
<tr>
<td>10</td>
<td>488302</td>
<td>3484</td>
<td>332346</td>
<td>5479</td>
</tr>
<tr>
<td>11</td>
<td>508717</td>
<td>3844</td>
<td>447353</td>
<td>8841</td>
</tr>
<tr>
<td>12</td>
<td>576163</td>
<td>4208</td>
<td>672249</td>
<td>16192</td>
</tr>
<tr>
<td>13</td>
<td>604318</td>
<td>4536</td>
<td>734703</td>
<td>17512</td>
</tr>
<tr>
<td>14</td>
<td>656100</td>
<td>4916</td>
<td>1113843</td>
<td>29069</td>
</tr>
</tbody>
</table>

TABLE 3.2: Transistor Count and Area Variations for the Two Designs
4. EXPERIMENTAL RESULTS

Three types of simulation-based analysis were performed on the designs presented in chapter 3. The critical signal path was found to determine the maximum clock frequency for each design. An evaluation of the spectral purity of each synthesizer was also computed as another figure of merit. Finally, dynamic power consumption was simulated as well.

Section 4.1 describes the variations and the different simulations that were performed. Section 4.2 presents the propagation delays for each design, section 4.3 shows the spectral content, and section 4.5 illustrates the simulated power consumption of the designs.

4.1 Simulation Variations

Each of the two designs simulations were performed for seven different bit-scales of implementation. Specifically each was implemented with approximately 9-15 bits of phase accuracy. Results listing show 8-14 bits as the designs sizes were referenced by the number of axis bits the circle interpolator used. Simulations of the two systems modulating frequency were performed only at 14 bits of circle interpolator size to estimate the peak signal performance of the two systems. The range of 20 increment words for these simulations gives an accurate performance trend.

4.2 Propagation Delay

The circle interpolator and non-memory portions of the phase accumulator were synthesized from a 0.5\( \mu m \) standard cell library. A static timing library com-
plenunetary to the cell library was used to perform static timing analysis with the Mentor Graphics SST Velocity application. In addition, both designs went through mask creation and parasitic information was extracted in order to calculate accurate cell interconnect delay. This information was back-annotated to the static timing analysis for more accurate results. A graph depicting this design flow is shown in figure 4.1. Figure 4.2 shows the critical-path propagation time for the two designs graphically. For all bit-size cases the phase accumulator clearly has a smaller critical path, and therefore a higher possible clock frequency.

The propagation path for the phase accumulator ROM was found in HSPICE by the address decoder, word-line, and output bit-line that fell in the longest active physical path in the circuit. This was necessary as it was not feasible to simulate the the large number of transistors the ROM circuit would incorporate. All transistor cells in this delay path were created through parasitic extraction of "one" and "zero" bit device layouts. Comparison of experimental data between figure 4.3 and figure 4.2 shows the ROM delay is smaller than the accumulator delay in all cases, thus determining the critical delay path.

The exponentially increasing delay of the ROM does deserve some scrutiny, however. This trend would be critical in the omission of data compression in the ROM as the read-time delay would exceed the accumulator portion's delay and be-
come the critical delay path. The delay of the ROM can be explained by examining the RC delays of the bit-lines and word-lines. Given the dimensions of a transistor ‘cell’ in the ROM and the oxide capacitance, the word-line polysilicon gate delay is estimated by

\[ C_{bit} = C_{ox} \cdot W \cdot L, \quad t_{row} = \sum_{k=1}^{N} R_{jk} C_k, \quad R_{jk} = \sum_{j=1}^{k} R_j \]

Assuming that \( R \) and \( C \) are constant through the word row-array

\[ t_{row} = \sum_{k=1}^{N} R \cdot k \cdot C \]

\[ = RC \sum_{k=1}^{N} k \]

\[ = RC \frac{N(N + 1)}{2} \approx \frac{RC}{2} N^2 \]
A similar assessment can be made of the output metal bit-lines, though the contributed capacitance will be smaller. As the length of the bit-line or word-line increases the corresponding propagation delay increases in an exponentially squared manner [15]. This justifies the use of data compression at the cost of spectral purity, in turn allowing a higher operational frequency and greater bandwidth.

4.3 Spectral Content

Analysis of the spectral content of each design was performed using the Fast-Fourier-Transform function (FFT) in the MATLAB application. These results are presented in table 4.1 and figure 4.4. The data demonstrates that the circle in-
terpolation synthesizer generally out-performs the phase accumulator in respect to peak spurious output noise away from the carrier. At the eight bit implementation, however, the two designs show comparable performance near 60 dB.

Figure 4.5 displays the relative noise power adjacent to the carrier frequency $f_c$ at offsets of 1kHz, 10kHz, and 100kHz in the frequency domain. As the sampled data is dimensionless in time the circuit critical delay path period was used to calculate the carrier frequency. At 1kHz and 10kHz the two designs differ greatly, with the phase accumulator generally out-performing the circle interpolator. Note that the “Inf” term in the lower two plots denotes a null in the spectrum at the particular frequency for the phase accumulator. For the 100kHz analysis at 14 bits of axis accuracy the phase accumulator carrier to noise value converges with the circle interpolator value. This is a result of a noise spur nearby the carrier that occurs at $f_c + 100kHz$ in the phase accumulator’s frequency spectrum.

The loss in spectral purity of the circle interpolator can be elucidated by examining the performance of the algorithm between the angles $\frac{\pi}{4}$ and $\frac{\pi}{2}$ of the interpolated arc, as illustrated by figure 4.6. As the arc is interpolated, the effective phase-angle increment is non-uniform, shown by the imposed vectors from the origin to each calculated point. Ideally the new coordinate values would be sampled at uniform increments of phase.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Circular Interpolator (dBc)</th>
<th>Phase Accumulator (dBc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>61.02</td>
<td>59.40</td>
</tr>
<tr>
<td>9</td>
<td>72.63</td>
<td>64.46</td>
</tr>
<tr>
<td>10</td>
<td>79.97</td>
<td>69.40</td>
</tr>
<tr>
<td>11</td>
<td>89.06</td>
<td>78.71</td>
</tr>
<tr>
<td>12</td>
<td>96.70</td>
<td>86.91</td>
</tr>
<tr>
<td>13</td>
<td>106.35</td>
<td>92.86</td>
</tr>
<tr>
<td>14</td>
<td>111.99</td>
<td>101.55</td>
</tr>
</tbody>
</table>

TABLE 4.1: Carrier Power Relative to Noise Floor Power
The resulting effect is a phase distortion that occurs two times per quadrant for each quadrature component sinusoid. The peak distortion difference relative to the expected ideal quantized value is given by

\[ err_{peak} = [r \cdot \cos(\phi)] - \left[ r \cdot \cos \left( r - \left[ r \cdot \cos(\phi) \right] + \left[ r \cdot \sin(\phi) \right] - \frac{\pi}{2} \right) \right] \]

\[ \phi = \frac{2\pi}{5} \quad r = 2^{N-2} - 1 \]

This distortion is periodic with near the same frequency as the desired sinusoid, resulting in the higher level of noise adjacent to the carrier in the frequency domain. The ratio of the carrier peak magnitude power to peak distortion power is plotted in figure 4.7. The convergent trend toward 30 dB is explained by considering the previous equation for large values of the radius, \( r \). The radius values in the right
cosine argument can be factored and eliminated, leaving the outside $r$ coefficients of the two cosine functions. As the entire equation is taken to be in a ratio against $r$, the outside cosine coefficients are eliminated as well. The resulting expression

$$err_{peak} = \cos(\phi) - \cos \left( \frac{1 - \cos(\phi) + \sin(\phi)}{2} \cdot \frac{\pi}{2} \right)$$

is a constant, independent of implementation radius, and depends only on the phase angle that results in its maximum magnitude. The outcome is a peak carrier to distortion value of 30.016 dB irregardless of bit precision.
4.4 Spectral Purity at Frequency Modulation

Carrier to noise floor signal power simulations were also performed at various modulation values at an implementation size of 14 bits. Figure 4.8 shows results for the two designs as they were simulated with increment values ranging from 1 to 20. The noise floor for the phase accumulator rises as the increment value changes from 1 to 8 steps, but comes back and is bounded from 102.67 dB to 98.75 dB. The circle interpolator, though, starts off with a higher value but steadily degrades as it it modulated up in frequency. The degradation of circle interpolator performance can be attributed to interpolation overshoot in conjunction with the non-uniform phase error described in section 4.3. The effect of interpolation overshoot is shown in figure 4.9. The consequence is that a maximum error of an increment step $i$ is
FIGURE 4.7: Overshoot Noise Power

FIGURE 4.8: Carrier to Noise Floor at Modulation
added to the non-uniform phase error. The total peak error at modulation is then bounded by

\[ err_{\text{peak}} = [r \cdot \cos(\phi)] - [r \cdot \cos \left( r - \frac{[r \cdot \cos(\phi)] + [r \cdot \sin(\phi)] \pi}{2r} \right) + i] \]

4.5 Power Dissipation

Power measurement in fixed voltage circuits is determined by the current draw measured over time. This measurement was facilitated by HSPICE and Mentor Graphics Eldo as well as computation of the average current value. All simulations were performed with a voltage \( V_{dd}=5V \) and nominal temperature settings. In addition, each design was simulated at the maximum clock frequency allowed by its
critical delay path. Due to the size constraints mentioned in section 4.2 the entire set of ROMs could not be simulated in their entirety. Instead average power analysis with an activity factor of $\alpha = 1$ was performed for the longest delay path of the ROM. Activity factors for other nodes in the ROM architecture were then calculated based on the known ROM contents and address decoding array. This information was used to extrapolate power consumption for the set of ROMs and was subsequently added to the simulated power of the accumulator and reconstruction logic of the phase accumulator.

Results in figure 4.10 and table 4.2 demonstrate that the phase accumulator uses more average power for all implementations. It is expected that as the size of the design increases the larger number of devices require more power dissipation. The circle interpolator does not follow this trend, however, and its rate of growth diverges from the phase accumulator. All sizes of both designs were simulated for the same constant number of clock cycles due to simulation resource constraints. As explained in section 3.1.2 the number of clock cycles required to complete a sinusoid period approximately doubles with each additional bit of axis-variable accuracy. Therefore, as the period length is extended and the simulated portion is constant, the simulated activity factor of the nodes decreases as less of the period is simulated and lowers the average power calculated.

In the phase accumulator a majority of power is lost due to the substantial bit-line and word-line capacitances of the larger ROMs. Without this component the average power used by the accumulator and reconstruction logic follows the declining trend shown by the circle interpolator. The non-monotonic peak at the ten bit implementation is caused by an increase in the width of the added fine-word inferring more gates in the reconstruction datapath. This jump in width occurs again at thirteen bits though the added power of the ROM is large enough at thirteen and fourteen bits to mask this trend.
TABLE 4.2: Dynamic Power Consumption as a Function of Design and Size

<table>
<thead>
<tr>
<th>Bits</th>
<th>Circular Interpolator (Watts)</th>
<th>Phase Accumulator (Watts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.0136</td>
<td>0.0160</td>
</tr>
<tr>
<td>9</td>
<td>0.0123</td>
<td>0.0147</td>
</tr>
<tr>
<td>10</td>
<td>0.0114</td>
<td>0.0168</td>
</tr>
<tr>
<td>11</td>
<td>0.0104</td>
<td>0.0156</td>
</tr>
<tr>
<td>12</td>
<td>0.0095</td>
<td>0.0165</td>
</tr>
<tr>
<td>13</td>
<td>0.0088</td>
<td>0.0184</td>
</tr>
<tr>
<td>14</td>
<td>0.0087</td>
<td>0.0188</td>
</tr>
</tbody>
</table>

FIGURE 4.10: Dynamic Power Dissipation
Dynamic power consumption is proportional to frequency as explained in section 2.5. Therefore, to make energy comparisons relevant the power consumption should be normalized against clock frequency resulting in the speed-power product (SPP). The SPP is a measure of circuit energy efficiency that allows direct comparison between the two designs, and is shown in figure 4.11. The phase accumulator is clearly a more efficient design up to eleven bits of relative axis accuracy. Past this size, however, the performance of the circle interpolator dominates differing from the circle interpolator at a maximum of 141.40 pJ at fourteen bits. The exchange of performance at larger implementations can be attributed to the increased power consumption of the ROM in the phase accumulator. Due to increasing bit-line and word-line capacitances as the number of data-bits increases the efficiency drops respectively.

4.6 Experimental Results Summary

In this chapter, the variations in simulation performed on the designs in the previous chapter have been explained. Section 4.2 presented the experimental data for propagation delay. For all sizes of the designs the circle interpolator was the slowest. Section 4.3 conferred the simulation information for the spectral content for each design. The circle interpolator was found to have the greater peak carrier to noise ratio for all sizes. However, in section 4.3 it was shown that the phase accumulator does not suffer from overshoot distortion as the circle interpolator does at high frequencies. Finally, in section 4.5 the power dissipation for each design was shown. For all sizes the phase accumulator was shown to consume the most average power when operated at its maximum clock frequency, though the circle interpolator showed better energy efficiency at larger implementations.
FIGURE 4.11: Speed Power Product
5. CONCLUSIONS

Conclusions based on the comparison of the new design simulation results to those of the existing method are presented. Optimal characteristics for both designs are reviewed. In addition, suggestions for future work are described as well.

5.1 Summary

A new design for digital frequency synthesis was presented, as well as a review and analysis of the design currently in use. Both designs were simulated and compared under varying conditions of size with regard to spectral purity, critical path delay, and power consumption.

The phase accumulator was consistently optimal in critical-path delay. However, the circle interpolator demonstrated better carrier to out-of-band noise power for the majority of the implementations. Also, at implementations of twelve bits and greater the circle interpolator shows better energy efficiency. Finally, the phase accumulator requires more transistors for all implementations while the new design is more efficient in respect to area for implementations greater than eleven bits.

Reviewing results for the various parameters, the new design does prove to be optimal for certain criteria. If carrier power to spurious out-of-band noise, size, and transistor count are critical required features, the new design is best for larger output words. For speed and high efficiency at small bit-accuracy the phase accumulator should be considered.
5.2 Future Research

The approach of discrete value interpolation can be applied to other forms of signal synthesis. This would be especially advantageous for complex signals that don’t benefit from ROM compression in phase accumulator architectures. In addition, alternative logic circuit implementations should be considered other than the standard-cell static CMOS gates used in this study. Alternate low power logic circuit formulations may reveal a savings in overall power consumption with regard to this implementation of the circle interpolator. Finally, $V_{dd}$ modulation in conjunction with threshold voltage modification should be considered in future studies.
REFERENCES


APPENDICES
A Circle Interpolator VHDL Design

Main Control Structure

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY arith_lib;
USE arith_lib.arith_utils.all;

ENTITY sin_gen IS
  GENERIC(width : positive := 8);
  PORT (clk,rst : IN std_logic;
        x_out,y_out : OUT std_logic_vector(width-2 DOWNTO 0);
        inc : IN std_logic_vector(width-3 DOWNTO 0));
END sin_gen;

ARCHITECTURE rtl OF sin_gen IS
  SIGNAL x, x_next, y, y_next, axis_new, axis,
       axis_alt, f_axis_cmp, f_axis :
       std_logic_vector(width-1 DOWNTO 0);
  SIGNAL f, f_next, f_err :
       std_logic_vector(width DOWNTO 0);
  SIGNAL inc_clkd, inc_muxd, inc_neg :
       std_logic_vector(width-3 DOWNTO 0); 
  SIGNAL sub_add, sign_ext :
       std_logic;

  COMPONENT fast_add is
    GENERIC (width : POSITIVE);
    PORT (A, B IN std_logic_vector(width-1 DOWNTO 0);
          CI IN std_logic;
          S OUT std_logic_vector(width-1 DOWNTO 0);
          CO OUT std_logic);
END COMPONENT fast_add;

  COMPONENT fast_comp is
    GENERIC (width : POSITIVE);
    PORT (WORD IN std_logic_vector(width-1 DOWNTO 0);
          CMP : OUT std_logic_vector(width-1 DOWNTO 0);
          CO : OUT std_logic;
          SEL : IN std_logic);
END COMPONENT fast_comp;

  COMPONENT multi_sum is
    GENERIC (width : POSITIVE);
    PORT (F IN std_logic_vector(width DOWNTO 0);
          AXIS_CMP IN std_logic_vector(width-1 DOWNTO 0);
          INC IN std_logic_vector(width-3 DOWNTO 0);
          S OUT std_logic_vector(width DOWNTO 0);
          CO : OUT std_logic);
END COMPONENT multi_sum;

BEGIN

  inc_comp: COMPONENT fast_comp 
             GENERIC MAP (width=>width-2) 
             PORT MAP (WORD => inc_muxd, 
                        CMP => inc_neg, 
                        CO => sign_ext, 
                        SEL => '1');

  RC_neg: COMPONENT fast_comp 
          GENERIC MAP (width => width) 
          PORT MAP (WORD => f_axis, 
                     CMP => f_axis_cmp, 
                     CO => OPEN, 
                     SEL => sub_add); --SEL is active high to negate

  axis_adder : COMPONENT fast_add 
               GENERIC MAP(width => width)
PORT MAP (A => axis_alt,  
B => axis,  
S => axis_new,  
CI => '0',  
CO => OPEN);

f_adder: component multi_sum
GENERIC MAP (width => width)
PORT MAP (F => f,  
AXIS_CMP => f_axis_cmp,  
INC => inc_muxd,  
S => f_err,  
CO => OPEN);

control:PROCESS(x,y,f,inc_neg,inc,inc_clkd,inc_muxd,f_err,axis_new,rst,sign_ext,clk) 
BEGIN

  y_max_chk := y(width-l) & y(width-2);--concatenates the bits together
  x_max_chk := x(width-l) & x(width-2);
  FOR i IN 1 TO width-2 LOOP
    const_ref(i-l) := '1';
    END LOOP;

  CASE y_max_chk IS
    WHEN "01" => y_out <= '1' & const_ref;
    WHEN "10" => y_out <= '0' & NOT(const_ref(width-3 DOWNTO 1)) & '1';
    WHEN "00" | "11" => y_out <= NOT(y(width-2)) & y(width-3 DOWNTO 0);
    WHEN OTHERS => y_out <= NOT(y(width-2)) & y(width-3 DOWNTO 0);
    END CASE;

  CASE x_max_chk IS
    WHEN "01" => x_out <= '1' & const_ref;
    WHEN "10" => x_out <= '0' & NOT(const_ref(width-3 DOWNTO 1)) & '1';
    WHEN "00" | "11" => x_out <= NOT(x(width-2)) & x(width-3 DOWNTO 0);
    WHEN OTHERS => x_out <= NOT(x(width-2)) & x(width-3 DOWNTO 0);
    END CASE;

  CASE sign_bits IS
    WHEN "010" | "111" => axis <= y;
    WHEN "010" => axis <= y;
    WHEN "00" | "10" => axis <= x;
    WHEN OTHERS => axis <= x;
  END CASE;
axis <= x;
axis_alt <= "00" & inc_muxd;
y_next <= y;
f_next <= f_err;
inc_muxd <= inc_clkd;
f_axis <= x;
sub_add <= '0';

WHEN "100" | "001" => x_next <= x;
axis <= y;
axis_alt <= "00" & inc_muxd;
f_axis <= y;
sub_add <= '0';
y_next <= axis_new;
IF (y = "00" & NOT(const_ref) AND x = "00" & const_ref) THEN
  inc_muxd <= inc;
  IF (sign_ext='l') THEN
    f_next <= "000" & NOT(const_ref);
  ELSE
    f_next <= f_err;
  END IF;
ELSE
  f_next <= f_err;
  inc_muxd <= inc_clkd;
END IF;

WHEN OTHERS =>
f_next <= '0' & '0' & '0' & NOT(const_ref);
y_next <= "00" & NOT(const_ref);
x_next <= "00" & const_ref;

END CASE;

IF (rst = '1') THEN
  f <= '0' & '0' & '0' & NOT(const_ref);
x <= "00" & const_ref;
y <= "00" & NOT(const_ref); -- Y axis value
  inc_clkd <= NOT(const_ref(width-4 DOWNTO 0)) &'1';
y_out <= '0' & NOT(const_ref);
x_out <= '0' & const_ref;
ELSIF (clk'EVENT AND clk ='1' AND clk'last_value = '0') THEN
  f <= f_next;
x <= x_next;
y <= y_next;
  inc_clkd <= inc_muxd;
END IF;

END PROCESS control;

END rtl;

Multiple Input Sum Structure

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
LIBRARY arith_lib;
-----------------------------------------------------------------------
ENTITY multi_sum IS
  GENERIC (width : POSITIVE);
  PORT (  
    F : IN std_logic_vector(width DOWNTO 0);
    AXIS_CMP : IN std_logic_vector(width-1 DOWNTO 0);
    INC : IN std_logic_vector(width-3 DOWNTO 0);
    S : OUT std_logic_vector(width DOWNTO 0);
    CO : OUT std_logic);
  END multi_sum;
-----------------------------------------------------------------------
ARCHITECTURE Structural OF multi_sum IS
SIGNAL SUM_RDNT_c : std_logic_vector(width-1 DOWNTO 0);
SIGNAL SUM_RDNT_ps : std_logic_vector(width-1 DOWNTO 0);

COMPONENT input_compress IS
  GENERIC (width : POSITIVE);
  PORT (AXIS_CMP : IN std_logic_vector(width-1 DOWNTO 0);
         F : IN std_logic_vector(width DOWNTO 0);
         INC : IN std_logic_vector(width-3 DOWNTO 0);
         SUM_RDNT_c : OUT std_logic_vector(width-1 DOWNTO 0);
         SUM_RDNT_ps : OUT std_logic_vector(width-1 DOWNTO 0);
         SUM_LSB : OUT std_logic);
END COMPONENT input_compress;

COMPONENT fast_add IS
  GENERIC (width : positive);
  port (A, B : IN std_logic_vector(width-1 DOWNTO 0);
        CI : IN std_logic;
        S : OUT std_logic_vector(width-1 DOWNTO 0);
        CO : OUT std_logic);
END COMPONENT fast_add;

BEGIN
  mult_input: COMPONENT input_compress
  GENERIC MAP (width => width) 
  PORT MAP (AXIS_CMP => AXIS_CMP,
         F => F,
         INC => INC,
         SUM_RDNT_c => SUM_RDNT_c,
         SUM_RDNT_ps => SUM_RDNT_ps,
         SUM_LSB => SCO));

  f_adder: COMPONENT fast_add 
  GENERIC MAP (width => width) 
  PORT MAP (A => SUM_RDNT_c,
         B => SUM_RDNT_ps,
         S => S(width DOWNTO 1),
         CI => '0',
         CO => CO);

END Structural;

Axis Adder Structure

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
LIBRARY arith_lib;

ENTITY fast_add IS
  GENERIC (width : POSITIVE := 5);
  PORT (A, B : IN std_logic_vector(width-1 DOWNTO 0);
         CI : IN std_logic;
         S : OUT std_logic_vector(width-1 DOWNTO 0);
         CO : OUT std_logic);
END fast_add;

ARCHITECTURE Structural OF fast_add IS
  SIGNAL GI_add, PI_add : std_logic_vector(width-1 DOWNTO 0);
  SIGNAL GO_add : std_logic_vector(width DOWNTO 0);
SIGNAL PT_add : std_logic_vector(width-1 DOWNTO 0);

COMPONENT Prefix_Kogge_Stone IS
GENERIC(width : POSITIVE);
PORT (GI, PI : in std_logic_vector(width-1 DOWNTO 0);
      GO : out std_logic_vector(width DOWNTO 0);
      CIN : in std_logic);
END COMPONENT Prefix_Kogge_Stone;

BEGIN
  -- calculate prefix input generate/propagate signals
  GI_add <= A AND B;
  PI_add <= A OR B;
  -- calculate adder propagate signals (PT = A XOR B)
  PT_add <= NOT GI_add AND PI_add;
  -- calculate prefix output generate/propagate signals with fast carry-in
  prefix : Prefix_Kogge_Stone
  GENERIC map Cwidth => width)
  PORT MAP CGI => GI_add,
     PI => PI_add,
     CIN => CI,
     GO => GO_add);
  -- calculate sum and carry-out bits
  S <= PT_add XOR GO_add(width-1 DOWNTO 0);
  CO <= GO_add(width);
END Structural;

Operand Negation Structure

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
LIBRARY arith_lib;
USE arith_lib.arith_utils.all;

ENTITY fast_comp IS
GENERIC Cwidth : POSITIVE := 5);
PORT CWORD IN std_logic_vector(width-1 DOWNTO 0);
CMP OUT std_logic_vector(width-1 DOWNTO 0);
CO OUT std_logic;
SEL IN std_logic);
END fast_comp;

ARCHITECTURE Structural OF fast_comp IS

SIGNAL INV_WORD: std_logic_vector(width-1 DOWNTO 0); -- prefix gen./prop. in
SIGNAL CMP_GEN: std_logic_vector(width-1 DOWNTO 0); -- prefix gen./prop. in

COMPONENT Prefix_KS_comp IS
GENERIC(width : POSITIVE);
port (WORD : IN std_logic_vector(width-1 DOWNTO 0)); -- gen./prop. in
      GO : OUT std_logic_vector(width-1 DOWNTO 0); -- gen out
      CO : OUT std_logic;
      SEL : IN std_logic);
END COMPONENT Prefix_KS_comp;

BEGIN
  -- calculate prefix input generate/propagate signals
negate:PROCESS(SEL,WORD)
BEGIN
   IF SEL='1' THEN
      INV_WORD <= NOT (WORD);
   ELSIF SEL='0' THEN
      INV_WORD <= WORD;
   end IF;
END PROCESS NEGATE;
--calculate prefix output generate signals with fast carry-in
prefix : Prefix_KS_comp
GENERIC MAP (width => width)
PORT MAP (WORD => INV_WORD,
   SEL => SEL,
   CO => CO,
   GO => CMP_GEN);
--xor it with the carry-propagated word
CMP <= INV_WORD XOR CMP_GEN;
END Structural;

Carry-Save Operand Compressor

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
LIBRARY arith_lib;
USE arith_lib.arith_utils.all;
------------------------------------------------------------------------
ENTITY input_compress IS
GENERIC 	(width positive:= 8);
PORT (AXIS_CMP : in std_logic_vector(width-l DOWNTO 0);
F : in std_logic_vector(width downto 0);
INC: in std_logic_vector(width-3 downto 0);
SUM_RDNT_ps: out std_logic_vector(width-l downto 0);
SUM_RDNT_c 	: out std_logic_vector(width-l downto 0);
SUM_LSB: OUT std_logic);
END input_compress;
------------------------------------------------------------------------
ARCHITECTURE Structural of input_compress IS
BEGIN -- Carry-save 3 operand input compressor recodes
   -- as carry and pseudo-sum for adder reconstruction
   SUM_LSB <= INC(0) XOR F(0);
c(0) <= INC(0) AND F(0);
in_array1: FOR i IN 1 TO width-3 GENERATE
   ps(i-1) <= INC(i) XOR F(i) XOR AXIS_CMP(i-1);
   c(i) <= (AXIS_CMP(i-1) AND F(i)) OR (AXIS_CMP(i-1) AND INC(i)) OR (F(i) AND INC(i));
END GENERATE in_array1;
in_array2: FOR i IN width-2 TO width GENERATE
   ps(i-1) <= AXIS_CMP(i-1) XOR F(i);
   c(i) <= AXIS_CMP(i-1) AND F(i);
END GENERATE in_array2;
SUM_RDNT_ps <= ps;
SUM_RDNT_c <= c(width-1 downto 0);
END Structural;
Negation Prefix Structure

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY arith_lib;
USE arith_lib.arith_utils.all;
-----------------------------------------------------------------------------
ENTITY Prefix_KS_comp IS
    GENERIC (width : POSITIVE := 8);
    PORT (WORD : IN std_logic_vector(width-1 downto 0); -- gen./prop. in --
           GO : OUT std_logic_vector(width-1 downto 0); -- gen out
           CO : OUT std_logic;
           SEL : IN std_logic);
END Prefix_KS_comp;
-----------------------------------------------------------------------------
ARCHITECTURE Structural OF Prefix_KS_comp IS
    CONSTANT n : POSITIVE := width; -- prefix structure width
    CONSTANT m : POSITIVE := log2ceil(width+1); -- prefix structure depth
    BEGIN
    -- Kogge Stone parallel-prefix carry-lookahead structure; quick RC complement
    local : BLOCK
        SIGNAL GT : std_logic_vector((n+1)*(m+1))-1 DOWNTO 0);
        BEGIN
        GT(0) <= SEL;
        GT(n DOWNTO 1) <= WORD;
        levels : FOR 1 IN 1 TO m GENERATE
            groups : FOR k IN 0 TO n GENERATE
                white : IF k < (2**(1-1)) GENERATE
                    GT((1*(n+1)+k) <= GT((1-1)*(n+1)+k); -- pass down row above
                END GENERATE white;
                black : IF (2**(1-1) <= k) GENERATE
                    GT((1*(n+1)+k) <=
                    GT((1-1)*(n+1)+k) AND
                    GT(((1-1)*(n+1)) *k - 2**(1-1));
                END GENERATE black;
            END GENERATE groups;
        END GENERATE levels;
        GO <= GT(((m+1)*(n+1))-2 DOWNTO (m)*(n+1));
        CO <= GT(((m+1)*(n+1))-1);
        END BLOCK local;
    END Structural;

Adder Prefix Structure

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY arith_lib;
USE arith_lib.arith_utils.all;
-----------------------------------------------------------------------------
ENTITY Prefix_Kogge_Stone IS
    GENERIC (width : POSITIVE := 8);
    PORT (GI, PI : IN std_logic_vector(width-1 DOWNTO 0);
          GO : OUT std_logic_vector(width DOWNTO 0);
          CIN : IN std_logic);
ARCHITECTURE Structural OF Prefix_Kogge_Stone IS

CONSTANT n : POSITIVE := width; -- prefix structure width
CONSTANT m : POSITIVE := log2ceil(width+1); -- prefix structure depth; adj for carry in

BEGIN

-- Kogge Stone parallel-prefix carry-lookahead structure
local : block
SIGNAL GT : std_logic_vector(((n+1)*(m+1))-1 DOWNTO 0);
SIGNAL PT : std_logic_vector(((m)*(n+1)-2**(m)+1)-1 DOWNTO 0);

BEGIN
GT(0) <= CIN;
GT(n DOWNTO 1) <= GI;
PT(n-1 DOWNTO 0) <= PI;
levels : FOR 1 IN 1 TO m GENERATE
groups : FOR k IN 0 TO n GENERATE
white : IF k < (2**(1-1)) GENERATE
GT((1*(n+1))+k) <= GT((1-1)*(n+1)+k); -- pass g from above row
END GENERATE white;
grey: IF (k >= 2**(1-1)) AND (k < 2**1) GENERATE
GT((1*(n+1))+k) <= 
GT(((1-1)*(n+1))+k) OR 
(PT(((1-1)*(n+1)) +k - 2**(1+1) AND 
GT(((1-1)*(n+1)) +k - 2**(1-1))); 
END GENERATE grey;
black : IF (2**1 <= k) GENERATE 
GT((1*(n+1))+k) <= 
GT(((1-1)*(n+1))+k) OR 
(PT(((1-1)*(n+1)) +k - 2**(1+1) AND 
GT(((1-1)*(n+1)) +k - 2**(1-1)));
END GENERATE black;
END GENERATE groups;
END GENERATE levels;
GO <= GT(((m+1)*(n+1))-1 DOWNTO (m)*(n+1));
END block local;
END Structural;

Scaling Arithmetic Utilities

library ieee;
use ieee.std_logic_1164.all;

package arith_utils is

function log2ceil (n : integer) return integer;
function log2floor (n : integer) return integer;
end arith_utils;

package body arith_utils is

-- purpose: computes ceil(log2(n))
function log2ceil (n : integer) return integer is
variable m, p : integer;
begin
m := 0;
p := 1;
for i in 0 to n loop
if \( p < n \) then
  \( m := m + 1; \)
  \( p := p \times 2; \)
end if;
end loop;
return m;
end log2ceil;

-- purpose: computes floor(log2(n))
function log2floor (n : integer) return integer is
  variable m, p : integer;
begin
  m := -1;
  p := 1;
  for i in 0 to n loop
    if \( p \leq n \) then
      m := m + 1;
      p := p \times 2;
    end if;
  end loop;
  return m;
end log2floor;
end arith_utils;
B Phase Accumulator VHDL Design

Main Interface Structure

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
LIBRARY arith_lib;
-----------------------------------------------------------------------------
ENTITY phs_acm IS
GENERIC (width : POSITIVE:=8;
          fr_width : POSITIVE:=2);
PORT (CLK : IN std_logic;
      RST : IN std_logic;
      FCW : IN std_logic_vector(width-1 DOWNTO 0);
      SIN_OUT : OUT std_logic_vector(width-3 DOWNTO 0);
      COS_OUT : OUT std_logic_vector(width-3 DOWNTO 0));
END phs_acm;
-----------------------------------------------------------------------------
ARCHITECTURE Structural OF phs_acm IS
SIGNAL ROM_ADDR_C, ROM_ADDR_F : std_logic_vector(width-2-((width-2)/3)-1 DOWNTO 0);
SIGNAL ROM_SIN_FINE : std_logic_vector(fr_width-1 DOWNTO 0);
SIGNAL ROM_COS_FINE : std_logic_vector(width-4 DOWNTO 0);
SIGNAL ROM_SIN_COARSE : std_logic_vector(width-4 DOWNTO 0);
SIGNAL ROM_COS_COARSE : std_logic_vector(width-4 DOWNTO 0);
COMPONENT phs_acm_strct IS
GENERIC (width : POSITIVE;
          fr_width : POSITIVE);
PORT (CLK : IN std_logic;
      RST : IN std_logic;
      FCW : IN std_logic_vector(width-1 DOWNTO 0);
      ROM_ADDR_F : OUT std_logic_vector(width-2-((width-2)/3)-1 DOWNTO 0);
      ROM_ADDR_C : OUT std_logic_vector(width-2-((width-2)/3)-1 DOWNTO 0);
      ROM_SIN_FINE : IN std_logic_vector(fr_width-1 DOWNTO 0);
      ROM_COS_FINE : IN std_logic_vector(width-4 DOWNTO 0);
      ROM_SIN_COARSE : IN std_logic_vector(width-4 DOWNTO 0);
      ROM_COS_COARSE : IN std_logic_vector(width-4 DOWNTO 0);
      SIN_OUT : OUT std_logic_vector(width-3 DOWNTO 0);
      COS_OUT : OUT std_logic_vector(width-3 DOWNTO 0));
END COMPONENT phs_acm_strct;
COMPONENT coarse_sin_rom IS
    PORT (addr: IN STD_LOGIC_VECTOR(3 downto 0);
          value: OUT STD_LOGIC_VECTOR(fr_width-1 DOWNTO 0));
END COMPONENT coarse_sin_rom;
COMPONENT coarse_cos_rom IS
    PORT (addr: IN STD_LOGIC_VECTOR(3 downto 0);
          value: OUT STD_LOGIC_VECTOR(width-4 DOWNTO 0));
END COMPONENT coarse_cos_rom;
COMPONENT fine_sin_rom IS
    PORT (addr: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
          value: OUT STD_LOGIC_VECTOR(fr_width-1 DOWNTO 0));
END COMPONENT fine_sin_rom;
COMPONENT fine_cos_rom IS
    PORT (addr: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
          value: OUT STD_LOGIC_VECTOR(width-4 DOWNTO 0));
END COMPONENT fine_cos_rom;
BEGIN
Accumulator and ROM Interface

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
LIBRARY arith_lib;

ENTITY phs_acm_strct IS
  GENERIC (width : POSITIVE:=8;
            fr_width : POSITIVE:=2);
  PORT (CLK : IN std_logic;
        RST : IN std_logic;
        FCW : IN std_logic;
        ROM_ADDR_F : OUT std_logic_vector(width-1 DOWNTO 0);
        ROM_ADDR_C : OUT std_logic_vector(width-1 DOWNTO 0);
        ROM_SIN_FINE : IN std_logic_vector(fr_width-1 DOWNTO 0);
        ROM_COS_FINE : IN std_logic_vector(width-4 DOWNTO 0);
        ROM_SIN_COARSE : IN std_logic_vector(width-4 DOWNTO 0);
        ROM_COS_COARSE : IN std_logic_vector(width-4 DOWNTO 0);
        ROM_ADDR_F_INV : OUT std_logic_vector(width-1 DOWNTO 0);
        ROM_ADDR_C_INV : OUT std_logic_vector(width-1 DOWNTO 0);
        ROM_SIN_FINE_INV : IN std_logic_vector(fr_width-1 DOWNTO 0);
        ROM_COS_FINE_INV : IN std_logic_vector(width-4 DOWNTO 0);
        ROM_SIN_COARSE_INV : IN std_logic_vector(width-4 DOWNTO 0);
        ROM_COS_COARSE_INV : IN std_logic_vector(width-4 DOWNTO 0);
        SIN_OUT : OUT std_logic_vector(width-3 DOWNTO 0);
        COS_OUT : OUT std_logic_vector(width-3 DOWNTO 0);
);
SIGNAL SIN_RC, SIN, COS_RC, COS : std_logic_vector(width-4 DOWNTO 0);
SIGNAL ROM_SIN_FINE_EXT : std_logic_vector(width-4 DOWNTO 0);
SIGNAL RC_SIN_ZRo, RC_CoS_ZRO, INV_COS_FLG, COS_SGN : std_logic;

COMPONENT accumulator IS
  GENERIC (width : POSITIVE);
  PORT (INC IN std_logic_vector(width-1 DOWNTO 0);
        CLK IN std_logic;
        RST IN std_logic;
        S OUT std_logic_vector(width-1 DOWNTO 0));
END COMPONENT accumulator;

COMPONENT fast_comp IS
  GENERIC (width : POSITIVE);
  PORT (WORD : IN std_logic_vector(width-1 DOWNTO 0);
        CMP OUT std_logic_vector(width-1 DOWNTO 0);
        CO : OUT std_logic;
        SEL : IN std_logic);
END COMPONENT fast_comp;

COMPONENT fast_add IS
  GENERIC (width : POSITIVE);
  PORT (A, B : IN std_logic_vector(width-1 DOWNTO 0);
        CI : IN std_logic;
        S OUT std_logic_vector(width-1 DOWNTO 0);
        CO : OUT std_logic);
END COMPONENT fast_add;

BEGIN
  A <= PHS_INV(width-3 DOWNTO (width-2)/3+(width-2)/3);
  B <= PHS_INV((width-2)/3+(width-2)/3-1 DOWNTO (width-2)/3);
  C <= PHS_INV((width-2)/3-1 DOWNTO 0);
  ROM_ADDR_C <= A & B;
  ROM_ADDR_F <= A & C;
  SIN_OUT <= (NOT(PHS(width-1)) OR RC_SIN_ZRO) & SIN;
  COS_OUT <= (PHS(width-1) XOR PHS(width-2)) OR RC_COS_ZRO) & COS;
  INV_COS_FLG <= PHS(width-1) XOR PHS(width-2);

  f_accum: COMPONENT accumulator
    GENERIC MAP (width => width)
    PORT MAP (INC => FCW,
              CLK => CLK,
              RST => RST,
              S => PHS);

  f_add_sin: COMPONENT fast_add
    GENERIC MAP (width => width-3)
    PORT MAP (A => ROM_SIN_FINE_EXT,
              B => ROM_SIN_COARSE,
              CI => '0',
              S => SIN_RC,
              CO => OPEN);

  f_add_cos: COMPONENT fast_add
    GENERIC MAP (width => width-3)
    PORT MAP (A => ROM_COS_FINE,
              B => ROM_COS_COARSE,
              CI => '0',
              S => COS_RC,
              CO => OPEN);

  f_comp_sin: COMPONENT fast_comp
    GENERIC MAP (width => width-3)
    PORT MAP (WORD => SIN_RC,
              CMP => SIN,
CO => RC_SIN_ZRO,
SEL => PHS(width-1));

f_comp_cos: COMPONENT fast_comp
GENERIC MAP (width => width-3)
PORT MAP (WORD => COS_RC,
CMP => COS,
CO => RC_COS_ZRO,
SEL => INVCOS_FLG);

extend:PROCESS(ROM_SIN_FINE, ROM_COS_FINE)
VARIABLE const_ref: std_logic_vector(width-fr_width-4 DOWNTO 0);
BEGIN
  FOR i IN 1 TO width-3-fr_width LOOP
    const_ref(i-1) := '0';
  END LOOP;
  ROM_SIN_FINE_EXT <= const_ref & ROM_SIN_FINE;
END PROCESS extend;

decode:PROCESS(PHS,PHS_INV)
BEGIN
  IF (PHS(width-2)='1') THEN
    PHS_INV <= NOT (PHS(width-3 DOWNTO 0));
  ELSE
    PHS_INV <= PHS(width-3 DOWNTO 0);
  END IF;
END PROCESS decode;
END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
LIBRARY arith_lib;

ENTITY accumulator IS
GENERIC (width POSITIVE:=8);
PORT (INC IN std_logic_vector(width-1 DOWNTO 0);
  CLK IN std_logic;
  RST IN std_logic;
  S OUT std_logic_vector(width-1 DOWNTO 0));
END accumulator;

ARCHITECTURE Structural OF accumulator IS

SIGNAL INC_CLKD : std_logic_vector(width-1 DOWNTO 0);
SIGNAL SUM_CLKD, SUM_TMP : std_logic_vector(width-1 DOWNTO 0);

COMPONENT fast_add IS
GENERIC (width: POSITIVE);
PORT (A, B : IN std_logic_vector(width-1 DOWNTO 0);
  CI : IN std_logic;
  S : OUT std_logic_vector(width-1 DOWNTO 0);
  CO : OUT std_logic);
END COMPONENT fast_add;

BEGIN

  f_adder: COMPONENT fast_add
GENERIC MAP (width => width)
PORT MAP (A => INC_CLKD,
  B => SUM_CLKD,
  S => SUM_TMP,
  CI => '0',';
CO => OPEN);

S <= SUM_CLKD;

control:PROCESS(INC,INC_CLKD,SUM_TMP,SUM_CLKD,RST,CLK)
VARIABLE const_ref: std_logic_vector(width-1 DOWNT0 0);
BEGIN

FOR i IN 1 TO width LOOP
  const_ref(i-1) := '0';
END LOOP;

IF (RST = '1') THEN
  SUM_CLKD <= const_ref;
  INC_CLKD <= const_ref;
ELSIF (CLK'EVENT AND CLK = '1' AND CLK'last_value = '0') THEN
  SUM_CLKD <= SUM_TMP;
  INC_CLKD <= INC;
END IF;

END PROCESS control;

END Structural;
#!/usr/local/bin/perl

#/usr/bin/perl
#Creates longest active path in SPICE from VHDL ROM code
#for read-time analysis, note time input is in ns
#Ryan Larson

#note:
col_bottom col_top gnd row_l row_r

open(INPUT, "./test_rom.vhd") || die("could n't open VHDL file ");
open(SPICE, ">test.sp") || die("could n't open output file ");

$inst=0;
$word=0;
$col=0;
$row=0;
$wrd_sp=16; # was 8, should control the # of output lines
$wrd_sp=~ARGV[0]; # get from command-line
$clktim=ARGV[2];
$vdd=ARGV[1];
$numargs=~ARGV;
$last_one=0;

printf SPICE ("**ROM test file

OO");

while ($line ne "") {
    if ($line =~ /<= 00[01]+/){
        $l = ~[01]+/; # have bit string
        $wrd_sz=length($l); # figures out size of word...this is redundant later
        if (($col+$wrd_sz*$i==$last_one) || ($row==1) || ($col==0)) {printf SPICE ("** New word Found
 OO");}
        if ($row==2) & ($col==0) {printf SPICE ("OO");}
        $line=$l;
        for ($i=0; $i<=$wrd_sz; $i++) {
            $line =~ /<= [01]+/;
            if ($row==1) || ($col+$wrd_sp*$i==$last_one) # need to tag this col value for later
            { # attempt to tag last 1 bit in row 0
                printf SPICE ("OOXY.d colY.d_Y.d col%d_Y.d 0 rowY.d_Y.d rowY.d_Y.d arrayl
 oo");
                $inst++;
                $col+$wrd_sp*$i,
                $row+1,$col+$wrd_sp*$i,$row,$row,$col+$wrd_sp*$i,$row,$row,$col+1+$wrd_sp*$i);
            }
            elsif (substr($line,$i,l) eq "0")
            { # attempt to tag last 1 bit in row 0
                printf SPICE ("OOXY.d colY.d_Y.d col%d_Y.d 0 rowY.d_Y.d rowY.d_Y.d arrayO
 oo");
                $inst++;
                $col+$wrd_sp*$i,
                $row+1,$col+$wrd_sp*$i,$row,$row,$col+$wrd_sp*$i,$row,$row,$col+1+$wrd_sp*$i);
            }
        }
        if ($row==0) {
            $last_one=$col+$wrd_sp*$i;
        }
    }
    else
    { # attempt to tag last 1 bit in row 0
        $last_one=$col+$wrd_sp*$i;
    }
    $line=<INPUT>;
}

printf SPICE ("**ROM test file

OO"); # last col (was 1)

close(INPUT); # OK, data matrix built, now time to write word-row decoder

print("$last_one
 OO"); # last col (was 1)
```plaintext
$num_row=$row;
print("rows = "$num_row "n");
printf SPICE ("n**Defining Row Decoder **n");
$m_inst=0;
for($row=0;$row<$num_row;$row++)
{
 printf SPICE ("**New addr word**n");
 for($col=0;2**($col)<$num_row;$col++)
 { 
 if($row < 1) || (2**($col+1)<$num_row)) # need to know if on outside address line
 { 
 $t=$row>>$col; $t&=1; #extract bits from row value
 printf SPICE ("X$4 rowD ADDR rowD ADDR 0 DLXD_i%D DLXD_i%D arrayD%D"n",$inst++,$row,
 2*$col,$row,2*$col+1,$row,1,$row,$row,$t);
 printf SPICE ("X$4 rowD ADDR rowD ADDR 0 DLXD ni%D DLXD ni%D arrayD%D"n",$inst++,$row,
 2*$col+1,$row,2*$col+2,$row,1,$row,$row,$t);
 if((($row < 1) && (2**($col+1)<$num_row)) # avoid case where 2^n exceeds # rows
 { 
 printf SPICE ("RDLY_i%D vdd DLXD_i%D 1E-05"n",$col,$row);
 printf SPICE ("RDLY ni%D 0 DLXD ni%D 1E-05"n",$col,$row);
 }
 }
}
printf SPICE ("n**Define grounding row resistors**n");
for($t=1;$t<$num_row;$t++)
{
 printf SPICE ("N_row=ADDR rowD ADDR 0 1E-05"n",$t,$t,$last_one); # test case delay path-tie to ground
}
printf SPICE ("n**Defining col prechg**n");
for($i=0; $i<$lolrd_sz*$lolrd_spc; $i++)
{
 printf SPICE ("m%D col%D O clkb vdd vdd P 1=0.6u lol=1.5u ad=1.35p as=1.35p n",$m_inst++,$i);
}
printf SPICE ("n**Define Row Pre-chg**n");
printf SPICE ("n**Define Wordline Driver**n");
printf SPICE ("n**Define address complementary buffer**n");
printf SPICE ("X$4 a_D ADDR DLXD ni%D DLXD ni%D arrayD%D 0 vbufc c.buf"n",$inst++,$col-1,$col-1,
 $num_row,$col-1,$num_row);
printf SPICE (.include c.buf.spn");
printf SPICE (.include row.buf.spn");
printf SPICE (.include array0.spn");
printf SPICE (.include array1.spn");
printf SPICE (.LIB '/nfs/tblsk/ai/mg/libraries/adk_lib/technology/accsim/ami05.mod' NOM"n");
printf SPICE ("OPTION ingold=1 numdte=4 co=132 post brief"n");
printf SPICE ("Vdd vdd 0 DC=X1.1f"n",$vdd);
printf SPICE ("Vbufc vbufc 0 DC=X1.1f"n",$vdd);
printf SPICE ("Vbufc vbufc 0 DC=X1.1f"n",$vdd);
if ($numargs==2) # test for col chrg time
{
 printf SPICE ("Vp1s a_D 0 PWL 0 X1.1f 0.1ns 0\n",$col-1,$vdd);
 printf SPICE ("Vclk clk 0 PWL 0 0 2ns 0 2.1ns X1.1f 4ns X1.1f \n",$vdd,$vdd);
 printf SPICE ("Vclk clk 0 PWL 0 0 2ns X1.1f 2.1ns 0 2.1ns 0 \n",$vdd,$vdd);
 printf SPICE (".tran .25ns 60ns\n")
}
elsif ($numargs==3) #adj for chrg and find fall time
{
 printf SPICE ("Vp1s a_D 0 PWL 0 X1.1f 0.1ns 0\n",$col-1,$vdd);
 printf SPICE ("Vclk clk 0 PWL 0 0 2ns 0 2.1ns X1.1f X3.3fns X1.1f X3.3fns 0\n",$vdd,
 $clktime+2.1,$vdd,$clktime+2.1);
 printf SPICE ("Vclk clk 0 PWL 0 0 2ns X1.1f 2.1ns 0 X3.3fns 0 X3.3fns X1.1f\n",$vdd,$vdd,$clktime+2.1,$clktime+2.1,$vdd);
```

printf SPICE (".tran %3.3fns %3.3fns\n\n",(2+4*($clktime+0.1))/100,2+4*($clktime+0.1));
if (($numargs==2) || ($numargs==3))
{
    printf SPICE (".print v(col%d_%d)\n",$last_one,$num_row);
}
elsif ($numargs==4) # power calc, hit it with a few clk pds
{
    printf SPICE ("Vpls a_%d 0 PWL 0 %1.1f 0.1ns 0\n",$col-1,$vdd); #output col active
    printf SPICE ("Vclk clk 0 PWL 0 0 %3.3fns %1.1f %3.3fns 0 R \n",$clktime,$vdd,$clktime+0.1,$vdd,2*$clktime,$vdd,2*$clktime+0.1);
    printf SPICE ("Vclkb clkb 0 PWL %1.1f %3.3fns %1.1f %3.3fns 0 %3.3fns %1.1f R \n",$vdd,$clktime,$vdd,$clktime+0.1,$vdd,$clktime+0.1,$vdd);
    printf SPICE (".tran %3.3fns %3.3fns\n\n",$clktime/10,8*$clktime);
    printf SPICE (".print i(m%d)\n",$last_one); #get row current for pwr calc
    printf SPICE (".print i(m%d)\n",$m_inst-1); #get row current for pwr calc
    printf SPICE (".measure TRAN AVGVAL1 AVG i(m%d)\n",$m_inst-1); #pull-up tran row pre-chg
    printf SPICE (".measure TRAN AVGVAL2 AVG i(m%d)\n",$last_one); #pull-up tran output col pre-chg
    printf SPICE (".measure TRAN AVGVAL3 AVG i(vbufs)\n"); # current used by row buffer
    printf SPICE (".measure TRAN AVGVAL4 AVG i(vbufs)\n"); # current used by addr buffer
    printf SPICE (".measure TRAN pwrl param='AVGVAL1*%1.1f'\n",$vdd);
    printf SPICE (".measure TRAN pwrl param='AVGVAL2*%1.1f'\n",$vdd);
    printf SPICE (".measure TRAN pwrl param='AVGVAL3*%1.1f'\n",$vdd);
    printf SPICE (".measure TRAN pwrl param='AVGVAL4*%1.1f'\n",$vdd);
}
printf SPICE (".END\n");
close(SPICE);
D MATLAB Simulation Files

Circle Interpolator Algorithm Simulator

function [Y_s,X_s,F_s] = circle_varinc(bits,inc,offset)
%function [Y_s,X_s,F_s] = circle_varinc(bits,inc,offset)
%bits=8; inc=[2]; offset=0;
YF=0;
XF=2^(bits-2)-1;
cw=0;
count=1;

fs=0;
xs=0;
ys=0;
X= XF;
Y=YF;
if offset==1
F=-128;
elseif offset==0
F=0;
end
index=1;
finflag=0;

while(finflag==0)
X_s(count)=X;
Y_s(count)=Y;
F_s(count)=F;
amp=1; Xamp=inc(index);
y1=xor(fs,xor(xs,ys));
if (y1==0)
xs=xor(cw,'(ys));
if(xs==0)
F=F+2*X*amp+inc(index)*amp;
X=X+inc(index);
else
F=F-2*X*amp+inc(index)*amp;
X=X-inc(index);
end
else
ys=xor(cw,xs);
if(ys==0)
F=F+2*Y*amp+inc(index)*amp;
Y=Y+inc(index);
else
F=F-2*Y*amp+inc(index)*amp;
Y=Y-inc(index);
end
end
if X<0
xs=1;
else
xs=0;
end
if Y<0
ys=1;
else
ys=0;
end
if F<0
fs=1;
else
fs=0;
end

end
Phase Accumulator ROM Compression Simulator

function [coarse,fine,wave,coarse_wave,fine_wave,coarse_addr,fine_addr]...
\quad = \sin\, \text{compr}2(\text{bits,step,sn\_flg})
\%function [coarse,fine,wave,coarse_wave,fine_wave,coarse_addr,fine_addr]...
\%= \sin\, \text{compr}2(\text{bits,step,sn\_flg})

\text{bits}=8; \quad \text{step}=1; \quad \text{sn\_flg}=0;
\text{amp}=2^{(\text{bits}-3)-1};
\text{bits}=\text{bits} \times 2; \quad \% \ 2 \text{ bits reserved for sign and quadrant}
A=\text{floor}(\text{bits}/3); \quad \% \text{ split into 3 wrds}
B=\text{floor}(\text{bits}/3); \quad C=B;

\text{for } i=0:2^{-A\times B}-1
\quad a=\text{bitshift}(i, -B); \quad b=\text{mod}(i, 2\times B);
\quad Fc(i+1)=\sin(pi/2\times(a\times 2^{-B}+b)/2^{-A\times B}+1/2^{-A\times B+C});
\text{end}

\text{if } (\text{sn\_flg}==0)
\quad \text{for } i=0:2^{-A\times C}-1
\quad \quad a=\text{bitshift}(i, -C); \quad c=\text{mod}(i, 2\times C);
\quad \quad \text{for } b=0:2\times B-1
\quad \quad \quad \text{Fc\_tmp}=\sin(pi/2\times(a\times 2^{-B}+b)/2^{-A\times B+C});
\quad \quad \quad \text{tmp}(b+1)=2^{-C-B}\times\{(\sin(pi/2\times(a\times 2^{-B}+b)/2^{-A\times B+C})+1/2^{-A\times B+C+1})-\text{Fc\_tmp}\};
\quad \quad \text{end}
\quad \quad \text{Ff}(i+1)=\sum\text{tmp};
\quad \quad \text{Ff\_s}(i+1)=\text{Ff}(i+1);
\quad \quad \text{Ff\_e}(i+1)=0.5*\max(2\times B*\text{tmp})-0.5*\min(2\times B*\text{tmp}); \quad i=i+1;
\quad \text{end}
\text{else } \% \text{do cosine calculation}
\quad \text{for } i=0:2^{-A\times B}-1
\quad \quad a=\text{bitshift}(i, -B); \quad b=\text{mod}(i, 2\times B);
\quad \quad Fc(i+1)=\cos(pi/2\times(a\times 2^{-B}+b)/2^{-A\times B}+1/2^{-A\times B+C});
\quad \text{end}

\text{for } i=0:2^{-A\times C}-1
\quad a=\text{bitshift}(i, -C); \quad c=\text{mod}(i, 2\times C);
\quad \text{for } b=0:2\times B-1
\quad \quad \text{Fc\_tmp}=\cos(pi/2\times(a\times 2^{-B}+b)/2^{-A\times B+C});
\quad \quad \text{tmp}(b+1)=2^{-B}\times\{(\cos(pi/2\times(a\times 2^{-B}+b)/2^{-A\times B+C})+1/2^{-A\times B+C+1})-\text{Fc\_tmp}\};
\quad \text{end}
\quad \text{Ff}(i+1)=\sum\text{tmp};
\quad \text{Ff\_s}(i+1)=\text{Ff}(i+1);
\quad \text{Ff\_e}(i+1)=0.5*\max(2\times B*\text{tmp})-0.5*\min(2\times B*\text{tmp}); \quad i=i+1;
\quad \text{end}
end

fine=round(amp*Ff);
coarse=round(amp*Fc);

% reconstruction test phase
i=0;
while((i*step<2^-(A+B+C)) | (mod(i*step,2^-(A+B+C+2)) = 1*step))
  if (~sn_flg & (mod(i*step,2^-(A+B+C+2)) >= 2^-(A+B+C+1))) | (sn_flg...
    & (((mod(i*step,2^-(A+B+C+2)) >= 2^-(A+B+C))& (mod(i*step,2^-(A+B+C+2))...
      <= 2^-(A+B+C+2)-2^-(A+B+C)))))
    sb=-1;
  else
    sb=1;
  end
  if mod(i*step,2^-(A+B+C+1)) >= 2^-(A+B+C)
    offst=2^-(A+B+C)-1;
  else
    offst=0;
  end
  p=abs(offst-mod(i*step,2^-(A+B+C)));
  a = bitshift(p,-(B+C))*2^-(C); b = mod(bitshift(p,-C),2^-B); c=mod(p,2^-C);
  coarse_wave(i+1)=coarse(a+b+1)*sb; coarse_addr(i+1)=a+b+1; a = bitshift(p,-(B+C))*2^-(C);
  fine_wave(i+1)=fine(a+c+1)*sb; fine_addr(i+1)=a+c+1;
  i=i+1;
end

coarse_wave=coarse_wave(1:length(coarse_wave)-1); fine_wave=fine_wave(1:length(fine_wave)-1);
wave=coarse_wave+fine_wave;