Asynchronous Direct Digital Synthesis Modulator

Implemented on a FPGA

by

Jennifer H. Shen

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 23, 1997

Author_____
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by_____
Professor James Kirtley Jr.
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Asynchronous Direct Digital Synthesis Modulator Implemented on a FPGA

by

Jennifer H. Shen

Submitted to the

Department of Electrical Engineering and Computer Science

May 23,1997

In Partial Fulfillment of the Requirements for the Degree of
Bachelor Science in Electrical Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

As IC chips get faster and smaller, the synchronous design method has hit upon the limitations of clock skew, higher power dissipation, and timing verifications. Now asynchronous design is being used to overcome those boundaries. It relies on request / acknowledge signalling instead of clock driven operation. This project will implement an asynchronous direct digital synthesis modulator which does quadrature phase-shift keying. Two data streams, I and Q, are passed through interpolation filters. The numerically controlled oscillator (NCO) takes in a set frequency and generates sine and cosine waves which are then modulated with the two filtered data streams respectively. The two waves are recombined for a final output to an external digital-to-analog convertor. Given the limited time frame, the system will be programmed on a field programmable gate array (FPGA) using VHDL programs that simulate the different blocks. The FPGA will be tested to verify the functionality of the asynchronous design. The focus of this thesis is on the asynchronous techniques developed.

Thesis Supervisor: Professor James Kirtley Jr.
Title: Professor of Electrical Engineering
Company Supervisor: Dr. John Grosspietsch
Title: Principal Staff Engineer, Motorola Chicago Corporate Research Labs

ACKNOWLEDGEMENTS

Thanks to Prof. James Kirtley, Jr. for being my thesis advisor.

My thanks to the engineers at the IC Design Lab at Motorola Chicago Corporate Research Labs and my manager Steve Gillig. Special thanks to Tim Rueger for all his help.

To all those special people that kept me sane and motivated: Edward Hwang, Nitin Kasturi, and Erica Pearson.

I would like to extend my sincerest appreciation to my company supervisor Dr. John Grosspietsch without whom there would not be any thesis.

Finally, I dedicate this work to my parents, Robert and Julia Shen for all their support.

TABLE OF CONTENTS

1. Introduction 9
2. Background 10
   - 2.1 Request/Acknowledge Signalling 10
   - 2.2 Micropipelines 14
   - 2.3 Self-Timed Circuits 15
3. Design 17
   - 3.1 Purpose 17
   - 3.2 Direct Digital Synthesis Modulator 17
     - 3.2.1 General Description 17
     - 3.2.2 Asynchronous Design Decisions 18
4. Implementation of Modules 21
   - 4.1 Control Signalling 21
     - 4.1.1 Completion Elements 21
     - 4.1.2 Four Phase HandShaking Elements and Registers 23
     - 4.1.3 Delay Insertions 23
   - 4.2 Self-Timed Blocks 24
     - 4.2.1 Self-Timed Adders 24
     - 4.2.2 Self-Timed Multipliers 25
   - 4.3 Sine and Cosine Wave Generation 26
     - 4.3.1 Numerically Controlled Oscillator 27
     - 4.3.2 Sine and Cosine ROMs 29
   - 4.4 Interpolation Filters 33
   - 4.5 Input Data Generation: Quadrature Phase-Shift Keying 34
     - 4.5.1 QPSK Rotation 34
     - 4.5.2 Random QPSK 35
     - 4.5.3 Integration of the Two QPSK Methods 36
5. Software and Hardware Configurations 38
   - 5.1 Software Design Flow 38
   - 5.2 FPGA Demonstration Board Setup 40
   - 5.3 Digital to Analog Convertor Board Setup 42
6. Results and Verification 44
   - 6.1 DDS Outputs: Sine and Cosine Generation 44
   - 6.2 QPSK 46
   - 6.3 Random QPSK 48
   - 6.4 System Observations 49
7. Conclusion 50

References 51
Literature 51
Web Sites 51
Appendices 53
VHDL Code for Modules 53

LIST OF FIGURES

1. Request Acknowledge Signalling 10
2. Two-Phase Signalling 11
3. Four-Phase Signalling 11
4. C-element and Example 12
5. Four-Phase Handshaking Circuitry 13
6. Different Cases of Four-Phase Signalling Inputs and Outputs 13
7. Micropipeline Structure From Sutherland's Paper [4] 14
8. DCVSL Logic 16
9. DCVSL Blocks with Control Blocks 16
10. Block Diagram for the Direct Digital Synthesis Modulator 18
11. Block Diagram of the DDSM With Asynchronous Control Paths 19
12. Design Flow 20
13. Simplified Block Diagram of XC4000-Series CLB 21
14. Gamble C-element and Truth Table [6] 22
15. Gamble's C-element Signals 22
16. Combinational C-element with the Y Input Inverted and Truth Table 23
17. Pipeline Control 23
18. Optimized C-element Acting as a Delay Element 24
19. Self-Timed Adder 25
20. Self-Timed Multiplier 26
21. Wavetable 27
22. Frequency Step Input to ROMs 27
23. NCO Schematic with Delay Insertions 28
24. Sine Wave Breakdown 29
25. Sixteen Point Sine Wave Beginning at Zero 30
26. Sixteen Point Sine Wave With Offset 31
27. 256 Point Sine Wave with Zeros and No Zeros 31
28. Sine and Cosine Generation Schematic with Delay Insertions 32
29. Interpolation Filter Schematic 33
30. Matlab Generated Z-transform of Interpolation Filter 34
31. QPSK Clockwise and Counterclockwise Rotation 35
32. Schematic of Pseudorandom Generator 36
33. QPSK schematic with Interpolation Filters 37
34. FPGA Demonstration Board Schematic 41
35. FPGA Demonstration Board Physical Layout 42
36. Schematic Layout of D/A convertor board 43

37. Generated Sine and Cosine Waves at 1.8 MHz....44
38. 1024-PT FFT vs. 78-PT FFT Carrier Frequency For Case A....45
39. Polaroid Showing the Generated Carrier Wave Spectrum....45
40. Theoretical Output Spectra of the CW and CCW for Case A....46
41. Leading and Lagging First and Third Harmonics....47
42. Matlab Generated Spectrums of the QPSK output CW, CCW....47
43. Polaroid Showing the Spectrums of the QPSK output CW, CCW....48
44. Polaroid of the Spectrum Output for Random QPSK....48

LIST OF TABLES

1. Gamble C-Element With Y Inverted 22
2. Combinational C-Element 23
3. Timing of Delay Elements 24
4. Partitioned Design Utilization Using Part 4010EPC84-2 for DDSMQPSK.VHD ... 40

Chapter 1. Introduction

While the synchronous approach has helped to simplify the design of IC chips, the limitations are becoming more evident as technology get smaller and faster. Each pass to increase the system clock for a higher throughput requires a redesign and re-verification of the full synchronous system, while the faster switching action dissipates more power throughout. We are gaining higher performance at the cost of higher power and longer re-design time [1]. Since synchronous design became popular, asynchronous research is now being revived to overcome those boundaries.

The lack of a global clock is the one trademark which separates asynchronous from its counterpart. Without any clocked circuits, there will be less switching which implies, but does not guarantee less power dissipated in the circuit. Another favorable condition is that the redesign becomes more modular. When a module is replaced by a faster one, there's no need to recheck timing parameters. Each module operates at its own pace whether fast or slow. When done, the module signals the next subsystem.

This thesis utilizes the asynchronous pipeline methodology to implement a digital system. In this case, a direct digital synthesis modulator is the proposed system. Given the fact that no commercial asynchronous standard cell libraries or asynchronous field programmable gate arrays (FPGAs) are available, we are limited to synchronous FPGAs. The time frame for this project is too short to develop a standard cell asynchronous library from scratch which would be more satisfactory. The focus of this thesis is the asynchronous techniques developed for this project. The resulting project is to be tested for functionality of the asynchronous system and the use of self-timed modules.

Chapter 2 provides the background material about asynchronous research applicable to this thesis. Chapter 3 proposes the design to be built. Chapter 4 presents goes over the work done and the decisions made for each block. While Chapter 5 discusses all the relevant software issues and hardware specifications for the FPGA, the demonstration board, and the digital-to-analog convertor. Chapter 6 covers the testing and verifying the functionality of the system with concluding remarks in Chapter 7. References and Appendices are located at the end of this thesis.

Chapter 2. Background

Asynchronous design has been around at least since the 1950's, but it was displaced in favor of synchronous techniques. [3] Present day limitations in synchronous design have marked a revival in asynchronous research. There are many branches to this field, but the one common trait to all of them is the absence of a global clock.

2.1 Request/Acknowledge Signalling

Without the clocks, the modules communicate using request/acknowledge signalling. For example, one module finishes and fires a request signal to the next one. It receives the request and grabs the valid data from the module or the bus. When finished, the receiver fires off an acknowledge to the sender module indicating that the transfer is complete. The modules goes into a stand-by mode which uses no or little power while waiting for the next request. [4] See Figure 1 for basic setup.

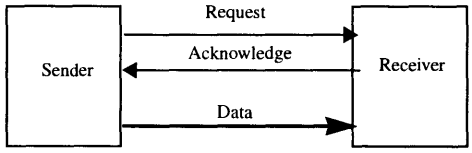

FIGURE 1. Request Acknowledge Signalling

There is one protocol that is adhered to. The data being passed to the receiver must be held valid from when the request fires until the acknowledges fires in return. There are two main types of request/acknowledge signalling: two-phase signalling [4,12] and four-phase signalling[4,11].

Two-phase signalling is based on event triggering. A rising or falling edge are both considered to be events. The request event is always answered by a acknowledge event after some indeterminate delay. The sender and receiver must wait for those events. Since both rising and falling edges are crucial in signalling, the resulting circuitry must be hazard free and delay insensitive. A hazard in the control circuitry will cause errors to result. See Figure 2 below. Valid data appears on the sender's outputs. The sender signals an event on the request line. (1) The receiver latches the valid data and responds with an event on the acknowledge wire. (2) The sender gets the acknowledge event and waits for the next request. Note that the data can become invalid after the acknowledge event (3).

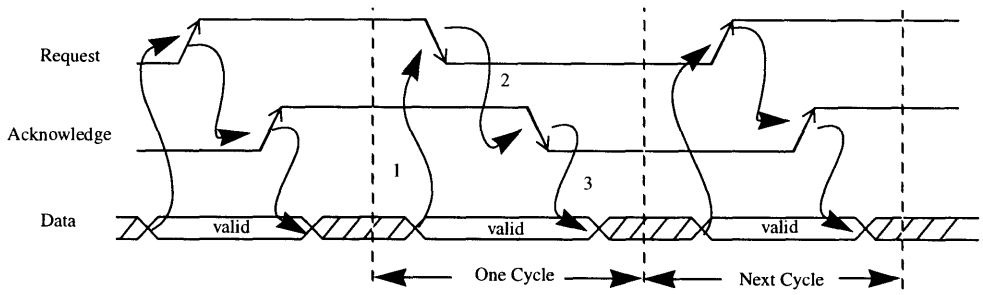

FIGURE 2. Two-Phase Signalling

Four-phase signalling requires that the request and acknowledge line must return to logic zero after each handshake as shown in Figure 3 below. The first step is for the sender to wait for data to become valid. The request line is pulled to logic one causing the receiver to work on the incoming data. (1) When done, the receiver pulls its acknowledge line high. (2) After this, the sender pulls it request line back to logic zero. (3) Answering in kind, the acknowledge line is pulled down. (4) This protocol has two transitions per signal for each communication. Four-phase signalling is easier to implement from a circuit standpoint. Logic for two-phase signalling is far more complex since both rising and falling transitions must trigger the same events.

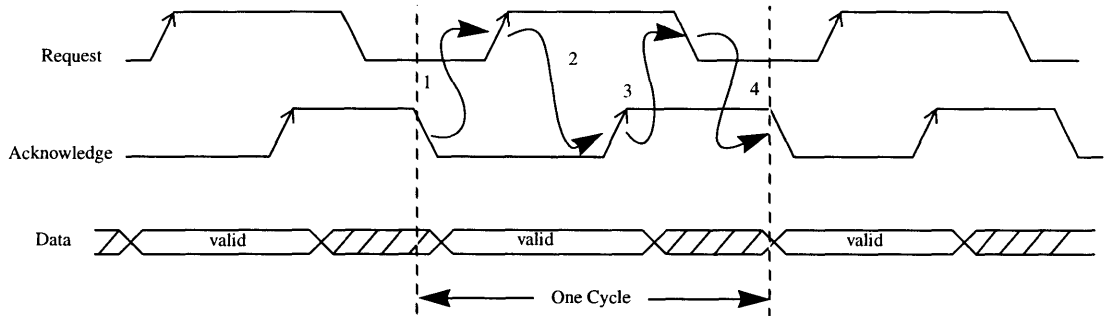

FIGURE 3. Four-Phase Signalling

One basic building block of asynchronous control signalling is the completion element (C-element). These C-elements are important for the control signalling between blocks. Without the clocks, the control signalling circuitry must be asynchronous and be able to wait indefinitely for single or multiple events [4]. A single C-element can have two or more inputs and one output. When all the inputs are logic one, then the output will be logic one. When all the inputs are logic zero, then the output will be logic zero. If all the inputs are not the same, the output remains the unchanged. In addition, a clear input pulls the output down to logic zero. A C-element and its timing diagram are shown in Figure 4.

The C-elements are used to signal the completion of multiple modules. An example is shown in the bottom half of Figure 4. Two sender modules each have a request line, an acknowledge line, and a data port.

A C-element can be used to synchronize the two requests to the receiver module. The acknowledge signal resets both sender modules once the receiver is done.

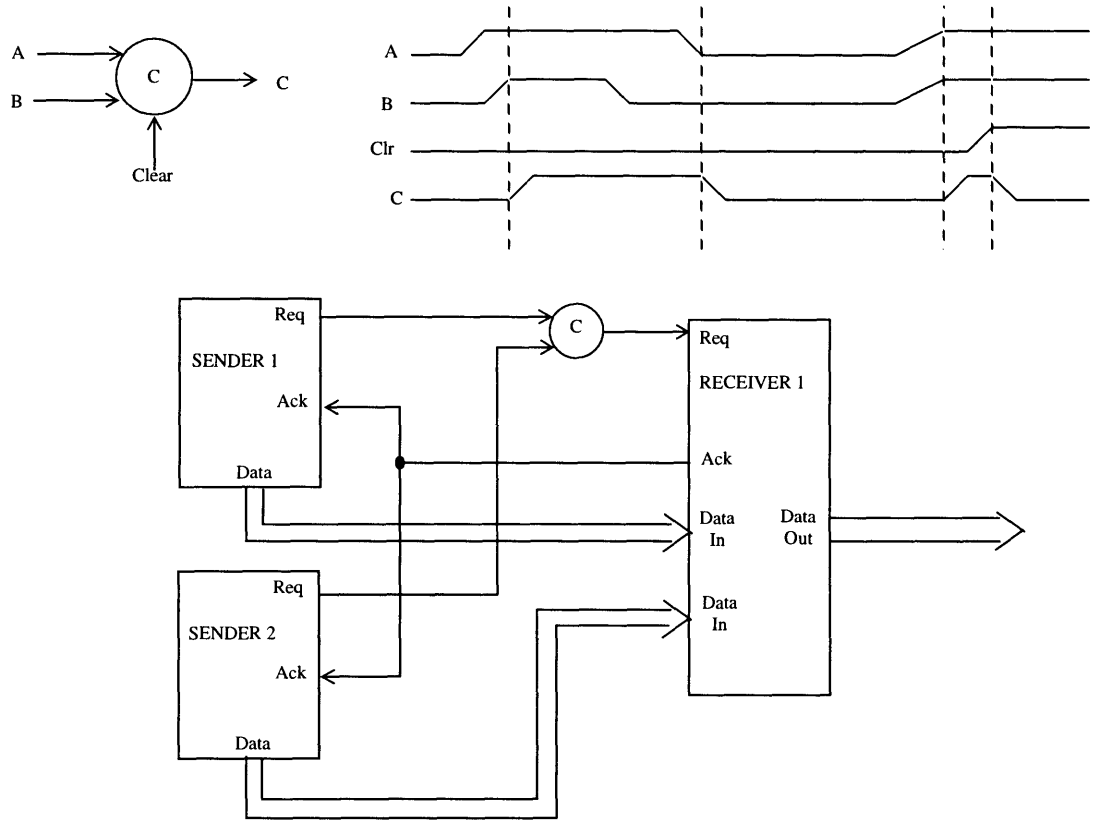

FIGURE 4. C-element and Example

C-elements should be built at the transistor level. Building a C-element with only gates can lead to meta-stability issues, and hazards can result. Hazard-free C-elements are definitely needed for asynchronous design. Any hazards caused in the control signalling can lead to false data being transferred between modules. Building C-elements on the transistor level is more ideal than using just logic gates. However for this thesis, the field-programmable gate array hinders us in that respect. Section 4.1.1 gives more details about the specific C-elements used for this project.

As shown above, one C-element can implement two-phase handshaking. Four-phase handshaking elements can be built out of C-elements. Figure 5 shows a four-phase handshaking structure. Notice that one of the inputs is inverted going into each C-element. The two C-elements have two handshaking pairs- (1) ReqIn with AckOut and (2) ReqOut with AckIn. ReqIn and AckIn are two inputs; ReqOut and AckOut are two outputs. The four-phase handshaking element is often used in pipelined structures [5].

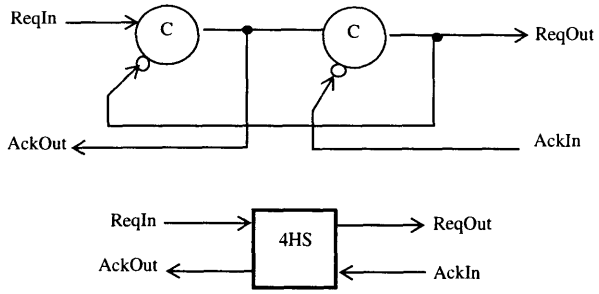

FIGURE 5. Four-Phase Handshaking Circuitry

The four-phase handshaking element is used to handle the request and acknowledge signaling between modules. Each request/acknowledge signal pair must follow the data valid protocol. Whether data is being sent or received, the data lines must remain valid between the request event (rising only) and the corresponding acknowledge (rising only). Below are three cases of possible inputs and output combinations for ReqIn, AckOut, ReqOut, and AckIn. Notice that the request/acknowledge pairs are following the four-phase handshaking protocol. See Figure 6 for signalling details.

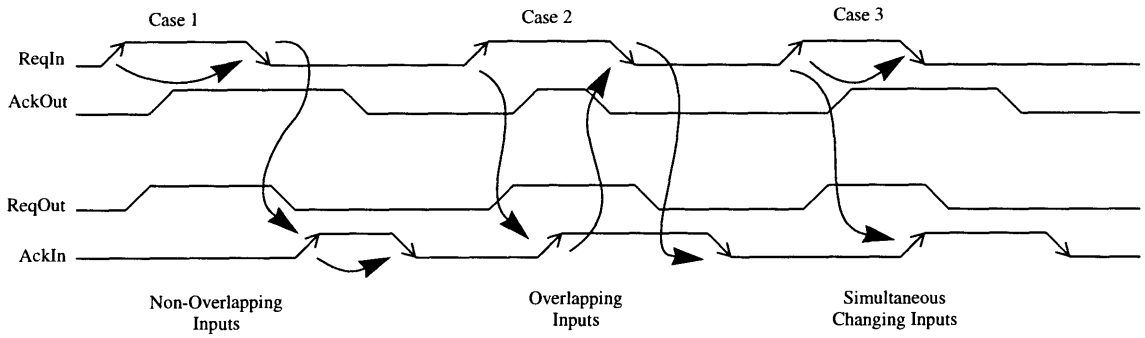

FIGURE 6. Different Cases of Four-Phase Signalling Inputs and Outputs

This asynchronous element must work correctly for all possible input combinations. Case 1 shows the inputs as non-overlapping. ReqIn goes high which pulls AckOut and ReqOut high. ReqIn falls before the AckIn is pulled high. Afterwards, both AckOut and ReqOut are pulled low. This case shows that ReqIn and AckIn are non-overlapping when high. Case 2 shows the inputs overlapping. When ReqIn is pulled high, then AckIn is pulled high afterwards. ReqIn is then pulled low followed by AckIn pulled low. This shows that AckIn and ReqIn are pulled high at different times and overlap. The last case shows the inputs changing simultaneously. ReqIn is pulled low at the same time AckIn is pulled high. These different cases show the hardiness of the four-phase signalling element for all different input combinations. Notice how the ReqOut and AckOut respond to the different cases. Control structures such as these are crucial for asynchronous design. Hazards must be checked for carefully.

2.2 Micropipelines

Pipelines are used to speed up the throughput of a repetitive process. Combinational logic is separated by clock driven registers, and the data flows through each stage. Ivan Sutherland modified a standard pipeline by replacing the clock signaling with request/acknowledge signalling. He called these modified pipelines "micropipelines". Figure 7 show more detail. These micropipelines used C-elements to signal between the stages, while event driven registers control the flow of data [4,7].

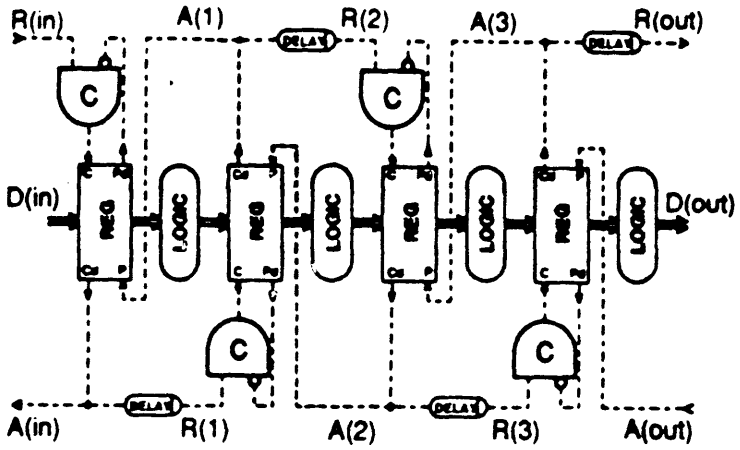

FIGURE 7. Micropipeline Structure From Sutherland's Paper [4]

The R(in) signal which stands for Request In triggers the pipeline. As the control signals propagate down the pipeline, the data is pushed through it. The two-phase signalling structure is accommodated by event driven registers (capture-pass) as seen in the figure. When the capture input (C) gets an event, the D(in) which is Data In is latched into the pipeline. The capture done (Cd) output from the register is delayed to the next C-element. When the pass input (P) is triggered, the latched data is transferred to the outputs. Pass done (Pd) is then fired. The combinational logic between the stages must finish before the capture signal of the next stage fires. In Figure 7, note the inserted delays between the C-elements.

For the micropipeline to work correctly, the delay through the C-element must be greater than the propagation delay through the registers and the propagation delay through the combinational logic [4]. The data must be valid going into the next stage before the request signal fires. Inserting delays are necessary for the correct data to be propagated through the pipeline.

$$T_{pdC} > T_{pdR} + T_{pdL} \quad \text{(EQ 1)}$$

T_{pdC} propagation delay through C-element

T_{pdR} propagation delay through register

T_{pdL} propagation delay through combinational logic

Sutherland's inserted delays are a drawback in some cases. Some circuit optimizers (i.e. FPGA compilers) will remove the inserted delays. Without these delays, the wrong data will be latched from stage to stage. The inserted delays keep the control signalling from reaching the register too fast. If the logic between registers is still computing when the control signal reaches the register, invalid data will be capture and sent on to the next stage. This invalid data would certainly corrupt the rest of the systems output. This possible complication stresses the importance for inserted delays. However, inserted delays need to be re-verify each time the logic is changed in the data path which adds to the re-design time. What is really needed is a completion signal from the combinational logic so that inserted delays become unnecessary for the designer.

2.3 Self-Timed Circuits

Self-timed circuits follow a request/acknowledge protocol. The request signal enables the circuit to evaluate its inputs. When the evaluation is done, the circuits pulls its acknowledge signal (completion or done signal) high. A self-timed circuit should be able to wait indefinitely for the request signal to activate. It does not need a clock edge for timing. This type of circuit is valuable for asynchronous design where delays are assumed indefinite.

Sutherland's micropipeline relied on inserted delays between C-elements to insure the correct operation of the pipeline. Using self-timed circuits in an asynchronous pipeline will eliminate the need for these inserted delays. The completion signal for the self-timed circuit indicates when the logic is finished evaluating. This "done" signal can be integrated into the control structure so that each stage waits for the completion signal before continuing. See Figure 9 for details. Without the need of inserted delays, this will simplify the work for the asynchronous logic designer.

One type of self-timed circuit is the differential cascode voltage switch logic (DCVSL). This transistor logic follows a precharge and evaluate style (dynamic logic). It has a request input (I) and a data valid output (DV). These signals follow a four-phase signalling protocol with differential inputs and outputs. When I is low, the pfets are turned on, and the internal nodes are both pulled up high. While Out and OutBar are both pulled low. When I goes high, the NMOS logic tree evaluates and pulls one of the nodes low. The DV signal goes high when one of the outputs is pulled low [5]. See Figure 8 for a transistor description.

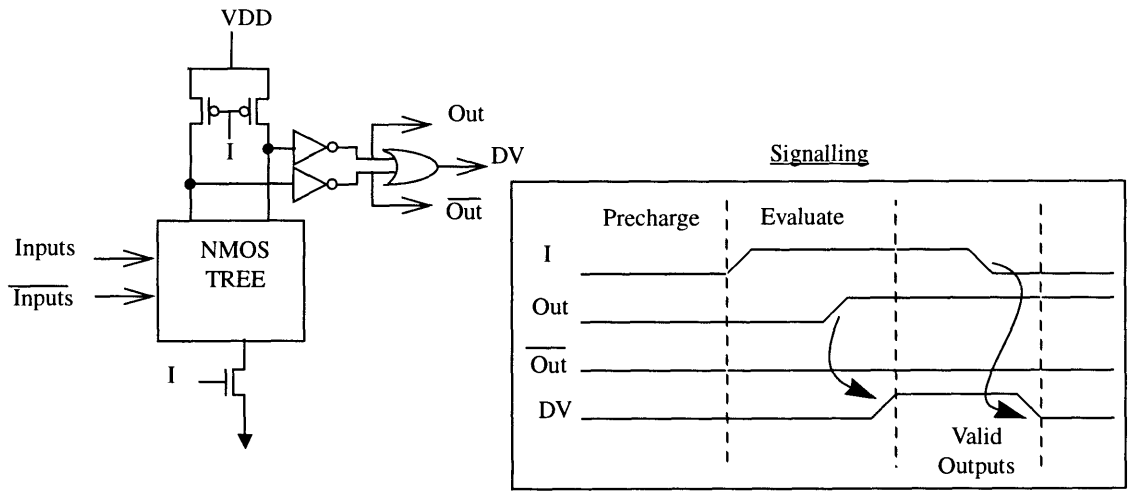

FIGURE 8. DCVSL Logic

All the combinational logic is laid out in N-fets in the NMOS tree. Four-phase signalling modules are needed to control the DCVSL structure. Inputs into the NMOS tree must remain valid while I is high. If no registers were inserted between the stages, then previous stage (j-1) would have to remain static while the current stage (j) evaluated. The following stage (j+1) would be precharging [8]. Inserting registers in between cuts down the number of stages. Figure 9 shows registers that have been inserted between the DCVSL blocks [5]. Notice how the I (request) and DV signals are integrated into the control structure.

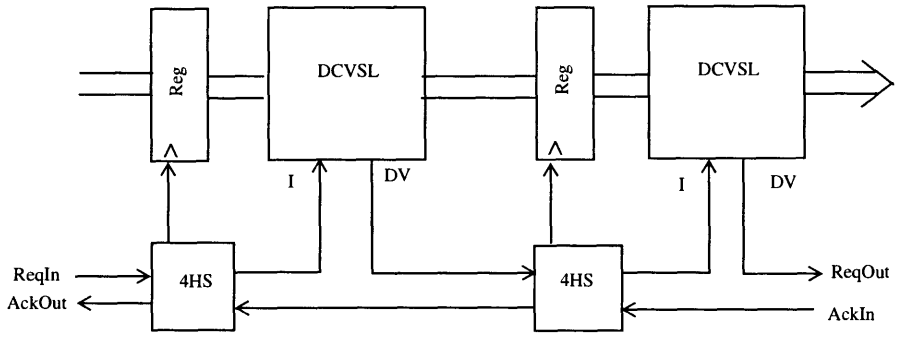

FIGURE 9. DCVSL Blocks with Control Blocks

The registers do not need to signal when the data is latched. This system assumes that the propagation delay of the register is smaller than the propagation delay through the interconnect circuitry. The registers do not follow the request acknowledge protocol but depend on inherent delays of the circuitry.

Chapter 3. Design

3.1 Purpose

Since most IC design efforts are directed toward synchronous design, the available software tools reflect that trend. Presently, there are no commercial tools for designing asynchronous systems. Only universities have developed the software to analyze and synthesize asynchronous circuits [9]. To do a fully self-timed IC design can take a few years to build, due to a lack of commercial asynchronous standard cell libraries. Since this thesis project has only a few months to be completed, we must utilize what technology we have. We implemented an asynchronous system on a field programmable gate array (FPGA) which was originally made for synchronous designs. There was some skillful manipulation of the software for this work to be accomplished.

The purpose of this thesis was to build an asynchronous direct digital synthesis modulator. We used current research to develop a functional design in a short timeframe. The FPGA implementation was to verify the correctness of the work. The research has taken different asynchronous techniques and combined them into a more workable methodology. Self-timed circuits and Sutherland's micropipelines were applied with the minimal use of inserted delays. Hopefully, the resulting asynchronous techniques can be used to implement a full custom IC, and other self-timed designs can be functionally tested on the FPGA.

3.2 Direct Digital Synthesis Modulator

This Section 3.2.1 contains a general description of the direct digital synthesis modulator (DDSM) without the asynchronous modifications. Section 3.2.2 elaborates on making the DDSM into a self-timed pipeline and the design-flow to accomplish this transformation.

3.2.1 General Description

A DDSM modulates data onto carrier waves. Figure 10 shows a general block diagram of the system. It synthesizes sine and cosine waves at a programmable frequency using numerically controlled oscillator (NCO). The I and Q data streams can be analog signals converted to digital. Or they can be digitally generated waveforms from other chips. The I and Q data streams are put through interpolation filters to match the sampling rate of the system [13]. Each filtered data stream is multiplied by the sine and cosine wave respectively. The outputs are added together for the resulting modulated waveform. An external digital-to-analog convertor converts the bitstream to an analog signal. Note that the block diagram does not show the asynchronous control paths. The asynchronous version of DDSM appears in the next section 3.2.2.

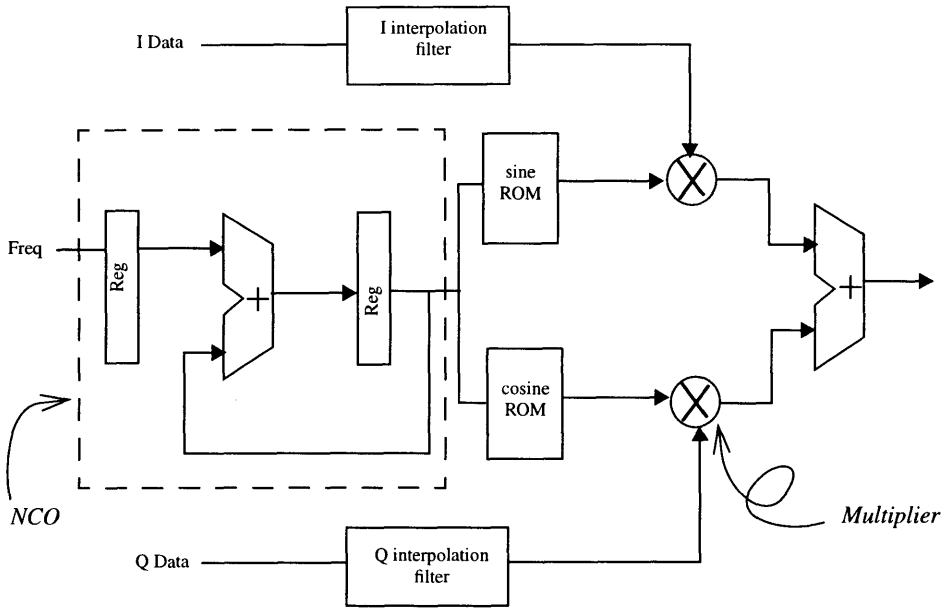

FIGURE 10. Block Diagram for the Direct Digital Synthesis Modulator

The DDSM system can be used for digital signal processing applications. It shifts the baseband data to the carrier frequency for transmission over a communications channel [15]. The addition of the two modulated signals can incorporate two independent data streams in the same bandwidth called quadrature-amplitude modulation [15]. One such example is quadrature phase-shift-keying (QPSK). I and Q change between +/-1 which cause the resultant phase shifts of +/- PI/4 and +/- 3*PI/4. These four phase shifts represent the four different bit combinations of I and Q (00,01,10,11). A more detailed theoretical description of QPSK is written in Section 4.5.1.

Currently, the commercial synchronous counterparts runs at about 100 MHz to 150 Mhz on average with range of 300 mWatts to 500mWatts. Since we are implementing on a FPGA, the system throughput and power was not expected to match up. However, the functionality of the design can be checked with this implementation.

3.2.2 Asynchronous Design Decisions

After researching different asynchronous techniques, a four-phase handshaking protocol was chosen to be implemented for this design. The last stage in the pipeline has its Rout and Ain tied together which resets all the control signals going back up the pipeline. This method has similarities to Sutherland's micropipeline [4] discussed in Section 2.2. For example, a single Rin event propagates one valid data point to the end of the pipeline while the acknowledge line resets the system. Whereas a synchronous pipeline with M stages needs at least M clock pulses to get a valid data point out. Also, the asynchronous pipeline can wait indefinitely for the next event instead of requiring a constant clock. Asynchronous pipelines are good for applications which have indeterminate periods of waiting between calculations. Unfortunately, the

DDSM system must operate continuously for signal modulation which requires an external clock driving the Rin input to the pipeline.

As with pipeline architecture, there are registers between each stage which are triggered by the acknowledge line. Each block is attached to the respective control signals to the blocks preceding and succeeding. This connection determines when a stage can operate depending on the operational state of the blocks next to it. Making the blocks self-timed can benefit the design since the FPGA compilers can optimize out set delays (i.e. double inverters). Ultimately, the multipliers and adder are self-timed, while the NCO and ROM's use inserted delays. Figure 11 shows the DDSM block diagram adapted to asynchronous control signals.

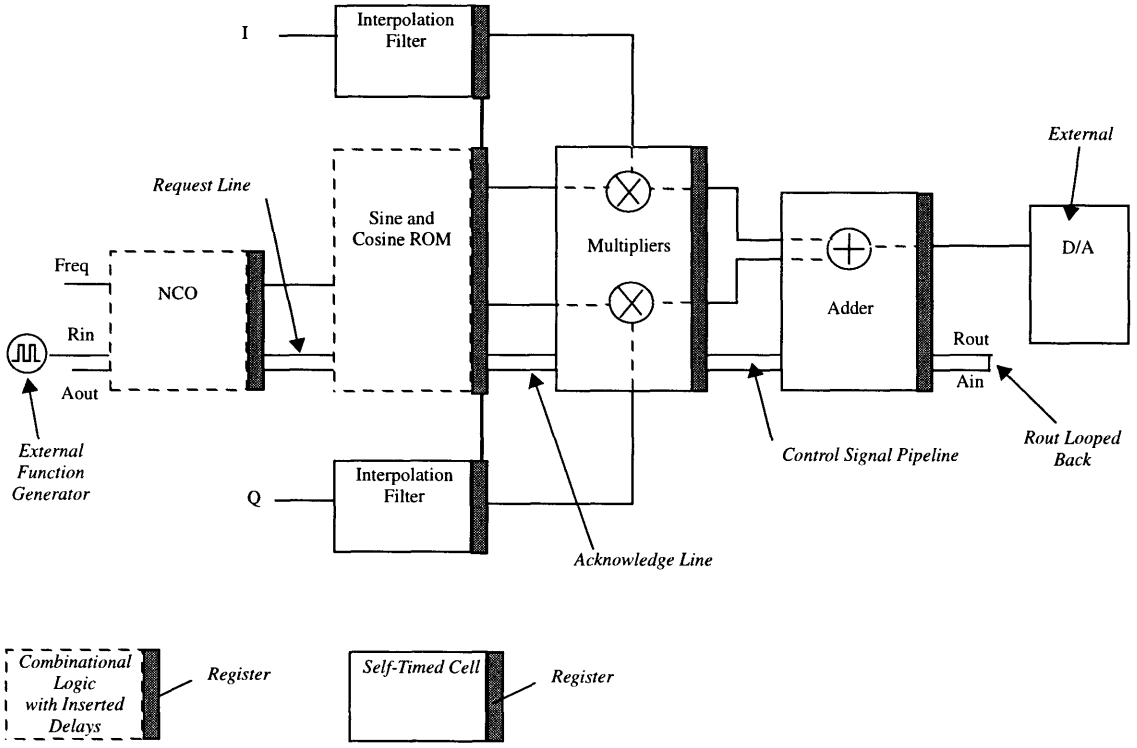

FIGURE 11. Block Diagram of the DDSM With Asynchronous Control Paths

The Xilinx XC4010E FPGA was chosen to implement this asynchronous system. This FPGA is a SRAM based family with look-up tables in each combinational logic block (CLB). The bigger the FPGAs are, the more CLBs are inside the hardware. The added CLB's allow for more complex designs and/or more bitwidth. With the XC4010E FPGA, there are 400 CLB's to be utilized.

The different blocks of the system are programmed and simulated in VHDL. Each module is compiled and simulated in QuickVHDL. Once the VHDL blocks are working, they are synthesized to logic gates by using Synopsys Design Analyzer (SDA). Synopsys is a software package which reads in VHDL and synthesizes the gate-level design. However, Synopsys takes a limited subset of VHDL commands. If a program

is unreadable by the design analyzer, the VHDL must be rewritten and retested until it can be read by Synopsys. The generated gate-level schematics are compiled into the netlist code for the FPGA software (XACT).

Once the XC4010E part is programmed, it is placed on a pre-made FPGA demonstration board. The digital inputs and outputs are tested for correct functionality. An external digital-to-analog device converts the output into a continuous signal for the spectrum analyzer. The design flow can be seen in Figure 12. Note the Design/Testing abstraction line.

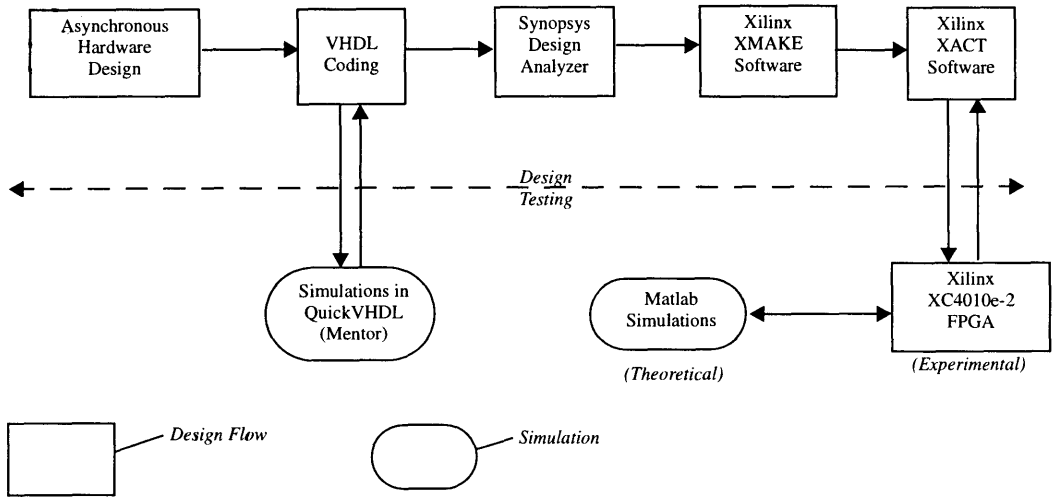

FIGURE 12. Design Flow

Obstacles and stumbling blocks are inherent to any large design. One particular software problem was compiling the VHDL programs through different software tools. For example, Synopsys only takes a subset of VHDL so some experimentation is required for correct programming. In addition, XACT (Xilinx's design editor software) must be able to read in the files generated from Synopsys. Another design problem was synthesizing the self-timed circuitry to the gate-level. In this case, transistor level design is desired but impossible to do on a FPGA which employs look-up tables and combinational logic.

A potential hardware problem is the size of the FPGA. The pre-made demonstration board can only take an 84 pin package. This restriction limits the size of the FPGA we can use. The largest FPGA for the 84 pin package was the XC4010E part. If the design exceeded the 400 CLBs, then we would need two demonstration boards to fit the project. Using two boards might limit the throughput of the pipeline since the largest delays are through the pads. An additional problem is disengaging the clock drivers to the edge-triggered flip-flops in the FPGA since the design calls for no global clock signals. Thankfully, all these problems were overcome in a short period of time [10].

Chapter 4. Implementation of Modules

The asynchronous system can be broken down into several different stages. Each section describes a particular module in detail. Theoretical calculations are also included when relevant unless otherwise noted. The VHDL code for all modules is located in Appendix.

4.1 Control Signalling

4.1.1 Completion Elements

Completion elements are used to handle the request/acknowledge signalling in the asynchronous pipeline. The basic function of C-elements was described in Section 2.1. For this design, multiple C-elements are needed for control signalling and delay elements. However, the FPGA has a limited number of logic gates. The Xilinx FPGA is made up of combinational logic blocks (CLB). The amount of CLBs in an FPGA limits the size of the design that can be downloaded into the hardware. By designing the C-element to fit in one Xilinx CLB structure, this will ultimately conserve logic gates.

Figure 13 shows the internal structure of a CLB [14]. The logic functions of G' and F' are SRAM lookup tables for up to 4 inputs, while logic function of H' is for 3 inputs. These function generators implement all the combinational logic in a design. Internal multiplexors select the various outputs of G', F', H', or other outputs to go through flip flops X and Y. Flip flops X and Y are clocked on the same signal (inverted or not inverted).

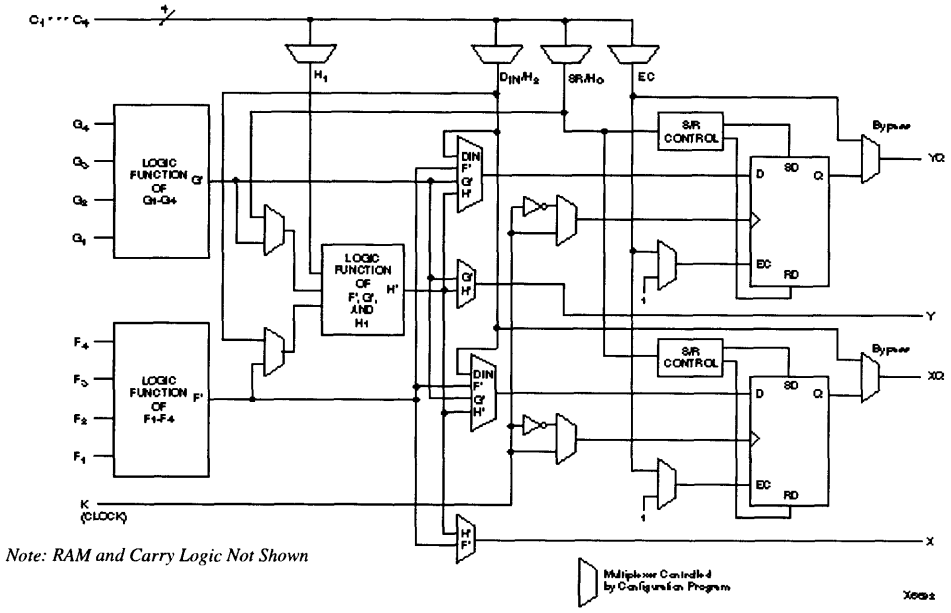

FIGURE 13. Simplified Block Diagram of XC4000-Series CLB

M. Gamble at the University of Manitoba, Canada constructed a micropipelined multiplier in a Xilinx XC4003 FPGA[6]. Their multiplier project included the design of a C-element with one inverted input which fit into one Xilinx CLB. Figure 14 shows the Gamble C-element and its truth table [6]. The three gates labeled A, B, and C map into the SRAM function generators G,F, and H respectively. Feedback outside the CLB loops the Z signal to the clock signal of the flip flops. One flip flop is used for this C-element. Since both flip flops in a CLB are clocked on the same signal, the second flip flop is unused.

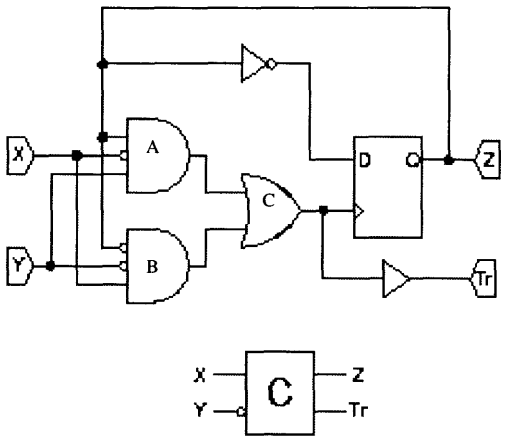

TABLE 1. Gamble C-Element With Y Inverted

X	~Y	Tr	Z(n+1)
0	0	0	Z(n)
0	1	↑↓	0
1	0	↑↓	1
1	1	0	Z(n)

FIGURE 14. Gamble C-element and Truth Table [6]

Gamble's C-element has the Y input inverted. The inverter and flip flop causes the flip flop to toggle between logic 1 and logic 0 whenever the Tr goes high. When X and Y are opposites and Z is opposite from X, the Tr signal goes high which clocks the register. The Z flips and Tr is pulled low immediately afterwards. The C-element then waits for its next input change. This Tr pulse is represented in Table 1 as the up and down arrows. The rising edge flip-flop in the CLB needs Tr reset to zero before the next possible input change. If the inputs do change while Tr is still high, the CLB may not work correctly. This hazard in Gamble's C-element is avoided by the four-phase handshaking configuration which we are using. Figure 15 shows the Gamble's respective signals for different possible situations.

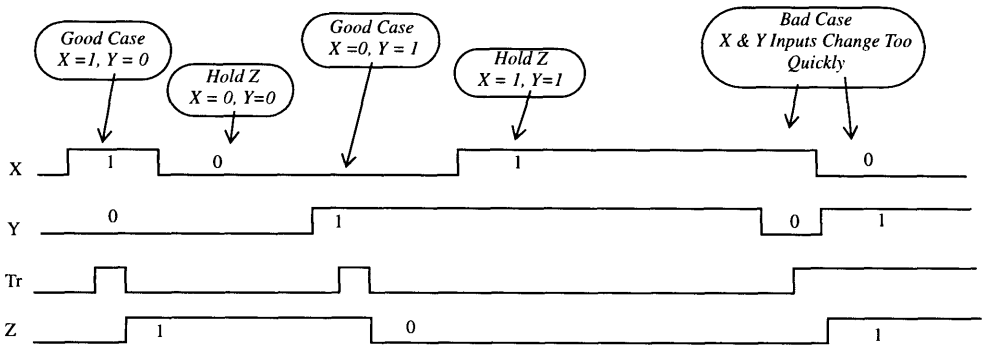

FIGURE 15. Gamble's C-element Signals

A C-element with the Y input inverted made with only combinational logic and asynchronous feedback was placed through the design flow for comparison. The straight combinational logic in Figure 16 took up 2 CLBs per C-element. With four phase handshaking and 2 C-elements per stage, we would have used up twice as many CLBs. For this primary reason, the Gamble C-element was used in this thesis.

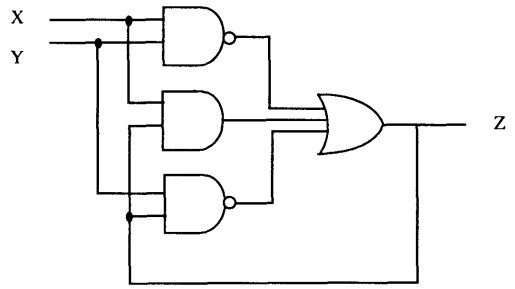

TABLE 2. Combinational C-Element

X	~Y	Z(n+1)
0	0	Z(n)
0	1	0
1	0	1
1	1	Z(n)

FIGURE 16. Combinational C-element with the Y Input Inverted and Truth Table

4.1.2 Four Phase HandShaking Elements and Registers

Generic four phase handshaking elements were described in Section 2.1. This four-phase element was built with two Gamble designed C-elements (Fig. 5) with a reset input. The reset input connects to the clear input to each C-element. The Aout signal from the four phase handshaking element is used to clock the pipeline registers. Figure 17 shows the pipeline control configuration with a combinational logic block between stages. Inserted delays are present if the combinational logic is not self-timed.

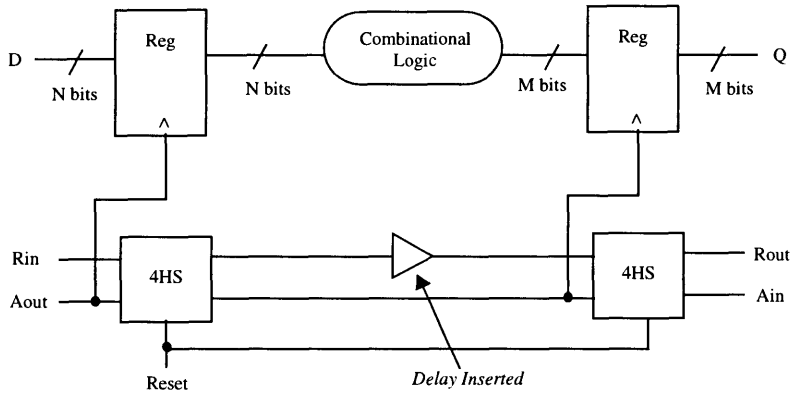

FIGURE 17. Pipeline Control

4.1.3 Delay Insertions

Delay insertions are a necessary part of asynchronous pipelines. Inserting buffers would act as delay elements in the design. However, the Synopsys software would optimize out double inverters and buffers. Instead, we inserted Gamble C-elements with their inputs tied together to act as delay elements. Synopsys does optimize the C-element, but does not eliminate it entirely. Figure 18 shows the optimized C-element

with its inputs tied together. The table shows the experimental delay times through one and more delay elements in series using the Xilinx FPGA architecture.

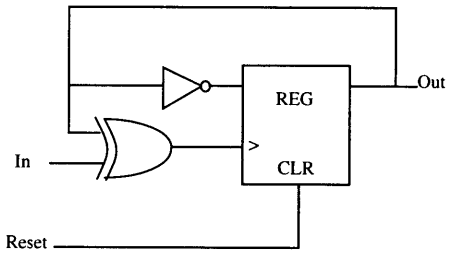

TABLE 3. Timing of Delay Elements

Number of C-elements	Propagation Delay
1	11.8 ns
2	24.92 ns
3	28.56 ns
4	40.46 ns

FIGURE 18. Optimized C-element Acting as a Delay Element

4.2 Self-Timed Blocks

There are two independent paths in an asynchronous pipeline system. One path is for the data being manipulated and the other path is for the handshaking signals. Normally with delay insertions only, the control path is independent of the data path. However with the self-timed architecture, the two paths become dependent on each other. The request/acknowledge (I/DV signals) protocol imposed on the logic blocks are used to break the abstraction. The I and DV signals complete the handshaking connection between each 4-phase handshaking module. See Figure 9 in Section 2.3 for illustration of concept. The self-timed architecture can be referred to as the "data-driven architecture" [16], since the combinational logic now controls the speed of the asynchronous pipeline.

4.2.1 Self-Timed Adders

The VHDL language allows adders to be inferred from the "+" sign. Synopsys reads the VHDL code and instantiates the appropriate adder. We used these instantiated adders to create a self-timed adder. A self-timed adder sums its addends only when the request input (I) is high. The sum is only valid when the data valid signal (DV) goes high. Self-timed circuits make inserted delays unnecessary. In transistor logic, a self-timed circuit is simpler to implement. However for gate level design, it requires more logic to build.

Two adders are used to build one self-timed adder. One adder sums the addends; while the other adder computes the inverse sum. For example, if 0011 and 0010 were added, the sum would be 0101; while the inverse of sum would be 1010. When I is high, one set of inputs is inverted and Cin is pulled high. This calculates the inverse sum. The sum and inverse sum bits are checked using exclusive OR's which then generates the data valid completion signal. When I is low, inputs to both adders are initialized so that the sum and inverse sum are both zero which in turn re-initializes DV to zero. Figure 19 shows a 4x4 self-timed adder. This configuration is easily adapted to a higher bit width.

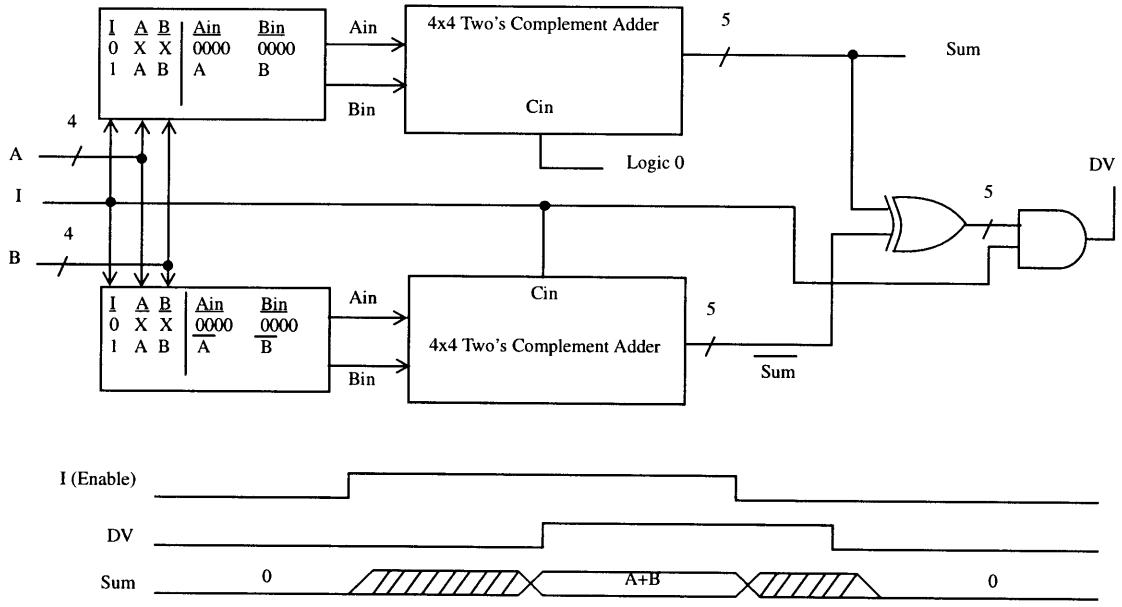

FIGURE 19. Self-Timed Adder

4.2.2 Self-Timed Multipliers

From the VHDL language, the "*" sign infers multipliers. Synopsys reads the VHDL code and instantiates the appropriate multiplier. By specifying the integer ranges of the multiplier and multiplicand, Synopsys interprets an unsigned binary multiplier versus a two's complement multiplier.

The instantiated multipliers are used to create a self-timed multiplier similar to a self-timed adder. Two multipliers compute the product and its inverse product. One multiplier has inverters on its output to compute the inverse product. The self-timing mechanism is created by initializing the inputs to a specified output. When the enable (I) signal is low, the data valid signal must be low (DV). When I is high, the product and inverse product are exact opposites in bits. Exclusive or's compare each bit and generates the DV signal. The inputs are set up that when I is low, the outputs of the two multipliers will be the same. When I goes high, the outputs are inverted and the DV signal goes high. Figure 20 shows the implemented self-timed multiplier.

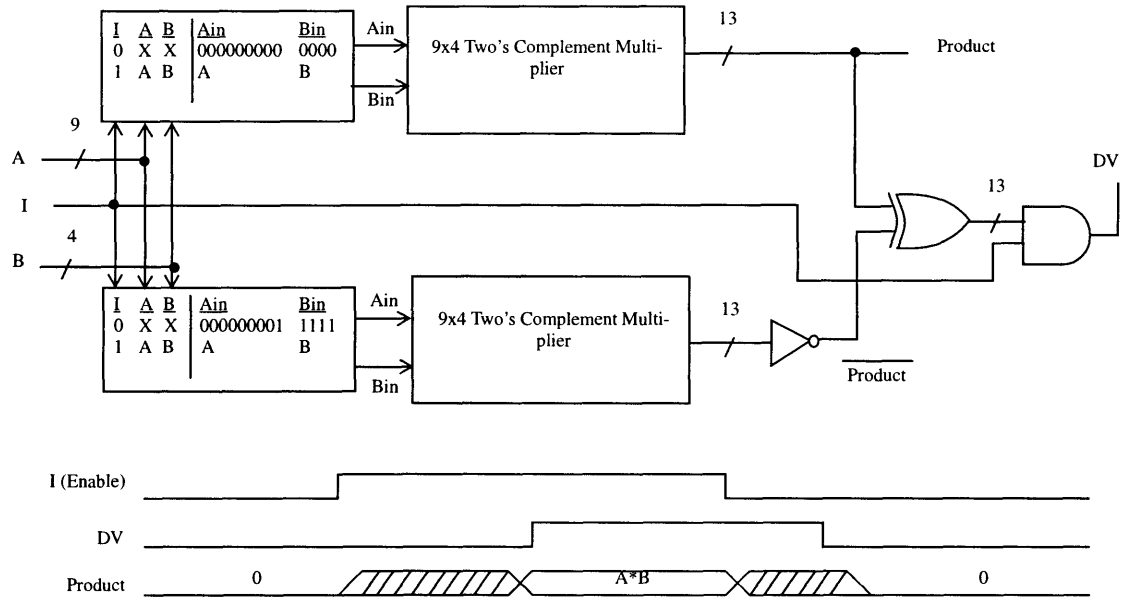

FIGURE 20. Self-Timed Multiplier

The inputs going into the top multiplier are conditioned using AND gates; while the inputs going in the bottom multiplier are conditioned using AND gates and OR gates with one inverted input. When I is low, the top multiplier calculates 0x0=0 while the bottom multiplier calculates 1x(-1)= -1. Since the bottom multplier's output is inverted, inverse product are all zeros. The top multiplier are also all zeros. This initializes the data valid (DV) signal to zero. When I goes high, 9-bit A and 4-bit B inputs are passed into the multipliers. When the calculation is completed, the products are inverses of each other and DV goes high. The bottom half of Figure 20 shows the pertinent signals.

For this self-timed implementation, we need the DV signal to be hazard free since it connects to the Rin input of the next stage. The hazards can result if the 13-bit product and inverse product match up as opposite bits during the calculation before the final product is completed. This causes the exclusive OR to pull high prematurely which in turn creates a hazard on the DV line. The long length of the bitword and the exact timing for each multiplier makes it less hazard prone, although the possibility is still there.

4.3 Sine and Cosine Wave Generation

The sine and cosine wave generation comprise about one-third of the overall system design. The programmable frequency allows the user the freedom to adjust the waveform immediately.

A section of memory can hold all the values from a single waveform. By sampling that table repeatedly, you can get a periodic waveform. Take a sine wave for example. If you take every value from the table, you get a complete copy of the waveform. Skipping every other point in the ROM will produce a sine wave that is half the length of the original. The more points that are skipped, the higher the frequency is of the

resultant sine wave. Figure 21 illustrates this property [W4]. This straight-forward method can easily be implemented in digital logic.

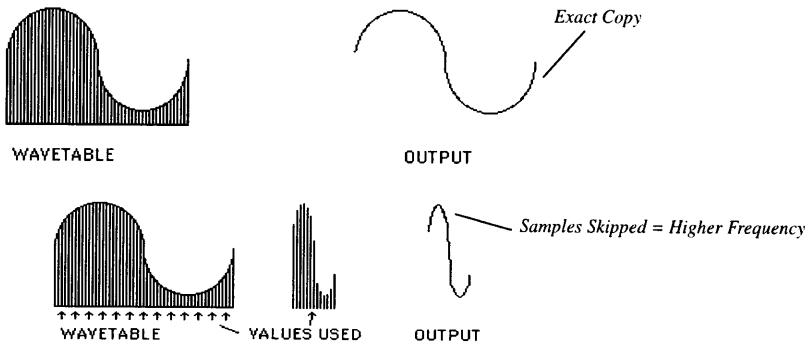

FIGURE 21. Wavetable

Section 4.3.1 describes the theory and implementation of the numerically controlled oscillator which accesses the waveform ROMs; while section 4.3.2 details the loading of the ROM's and its implementation in the FPGA. Both sections discusses the asynchronous implementation of each part.

4.3.1 Numerically Controlled Oscillator

The numerically controlled oscillator (NCO) runs through the samples in the wavetable. Its output serves as the address into the ROMs. Similarly, it represents the 0 to 2*PI index for the waveforms. Skipping addresses in the sequence is similar to counting in bigger increments to 2*PI. The NCO is an adder and an accumulator. Depending on the bitwidth (M bits) of the NCO, the number of frequency steps between 0 to 2*PI varies by 2^M. The lowest bitword maps to 0, while the highest bitword maps to 2*PI. The adder will eventually start back at zero when the most significant carry changes. The output of the NCO inputs to the cosine and sine ROMs. This generates cosine and sine waves at that particular frequency. Figure 22 shows a graphical depiction.

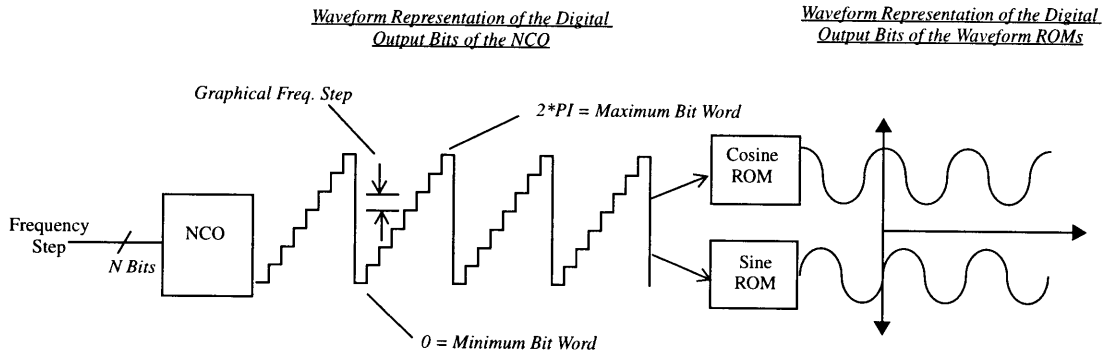

FIGURE 22. Frequency Step Input to ROMs

Finer frequency control was added to the system by increasing the bitwidth of the adder and taking the higher significant bits off the accumulator. Theoretical calculations were done to predict the frequency step and the minimum and maximum frequency of the NCO. Frequency inputs bits can be offset into the adder by A bits. And the NCO output bits are offset from the accumulator's LSB bits by B bits. The accumulator is clocked at a certain Rin frequency. These parameters combine in Eq. 2 to determine the current frequency of NCO based on the frequency input bitword with 8 bits for the NCO output. One can determine the smallest frequency step of the system by setting F = 1. Setting F to the maximum bitword allowed gives the highest frequency. For example, if the frequency input has 8 bits, then setting F to 255 will determine the highest output frequency for the NCO.

$$NCO_{freq} = \frac{Rin \cdot F}{2^{(8+B-A)}} \qquad \text{(EQ 2)}$$

where:

Rin clocking frequency of the accumulator
F value of frequency step bitword
A number of offset bits of frequency bitword into adder
B number of offset bits of the NCO output from accumulator

Based on the above equation, a particular NCO was determined for a frequency range about 1 MHz with a 5-10 kHz minimum increment. The design has a 8 bit frequency input choosing Rin to approximately be around 10 MHz. A and B were chosen to be 0 and 3 respectively in order to meet the design criteria. For this particular configuration, the theoretical minimum frequency step is 4.88 kHz, while the theoretical fastest frequency will be 1.245 MHz. Experimental results should match these calculations. At the slowest frequency, the NCO will make a 256 point ROM into a 2048 point waveform. Figure 23 shows the NCO design. The adder's 2 input MSBs are tied to logic 0.

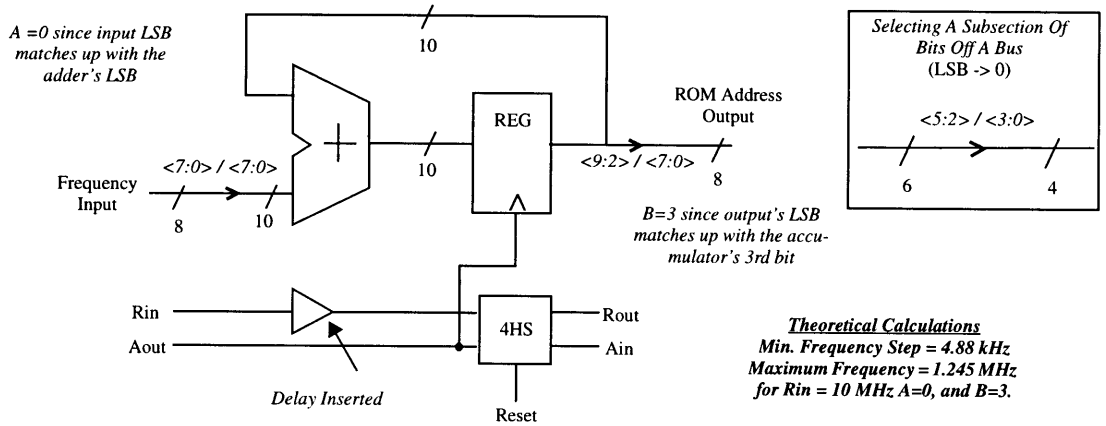

FIGURE 23. NCO Schematic with Delay Insertions

Two Gamble C-element delays are inserted in the control signal pipeline for the NCO stage. We want this stage to regularly output addresses to the sine and cosine ROMs. The NCO is the first stage of the pipeline and we want it to keep up with the external function generator which is driving the pipeline. Ideally, the

Rin signal must rise when the Aout signal goes low which follows the four-phase handshaking protocol. But to have sine and cosine generation, we need the accumulator clocking at a set speed, which requires the need for an external clock. Placing an external function generator on the Rin input regulates the NCO output. It is assumed that Rout, Ain, and Aout will trigger around the Rin signal so that the four-phase protocol is satisfied.

For this implementation, we chose to place an external function generator at the first stage's Rin input and loop the last stage's Rout to its Ain. Figure 11 shows this method in Section 3.2.2. Theoretically, one can also trigger the asynchronous pipeline from the other end. An external function generator can clock the last stage's Ain while looping the first stage's Aout through an inverter to the first stage's Rin. We chose the first method instead of the second method due to VHDL simulator difficulties initializing the control signalling pipeline.

4.3.2 Sine and Cosine ROMs

While the NCO counts between 0 to 2*PI, the sine and cosine ROMs hold only 0 to PI/2 of datapoints. Since sines and cosines are very regular, only one-fourth of the period (0 to PI/2) needs to be stored in the ROMs. Extra combinational logic determines how to read the memory using the two MSBs of the address bitword to generate the full sine and cosine. Storing only the first quadrant of the waveforms keeps the design smaller.

Using only one quadrant of the waveform, the full sine wave can be generated. The first quadrant is generated by sampling the ROM in order from the lowest address to the highest. The second quadrant is generated by sampling the ROM backwards from the highest address to the lowest. The third quadrant is read forward but the data must be negative. The fourth quadrant must also be converted into negative numbers while reading the ROM backwards. See Figure 24 for the breakdown of sine wave. Constructing the cosine wave has a similar methodology.

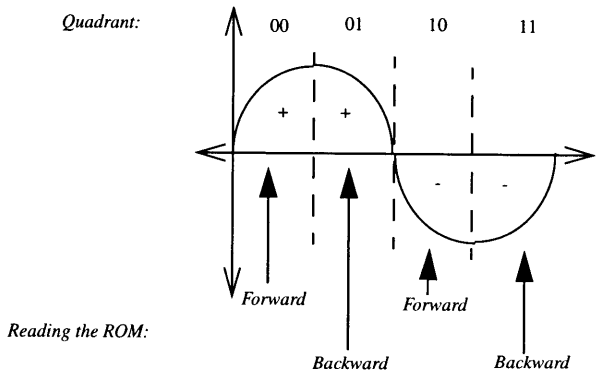

FIGURE 24. Sine Wave Breakdown

Another consideration are the particular data points placed in the ROM. We used Matlab to determine the validity of these points. Take a 16 point sine wave as an example. This translates to 4 points per quadrant.

Using Eq. 3 to pick the ROM points, the first point is always 0; while the last point in the ROM is never 1. The first four discrete points are stored in a ROM. Matlab code took those four points and reconstructed the sine wave based on method from Fig. 15. The top graph in Figure 25 display the generated waveform results. Notice that the repeated zeros create a discontinuity in the sine wave.

$$y(n) = \sin\left(\frac{\Pi}{2} \cdot \frac{n}{N_o}\right) \qquad \text{(EQ 3)}$$

where:
y(n) value of a point in the ROM at the nth address
N_0 number of points in the ROM
n ranges from 0 to (N_0 - 1)

The dotted line is the real sine wave overlaid on the discrete sine wave indicated by the lollipops. The bottom graph in Figure 25 shows the difference between the two curves. The maximum difference between the two curves is as high as 40% with an abrupt jumps around the midway point. This discontinuity must be smoothed out. A periodic discontinuity in the sine and cosine waveforms creates unwanted harmonics in the frequency spectrum. These harmonics can obscure the desired frequency response making it harder to read the output.

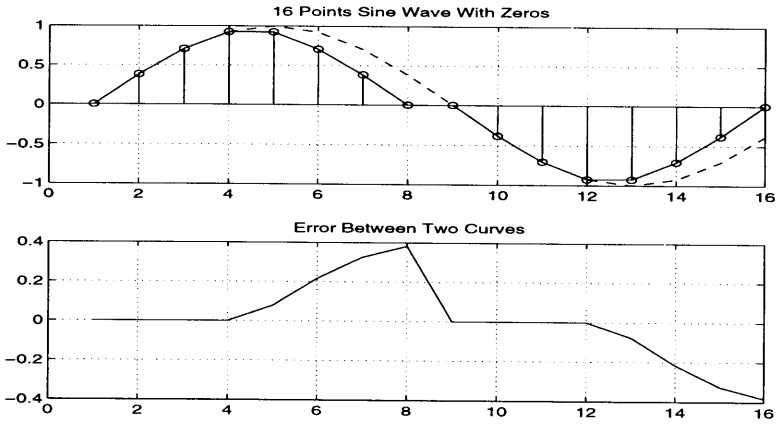

FIGURE 25. Sixteen Point Sine Wave Beginning at Zero

To solve this problem we generated the points for the ROM by excluding the zero. We modified Eq. 3 by adding an offset resulting in Eq. 4. This offset was calculated by dividing the 0 to 1 interval by 2No. This shifted the values for the ROM off the zero value and closer to 1.

$$y(n) = \sin\left(\frac{\Pi}{2} \cdot \frac{n}{N_o} + \frac{\Pi}{2} \cdot \frac{1}{2N_0}\right) \qquad \text{(EQ 4)}$$

Figure 26 below shows the new ROM values calculated with the offset from Eq. 4. Notice that the error function is smoother and that the discontinuities in the discrete sine wave are gone. A slight phase shift is the only difference between the real sine wave and the discrete sine wave.

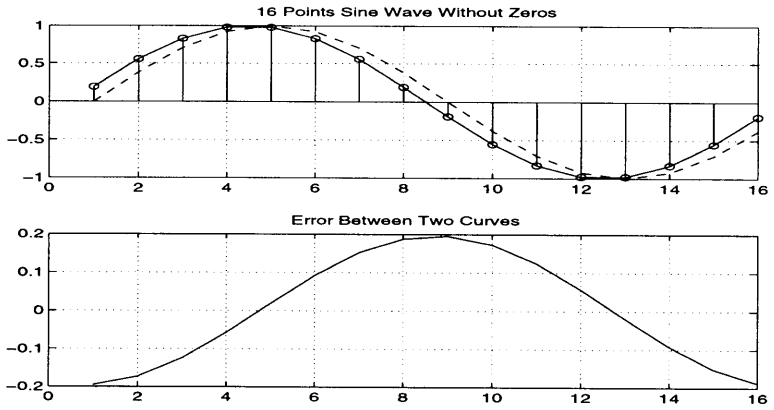

FIGURE 26. Sixteen Point Sine Wave With Offset

Using a larger ROM, the error function will go down as the discrete sine approximation approaches the real sine wave. Figures 27 show 256 point sine wave using 64 point ROM. The maximum error for the zero method is 0.0245 (2.4%); while the maximum error for the offset method is 0.0123 (1.2%). While the errors for both are smaller, the zero method still has that abrupt discontinuity which is undesirable. Based off these Matlab computations, we choose to use the offset method of Eq. 4 to generate the ROM data points.

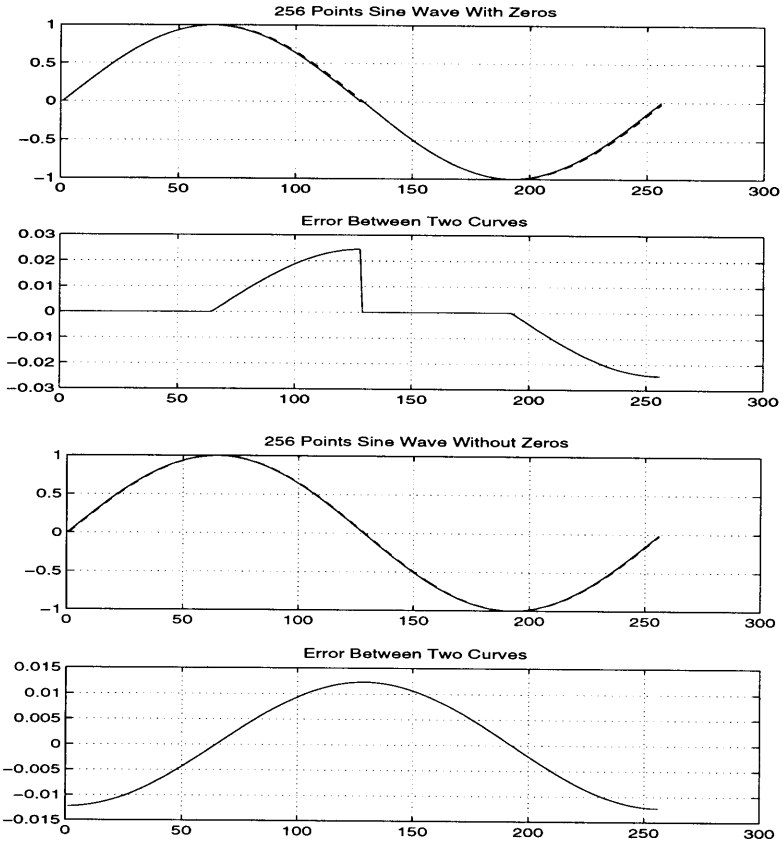

FIGURE 27. 256 Point Sine Wave with Zeros and No Zeros

Once the method was determined, the actual hardware was set up to generate a two's complement sine and cosine wave. The 64x8 ROM was read forward and backwards alternating using the 8 address bits outputted by the NCO (Section 4.3.1). Since the addresses to the ROM are unsigned binary, we can invert the lower 6 address bits <5:0> to read the ROM backwards. The remaining 2 address bits <7:6> are used to keep track of the four quadrants. Exclusive OR's are used to invert the address bits and the data bits going in and out of the ROM's. The data points in the ROM's are unsigned binary. Converting to two's complement requires adding one to every number that is inverted. However, adding 1 to the LSB of an 8 bit number is not going to change the value drastically. Also, it would require another adder stage for that one added bit, so we decided not to add the one. Additionally, adding a one can cause unwanted harmonics in the frequency spectrum. Figure 28 shows the schematic for the sine and cosine ROMs with the extra logic. The bit that inverts the data points to negative was used for the MSB for the sine and cosine bitword. This increases the bitwidth of the sine and cosine waveforms to 9 bits.

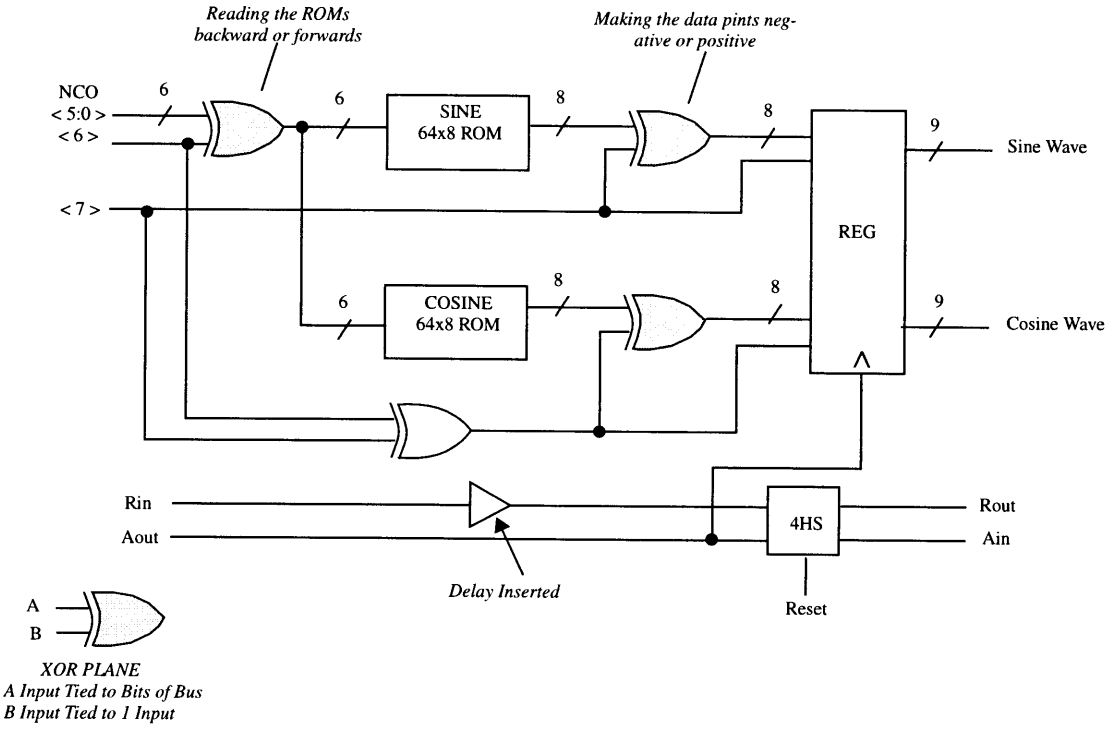

FIGURE 28. Sine and Cosine Generation Schematic with Delay Insertions

Sine and cosine must always be 90 degrees out of phase. To maintain this relation, the first quadrant values for the sine ROM are reversed in the cosine ROM. Reading an same address in the sine ROM and the cosine ROM will result in two data points that are 90 degrees out of phase always. This is why the ROMs in Figure 28 have the same address accessing them at all times. ROM decimal values were generated using Matlab and converted to binary using Perl programs.

Attempts at making this stage self-timed were hindered by the optimization software in Synopsys. We tried to include I and DV signals with duplicate ROMs for sine and cosine. However once optimized through Synopsys, the I and DV signals were connected directly by a wire instead of through combinational logic. Obviously, the DV signal would go high too soon for before the stage completed. The only choice was to insert 6 Gamble delay insertions. This shows that the synthesis optimizer is fallible.

4.4 Interpolation Filters

The interpolation filters change the sampling rate of the I and Q data to the sampling rate of the system. It moves the baseband signal to the new sampling frequency. It also filters out the replicated baseband signal at multiplies of the lower frequency and leaves the replicated baseband signal at multiplies of the higher frequency. Figure 29 displays the schematic for the interpolation filter.

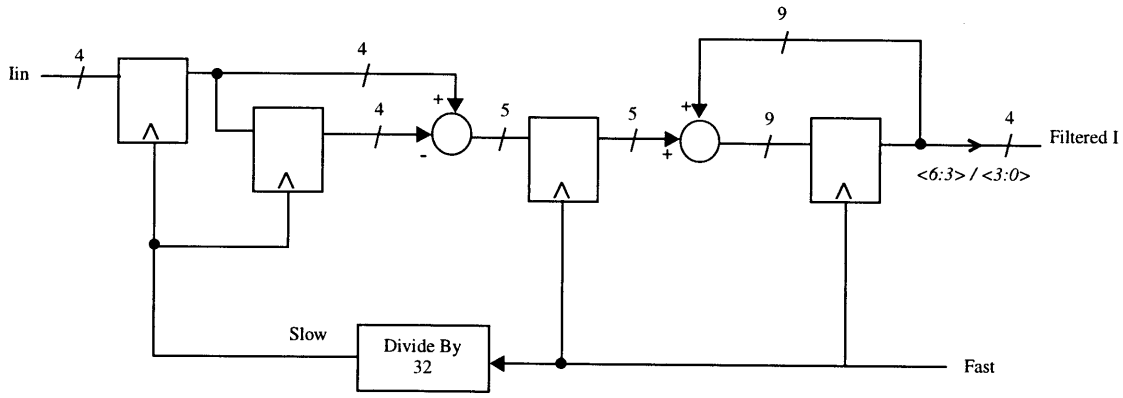

FIGURE 29. Interpolation Filter Schematic

The Fast signal clocks the registers in the back-end of the interpolation filter. The Slow signal is the Fast signal divided down by 32 which clocks the front end registers. Since the data from the interpolation stages and the ROM stages both enter the self-timed multiplier stages, the registers are clocked off the same signal in the request/acknowledge pipeline. The Fast signal is the same clocking signal that goes to the ROM output registers. Eq. 5 shows the system function for the interpolation filters. If Fast was divided by N to produce the Slow signal, then the generic numerator will be [1 - (z^-N)]. The N = 32 was chosen since it was easily implemented in registers.

$$H(z) = \left(\frac{1 - z^{-32}}{1 - z^{-1}}\right)^2 \qquad \text{(EQ 5)}$$

Matlab was used to generate the z-transform from Eq. 4. The zero points in the transform will cancel out the multiples of the lower frequency.

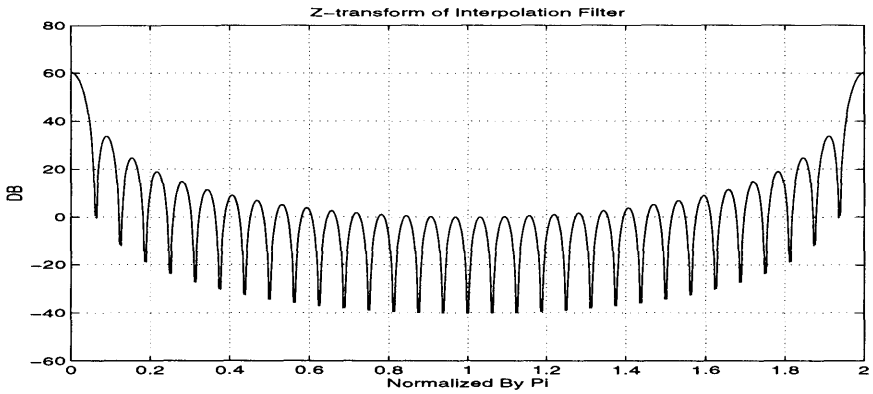

FIGURE 30. Matlab Generated Z-transform of Interpolation Filter

4.5 Input Data Generation: Quadrature Phase-Shift Keying

A data generation technique must be applied to the I and Q inputs to test the asynchronous system. Quadrature phase-shift keying (QPSK) was chosen to test the DDSM system. It is a phase modulation system which utilizes the DDSM setup and incorporates two independent data streams in the same bandwidth. Two data streams of +/-1's make four possible input combinations [(+1,+1);(+1,-1);(-1,+1);(-1,-1)]. How these +/- 1's are multiplied to the sine and cosine of the carrier frequency affects their phase. Adding the two modulated waves together results in one output wave with four potentially different phase configurations. A receiver (although not needed for this thesis) can decode the output to determine the original I and Q data streams from the single waveform input. QPSK will be used to determine the functionality of the DDSM system. Entering a known I and Q data streams into the system, we can check the output for validity against theoretical models done in Matlab.

4.5.1 QPSK Rotation

The easiest way to enter the I/Q data is to cycle through the four possible points in a certain rotation. Visualize the four points as radial points in a circle. Following the clockwise rotation decreases the phase of the resultant waveform, while the counterclockwise rotation increases the phase. The increments are PI/4. Figure 31 shows the phase relationship for QPSK rotation. The needed I and Q input waveforms are shown next to each rotation. Using 2 flip-flops and 1 inverter in a divide-by-4 configuration will generate the required I and Q test data streams.

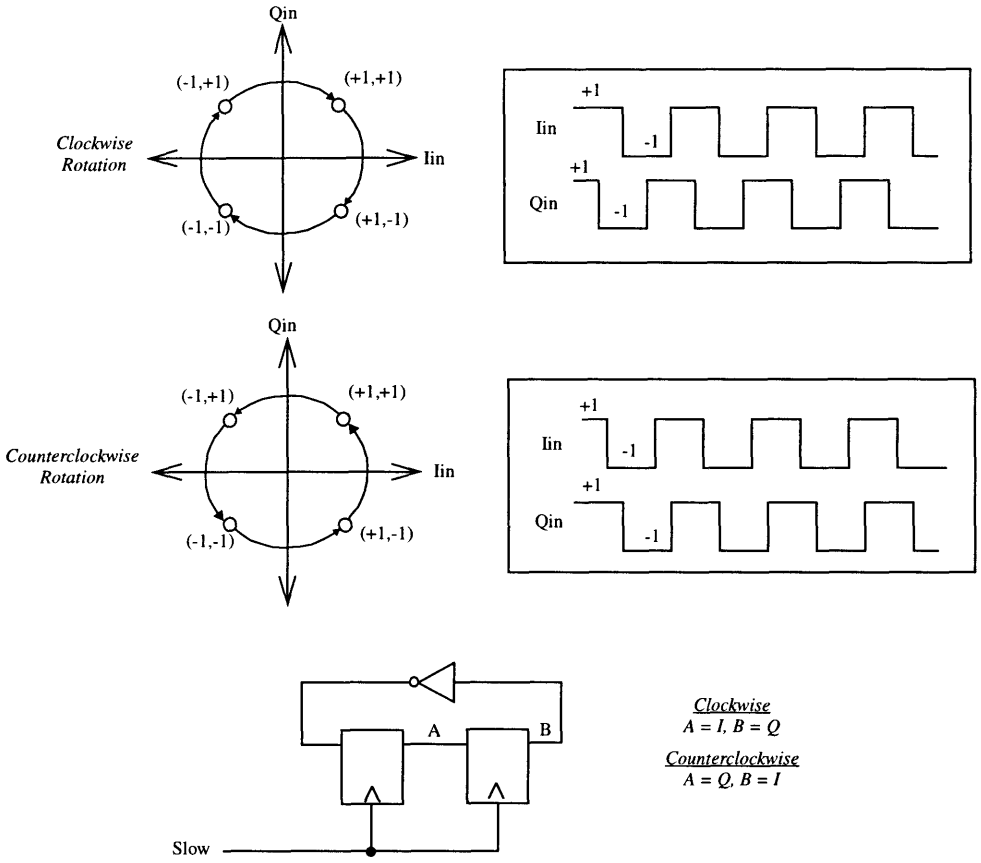

FIGURE 31. QPSK Clockwise and Counterclockwise Rotation

The clockwise (CW) and counterclockwise (CCW) rotations are used to test the DDSM system by comparing the experimental output of the system with the theoretical simulations. The above QPSK system was integrated with a few muxes and the interpolation filters as shown in Figure 33 in the Section 4.5.3. The muxes allow us to switch the rotation.

4.5.2 Random QPSK

Pseudo-random number sequence of length 1024 is generated using a 10-bit shift register and one exclusive OR. One pseudo-random generator drives the input into the I and another into Q. Both generators are similar in structure with different reset words. See Figure 32 for differences. This guarantees that a sufficiently random sequence is going into the interpolation filters.

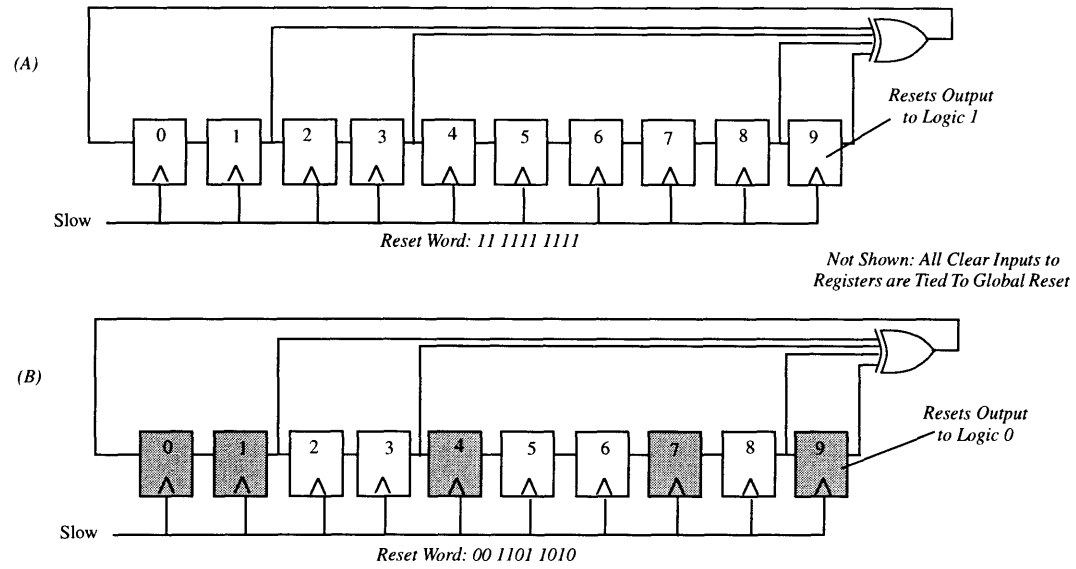

FIGURE 32. Schematic of Pseudorandom Generator

Setup A has a reset word of 11 1111 1111; while setup B has a reset word of 00 1101 1010. When global reset is pulled active, then the flip-flops reset to either 0 or 1 depending on the flip-flop type. Xilinx can provide both types of flip-flops. The VHDL code enables the programmer to specify the reset value for the flip flops. Both setups provide a pseudo-random sequence which repeats after 1024 times.

Each pseudo-random setup, A and B, provides one random bit to the Iin and Qin inputs respectively of the interpolation filters. The I and Q bits are both pulled off the output of the sixth register of each pseudo-random number generator. Randomizing between the four points allow us to visualize the interpolation filter's spectrum modulated around the carrier frequency. Figure 33 in next section shows the block implementation of the system with the interpolation filters.

4.5.3 Integration of the Two QPSK Methods

The QPSK rotation and random method were integrated together into the same design. This allows for easy switching between the two methods. One switch selects the method while another switch determines clockwise or counter-clockwise when the QPSK rotation method is chosen. For this particular interpolation filter, we chose to enter +/-7 into the filters instead. Still the same four possible points but the interpolation filters output waveform is bigger. This is required since we need to truncate the bitword from 9 bits to 4 bits going from the interpolation filters to the self-timed multipliers. Figure 33 shows a the relevant block diagram.

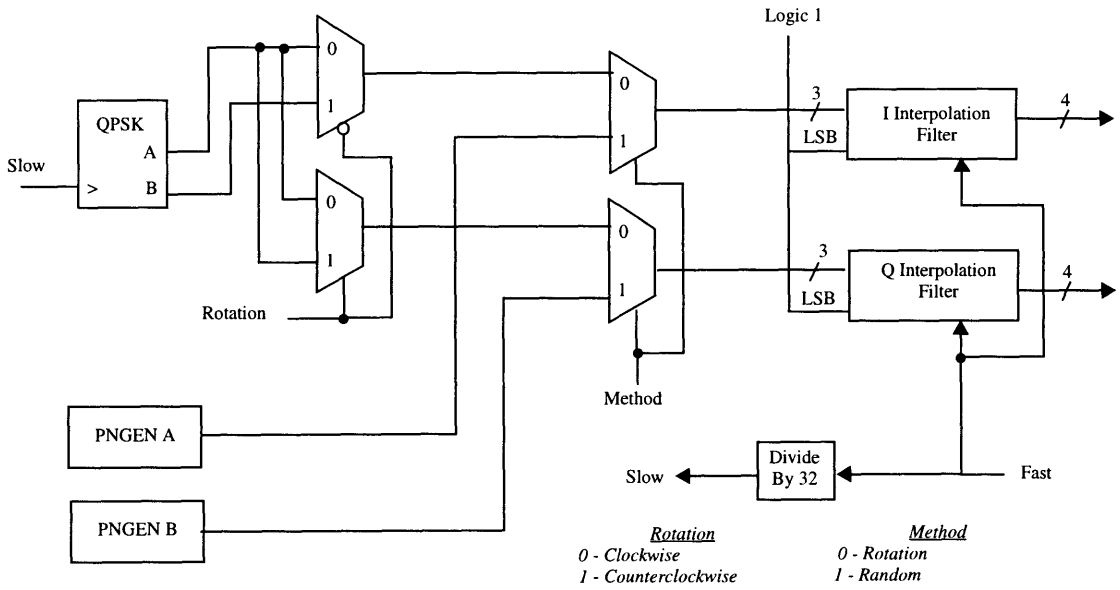

FIGURE 33. QPSK schematic with Interpolation Filters

For a real design, input generation for I and Q would be done off-chip. However, it was easier to program the QPSK modules onto the FPGA and monitor the DDSM system output, instead of trying to build another system and debug both. Our emphasis is on verifying the asynchronous design techniques and not the optimal interpolation filter or QPSK generator.

Chapter 5. Software and Hardware Configurations

5.1 Software Design Flow

The Synopsis Design Analyzer software reads in VHDL programs and can create equivalent gate-level schematics. Ideally, this tool allows the designer to specify a behavioral specification while Synopsis cranks out the gate level translation. Unfortunately this is often not the case. Synopsys can only take a subset of the VHDL language which limits the coding strategies for hardware designs. A good portion of time was spent trying to determine the correct VHDL coding for Synopsis using trial-and-error and reading on-line manuals. While coding basic structures for multipliers and adders were very straightforward using only the "+" and "*" operators, asynchronous feedback such as C-elements caused problems with the software. Synopsys required us to experiment with different C-element VHDL coding until the system accepted the design.

An example of a acceptable and unacceptable VHDL programs is shown below. The entity heading shows the input and outputs for the register. The "Req" input is the clocking input for the flip-flop.

```
-- Module for REGISTER
entity reg is
      port (  d       :in std_logic;
              req     :in std_logic;
              clear   :in std_logic;
              q       :out std_logic);
end reg;
```

The following two architectures show two different implementations for the rising edge flip flop. Synopsis can read the acceptable program (below left) and instantiate a rising edge register into the gate-level schematic. While the unacceptable program (below right) cannot instantiate a generic block which can't be compiled or saved. While both are functionally correct, only one will compile with Synopsys. Typically, it helped to write the VHDL program with the gate level schematic already in mind.

```
-- ACCEPTABLE Rising Edge Flip Flop
architecture reg_logic of reg is
begin
reg:process (req,clear)
 begin
 if (clear = '1') then
            q <= '0';
 elsif (req'event and req = '1') then
            q <= d;
 end process;
end reg_logic;
```

```
--NOT ACCEPTABLE
architecture reg_logic of reg is
begin
reg:process (req,clear)
 begin
 if (req'event and
      req ='1' and
      clear='0') then
            q <= d;
 elsif (clear = '1') then
            q <= 0;
 end process;
end reg_logic;
```

Once the VHDL modules were suitable for Synopsis, the next step was to compile the code for the proper FPGA. Synopsis points at a Xilinx library for the particular family that we are using (XC4000). The Synopsys design flow script for the thesis is shown below. The read statements at the beginning of the file read in the lower hierarchy modules into the design analyzer. The analyze and elaborate commands read in

the VHDL entity and selects the architecture. Every port that is connected to a pin on the FPGA must be specified using the set_port_is_pad command. In this example, all input and output ports will be connected to a pin. Specific to the Xilinx FPGA is the option to set the slew rate on the pins as high or low. Slew rate keeps the outputs from being so noisy. The ungroup commands flattens the hierarchy of the smaller modules to their gate-level designs which is required for successful compile. The rest of the commands deal with the actual compile commands and saving the resulting schematic into a *.SXNF file. The *.SXNF file is the gate level netlist to be read by the Xilinx FPGA software.

```
read -format db {"/home/shen/project/reset/reset_block.db"}
read -format db {"/home/shen/project/front/front64/front64.db"}
read -format db {"/home/shen/project/qpskpart/qpskpart.db"}
read -format db {"/home/shen/project/mst/mst9x4/mst9x4.db"}
read -format db {"/home/shen/project/ast/ast12x12.db"}
read -format db {"/home/shen/project/mux/mux26to13.db"}

analyze -format vhdl -lib WORK {"/home/shen/project/ddsmqpsk/ddsmqpsk.vhd"}
elaborate ddsmqpsk -arch "ddsmqpsk_logic" -lib WORK -update

set_port_is_pad "*"
set_pad_type -slewrate HIGH all_outputs()

ungroup {"re0"}
ungroup {"f0"}
ungroup {"q0"}
ungroup {"m0"}
ungroup {"a0"}
ungroup {"mx0"}

insert_pads
compile -map_effort medium
replace_fpga

write -format db -hierarchy -output "/home/shen/project/ddsmqpsk/ddsmqpsk.db" {"/home/shen/
project/ddsmqpsk.db:ddsmqpsk"}
write -format xnf -hierarchy -output "/home/shen/project/ddsmqpsk/ddsmqpsk.sxnf" {"/home/shen/
project/ddsmqpsk.db:ddsmqpsk"}
```

The XMAKE program was used to transfer the *.SXNF file into a Xilinx place and route file (.LCA) and a download file (.BIT). The download file is used by the Xchecker which is the cable attachment between the HP UNIX Station (RS232 port) and the Xilinx FPGA demonstration board. The demonstration board will be explained more fully in Section 5.2.

The design size was a concern early on in the project. There was a possibility that we needed to fit the digital system onto two XC4010e-2 FPGAs instead of one. Luckily that was not required. The entire design was placed on one XC4010e-2 FPGA. The table below tells the final figures for the size the thesis design. The number of combinational logic blocks (CLBs) utilized appear to be fully used. However, the usage of the F',G', and H' function generators inside each CLB shows a more accurate usage of the FPGA. For future applications in FPGAs, users should use these numbers to correctly estimate the size of the FPGA needed.

TABLE 4. Partitioned Design Utilization Using Part 4010EPC84-2 for DDSMQPSK.VHD

	No. Used	Max Available	%Used
Occupied Combination Logic Blocks (CLB's)	398	400	99%
Bonded I/O Pins	54	61	88%
F and G Function Generators	572	800	**71%**
H Function Generators	105	400	**26%**
CLB Flip Flops	166	800	20%
CLB Fast Carry Logic	43	400	10%

5.2 FPGA Demonstration Board Setup

An Xilinx FPGA Demonstration Board was used to test the programmed FPGA for this thesis. The schematic and layout of the demo board were taken from the Xilinx Hardware Peripheral Guide. The schematic is shown in Figure 34. The FPGA demo board has sockets for a XC3000 chip and a XC4000 chip. The XC4000 socket is for a 84 pin PLCC package. Based on this size restriction, the largest FPGA that can be placed on the board is a XC4010 with 400 CLBs and 61 I/O ports. And for ease of programming, the entire digital design was placed on a XC4000 chip. Therefore, the XC3000 socket was not used at all.

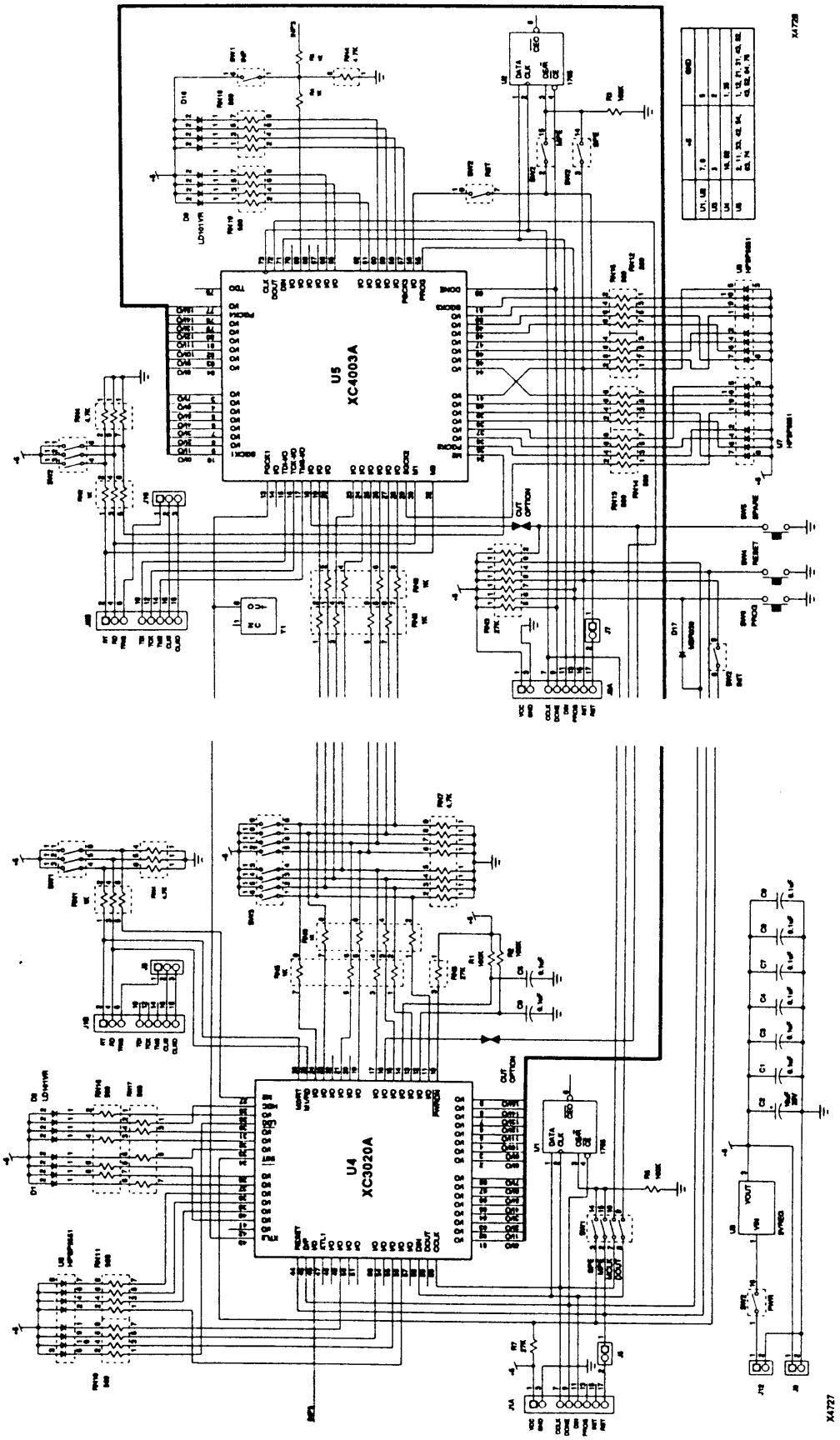

FIGURE 34. FPGA Demonstration Board Schematic

Since the XC3000 socket is unused, we can utilize the some of the configuration switches as input switches to the XC4000 chip. After removing the XC3000 chip, we were able to use 4 switches from SW1 and wire them to I/O pins to the XC4000 socket. (SW1 is a set of 8 switches in the upper left-hand corner of the FPGA demonstration board.) The physical layout of the board is shown in Figure 35. This gave us a total of 12 input switches for the FPGA. Ten of the I/O pins are reserved for the digital-to-analog convertor inputs. The rest of the pins are used by the logic analyzer for debugging purposes.

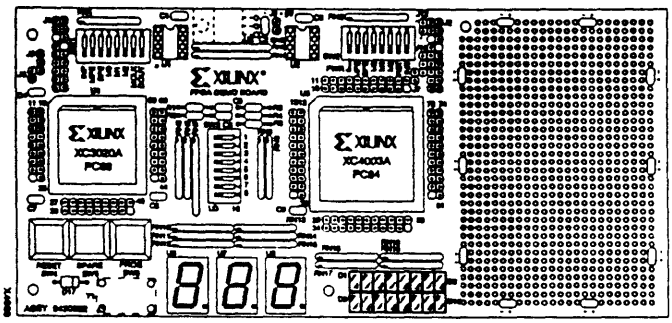

FIGURE 35. FPGA Demonstration Board Physical Layout

The reset button (SW4) was used on the FPGA board. It resets all the internal flip-flops to logic zero. This was useful for resetting the asynchronous control signalling path. The LED's for the board are hard-wired to some I/O pins. LED's flashing were an indication that the asynchronous pipeline was functioning. No flashing was an indication that the control signalling pipeline was stopped somewhere along the line. When there is no output from an asynchronous pipeline, this is good indicator that something is wrong with the system unlike synchronous systems which could spew out invalid data continuously.

Using a premade demonstration board cuts down on implementation time for the thesis. Building a separate board would have taken time away from the design and testing of the thesis.

5.3 Digital to Analog Convertor Board Setup

The DDSM 10 bit output word was inputted to a digital-to-analog convertor that was placed on a separate PC board. The resulting analog output was placed into a spectrum analyzer for analysis. The spectrum analyzer allows us to easily view the frequency components of the output.

Setting up the D/A convertor PC board was specific to the 40 pin ADV7121 device (Analog Devices Video 10-Bit DAC). This device contains three 10-bit DACs for color video purposes. For our purposes, we only used one DAC and tied the inputs to the other two to ground. The PC board setup follows almost straight from the specifications with a few variations. Figure 36 shows a schematic of the board. Notice that the analog ground is tied to digital ground by one connection through a ferrite bead. Same with analog power to digital power. A co-axial connection port to the analog board is not pictured in this schematic. This

was for ease of attaching a co-axial cable from the D/A to the spectrum analyzer and oscilloscope. There is a 75 ohm resistor for termination.

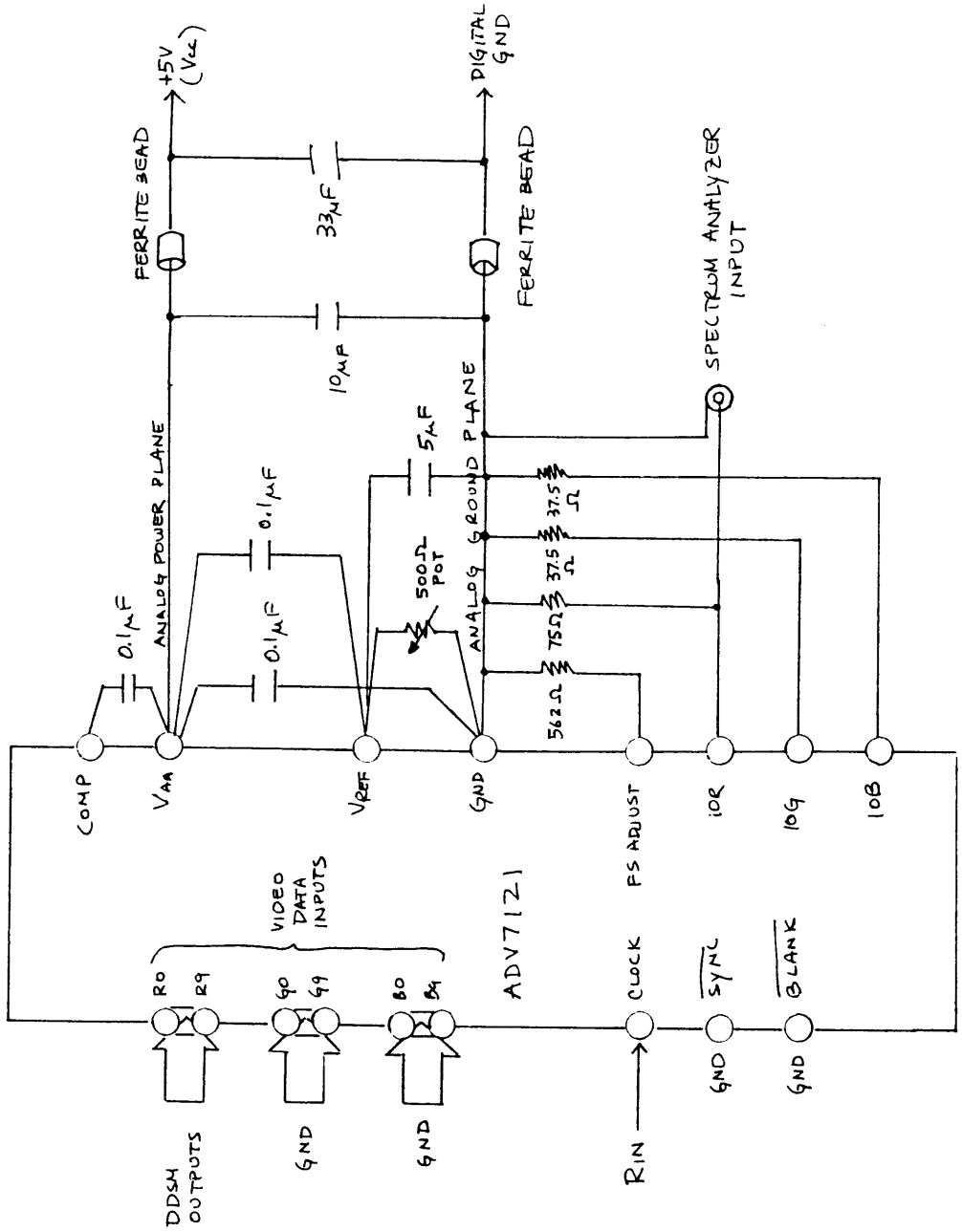

FIGURE 36. Schematic Layout of D/A convertor board

Chapter 6. Results and Verification

Verifying the functionality of the frequency generator was the first step in debugging the system. Section 6.1 elaborated on this analysis. Once verified, the full DDSM was assembled and tested. The lab results were compared to theoretical models done in Matlab (Section 6.2 and 6.3). Additional observations of the completed system were detailed in Section 6.4.

6.1 DDS Outputs: Sine and Cosine Generation

Using a logic analyzer, the actual data points of the sine and cosine waveform were captured. These were read into Matlab to graph the carrier and to examine its spectrum.

According to Eq. 2 in Section 4.3, setting the Frequency word to 1111 1111 and running the system at 14.5 MHz should generate a carrier frequency at 1.8054 MHz. Using a spectrum analyzer, the actual measured frequencies from the output was 1.8050 MHz. From this experiment, Eq. 2 which calculates the carrier frequency proves to be exact. The experimental data was captured by the logic analyzer; and the hard data which we will designate as Case A was reformatted and placed into Matlab.

The left Matlab graph in Figure 37 shows the generated sine and cosine waveforms for Case A. The right graph shows an overlay of the FFT of the experimentally generated signal (solid line) and the FFT of the theoretically derived carrier signal (dotted). The peaks are aligned at the same frequency and DB level.

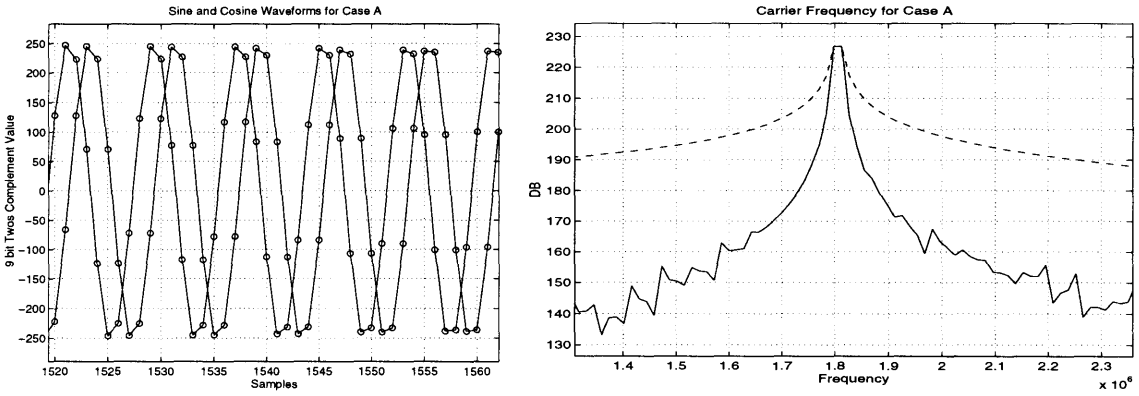

FIGURE 37. Generated Sine and Cosine Waves at 1.8 MHz

Notice the smoothness in the tails of the ideal spectrum peak in Figure 37 in contrast to the jaggedness around the experimental spectrum peak. How do we explain this noisiness? Remember that the NCO and ROM stages were implemented with inserted delays in the control signalling pipeline. This creates the possibility that the wrong carrier points can be captured. These erroneous data points in the frequency domain equates to the jaggedness that we see in Figure 37. To produce a smoother spectrum, we need to perform a shorter FFT on the carrier signal. Using a shorter FFT increases the chances that the carrier points are accurate. Figure 38 illustrates the difference in neglecting bad points in the data. The 1024-PT includes some

erroneous data which causes the noisiness. The shorter FFT was taken from a correct sequence of data points which is why it is less jagged in comparison. Less discontinuities mean less unwanted harmonics in the frequency spectrum as mentioned in Section 4.3.2.

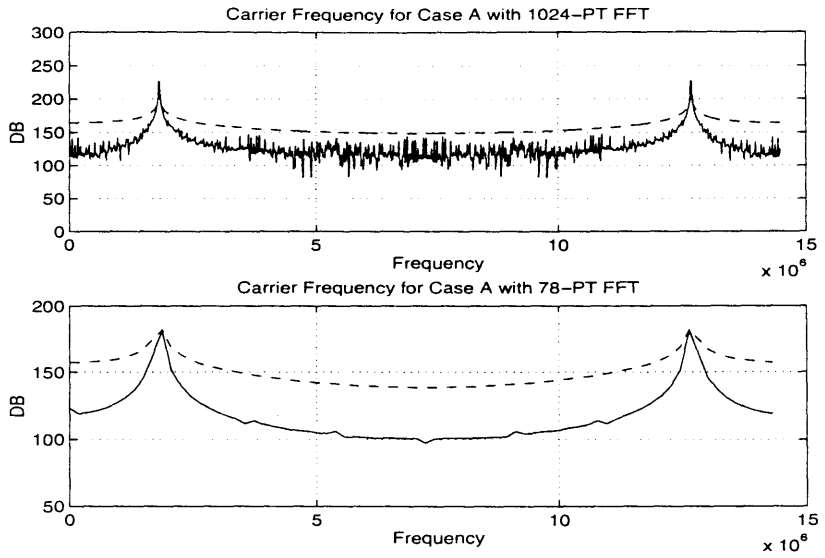

FIGURE 38. 1024-PT FFT vs. 78-PT FFT Carrier Frequency For Case A

While the tails of waveform appear noisy around the carrier frequency, the DB level is much lower than its ideal counterpart's tails. This ensures that the signal-to-noise ratio is still high. A Polaroid of the carrier spectrum is shown in Figure 39. This was taken from the spectrum analyzer which is attached to the output of the digital-to-analog convertor.

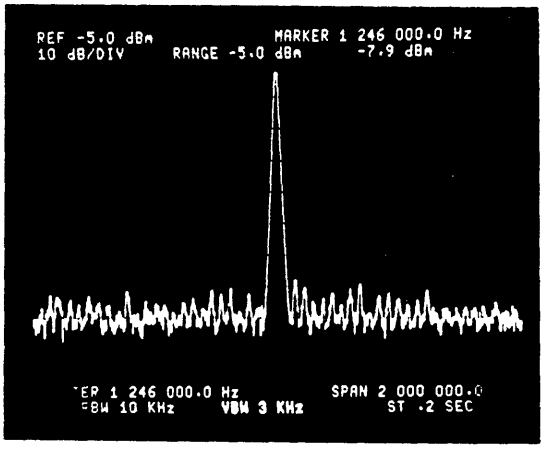

FIGURE 39. Polaroid Showing the Generated Carrier Wave Spectrum

6.2 QPSK

After verifying the functionality of the frequency generator, we can now verify the rest of the system. First, let's examine the theory behind the system and figure out what to expect from the output. A theoretical model was built in Matlab. In Figure 30, two FFT's are displayed for clockwise (CW) and counterclockwise (CCW) respectively. These particular graphs are generated for Case A as described in the previous section.

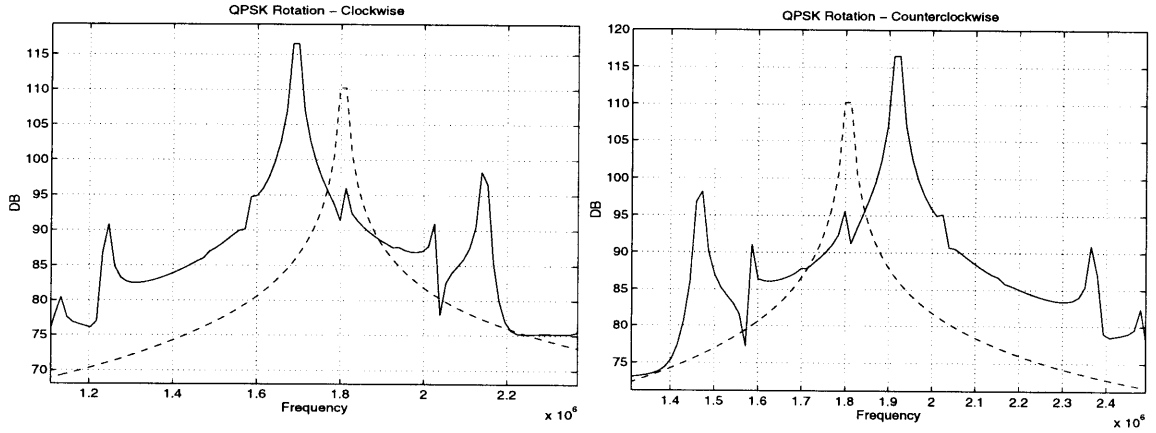

FIGURE 40. Theoretical Output Spectra of the CW and CCW for Case A

These graphs show the theoretical output from the Matlab model. Both waveforms are centered around the carrier frequency shown with a dotted line. Notice that difference between CW and CCW is that the 1st (largest peak) and 3rd harmonics (second largest peak) are flipped around the center frequency. Notice that the first harmonic is left of the carrier frequency peak in the CW graph; while in the CCW graph, it is to the right of the carrier frequency peak. The same applies to the 3rd harmonic but in reverse. Looking at these spectra, is this what we expect?

Why are the 1st and 3rd harmonic flipped around the carrier frequency? This is easily derived from graph below. Think of the rotation input of Q and I which are just periodic signals 90 degrees phase shifted. Drawing in the 1st and 3rd harmonics for both Q and I shows the relations when you add the two waveforms. One can see below that the 1st harmonic for I is leading the 1st harmonic for Q (counterclockwise). However, notice that the 3rd harmonic waveform for I is lagging behind the 3rd harmonics for Q. If the I and Q labels were switched (clockwise), then the 3rd harmonic would be leading and the 1st harmonic would be lagging. This leading and lagging of the harmonics are centered around the carrier freqeuncy during the modulation. This explains the phenomena of the spectrum peaks as seen in the Figure 39.

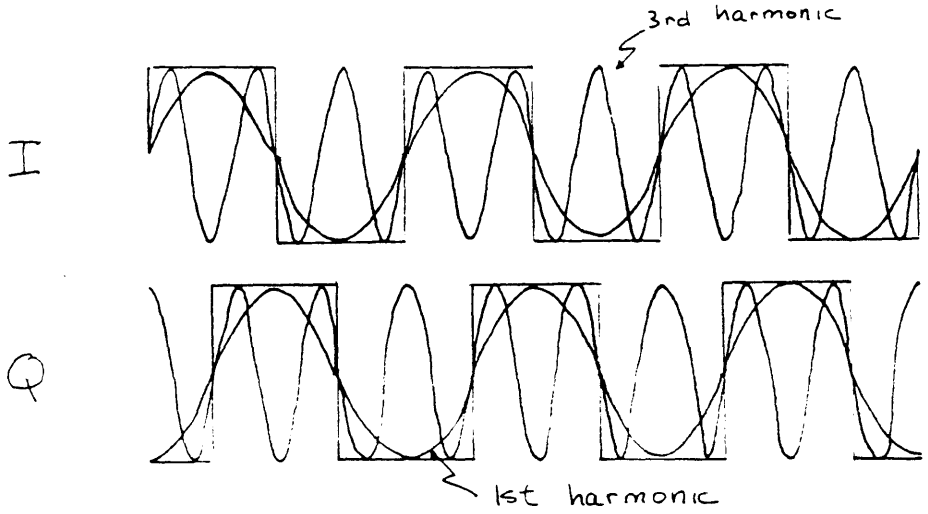

FIGURE 41. Leading and Lagging First and Third Harmonics

Now that we understand the expected output of the systems, let see the experimental results to check our functionality. Generating the same 1.8050 MHz carrier frequency for the system, the system data was captured in two ways. Using the logic analyzer, the system output datastream was placed on disk and massaged in Matlab to see the spectrum output. The other method was to use a digital-analog convertor and output to a spectrum analyzer. A Polaroid scope camera was used to capture the spectrum. These two methods both prove the functionality of the system for QPSK rotation. Figure 42 below show the Matlab generated spectrums clockwise and counterclockwise. Figure 43 shows the corresponding polaroid.

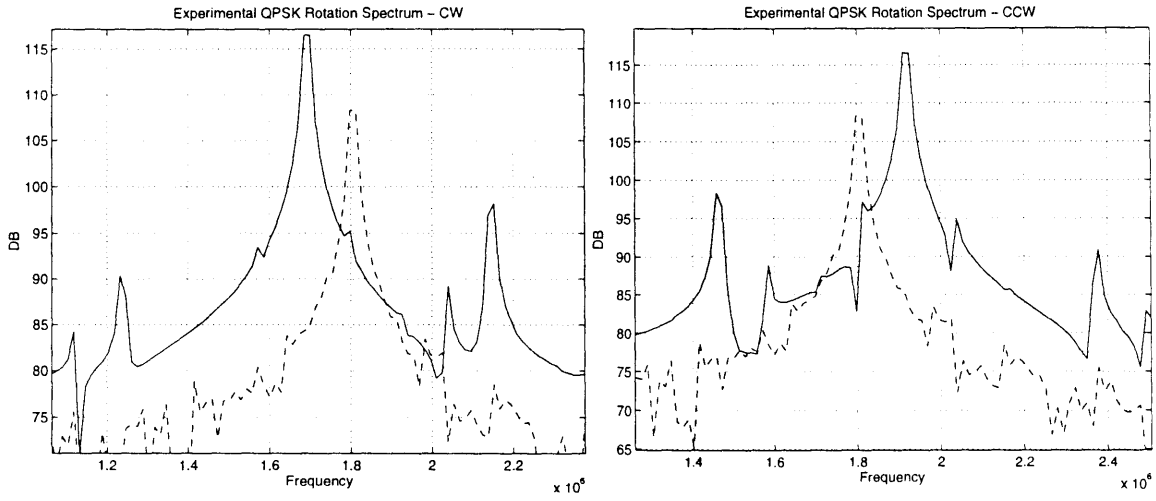

FIGURE 42. Matlab Generated Spectrums of the QPSK output CW, CCW

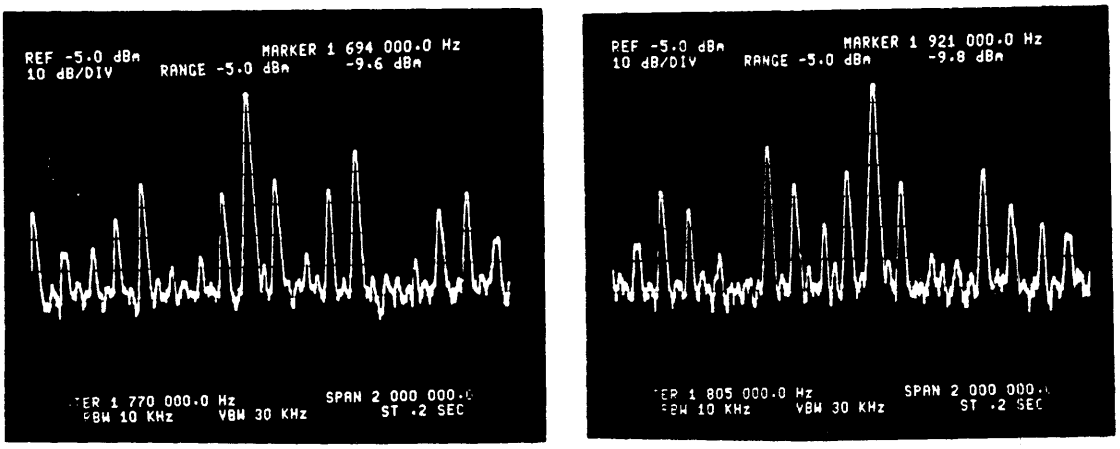

FIGURE 43. Polaroid Showing the Spectrums of the QPSK output CW, CCW.

6.3 Random QPSK

Randomizing the inputs into the I and Q as described in section 4.5.2 will cause the system to output something different from the rotation experiments done previously. Since there is random data going into the system, then the interpolation filter spectrum (as seen Figure 30) should be seen on the spectrum analyzer of the output.

The system was run at 14.2 MHz with a Frequency word of 1111 1111. The generated carrier frequency was measured at 1.770 MHz in contrast to the theoretically derived 1.7681 MHz. Figure 43 below shows the Polaroid shot of the spectrum of the system output. It definitely matches the Z-transform of the interpolation filter's theoretical spectrum.

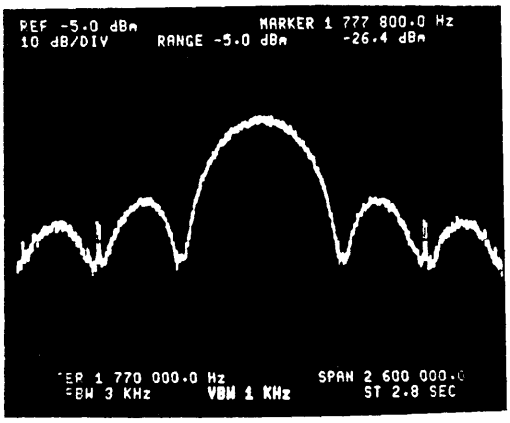

FIGURE 44. Polaroid of the Spectrum Output for Random QPSK

6.4 System Observations

The results from Section 6.1-3 proves the functionality of the asynchronous DDSM system. The actual system was tested from 10 MHz up to 15 MHz which worked for the whole range. However, some frequencies were noisier than others on the spectrum analyzer. One would guess that the 15 MHz would have the noisiest output and 10 MHz would not. This was not the case meaning that linearly increasing the system clock does not linearly increase the noise. Viewing the spectrum on the logic analyzer while increasing the system clock a local minimum was found for noise in the operating frequency range. The best rotation spectrum with low noise and high DB was found at an operating frequency of 14.5 MHz. While the best frequency for random QPSK was 14.2 MHz. This explains why those particular carrier frequencies were chosen as examples in earlier sections.

Power consumption for this part would not beat its synchronous counterpart for a few reasons. Since DDSM is a digital signal processing application, the asynchronous system is clocked synchronously to ensure that a regular output to the digital-to-analog convertor. Running it continuously does not take advantage of the system's possible power savings. A better application would be a digital compact cassette error corrector which does not require continuous use [11]. A power test should be averaged over time a long period of time for better apples-to-apples results between a synchronous system versus an asynchronous system. Another reason for bad power consumption is the use of the FPGA to implement the design. Since we were not dealing at a transistor level, the implementations of the self-timed modules required twice as many adders and multipliers on the FPGA. This would definitely increase power consumption.

Chapter 7. Conclusion

While asynchronous design is still seen as a research topic, some examples of its influence are being felt in the industry. Sharp has just recently introduced a video processor with eight self-timed cores as a product [16]. The EE Times newspaper article highlights the fact that asynchronous technology has begun the transition from the research lab to the commercial marketplace. In the past few years, international conferences and dedicated web sites have sprung up to discuss the latest theories and applications. These examples illustrate the gathering momentum in the asynchronous design community where research is increasing.

Given the large interest in asynchronous design, how do the results of this thesis impact the field? Before this research, only a 4x4 self-timed multiplier had ever been implemented on a FPGA [6]. As proven in Chapter 6, a large asynchronous digital signal processing application can work functionally on the same medium. The ease of implementing asynchronous pipeline designs has definitely been demonstrated. A possible follow-up project would be a full custom chip implementing asynchronous pipelines. This will allow a better feeling of power consumption and layout constraints of asynchronous designs.

An article in IEEE Transactions by Ravi Ramachandran and Shih-Lien Lu discusses a self-time arithmetic module they designed. Their asynchronous module needed an interface between itself and to synchronous structures. Once generic interfaces are designed, asynchronous structures can replace synchronous structures in already made clocked IC chips. The easy module replacement is a another benefit for asynchronous design. This can be considered a gradual method of conversion of synchronous chips to asynchronous.

The actual implementation of this project and design flow were other benefits from this thesis. Using VHDL to model the asynchronous logic proved adequate for our needs. In addition, a technique to produce self-timed circuits in a FPGA overcame the need for inserted delays in the FPGA. Currently, an asynchronous FPGA called MONTAGE [2] is being developed to provide designers with a new medium for implementation. With all this knowledge- new and old, future asynchronous designs can be implemented in a FPGA medium for quicker results.

References

Literature

[1] S. Furber, P. Day, J. Garside, N. Paver, and J. Woods. "A Micropipelined ARM" Dept. of Computer Science. Manchester University. UK 10 pages.

[2] S. Hauck, S.Burns, G.Borriello, and C.Ebeling. "An FPGA for Implementing Asynchronous Circuits." IEEE Design and Test of Computers, Vol. 11 No. 3, 1994.

[3] S. Hauck, "Asynchronous Design Methodologies: An Overview" in Proceedings of the IEEE Vol. 83, No.1, pp. 69-93, January 1995.

[4] I.E. Sutherland, "Micropipelines" in Communications of the ACM, Vol. 32, No. 6, p. 720-738, June 1989.

[5] G. Jacobs, R. Brodersen."A Fully-Asynchronous Digital Signal Processor Using Self-Timed Circuits" in IEEE International Solid State Circuits Vol. 25, No. 6, p. 1526-37. December 1990.

[6] M. Gamble, B. Rahardjo, R.D. McLeod. "Reconfigurable FPGA Micropipelines" Technical Report of University of Manitoba, 7 pages, 1994.

[7] S. Lu, L. Merani, "Micro Data Flow" in Proceedings of the 5th Annual IEEE International ASIC Con ference and Exhibit. p. 301-4.

[8] M. Renaudin, B. El Hassan, and A. Guyot. "A New Asynchronous Pipeline Scheme: Application to the Design of a Self-Timed Ring Divider." IEEE Journal of Solid-State Circuits, Vol. 31, No. 7, p. 1001-1013, July 1996.

[9] T. Meng, R. Brodersen, D. Messerchmitt. "Automatic Synthesis of Asynchronous Circuits from high-Level Specifications" in IEEE Transactions on Computer-Aided Design, Vol. 8, No. 11, p. 1185-205. Nov. 1989.

[10] Xilinx. The Programmable Logic Data Book. 1994.

[11] K Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, F. Schaliji, "A Fully Asynchronous Low-Power Error Corrector for the DCC Player" in IEEE Journal of Solid-State Circuits, VOL. 29, No. 12, p. 1429-39, Dec. 1994.

[12] S. Lu, L. Merani, "Micro Data Flow" in Proceedings of the 5th Annual IEEE International ASIC Conference and Exhibit. p. 301-4.

[13] A. Oppenheim, R. Schafer. Discrete-Time Signal Processing. Prentice Hall, New Jersey. p. 109, 1989.

[14] Xilinx. XC4000 Series Field Programmable Gate Arrays. Version 1.02, June 1996.

[15] R.W. Lucky, J. Salz, E. Weldon Jr. *Principles of Data Communication*. McGraw-Hill Book Company. New York, 1968.

[16] Junko Yoshida. "Sharp's Processor Beats the Clock: Data Driven Architecture Leaves the Lab" in *Electronic Engineering Times*, p. 1,6. November 25, 1996.

[17] Ravi Ramachandran, Shih-Lien Lu. "Efficient Arithmetic Using Self-Timing" in IEEE Transactions on VLSI Systems, Vol. 4, No. 4. p. 445-54. December 1996.

Web Sites

These are references for web sites.

[W1] Asynchronous Logic Home Page. http://maveric0.uwaterloo.ca:80/amulet/async/

[W2] Asynchronous Software Tools (synthesizing and/or verification). http://www.ee.umanitoba.ca/~rahard/asynctools.html

[W3] Self-Timed FPGAs. http://www.dcs.ed.ac.uk/home/rep/selfTimedFPGA/selfTimedFPGA.html

[W4] Direct Digital Synthesis Explanations. http://arts.ucsc.edu/EMS/Music/tech_background/TE-17/teces_17.html

[W5] Direct Digital Synthesis Explanations. http://www.batnet.com/srsys/ddsnote.html

Appendices

VHDL Code for Modules

File: ddsmfinal.vhd

```
-- Direct Digital Synthesis Modulator With QPSK Rotation/Random Input
-- Contains the  Front64, Qpskpart, MST9x4
-- Start Date: January 14, 1997

-- Modifications:
-- 11/18/96 - Convert two's complement output to unsigned
-- binary by inverting the MSB.
-- 1/8/97 - Combined both qpsk parts into one. Method switches
-- between the two.
-- 1/14/97 - Copied from ddsmqpsk.vhd and modified by outputting
-- carrier 9 bits - sin.

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity ddsmfinal is
    port (  freq            :in std_logic_vector(7 downto 0);
            ddsm_out        :inout std_logic_vector(12 downto 0);
            d2abits         :out std_logic_vector(9 downto 0);
            sin             :inout std_logic_vector(8 downto 0);
            mwave           :out std_logic_vector(12 downto 0);
            wavesel         :in std_logic;
            reset_button    :in std_logic;
            rin,ain          :in std_logic;
            rout            :inout std_logic;
            rot,method      :in std_logic);
    end ddsmfinal;

architecture ddsmfinal_logic of ddsmfinal is
  component reset_block
      port (button    :in std_logic;
            reset     :inout std_logic);
  end component;
  component front64
      port (freq      in std_logic_vector(10 downto 0);
            rin,ain   :in std_logic;
            aout,rout :inout std_logic;
            reset     :in std_logic;
            sine,cos  :out std_logic_vector(8 downto 0);
            fast      :inout std_logic);
  end component;
  component qpskpart
      port (ilong,qlong:inout std_logic_vector(8 downto 0);
            rot       :in std_logic;
            method    :in std_logic;
            fast      :in std_logic;
            reset     :in std_logic);
  end component;
  component mst9x4
      port (rin,ain     :in std_logic;
            aout,rout   :inout std_logic;
            reset       :in std_logic;
            sin,cos     :in std_logic_vector(8 downto 0);
            I,Q         :in std_logic_vector(3 downto 0);
            msin,mcos:out std_logic_vector(12 downto 0));
  end component;
  component ast12x12
      port (A         :in std_logic_vector(11 downto 0);
            B         :in std_logic_vector(11 downto 0);
            Sum       :out std_logic_vector(12 downto 0);
            reset     :in std_logic;
            rin,ain   :in std_logic;
            aout,rout :inout std_logic);
  end component;
  component mux26to13
      port (A         :in std_logic_vector(12 downto 0);
            B         :in std_logic_vector(12 downto 0);
            Q         :out std_logic_vector(12 downto 0);
            sel       :in std_logic);
  end component;
  signal aout :std_logic;
  signal reset,fast :std_logic;
  signal a,b,c,d :std_logic;
  signal cos :std_logic_vector(8 downto 0);
  signal I,Q :std_logic_vector(3 downto 0);
  signal ilong,qlong :std_logic_vector(8 downto 0);
```

```
   signal freqexp :std_logic_vector(10 downto 0);
   signal msin,mcos : std_logic_vector(12 downto 0);
begin
      -- reset circuitry
      re0: reset_block port map (reset_button, reset);

      -- Expanding the Frequency Input
      freqexp(10 downto 8) <= "000";
      freqexp(7 downto 0) <= freq(7 downto 0);

      f0: front64 port map (freqexp,rin,b,aout,a,reset,sin,cos,fast);

      -- Reducing Ilong, Qlong
      I(3 downto 0) <= ilong (8 downto 5);
      Q(3 downto 0) <= qlong (8 downto 5);
      q0: qpskpart port map (ilong,qlong,rot,method,fast,reset);

      -- Multiplication
      m0: mst9x4 port map (a,d,b,c,reset,sin,cos,I,Q,msin,mcos);

      -- Selecting Modulated Waveforms out of the Multplier
      mx0: mux26to13 port map (msin,mcos,mwave,wavesel);

      -- Reducing Msin and Mcos
      a0: ast12x12 port map (msin(11 downto 0),mcos(11 downto0), ddsm_out, reset,
         c,ain,d,rout);

      -- Converting Two's Complement to Unsigned Binary
      d2abits(8 downto 0) <= ddsm_out(11 downto 3);
      d2abits(9) <= not(ddsm_out(12));
end ddsmfinal_logic;
```

File: front64.vhd

```
-- Front end with NCO and AROM16x8 (9 bit sine and cosine waves)
-- Start Date: October 1, 1996

-- 6 Delay

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity front64 is
      port (  freq              :in std_logic_vector(10 downto 0);
              rin,ain           :in std_logic;
              aout,rout         :inout std_logic;
              reset             :in std_logic;
              sine,cos          :out std_logic_vector(8 downto 0);
              fast              :inout std_logic);
end front64;

architecture front64_logic of front64 is
   component reset_block
         port (button    :in std_logic;
               reset     :inout std_logic);
   end component;
   component arom64
         port (addr      :in std_logic_vector(7 downto 0);
               sine,cos  :out std_logic_vector(8 downto 0);
               rin,ain   :in std_logic;
               rout,aout :inout std_logic;
               reset     :in std_logic;
               fast      :inout std_logic);
   end component;
   component nco
         port (rin,ain   :in std_logic;
               rout,aout :inout std_logic;
               reset     :in std_logic;
               freq      :in std_logic_vector(10 downto 0);
               index     :inout std_logic_vector(7 downto 0));
   end component;
```

```
    signal g,h:std_logic;
    signal addr :std_logic_vector(7 downto 0);

begin
    -- NCO
    n0: nco port map (rin,h,g,aout,reset,freq,addr);

    -- AROM64
    a0: arom64 port map (addr,sine,cos,g,ain,rout,h,reset,fast);
end front64_logic;
```

File: nco.vhd

```
-- Numerically Controlled Oscilator (NCO)
-- Start Date: September 30, 1996

-- Modified 10/24/96 for a finer control.
-- Reset Block is Commented out
-- Modified 10/28/96 Adder24 down to Adder14 to reduce CLB's

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
-- use work.CONV_PACK_add.all;

entity nco is
    port (  rin,ain             :in std_logic;
            rout,aout           :inout std_logic;
            reset               :in std_logic;
            freq                :in std_logic_vector(10 downto 0);
            index               :inout std_logic_vector(7 downto 0));
end nco;

architecture nco_logic of nco is
  component reset_block
      port (button    :in std_logic;
            reset     :inout std_logic);
  end component;
  component ADD_SUB_TWO_COMP_14
      port(C_IN, ADD_SUB :in std_logic;
           B, A     : in std_logic_vector (0 to 13);
           FUNC     : out std_logic_vector (0 to 13));
  end component;
  component reg
      port (d       :in std_logic;
            req     :in std_logic;
            clear   :in std_logic;
            q       :out std_logic);
  end component;
  component cg4
  port (rin   :in std_logic;
```

```
            ain         :in std_logic;
            reset       :in std_logic;
            rout        :nout std_logic;
            aout        :inout std_logic;
            req         :out std_logic);
    end component;
    component delay_c2
        port (g0        :in std_logic;
            g1          :inout std_logic;
            reset       :in std_logic);
    end component;
    signal g,t :std_logic;
    signal logic1,logic0 :std_logic;
    signal indexexp,freqexp,step :std_logic_vector(13 downto 0);
begin
    logic1 <= '1';
    logic0 <= '0';

    freqexp(13 downto 11) <= "000";
    freqexp(10 downto 0) <= freq(10 downto 0);

    -- adders
    a0: ADD_SUB_TWO_COMP_14 port map (logic0, logic1,index-
      exp,freqexp,step);

    -- async. control and registers for nco
    cg0: cg4 port map (g,ain,reset,rout,aout,t);
    d0: delay_c2 port map (rin,g,reset);

    r0: reg port map(step(0),t,reset,indexexp(0));
    r1: reg port map(step(1),t,reset,indexexp(1));
    r2: reg port map(step(2),t,reset,indexexp(2));
    r3: reg port map(step(3),t,reset,indexexp(3));
    r4: reg port map(step(4),t,reset,indexexp(4));
    r5: reg port map(step(5),t,reset,indexexp(5));
    r6: reg port map(step(6),t,reset,indexexp(6));
    r7: reg port map(step(7),t,reset,indexexp(7));
    r8: reg port map(step(8),t,reset,indexexp(8));
    r9: reg port map(step(9),t,reset,indexexp(9));
    r10: reg port map(step(10),t,reset,indexexp(10));
    r11: reg port map(step(11),t,reset,indexexp(11));
    r12: reg port map(step(12),t,reset,indexexp(12));
    r13: reg port map(step(13),t,reset,indexexp(13));

    index(7 downto 0) <=  indexexp(10 downto 3);
end nco_logic;
```

File: arom64.vhd

```
-- AROM - XOR plane with Rom 16x8 with XOR PLANE
-- pipeline slice
-- Determine the DELAY needed in the control signals
-- Start Date: October 15, 1996

-- 2 Delay

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity arom64 is
    port (  addr            :in std_logic_vector(7 downto 0);
            sine,cos        :out std_logic_vector(8 downto 0);
            rin,ain         :in std_logic;
            rout,aout       :inout std_logic;
            reset           :in std_logic;
            fast            :inout std_logic);
    end arom64;

architecture arom64_logic of arom64 is
  component rom64x8
      port (addr    :in std_logic_vector(5 downto 0);
            data    :out std_logic_vector(7 downto 0));
  end component;
  component rom64x8c
      port (addr    :in std_logic_vector(5 downto 0);
            data    :out std_logic_vector(7 downto 0));
  end component;
  component xorp6
      port (A       :in std_logic_vector(5 downto 0);
            X       :out std_logic_vector(5 downto 0);
            I       :in std_logic);
  end component;
  component xorp8
      port (A       :in std_logic_vector(7 downto 0);
            X       :out std_logic_vector(7 downto 0);
            I       :in std_logic);
  end component;
  component cg4
      port (rin     :in std_logic;
            ain     :in std_logic;
            reset   :in std_logic;
            rout    :inout std_logic;
            aout    :inout std_logic;
            req     :out std_logic);
  end component;
  component delay_c6
      port (g0      :in std_logic;
            g1      :inout std_logic;
            reset   :in std_logic);
  end component;
  component reg
      port (d       :in std_logic;
            req     :in std_logic;
            clear   :in std_logic;
            q       :out std_logic);
  end component;
  signal g,t :std_logic;
  signal fa :std_logic_vector(5 downto 0);
  signal rds,rdc :std_logic_vector(7 downto 0);
  signal ids,idc :std_logic_vector(7 downto 0);
  signal datab :std_logic_vector(7 downto 0);
  signal flip :std_logic;
  signal invert_sine,invert_cos :std_logic;
begin
    flip <= addr(6);
    invert_sine <= addr(7);
    invert_cos <= addr(6) xor addr(7);

    -- xorplane into the rom
    x0: xorp6 port map(addr(5 downto 0),fa,flip);

    -- address bits into ROM16x8
    m0: rom64x8 port map (fa,rds);
    m1: rom64x8c port map (fa,rdc);
```

```
    -- xorplane out of the rom
    x1: xorp8 port map(rds,ids,invert_sine);
    x2: xorp8 port map(rdc,idc,invert_cos);

    -- delay of the pipeline
    d0: delay_c6 port map (rin,g,reset);

    -- end of pipeline
    cg0: cg4 port map (g,ain,reset,rout,aout,t);

    -- sine output
    r0: reg port map (ids(0),t,reset,sine(0));
    r1: reg port map (ids(1),t,reset,sine(1));
    r2: reg port map (ids(2),t,reset,sine(2));
    r3: reg port map (ids(3),t,reset,sine(3));
    r4: reg port map (ids(4),t,reset,sine(4));
    r5: reg port map (ids(5),t,reset,sine(5));
    r6: reg port map (ids(6),t,reset,sine(6));
    r7: reg port map (ids(7),t,reset,sine(7));
    r8: reg port map (invert_sine,t,reset,sine(8));

    -- cosine output
    r9: reg port map (idc(0),t,reset,cos(0));
    r10: reg port map (idc(1),t,reset,cos(1));
    r11: reg port map (idc(2),t,reset,cos(2));
    r12: reg port map (idc(3),t,reset,cos(3));
    r13: reg port map (idc(4),t,reset,cos(4));
    r14: reg port map (idc(5),t,reset,cos(5));
    r15: reg port map (idc(6),t,reset,cos(6));
    r16: reg port map (idc(7),t,reset,cos(7));
    r17: reg port map (invert_cos,t,reset,cos(8));

    fast <= t;

end arom64_logic;
```

File: rom64x8.vhd

```
-- ROM data generated with the offset method (ie. with peaks)
-- Start Date: August 26, 1996

-- program modified to take std_logic
-- integer conversion done here
-- ******************************************************
library IEEE;
use IEEE.std_logic_1164.all;
package CONV_PACK_add is

-- define attributes
attribute ENUM_ENCODING : STRING;

  -- Declarations for conversion functions.
  function integer_to_unsigned(arg, size : in INTEGER) return std_logic_vector;
  function unsigned_to_integer(arg : in std_logic_vector) return INTEGER;
  function unsigned_bit_to_integer(arg : in std_logic) return INTEGER;

end CONV_PACK_add;

package body CONV_PACK_add is

  -- integer type to std_logic_vector function
  function integer_to_unsigned(arg, size : in INTEGER) return std_logic_vector
  is
    variable result: std_logic_vector(size-1 downto 0);
    variable temp: INTEGER;
    -- synopsys built_in SYN_INTEGER_TO_UNSIGNED
  begin
    temp := arg;
    for i in 0 to size-1 loop
      if (temp mod 2) = 1 then
        result(i) := '1';
      else
        result(i) := '0';
      end if;
      temp := temp / 2;
    end loop;
```

```
    return result;
  end;

  -- std_logic_vector to integer type function
  function unsigned_to_integer(arg : in std_logic_vector) return INTEGER is
    variable result: INTEGER;
    -- synopsys built_in SYN_UNSIGNED_TO_INTEGER;
  begin
    result := 0;
    for i in arg'range loop
      result := result * 2;
      if arg(i) = '1' then
        result := result + 1;
      end if;
    end loop;
    return result;
  end;

  -- std_logic bit to integer type function
  function unsigned_bit_to_integer(arg : in std_logic) return INTEGER is
    variable result: INTEGER;
    -- synopsys built_in SYN_UNSIGNED_TO_INTEGER;
  begin
    result := 0;
    if (arg = '1') then
      result := 1;
    end if;
    return result;
  end;

end CONV_PACK_add;
-- ***************************************************

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
use work.CONV_PACK_add.all;

entity rom64x8 is
    port (  addr                :in std_logic_vector(5 downto 0);
            data                :out std_logic_vector(7 downto 0));
end rom64x8;

architecture rom64x8_behav of rom64x8 is
    signal ia: integer range 0 to 63;
    subtype rom_word is std_logic_vector(7 downto 0);
    type rom_table is array (0 to 63) of rom_word;
    constant rom: rom_table := rom_table'(
            rom_word'("00000011"),
            rom_word'("00001001"),
            rom_word'("00001111"),
            rom_word'("00010101"),
            rom_word'("00011100"),
            rom_word'("00100010"),
            rom_word'("00101000"),
            rom_word'("00101110"),
            rom_word'("00110101"),
            rom_word'("00111011"),
            rom_word'("01000001"),
            rom_word'("01000111"),
            rom_word'("01001101"),
            rom_word'("01010011"),
            rom_word'("01011001"),
            rom_word'("01011111"),
            rom_word'("01100100"),
            rom_word'("01101010"),
            rom_word'("01110000"),
            rom_word'("01110101"),
            rom_word'("01111011"),
            rom_word'("10000000"),
            rom_word'("10000110"),
            rom_word'("10001011"),
            rom_word'("10010000"),
            rom_word'("10010101"),
            rom_word'("10011011"),
            rom_word'("10011111"),
            rom_word'("10100100"),
            rom_word'("10101001"),
```

```
        rom_word'("10101110"),
        rom_word'("10110010"),
        rom_word'("10110111"),
        rom_word'("10111011"),
        rom_word'("10111111"),
        rom_word'("11000011"),
        rom_word'("11000111"),
        rom_word'("11001011"),
        rom_word'("11001111"),
        rom_word'("11010011"),
        rom_word'("11010110"),
        rom_word'("11011001"),
        rom_word'("11011101"),
        rom_word'("11100000"),
        rom_word'("11100011"),
        rom_word'("11100110"),
        rom_word'("11101000"),
        rom_word'("11101011"),
        rom_word'("11101101"),
        rom_word'("11101111"),
        rom_word'("11110010"),
        rom_word'("11110100"),
        rom_word'("11110101"),
        rom_word'("11110111"),
        rom_word'("11111001"),
        rom_word'("11111010"),
        rom_word'("11111011"),
        rom_word'("11111100"),
        rom_word'("11111101"),
        rom_word'("11111110"),
        rom_word'("11111111"),
        rom_word'("11111111"),
        rom_word'("11111111"),
        rom_word'("11111111"));

begin
    ia <=INTEGER(unsigned_to_integer(addr));
    data <= rom(ia);
end rom64x8_behav;
```

File: rom64x8c.vhd

```
-- ROM data generated with the offset method (ie. with peaks)
-- Start Date: October 3, 1996

-- Integer Conversion in ~/project/romrom64x8/rom64x8.vhd
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
use work.CONV_PACK_add.all;

entity rom64x8c is
    port (  addr            :in std_logic_vector(5 downto 0);
            data            :out std_logic_vector(7 downto 0));
end rom64x8c;

architecture rom64x8c_behav of rom64x8c is
    signal ia: integer range 0 to 63;
    subtype rom_word is std_logic_vector(7 downto 0);
    type rom_table is array (0 to 63) of rom_word;
    constant rom: rom_table := rom_table'(
            rom_word'("11111111"),
            rom_word'("11111111"),
            rom_word'("11111111"),
            rom_word'("11111111"),
            rom_word'("11111110"),
            rom_word'("11111101"),
            rom_word'("11111100"),
            rom_word'("11111011"),
            rom_word'("11111010"),
            rom_word'("11111001"),
            rom_word'("11110111"),
            rom_word'("11110101"),
            rom_word'("11110100"),
            rom_word'("11110010"),
            rom_word'("11101111"),
            rom_word'("11101101"),
            rom_word'("11101011"),
            rom_word'("11101000"),
            rom_word'("11100110"),
```

```
rom_word'("11100011"),
rom_word'("11100000"),
rom_word'("11011101"),
rom_word'("11011001"),
rom_word'("11010110"),
rom_word'("11010011"),
rom_word'("11001111"),
rom_word'("11001011"),
rom_word'("11000111"),
rom_word'("11000011"),
rom_word'("10111111"),
rom_word'("10111011"),
rom_word'("10110111"),
rom_word'("10110010"),
rom_word'("10101110"),
rom_word'("10101001"),
rom_word'("10100100"),
rom_word'("10011111"),
rom_word'("10011011"),
rom_word'("10010101"),
rom_word'("10010000"),
rom_word'("10001011"),
rom_word'("10000110"),
rom_word'("10000000"),
rom_word'("01111011"),
rom_word'("01110101"),
rom_word'("01110000"),
rom_word'("01101010"),
rom_word'("01100100"),
rom_word'("01011111"),
rom_word'("01011001"),
rom_word'("01010011"),
rom_word'("01001101"),
rom_word'("01000111"),
rom_word'("01000001"),
rom_word'("00111011"),
rom_word'("00110101"),
rom_word'("00101110"),
rom_word'("00101000"),
rom_word'("00100010"),
rom_word'("00011100"),
rom_word'("00010101"),
rom_word'("00001111"),
rom_word'("00001001"),
rom_word'("00000011"));

begin
    ia <=INTEGER(unsigned_to_integer(addr));
    data <= rom(ia);
end rom64x8c_behav;
```

File: qpskpart.vhd

```
-- QPSK Rotation / QPSK Random attached to Interpolation Filters
-- Start Date: January 8,1997

-- Note:
-- ilong,qlong <8:5> <-> <3:0>

--*******************************
library IEEE;
use IEEE.std_logic_1164.all;

package CONV_PACK_mult is

-- define attributes
attribute ENUM_ENCODING : STRING;

  -- Declarations for conversion functions.
  function integer_to_signed(arg, size : in INTEGER) return std_logic_vector;
  function signed_to_integer(arg : in std_logic_vector) return INTEGER;

end CONV_PACK_mult;

package body CONV_PACK_mult is
  -- integer type to std_logic_vector function
  function integer_to_signed(arg, size : in INTEGER) return std_logic_vector
  is
    variable result: std_logic_vector(size-1 downto 0);
    variable temp: INTEGER;
    -- synopsys built_in SYN_INTEGER_TO_SIGNED
  begin
    temp := arg;
    if temp < 0 then
      result(size-1) := '1';
      -- make the number positive so /2 is a shift.
      temp := temp + (2**30);
      -- for really big numbers need to add more.
      if size = 32 then
        temp := temp + (2**30);
      end if;
    else
      result(size-1) := '0';
    end if;

    for i in 0 to size-2 loop
      if (temp mod 2) = 1 then
        result(i) := '1';
      else
        result(i) := '0';
      end if;
      temp := temp / 2;
    end loop;
    return result;
  end;

  -- std_logic_vector to integer type function
  function signed_to_integer(arg : in std_logic_vector) return INTEGER is
    variable result: INTEGER;
    -- synopsys built_in SYN_SIGNED_TO_INTEGER;
  begin
    result := 0;
    for i in arg'range loop
      if i = arg'left then
        next;
      end if;
      result := result * 2;
      if arg(i) = '1' then
        result := result + 1;
      end if;
    end loop;
    if arg(arg'left) = '1' then
      if arg'length = 32 then
        result := (result - 2**30) - 2**30;
      else
        result := result - (2 ** (arg'length-1));
      end if;
    end if;
    return result;
  end;
```

```
end CONV_PACK_mult;

--*******************************
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
use work.CONV_PACK_mult.all;

entity qpskpart is
      port (  ilong,qlong :inout std_logic_vector(8 downto 0);
              rot                    :in std_logic;
              method                 :in std_logic;
              fast                   :in std_logic;
              reset                  :in std_logic);
end qpskpart;

architecture qpskpart_logic of qpskpart is
   component interp
         port (Iin       :in integer range -8 to 7;
               Qin       :in integer range -8 to 7;
               Ilong     :inout integer range -256 to 255;
               Qlong     :inout integer range -256 to 255;
               fast      :in std_logic;
               slow      :inout std_logic;
               reset     :in std_logic);
   end component;
   component div4
         port (clk       :in std_logic;
               reset     :in std_logic;
               I,Q       :inout std_logic);
   end component;
   component mux2to1
         port (A         :in std_logic;
               B         :in std_logic;
               Q         :out std_logic;
               sel       :in std_logic);
   end component;
   component pngen
         port (bits      :out std_logic_vector(9 downto 0);
               req       :in std_logic;
               clear     :in std_logic);
   end component;
   component pngen1
         port (bits      :out std_logic_vector(9 downto 0);
               req       :in std_logic;
               clear     :in std_logic);
   end component;
   signal logic0,logic1 :std_logic;
   signal iin,qin :std_logic_vector(3 downto 0);
   signal iin_int, qin_int: integer range -8 to 7;
   signal aa,bb,cc,dd,ee,ff: std_logic;
   signal ilong_int, qlong_int: integer range -256 to 255;
   signal slow,dumslow,rotbar: std_logic;
   signal xx,yy :std_logic_vector(9 downto 0);
begin
      logic0 <= '0';
      logic1 <= '1';

      -- (0) QPSK Rotation
      d0: div4 port map (slow,reset,aa,bb);
      m0: mux2to1 port map (aa,bb,cc,rotbar);
      m1: mux2to1 port map (aa,bb,dd,rot);
      rotbar <= not(rot);

      -- (1) QPSK Random
      p0: pngen port map(xx,slow,reset);
      p1: pngen1 port map(yy,slow,reset);

      -- Selection of the QPSK method
      m2: mux2to1 port map (cc,xx(6),ee,method);
      m3: mux2to1 port map (dd,yy(6),ff,method);

      -- Vector Inputs
      iin(0)<= logic1;
      iin(1)<= ee;
      iin(2)<= ee;
      iin(3)<= not(ee);
```

```
    qin(0)<= logic1;
    qin(1)<= ff;
    qin(2)<= ff;
    qin(3)<= not(ff);

    -- std_logic_vector converion to integer
    iin_int <= signed_to_integer(iin);
    qin_int <= signed_to_integer(qin);

    -- I
    i0: interp port map (iin_int,qin_int,ilong_int,qlong_int,fast,slow,reset);

    -- integer conversion to std_logic_vector
    ilong <= integer_to_signed(ilong_int,9);
    qlong <= integer_to_signed(qlong_int,9);
end qpskpart_logic;
```

File: interp.vhd

```
-- Interpolation Filter - 9 bit outputs (I and Q)
-- Start Date: October 11th, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity interp is
    port (  Iin             :in integer range -8 to 7;
            Qin             :in integer range -8 to 7;
            Ilong           :inout integer range -256 to 255;
            Qlong           :inout integer range -256 to 255;
            fast            :in std_logic;
            slow            :inout std_logic;
            reset           :in std_logic);
end interp;

architecture interp_logic of interp is
  component regint4
      port (d       :in integer range  -8 to 7;
            req     :in std_logic;
            clear   :in std_logic;
            q       :out integer range -8 to 7);
  end component;
  component regint5
      port (d       :in integer range -16 to 15;
            req     :in std_logic;
            clear   :in std_logic;
            q       :out integer range -16 to 15);
  end component;
  component regint9
      port (d       :in integer range -256 to 255;
            req     :in std_logic;
            clear   :in std_logic;
            q       :out integer range -256 to 255);
  end component;
  component div32
      port (fast    :in std_logic;
```

```
            reset       :in std_logic;
            slow        :inout std_logic);
  end component;
  signal AQ,AI,BQ,BI:integer range -8 to 7;
  signal CQ,CI,DQ,DI :integer range -16 to 15;
  signal EQ,EI :integer range -256 to 255;
begin
    -- conversion of fast to slow
    d0: div32 port map (fast,reset,slow);

    -- I
    -- slow clock
    ri0: regint4 port map (Iin,slow,reset,AI);
    ri1: regint4 port map (AI,slow,reset,BI);

    --adders
    CI <= AI - BI;
    ri3: regint5 port map (CI,fast,reset,DI);
    EI <= DI + Ilong;

    -- fast clock
    ri2: regint9 port map (EI,fast,reset,Ilong);


    -- Q
    -- slow clock
    rq0: regint4 port map (Qin,slow,reset,AQ);
    rq1: regint4 port map (AQ,slow,reset,BQ);

    --adders
    CQ <= AQ - BQ;
    rq3: regint5 port map (CQ,fast,reset,DQ);
    EQ <= DQ + Qlong;

    -- fast clock
    rq2: regint9 port map (EQ,fast,reset,Qlong);

end interp_logic;
```

File: div4.vhd

```
-- Divide by 4 using flip-flops
-- Start Date: October 18, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity div4 is
    port (  clk             :in std_logic;
            reset           :in std_logic;
            I,Q             :inout std_logic);
end div4;


architecture div4_logic of div4 is
  component reg
      port (d       :in std_logic;
            req     :in std_logic;
            clear   :in std_logic;
            q       :out std_logic);
  end component;
  signal qb: std_logic;
begin
    --div4
    r1: reg port map (qb,clk,reset,I);
    r2: reg port map (I,clk,reset,Q);
    qb <= not (Q);
end div4_logic;
```

File: div32.vhd

```
-- Divide by 32 using flip-flops .Start Date: October 18, 1996
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity div32 is
      port (  fast                    :in std_logic;
              reset                   :in std_logic;
              slow                    :inout std_logic);
end div32;

architecture div32_logic of div32 is
   component reg
         port (d          :in std_logic;
              req         :in std_logic;
              clear       :in std_logic;
              q           :out std_logic);
   end component;
   signal d:std_logic_vector(16 downto 1);
begin
      r1: reg port map (d(1),fast,reset,d(2));
      r2: reg port map (d(2),fast,reset,d(3));
      r3: reg port map (d(3),fast,reset,d(4));
      r4: reg port map (d(4),fast,reset,d(5));
      r5: reg port map (d(5),fast,reset,d(6));
      r6: reg port map (d(6),fast,reset,d(7));
      r7: reg port map (d(7),fast,reset,d(8));
      r8: reg port map (d(8),fast,reset,d(9));
      r9: reg port map (d(9),fast,reset,d(10));
      r10: reg port map (d(10),fast,reset,d(11));
      r11: reg port map (d(11),fast,reset,d(12));
      r12: reg port map (d(12),fast,reset,d(13));
      r13: reg port map (d(13),fast,reset,d(14));
      r14: reg port map (d(14),fast,reset,d(15));
      r15: reg port map (d(15),fast,reset,d(16));
      r16: reg port map (d(16),fast,reset,slow);
      d(1) <= not (slow);
end div32_logic;
```

File: pngen.vhd

```
-- Pseudo-Random Number Generator
-- Start Date: November 7th, 1996

-- Modified:
-- 12/4/96: changed pngen to repeat every 1024
-- (originally 256)

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity pngen is
      port (  bits                    :out std_logic_vector(9 downto 0);
              req                     :in std_logic;
              clear                   :in std_logic);
end pngen;

architecture pngen_logic of pngen is
   signal d :std_logic_vector(9 downto 0);
   signal x :std_logic;
begin
pngen: process (req,clear)
    begin
      if (clear = '1') then
              d(9 downto 0) <= "1111111111";
      elsif (req'event and req = '1') then
              d(9 downto 1) <= d(8 downto 0);
              d(0) <= x;
      end if;
    end process;
bits <= d;
x <= d(1) xor d(3) xor d(8) xor d(9);
end pngen_logic;
```

File: pngen1.vhd

```
-- Pseudo-Random Number Generator
-- Start Date: November 25th, 1996

-- Modified:
-- 12/4/96: changed pngen to repeat every 1024
-- (originally 256)

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity pngen1 is
      port (  bits              :out std_logic_vector(9 downto 0);
              req               :in std_logic;
              clear             :in std_logic);
end pngen1;

architecture pngen1_logic of pngen1 is
   signal d :std_logic_vector(9 downto 0);
   signal x :std_logic;
begin
pngen1: process (req,clear)
    begin
      if (clear = '1') then
              d(9 downto 0) <= "0011011010";
      elsif (req'event and req = '1') then
              d(9 downto 1) <= d(8 downto 0);
              d(0) <= x;
      end if;
    end process;
bits <= d;
x <= d(1) xor d(3) xor d(8) xor d(9);
end pngen1_logic;
```

File: mst9x4.vhd

```
-- Self_timed Multiplier With HandSaking Signals and Resgisters
-- Start Date: October 17, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity mst9x4 is
      port (  rin,ain           :in std_logic;
              aout,rout         :inout std_logic;
              reset             :in std_logic;
              sin,cos           :in std_logic_vector(8 downto 0);
              I,Q               :in std_logic_vector(3 downto 0);
              msin,mcos         :out std_logic_vector(12 downto 0);
              sdv,cdv,mdv       :inout std_logic);
end mst9x4;

architecture mst9x4_logic of mst9x4 is
   component mult_st9x4a
         port (I        :in std_logic;
               dv       :out std_logic;
               A        :in std_logic_vector(8 downto 0);
               B        :in std_logic_vector(3 downto 0);
               Prod     :out std_logic_vector(12 downto 0));
   end component;
   component cg4
         port ( rin     :in std_logic;
               ain      :in std_logic;
               reset    :in std_logic;
               rout     :inout std_logic;
               aout     :inout std_logic;
               req      :out std_logic);
   end component;
   component reg
         port (d        :in std_logic;
               req      :in std_logic;
               clear    :in std_logic;
               q        :out std_logic);
```

```
  end component;
  signal sdv,cdv,mdv :std_logic;
  signal t :std_logic;
  signal ms,mc :std_logic_vector(12 downto 0);
begin
    -- self-timed multiplier
     -- sine modulator
    ms0: mult_st9x4a port map (rin,sdv,sin,I,ms);
     -- cosine modulator
    ms1: mult_st9x4a port map (rin,cdv,cos,Q,mc);
     -- data valid for both multipliers
    mdv <= sdv and cdv;

    -- Handshaking
    c0: cg4 port map (mdv,ain,reset,rout,aout,t);

    -- Registers
    -- modulated sine
    rs0: reg port map (ms(0),t,reset,msin(0));
    rs1: reg port map (ms(1),t,reset,msin(1));
    rs2: reg port map (ms(2),t,reset,msin(2));
    rs3: reg port map (ms(3),t,reset,msin(3));
    rs4: reg port map (ms(4),t,reset,msin(4));
    rs5: reg port map (ms(5),t,reset,msin(5));
    rs6: reg port map (ms(6),t,reset,msin(6));
    rs7: reg port map (ms(7),t,reset,msin(7));
    rs8: reg port map (ms(8),t,reset,msin(8));
    rs9: reg port map (ms(9),t,reset,msin(9));
    rs10: reg port map (ms(10),t,reset,msin(10));
    rs11: reg port map (ms(11),t,reset,msin(11));
    rs12: reg port map (ms(12),t,reset,msin(12));

    -- modulated cosine
    rc0: reg port map (mc(0),t,reset,mcos(0));
    rc1: reg port map (mc(1),t,reset,mcos(1));
    rc2: reg port map (mc(2),t,reset,mcos(2));
    rc3: reg port map (mc(3),t,reset,mcos(3));
    rc4: reg port map (mc(4),t,reset,mcos(4));
    rc5: reg port map (mc(5),t,reset,mcos(5));
    rc6: reg port map (mc(6),t,reset,mcos(6));
    rc7: reg port map (mc(7),t,reset,mcos(7));
    rc8: reg port map (mc(8),t,reset,mcos(8));
    rc9: reg port map (mc(9),t,reset,mcos(9));
    rc10: reg port map (mc(10),t,reset,mcos(10));
    rc11: reg port map (mc(11),t,reset,mcos(11));
    rc12: reg port map (mc(12),t,reset,mcos(12));

end mst9x4_logic;
```

File: mult_st9x4a.vhd

```
-- Self_timed Multiplier 9x4 2's complement with I and DV
-- Start Date: October 17, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
use work.CONV_PACK_mult.all;

entity mult_st9x4a is
     port (  I                  :in std_logic;
             dv                 :out std_logic;
             A                  :in std_logic_vector(8 downto 0);
             B                  :in std_logic_vector(3 downto 0);
             Prod               :out std_logic_vector(12 downto 0));
end mult_st9x4a;

architecture mult_st9x4a_logic of mult_st9x4a is
  component mult9x4
       port (A        :in integer range -256 to 255;
             B        :in integer range -8 to 7;
             Prod     :out integer range -4096 to 4095);
  end component;
  signal pprod,npd,nprod,dprod:std_logic_vector(12 downto 0);
  signal aip,ain:std_logic_vector(8 downto 0);
  signal bip,bin:std_logic_vector(3 downto 0);
  signal pprod_int, npd_int:integer range -4096 to 4095;
  signal aip_int, ain_int: integer range -256 to 255;
  signal bip_int, bin_int: integer range -8 to 7;
begin
     -- Positive Inputs
     aip(8) <= I and A(8);
     aip(7) <= I and A(7);
     aip(6) <= I and A(6);
     aip(5) <= I and A(5);
     aip(4) <= I and A(4);
     aip(3) <= I and A(3);
     aip(2) <= I and A(2);
     aip(1) <= I and A(1);
     aip(0) <= I and A(0);

     bip(3) <= I and B(3);
     bip(2) <= I and B(2);
     bip(1) <= I and B(1);
     bip(0) <= I and B(0);

     aip_int <= signed_to_integer(aip);
     bip_int <= signed_to_integer(bip);

     -- Negative Inputs
     ain(8) <= I and A(8);
     ain(7) <= I and A(7);
     ain(6) <= I and A(6);
     ain(5) <= I and A(5);
     ain(4) <= I and A(4);
     ain(3) <= I and A(3);
     ain(2) <= I and A(2);
     ain(1) <= I and A(1);
     ain(0) <= not(I) or A(0);

     bin(3) <= not(I) or B(3);
     bin(2) <= not(I) or B(2);
     bin(1) <= not(I) or B(1);
     bin(0) <= not(I) or B(0);

     ain_int <= signed_to_integer(ain);
     bin_int <= signed_to_integer(bin);

     -- Positive multiplier (P)
     m0: mult9x4 port map(aip_int,bip_int,pprod_int);
     pprod <= integer_to_signed(pprod_int,13);
     Prod <= pprod;

     -- Negative multiplier (N)
     m1: mult9x4 port map(ain_int,bin_int,npd_int);
     npd <= integer_to_signed(npd_int,13);
     nprod <= not(npd);
```

```
    -- completion signal
    dprod(12) <= pprod(12) xor nprod(12);
    dprod(11) <= pprod(11) xor nprod(11);
    dprod(10) <= pprod(10) xor nprod(10);
    dprod(9) <= pprod(9) xor nprod(9);
    dprod(8) <= pprod(8) xor nprod(8);

    dprod(7) <= pprod(7) xor nprod(7);
    dprod(6) <= pprod(6) xor nprod(6);
    dprod(5) <= pprod(5) xor nprod(5);
    dprod(4) <= pprod(4) xor nprod(4);

    dprod(3) <= pprod(3) xor nprod(3);
    dprod(2) <= pprod(2) xor nprod(2);
    dprod(1) <= pprod(1) xor nprod(1);
    dprod(0) <= pprod(0) xor nprod(0);

    dv <= dprod(12) and dprod(11) and dprod(10) and dprod(9) and
        dprod(8) and dprod(7) and dprod(6) and dprod(5) and
        dprod(4) and dprod(3) and dprod(2) and dprod(1) and
        dprod(0) and I;

end mult_st9x4a_logic;
```

File: mult9x4.vhd

```
-- Multiplier
-- Start Date: October 3, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity mult9x4 is
    port (  A               :in integer range -256 to 255;
            B               :in integer range -8 to 7;
            Prod            :out integer range -4096 to 4095);
end mult9x4;

architecture mult9x4_logic of mult9x4 is
begin
    Prod <= A * B;
end mult9x4_logic;
```

File: ast12x12.vhd

```
-- 12x12 self-timed 2's Complement Adder
-- With Request Acknowledge Signalling and Registers
-- Start Date: November 4th, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity ast12x12 is
    port (  A                   :in std_logic_vector(11 downto 0);
            B                   :in std_logic_vector(11 downto 0);
            Sum                 :out std_logic_vector(12 downto 0);
            reset               :in std_logic;
            rin,ain             :in std_logic;
            aout,rout :inout std_logic);
end ast12x12;

architecture ast12x12_logic of ast12x12 is
  component reg
      port (d       :in std_logic;
            req     :in std_logic;
            clear   :in std_logic;
            q       :out std_logic);
  end component;
  component cg4
  port (rin  :in std_logic;
            ain     :in std_logic;
            reset   :in std_logic;
            rout    :nout std_logic;
            aout    :inout std_logic;
            req     :out std_logic);
  end component;
  component add_st12x12
      port (Enable  :in std_logic;
            DV      :out std_logic;
            A       :in std_logic_vector(11 downto 0);
            B       :in std_logic_vector(11 downto 0);
            Sum     :inout std_logic_vector(12 downto 0));
  end component;
  signal F:std_logic_vector(12 downto 0);
  signal v: std_logic;
  signal add_enable, add_dv :std_logic;
begin
    add_enable <= rin;

    -- Self_timed Adder
    a0: add_st12x12 port map (add_enable,add_dv,a,b,F);

    -- end of pipeline
    cg0: cg4 port map (add_dv,ain,reset,rout,aout,v);
    r0: reg port map (F(0),v,reset,Sum(0));
    r1: reg port map (F(1),v,reset,Sum(1));
    r2: reg port map (F(2),v,reset,Sum(2));
    r3: reg port map (F(3),v,reset,Sum(3));
    r4: reg port map (F(4),v,reset,Sum(4));
    r5: reg port map (F(5),v,reset,Sum(5));
    r6: reg port map (F(6),v,reset,Sum(6));
    r7: reg port map (F(7),v,reset,Sum(7));
    r8: reg port map (F(8),v,reset,Sum(8));
    r9: reg port map (F(9),v,reset,Sum(9));
    r10: reg port map (F(10),v,reset,Sum(10));
    r11: reg port map (F(11),v,reset,Sum(11));
    r12: reg port map (F(12),v,reset,Sum(12));

end ast12x12_logic;
```

File: add_st12x12.vhd

```
-- 12x12 self-timed 2's Complement Adder
-- Start Date: November 4, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity add_st12x12 is
     port (  Enable              :in std_logic;
             DV                  :out std_logic;
             A                   :in std_logic_vector(11 downto 0);
             B                   :in std_logic_vector(11 downto 0);
             Sum                 :inout std_logic_vector(12 downto 0));
end add_st12x12;

architecture add_st12x12_logic of add_st12x12 is
  component ADD_SUB_TWO_COMP_14
       port(C_IN, ADD_SUB :in std_logic;
            B, A      : in std_logic_vector (0 to 13);
            FUNC      : out std_logic_vector (0 to 13));
  end component;
  signal logic1,logic0 :std_logic;
  signal c,d,e,f,sumlong,sumbar,x: std_logic_vector(13 downto 0);
begin
     logic1 <= '1';
     logic0 <= '0';

     -- Inputs into the Positive Adder (Sum)
     -- Note: c(13) = c(12) Sign Extension
     c(13) <= enable and a(11);
     c(12) <= enable and a(11);
     c(11) <= enable and a(11);
     c(10) <= enable and a(10);
     c(9) <= enable and a(9);
     c(8) <= enable and a(8);
     c(7) <= enable and a(7);
     c(6) <= enable and a(6);
     c(5) <= enable and a(5);
     c(4) <= enable and a(4);
     c(3) <= enable and a(3);
     c(2) <= enable and a(2);
     c(1) <= enable and a(1);
     c(0) <= enable and a(0);

     -- Note: d(13) = d(12) Sign Extension
     d(13) <= enable and b(11);
     d(12) <= enable and b(11);
     d(11) <= enable and b(11);
     d(10) <= enable and b(10);
     d(9) <= enable and b(9);
     d(8) <= enable and b(8);
     d(7) <= enable and b(7);
     d(6) <= enable and b(6);
     d(5) <= enable and b(5);
     d(4) <= enable and b(4);
     d(3) <= enable and b(3);
     d(2) <= enable and b(2);
     d(1) <= enable and b(1);
     d(0) <= enable and b(0);

     -- Inputs into the Negative Adder (SumBar)
     -- Note: e(13) = e(12) Sign Extension
     e(13) <= not(enable) nor a(11);
     e(12) <= not(enable) nor a(11);
     e(11) <= not(enable) nor a(11);
     e(10) <= not(enable) nor a(10);
     e(9) <= not(enable) nor a(9);
     e(8) <= not(enable) nor a(8);
     e(7) <= not(enable) nor a(7);
     e(6) <= not(enable) nor a(6);
     e(5) <= not(enable) nor a(5);
     e(4) <= not(enable) nor a(4);
     e(3) <= not(enable) nor a(3);
     e(2) <= not(enable) nor a(2);
     e(1) <= not(enable) nor a(1);
     e(0) <= not(enable) nor a(0);
```

```
    -- Note: f(13) = f(12) Sign Extension
    f(13) <= not(enable) nor b(11);
    f(12) <= not(enable) nor b(11);
    f(11) <= not(enable) nor b(11);
    f(10) <= not(enable) nor b(10);
    f(9) <= not(enable) nor b(9);
    f(8) <= not(enable) nor b(8);
    f(7) <= not(enable) nor b(7);
    f(6) <= not(enable) nor b(6);
    f(5) <= not(enable) nor b(5);
    f(4) <= not(enable) nor b(4);
    f(3) <= not(enable) nor b(3);
    f(2) <= not(enable) nor b(2);
    f(1) <= not(enable) nor b(1);
    f(0) <= not(enable) nor b(0);

    -- Positive Adder (Sum)
    a0: ADD_SUB_TWO_COMP_14 port map (C_IN =>logic0,
      ADD_SUB=>logic1,
          B(0 to 13) => d(13 downto 0),
          A(0 to 13)=> c(13 downto 0),
          Func(0 to 13)=>sumlong(13 downto 0));

    -- Negative Adder (SumBar)
    a1: ADD_SUB_TWO_COMP_14 port map (C_IN =>Enable,
      ADD_SUB=>logic1,
          B(0 to 13) =>f(13 downto 0),
          A(0 to 13)=>e(13 downto 0),
          Func(0 to 13)=>sumbar(13 downto 0));

    x(13 downto 0) <= sumlong(13 downto 0) xor sumbar(13 downto 0);
    DV <= x(13) and x(12) and x(11) and x(10) and x(9) and
       x(8) and x(7) and x(6) and x(5) and x(4) and x(3) and
       x(2) and x(1) and x(0) and Enable;

    sum(12 downto 0) <= sumlong(12 downto 0);
end add_st12x12_logic;
```

File: reset_block.vhd

```
-- reset button (active low input to active high output)
-- Start Date: September 13, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity reset_block is
    port (  button          :in std_logic;
            reset           :inout std_logic);
end reset_block;

architecture reset_block_logic of reset_block is
begin
    reset <= not (button);
end reset_block_logic;
```

File: cg4.vhd

```
-- 4 - phase handshaking element using cpart.vhd
-- Start Date: September 3, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity cg4 is
      port (  rin                    : in std_logic;
              ain                    : in std_logic;
              reset                  : in std_logic;
              rout                   : inout std_logic;
              aout                   : inout std_logic;
              req                    : out std_logic);
end cg4;

architecture cg4_logic of cg4 is
   component cg
         port (X          : in std_logic;
               Y          : in std_logic;
               Z          : inout std_logic;
               Tr         : inout std_logic;
               reset      : in std_logic);
   end component;
   signal a,b:std_logic;
begin
      c0: cg port map (rin,rout,aout,a,reset);
      c1: cg port map (aout,ain,rout,b,reset);
      req <= aout;
end cg4_logic;
```

File: cg.vhd

```
-- C-element using Gamble architecture
-- Start Date: August 29, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity cg is
      port (  X                    : in std_logic;
              Y                    : in std_logic;
              Z                    : inout std_logic;
              Tr                   : inout std_logic;
              reset                : in std_logic);
end cg;

architecture cg_logic of cg is
   component reg
         port (d          :in std_logic;
               req        :in std_logic;
               clear      :in std_logic;
               q: out std_logic);
   end component;
   signal a,b,c: std_logic;
begin
      r0: reg port map (a,Tr,reset,Z);
      a <= not (Z);
      b <= (Z and not(X) and Y);
      c <= (not(Z) and X and not(Y));
      Tr <= b or c;
end cg_logic;
```

File: delay_c2.vhd

```
-- Delay element using TWO cg
-- Start Date: September 25, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity delay_c2 is
    port (  g0          :in std_logic;
            g1          :inout std_logic;
            reset       :in std_logic);
end delay_c2;

architecture delay_c2_logic of delay_c2 is
  component cg
      port (X       : in std_logic;
            Y       : in std_logic;
            Z       : inout std_logic;
            Tr      : inout std_logic;
            reset   : in std_logic);
  end component;
  signal a: std_logic;
  signal d0,d1:std_logic;
  signal x0,x1:std_logic;
begin

    x0 <= not(g0);
    x1 <= not(a);
    c0: cg port map (g0,x0,a,d0,reset);
    c1: cg port map (a,x1,g1,d1,reset);
end delay_c2_logic;
```

File: delay_c6.vhd

```
-- element using SIX cg
-- Start Date: September 25, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity delay_c6 is
    port (  g0          :in std_logic;
            g1          :inout std_logic;
            reset       :in std_logic);
end delay_c6;

architecture delay_c6_logic of delay_c6 is
  component cg
      port (X       : in std_logic;
            Y       : in std_logic;
            Z       : inout std_logic;
            Tr      : inout std_logic;
            reset   : in std_logic);
  end component;
  signal a,b,c,d,e: std_logic;
  signal d0,d1,d2,d3,d4,d5:std_logic;
  signal x0,x1,x2,x3,x4,x5:std_logic;
begin
    x0 <= not(g0);
    x1 <= not(a);
    x2 <= not(b);
    x3 <= not(c);
    x4 <= not(d);
    x5 <= not(e);
    c0: cg port map (g0,x0,a,d0,reset);
    c1: cg port map (a,x1,b,d1,reset);
    c2: cg port map (b,x2,c,d2,reset);
    c3: cg port map (c,x3,d,d3,reset);
    c4: cg port map (d,x4,e,d4,reset);
    c5: cg port map (e,x5,g1,d5,reset);
end delay_c6_logic;
```

File: mux26to13.vhd

```
-- 26 to 13  Mux
-- Start Date: November 18, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity mux26to13 is
     port (  A                  :in std_logic_vector(12 downto 0);
             B                  :in std_logic_vector(12 downto 0);
             Q                  :out std_logic_vector(12 downto 0);
             sel                :in std_logic);
end mux26to13;

architecture mux26to13_logic of mux26to13 is
  component mux2to1
       port (A          :in std_logic;
             B          :in std_logic;
             Q          :out std_logic;
             sel        :in std_logic);
  end component;
begin
     m0: mux2to1 port map(A(0),B(0),Q(0),sel);
     m1: mux2to1 port map(A(1),B(1),Q(1),sel);
     m2: mux2to1 port map(A(2),B(2),Q(2),sel);
     m3: mux2to1 port map(A(3),B(3),Q(3),sel);
     m4: mux2to1 port map(A(4),B(4),Q(4),sel);
     m5: mux2to1 port map(A(5),B(5),Q(5),sel);
     m6: mux2to1 port map(A(6),B(6),Q(6),sel);
     m7: mux2to1 port map(A(7),B(7),Q(7),sel);
     m8: mux2to1 port map(A(8),B(8),Q(8),sel);
     m9: mux2to1 port map(A(9),B(9),Q(9),sel);
     m10: mux2to1 port map(A(10),B(10),Q(10),sel);
     m11: mux2to1 port map(A(11),B(11),Q(11),sel);
     m12: mux2to1 port map(A(12),B(12),Q(12),sel);
end mux26to13_logic;
```

File: mux2to1.vhd

```
-- 2to1 Mux
-- Start Date: October 18, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity mux2to1 is
     port (  A                  :in std_logic;
             B                  :in std_logic;
             Q                  :out std_logic;
             sel                :in std_logic);
end mux2to1;

architecture mux2to1_logic of mux2to1 is
begin
   -- 0-A, 1-B
   process (sel,A,B)
   begin
     if (sel = '0') then
             Q <= A;
     elsif (sel = '1') then
             Q <= B;
     end if;
   end process;
end mux2to1_logic;
```

File: xorp6.vhd

```
-- 6-bit xorplane
-- Start Date: September 17, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity xorp6 is
    port (  A               :in std_logic_vector(5 downto 0);
            X               :out std_logic_vector(5 downto 0);
            I               :in std_logic);
end xorp6;

architecture xorp6_logic of xorp6 is
begin
    -- template X() <= A() xor I;

    X(5) <= A(5) xor I;
    X(4) <= A(4) xor I;

    X(3) <= A(3) xor I;
    X(2) <= A(2) xor I;
    X(1) <= A(1) xor I;
    X(0) <= A(0) xor I;
end xorp6_logic;
```

File: xorp8.vhd

```
-- 8-bit xorplane
-- Start Date: September 17, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity xorp8 is
    port (  A               :in std_logic_vector(7 downto 0);
            X               :out std_logic_vector(7 downto 0);
            I               :in std_logic);
end xorp8;

architecture xorp8_logic of xorp8 is
begin
    -- template X() <= A() xor I;

    X(7) <= A(7) xor I;
    X(6) <= A(6) xor I;
    X(5) <= A(5) xor I;
    X(4) <= A(4) xor I;

    X(3) <= A(3) xor I;
    X(2) <= A(2) xor I;
    X(1) <= A(1) xor I;
    X(0) <= A(0) xor I;
end xorp8_logic;
```

File: reg.vhd

```
-- Module for REGISTER
-- Start Date: August 27, 1996
-- Note: Trying to get the XILINX software to use s flip flop.

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity reg is
      port (  d                   :in std_logic;
              req                 :in std_logic;
              clear               :in std_logic;
              q                   :out std_logic);
end reg;

architecture reg_logic of reg is
begin
      reg: process (req,clear)
      begin
      if (clear = '1') then
              q <= '0';
      elsif (req'event and req = '1') then
              q <= d;
      end if;
      end process;
end reg_logic;
```

File: regint9.vhd

```
-- Module for Integer Register 9
-- Start Date: October 22, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity regint9 is
      port (  d                   :in integer range -256 to 255;
              req                 :in std_logic;
              clear               :in std_logic;
              q                   :out integer range -256 to 255);
end regint9;

architecture regint9_logic of regint9 is
begin
      regint9: process (req,clear)
      begin
      if (clear = '1') then
              q <= 0;
      elsif (req'event and req = '1') then
              q <= d;
      end if;
    end process;
end regint9_logic;
```

File: regint5.vhd

```
-- Module for Integer Register
-- Start Date: October 9, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity regint5 is
      port (  d                    :in integer range -16 to 15;
              req                  :in std_logic;
              clear                :in std_logic;
              q                    :out integer range -16 to 15);
end regint5;

architecture regint5_logic of regint5 is
begin
      regint5: process (req,clear)
      begin
      if (clear = '1') then
              q <= 0;
      elsif (req'event and req = '1') then
              q <= d;
      end if;
    end process;
end regint5_logic;
```

File: regint4.vhd

```
-- Module for Integer Register
-- Start Date: October 9, 1996

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity regint4 is
      port (  d                    :in integer range -8 to 7;
              req                  :in std_logic;
              clear                :in std_logic;
              q                    :out integer range -8 to 7);
end regint4;

architecture regint4_logic of regint4 is
begin
      regint4: process (req,clear)
      begin
      if (clear = '1') then
              q <= 0;
      elsif (req'event and req = '1') then
              q <= d;
      end if;
    end process;
end regint4_logic;
```