NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN ALL-DIGITAL IMAGE SYNTHESIZER FOR
COUNTERING HIGH-RESOLUTION
IMAGING RADARS

by

Stig R.T. Ekestorm and
Christopher Karow

September 2000

Thesis Advisor: Phillip E. Pace
Second Reader: Robert E. Surratt

Approved for public release; distribution is unlimited

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 2000	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE: An All-Digital Image Syntesizer for Countering High-Resolution Radars		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Stig R.T Ekestorm Christopher Karow			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Research Laboratory		**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**	

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*
The subject of this thesis is a digital image synthesizer (DIS), which is especially useful as a counter-targeting signal repeater, i.e., for synthesizing the characteristic echo signature of a pre-selected target. The DIS has a digital radio frequency memory (DRFM) and associated circuitry, including digital tapped delay lines and a modulator in each delay line to impose both amplitude and frequency modulation in each line. A unique property of the digital image synthesizer is its ability to synthesize false targets using wideband chirp signals of any duration. The *system-on-a-chip* uses a scalable CMOS technology that increases the bandwidth and sensitivity of such a repeater over prior analog-based systems. The application-specific integrated-circuit reduces the noise of the repeated signal, reduces the size and cost of such a system and permits real-time alteration of operating parameters, permitting rapid and adaptive shifting among different types of targets to be synthesized.

14. SUBJECT TERMS Inverse Synthetic Aperture Radars, ISAR, Countermeasure, Digital Radio Frequency Memory, DRFM, Image Synthesizer, Field Programmable Gate Array, FPGA, Application Specific Integrated Circuit, ASIC, Chip Design			**15. NUMBER OF PAGES** 374
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

THIS PAGE INTENTIONALLY LEFT BLANK

AN ALL-DIGITAL IMAGE SYNTHESIZER FOR COUNTERING HIGH-RESOLUTION IMAGING RADARS

Stig R.T Ekestorm
LTC, Swedish Army
BSSE, Swedish National Defense College, 1996

Christoper Karow
LCDR, German Navy

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2000

Authors: _____
Stig R.T. Ekestorm
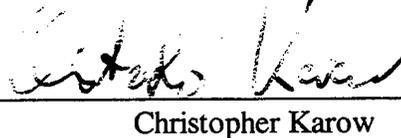
_____
Christopher Karow
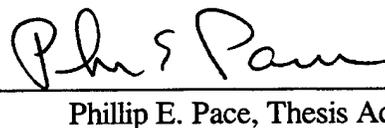
Approved by: _____
Phillip E. Pace, Thesis Advisor
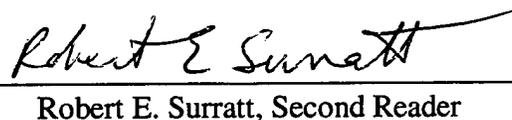
_____
Robert E. Surratt, Second Reader
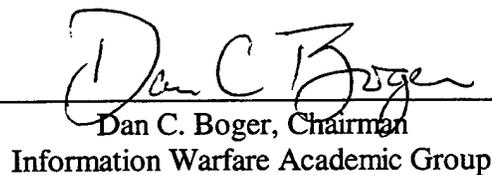
_____
Dan C. Boger, Chairman
Information Warfare Academic Group

iii

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The subject of this thesis is a digital image synthesizer (DIS), which is especially useful as a counter-targeting signal repeater, i.e., for synthesizing the characteristic echo signature of a pre-selected target. The DIS has a digital radio frequency memory (DRFM) and associated circuitry, including digital tapped delay lines and a modulator in each delay line to impose both amplitude and frequency modulation in each line. A unique property of the digital image synthesizer is its ability to synthesize false targets using wideband chirp signals of any duration. The *system-on-a-chip* uses a scalable CMOS technology that increases the bandwidth and sensitivity of such a repeater over prior analog-based systems. The application-specific integrated-circuit reduces the noise of the repeated signal, reduces the size and cost of such a system and permits real-time alteration of operating parameters, permitting rapid and adaptive shifting among different types of targets to be synthesized.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

xvii

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

EXECUTIVE SUMMARY

The subject of this thesis is a digital image synthesizer (DIS), which is especially useful as a counter-targeting signal repeater, i.e., for synthesizing the characteristic echo signature of a pre-selected target. The DIS has a digital radio frequency memory (DRFM) and associated circuitry, including digital tapped delay lines and a modulator in each delay line to impose both amplitude and frequency modulation in each line. A unique property of the digital image synthesizer is its ability to synthesize false targets using wideband chirp signals of any duration. To generate the false target, the user can program the target extent (number of taps) and the amplitude and Doppler frequency of each range-Doppler cell within the image. The algorithm of the DIS has been computer simulated and has verified the theory behind it. A concept demonstrator has been developed using a field programmable gate array technique. The DIS has developed further toward physical implementation as an application specific integrated circuit. The *system-on-a-chip* uses a scalable CMOS technology that increases the bandwidth and sensitivity of such a repeater over prior analog-based systems. The application-specific integrated-circuit reduces the noise of the repeated signal, reduces the size and cost of such a system, and permits real time alteration of operating parameters, permitting rapid and adaptive shifting among different types of targets to be synthesized. A scan-path test capability is also included to allow intra-chip signal analysis and verification.

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

We would like to thank Professor Phillip E. Pace and Professor Douglas J. Fouts for their technical directions and assistance throughout the course of this thesis. Working together as a team provided dimensions and analysis that could not have occurred if we had been working individually.

Furthermore, we would like to thank Dr. Harry Hurt and CDR Dan Gahagan of the Office of Naval Research for their inspiration and support during this project. We would also like to express our sincere appreciation to Mr. Robert E. Surratt, Section Head, Code 5760, Integrated Electronic Warfare Simulation Branch, Tactical Electronic Warfare Division, Naval Research Laboratory for his time and for his helpful comments on our work. With the help of these extraordinary gentlemen, this thesis, funded in part by the Office of Naval Research and the Naval Research Laboratory, Code 5740, proved to be fascinating, edifying, and highly productive.

Stig Ekestorm would also like to thank his colleagues in the Swedish Armed Forces personally for their support in making his studies in the United States possible. He would like to encourage his friends at the Lapland Ranger Regiment who are at this moment facing hard times. May the true spirit of an Arctic Ranger always follow you. Most of all, Stig Ekestorm would like to single out his wife Kristina, and his son, Oskar, for their love, support and encouragement throughout his educational experience at the Naval Postgraduate School.

I. COUNTERING THE SENSOR-SHOOTER ENGAGEMENT

Future Navy electronic warfare (EW) systems must be designed to operate in the RF environment to provide a layered EW defense and also to serve as a fully-integrated shipboard combat system sensor. The next generation EW systems must also provide threat identification and a complete situational awareness to allow the quick reaction modes required to counter the modern anti-ship cruise missile (ASCM) threat. Figure 1 shows the sequence of events taken by the enemy sensor-shooter in order to place a missile on a target (hard kill). A typical sequence begins with the enemy's electronic support surveillance sensor detecting the target of interest (e.g., with a long range over-the-horizon targeting radar). After acquiring a number of hits on the target, one can identify the target by using an additional high-resolution sensor, such as an airborne inverse synthetic aperture radar (ISAR) imager. This type of radio frequency (RF) sensor forms an image of the target that can be used for recognition and identification.

Surveillance	Acquisition and Identification	Engage Decision	Weapon Selection and Launch	Mid-Course Guidance	Acquisition	Terminal Homing

HARD KILL →

| Counter-Surveillance | Counter-Identification | Counter-Targeting | Counter Lock-on | Counter-Terminal |
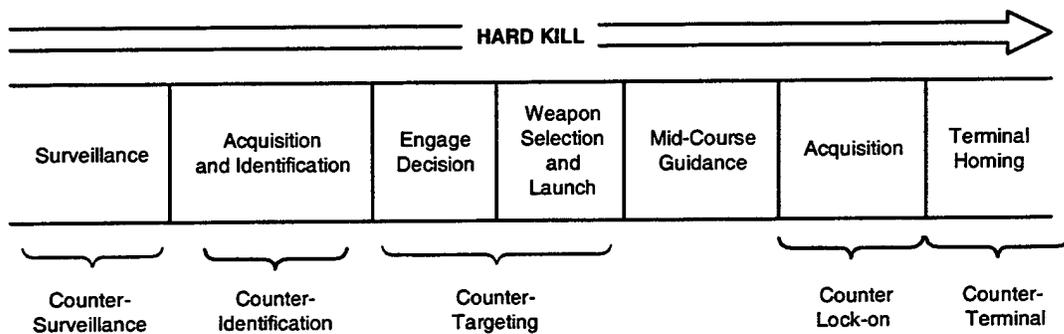
Figure 1. Sequence of Steps Necessary to Land a Missile on a Target

Depending on the target identification, the decision to engage the target and launch a weapon (such as an ASCM) is made using the inputs, for example, from the

ISAR imager. After the ASCM is launched, acquisition and terminal homing of the missile is again accomplished using the missile's ISAR. Use of an ISAR in the terminal phase of the missile allows good aimpoint accuracy and greater probability of kill.

To avoid the ASCM hard kill, one can use a number of countering techniques including counter-surveillance, counter-identification, counter-targeting, counter-lock-on and counter-terminal. Counter-surveillance and counter-identification include the use of low-radar cross-section materials, stealth and deception devices. Counter-targeting includes the use of active electronic attack (EA) and the use of decoy repeaters. Counter-lock-on and counter-terminal techniques consist of EA, distraction and seduction chaff as well as decoy repeaters.

Counter-identification and counter-targeting systems can begin the electronic attack well before the opposition launches any missiles due to the generation of a lower probability of target acquisition. Since acquisition systems and future missile seekers will employ pulse-to-pulse spread spectrum using unfocused SAR and ISAR to improve target recognition and decoy rejection, the need for coherent countering of these imaging sensors/seekers remains a high priority for EA systems. Countering-identification and counter-targeting techniques employ a false target image generated or synthesized with the objective of deceiving the imaging radar into believing the false target is a real one. Imaging sensors use coherent range-Doppler processing. Consequently, various forms of complex modulations must be imposed on the intercepted wideband waveforms in order to enable the imager to integrate the false target properly.

In this report, the design, analysis and fabrication of an all-digital image synthesizer for pulse-to-pulse countering of high resolution RF imaging sensors (e.g.,

SAR, ISAR) is presented. The signal processing used in the digital image synthesizer circuit is especially useful as a signal repeater, i.e., for synthesizing the characteristic echo signature of a pre-selected target. The entire system has a digital radio-frequency memory (DRFM) and associated circuitry, including a digital-tapped delay line and a modulator in each delay line to impose both amplitude and frequency modulation in each range-cell. The use of digital semiconductor technology (0.5/0.35μm CMOS) increases the bandwidth and sensitivity of the repeater over prior analog-based systems and reduces the noise of the repeated signal. It also reduces the size and cost of such a system and permits real-time alteration of operating parameters, permitting rapid and adaptive shifting among different kinds of targets to be synthesized. The integrated circuit is designed so that it can easily be integrated with a number of phase-sampling DRFM architectures.

For completeness, Chapter II provides a brief introduction of ISAR and ISAR signal processing. Chapter III discusses the digital image synthesizer concept and how the false target is generated. Chapter IV describes a set of modular Matlab programs that is easy to use and maintain for hardware simulation and evaluation of concept alternatives. Chapter V presents an Altera field-programmable gate-array (FPGA) implementation of the image synthesizer concept. To increase the bandwidth of the device, Chapter VI describes the investigation into converting the FPGA design into an application specific integrated circuit (ASIC). In Chapter VII, the schematic of an ASIC design in scalable CMOS is described in detail. Chapter VIII addresses timing and control of the ASIC. In Chapter IX, simulation results are presented including full-scale simulation of a radar pulse. Comparison of the results with the Matlab simulation is also

presented in order to verify the concept and detail the advantages of the architecture.

Finally, in Chapter X, layout and fabrication issues are discussed.

II. INTRODUCTION TO

INVERSE SYNTHETIC APERTURE RADAR

ISAR is a high-resolution technique for imaging isolated moving targets, such as ships and aircraft. The technique used by both targeting sensors and ASCMs closely parallels the SAR imaging approach in which the image (or map) is generated from the return signals being reflected off the target as the radar moves past the target area. In the ISAR technique, the target imaging is generated from the return signals being reflected off the target as the *target rotates* within the radar illumination. To illustrate this duality Figure 2 (a) shows a spotlight SAR in which the radar transverses a circular path about the target while collecting the return signals (focused spotlight) [Ref. 1]. The radar antenna in the spotlight SAR continually tracks the target. Note that the same signal returns could be collected if the radar were stationary and the target was put through a rotation as shown in Figure 2 (b).
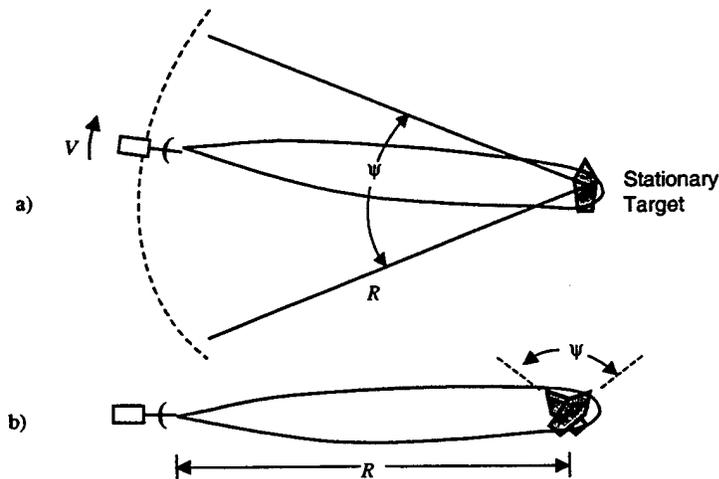
Figure 2. Comparison of the Geometrical Relationship between

(a) Focused Spotlight SAR and (b) ISAR (From Ref. [1])

5

A. RANGE-DOPPLER IMAGING

The range-Doppler image consists of resolution cells, each containing estimates of the target's magnitude and position of scatterers in both range and cross-range (Doppler). The orientation of the range-Doppler image is determined by the target's rotation relative to the ISAR. The range dimension within the range-Doppler image is oriented along the radar line of sight (LOS). Range focusing is based on the range-independent point target response determined by the wideband chirp waveform. The cross-range dimension of the range-Doppler image is the dimension lying *perpendicular* to the plane contained by the radar LOS and contains the Doppler frequency of the resolved scatterers in range. Determining the rotational motion during data collection and calculating the compressions for the sharpest focus accomplish the azimuth focusing. The Doppler frequency shift produced by a range resolved scatterer is proportional to the angular rotation rate ω and the cross-range distance between the scatterer and the center of target rotation [Ref. 1].

B. RANGE COMPRESSION PROCESS

High range resolution ISAR uses an analog-frequency coding technique called chirp. A chirp pulse waveform is shown in Figure 3. The transmitted chirp can be expressed as a complex narrowband signal

$$S_t(t) = a(t)e^{j\Phi(t)} = \text{rect}\left(\frac{t}{T}\right)e^{j2\pi\left(f_c t + Kt^2/2\right)} \tag{2.1}$$

6

$$\text{rect}\left(\frac{t}{T}\right) = \begin{cases} 1 & \text{for } \left|\frac{t}{T}\right| < \frac{1}{2} \\ \\ 0 & \text{for } \left|\frac{t}{T}\right| > \frac{1}{2} \end{cases} \qquad (2.2)$$

where f_c is the carrier frequency, Δ is the linear frequency sweep or bandwidth of the transmitted signal, K is the slope or chirp rate ($K = \Delta/T$), T is the pulse width and the instantaneous frequency (time-dependent frequency) is obtained as:

$$f(t) = \frac{1}{2\pi}\frac{d\Phi}{dt} = f_c + Kt \qquad (2.3)$$
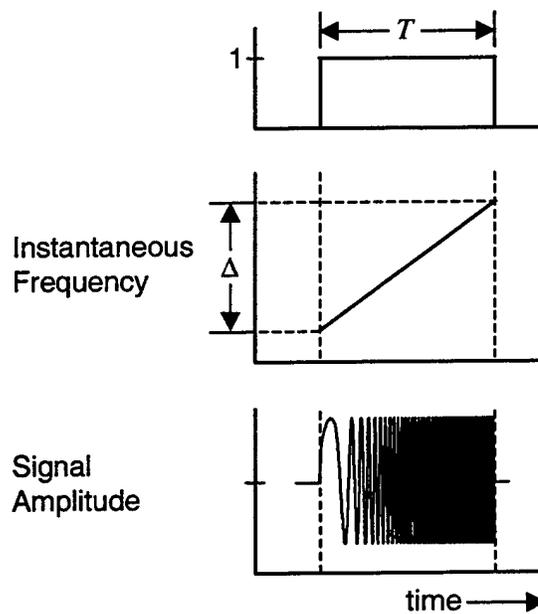
Figure 3. Chirp Pulse Waveform

Within the pulse duration T, the instantaneous frequency changes from $f_c - KT/2$ to $f_c + KT/2$. The dispersion D or time-bandwidth product of the waveform is $D = T\Delta$ [Ref. 1].

7

1. Analog Range Compression Network Example

The chirp pulse waveform can be compressed using an analog pulse compression network as shown in Figure 4.
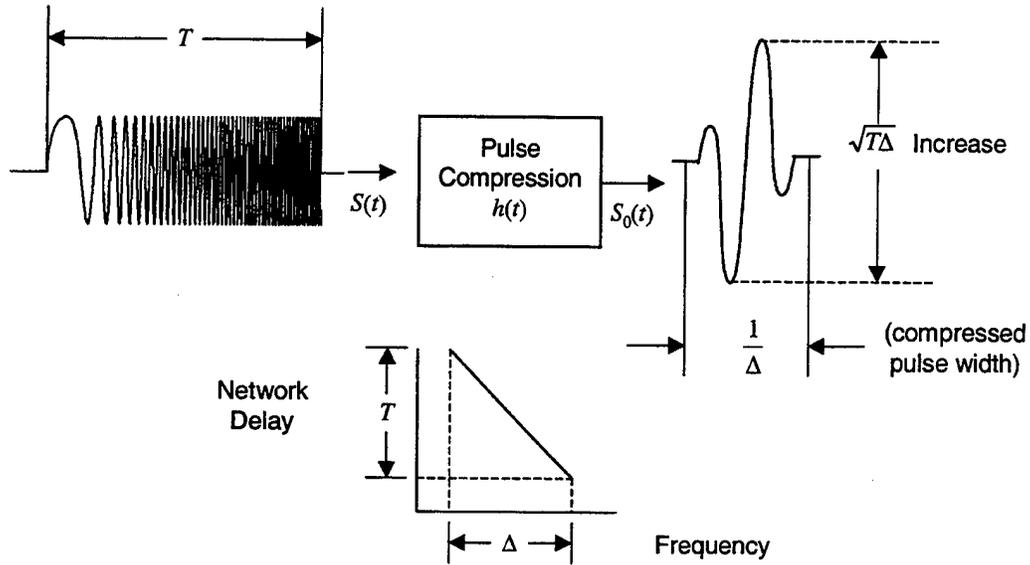
Figure 4. Chirp Pulse Compressed using Analog Pulse Compression Network

This common form of a pulse compression network is called a *phase equalizer* and equalizes the slope of the linear frequency sweep. The transfer function of the pulse compression network can be written as:

$$H(f) = e^{j2\pi/K(f-f_c)^2} \tag{2.4}$$

The corresponding impulse response can be expressed as:

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{j2\pi ft} df \tag{2.5}$$

or

$$h(t) = \sqrt{\frac{j\Delta}{T}} e^{j2\pi(f_c t - Kt^2/2)} \tag{2.6}$$

The complex matched filter output is obtained by convolving the chirp signal with the impulse response as:

$$S_0(t) = h(t) * S(t) = \sqrt{Dj} \frac{\sin(\pi \Delta t)}{\pi \Delta t} e^{j2\pi(f_c t - Kt^2/2)} \tag{2.7}$$

The compressed pulse duration of the envelope at the $2/\pi$ points is $T_c = 1/\Delta$ (Raleigh resolution). The corresponding range resolution is then:

$$dr = \frac{c}{2\Delta} \tag{2.8}$$

Note the wider the bandwidth of the ISAR chirp signal transmitted, the smaller the range-bin size.

2. Digital Range Compression

If the pulse compression is performed digitally on the baseband return samples, it is possible to control the matched filter transfer function adaptively. The range resolution is determined by the ADC sampling rate. The convolution can be carried out in the frequency domain using the advantages of the fast Fourier transform (FFT) as:

$$S_0(f) = F\{S(t) * h(t)\} = S(f)H(f) \tag{2.9}$$

and is the time domain convolution carried out by multiplication in the frequency domain where $S(f)$ is the spectrum of the returns from one transmitted pulse and $H(f)$ is the transfer function (reference function) of the pulse compression, filter which is stored as a series of complex pairs (constant for a particular chirp waveform). The range compression signal processing is shown in Figure 5.
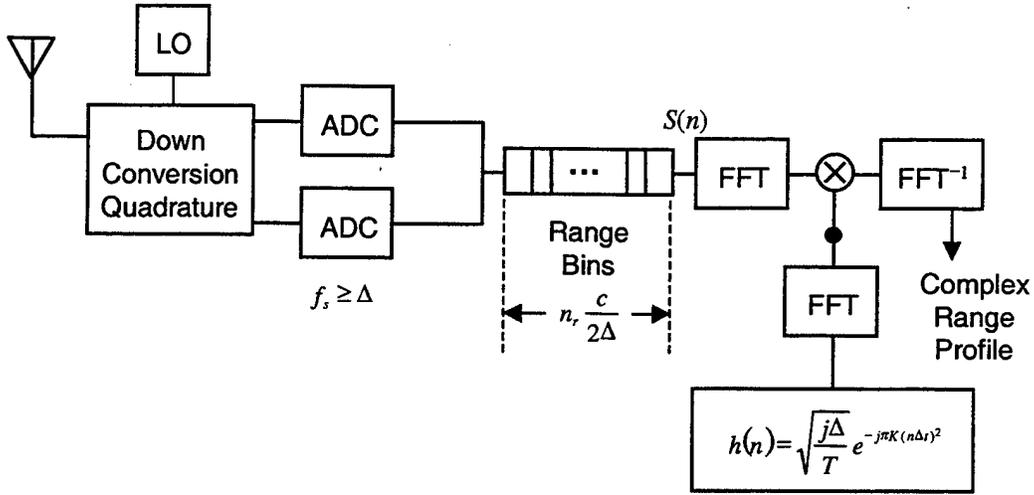
Figure 5. ISAR Range Compression Signal

The number of samples required for both $S(n)$ and $h(n)$ to avoid a circular convolution is

$$N \geq \frac{T + \frac{2(R_2 - R_1)}{c}}{\Delta t} - 1 \qquad (2.10)$$

where R_2 and R_1 are the edges of the range window to be processed and $\Delta t = 1/f_s$ is the ADC sampling period. Zeroes must be added to the signal and to the $T/\Delta t$ samples of the impulse response (common period of length N). Also note that $N = 2^\alpha$ (where α is an integer) due to the constraint on the FFT algorithm. The unambiguous range extent of the ISAR is

$$R_u = \frac{n_r c}{2\Delta} = \frac{n_r c}{2 f_s} \qquad (2.11)$$

and depends on the bandwidth of the chirp signal. A two-dimensional high-resolution spectral analysis algorithm based on 2-D linear prediction using autoregressive estimation for ISAR has been presented in [Ref. 2]. This approach is superior to the FFT method mentioned above.

C. AZIMUTH COMPRESSION PROCESS

If the target rotates at a rate of ω rad/s towards the radar, a scatterer at a cross range distance a has an instantaneous velocity ωa toward the radar with a corresponding Doppler frequency shift:

$$f_d = \frac{2\omega a}{\lambda} \qquad (2.12)$$

Considering two scatterers in the same slant range cell separated by da then:

$$df_d = \frac{2\omega da}{\lambda} \qquad (2.13)$$

resulting in a cross-range resolution of:

$$da = \frac{\lambda}{2\omega} df_d \qquad (2.14)$$

The Doppler resolution is related to the inverse synthetic integration (frame) time $df_d = \frac{1}{T}$ giving a cross-range resolution of (see Figure 6):

$$da = \frac{\lambda}{2\omega T} = \frac{\lambda}{2\psi} \qquad (2.15)$$

A cross-range profile exists for each range-bin. Samples that are integrated to form a cross-range profile come from the same range-bin separated by a pulse repetition interval (PRI) as shown in Figure 6.

11
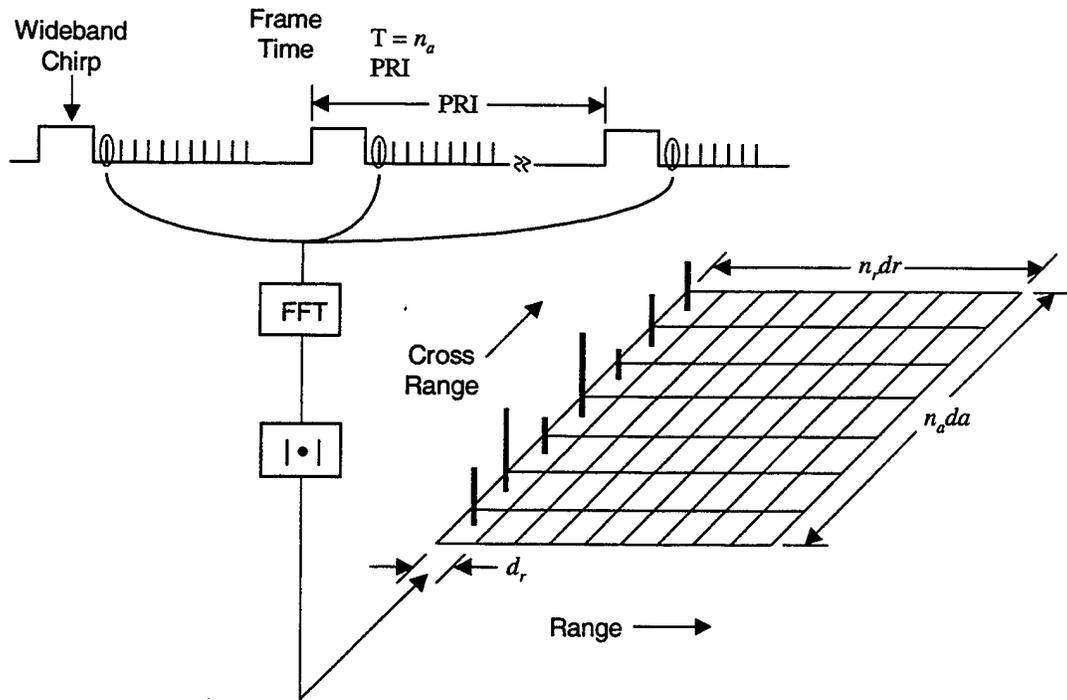
Figure 6. ISAR Azimuth Compression Processing

The unambiguous cross-range extent corresponds to the *target size* in the cross-range. The required PRF for unambiguous sampling a target of cross-range extent A_u is

$$PRF = \frac{2\omega A_u}{\lambda} \tag{2.16}$$

and the number of range samples needed is

$$n_a = \frac{2\omega A_u T}{\lambda} \tag{2.17}$$

The cross-range extent is

$$A_u = n_a da = \frac{n_a \lambda}{2\psi} \tag{2.18}$$

12

A summary of the ISAR compression process is shown in Figure 7.
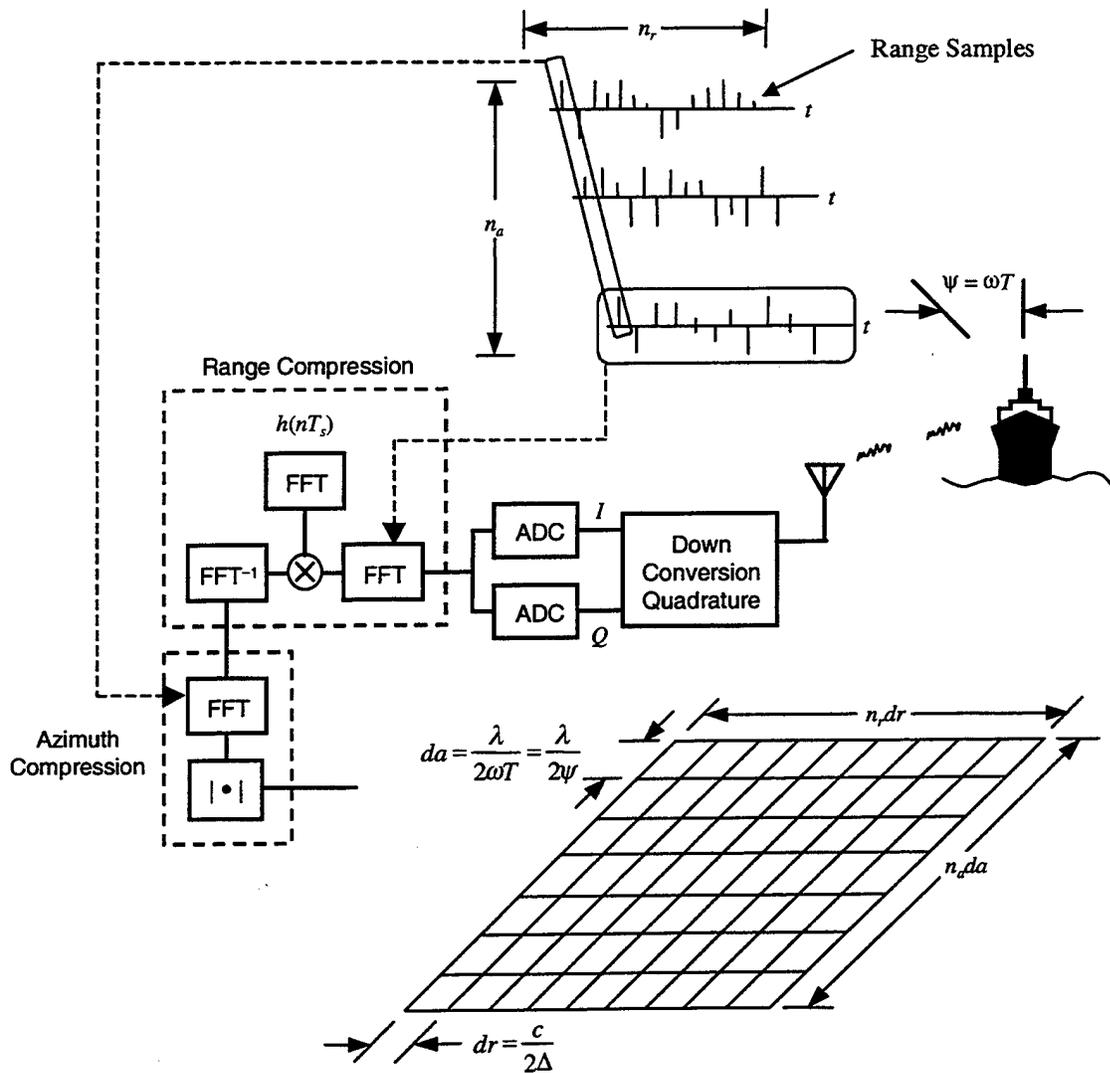
Figure 7. Summary of ISAR Compression Processing

13

THIS PAGE INTENTIONALLY LEFT BLANK

III. THE DIGITAL IMAGE SYNTHESIZER CONCEPT

A. SCATTERING PHYSICS OF A TARGET

An object will modify any signal reflected from it according to the object's shape, surface material properties, and the object's velocity relative to the signal. This permits an enemy sensor to identify the nature of such objects, which, if the objects are military platforms like warships or aircraft, is not desirable. One solution is to artificially synthesize false characteristic echo signatures by responding to an interrogating signal. Figure 8 shows a ship and an aircraft, in the line of sight of an interrogating radar signal. As the signal hits the aircraft and the ship, it is reflected from their major scattering surfaces. The return signal from the ship and the aircraft will be the superposition of the reflections from the various surfaces such as the hull, superstructure, the aircraft wings and nose.
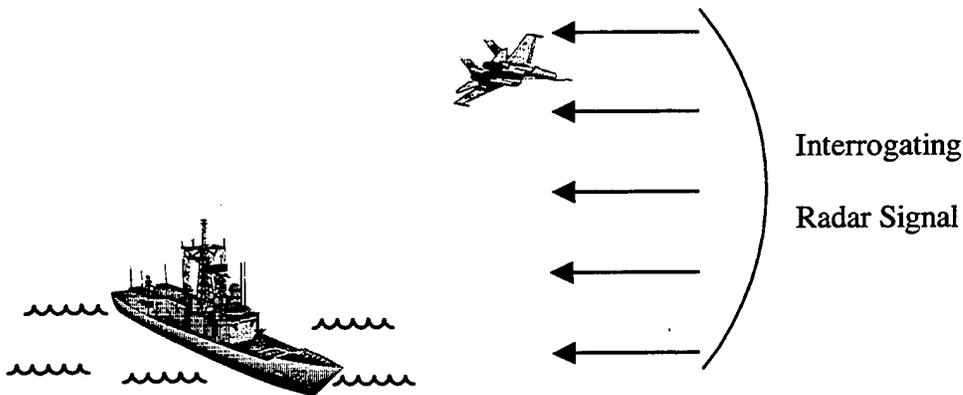
Figure 8. A Ship and an Aircraft in the Line of Sight of an Interrogating Radar Signal

The reflections of these surfaces are at different places along the radar's line of sight. These superimposed reflections will be out of phase with one another, owing to the varying times of signal propagation to each reflecting surface.

This tends to lengthen the return radar pulse by an amount equal to the round trip propagation time of the radar signal between the nearest and farthest major reflector and to make the reflection magnitude vary as dictated by the varying radar cross sections of the reflecting surfaces. Furthermore, movements of the aircraft or ship relative to the radar signal will Doppler shift the returned reflections. That is, any platform that reflects the radar signal will frequency-modulate the signal, so the returned reflections permit the radar to calculate the nature and motion of the platform.

The most common way to detect a Doppler spectrum in the return signal is to compare the reflections from consecutive pulses. Thus, an imaging sensor, such as a search radar, SAR, or ISAR can calculate the Doppler by comparing consecutive return pulses on a range-bin by range-bin basis. The Doppler spectrum is conventionally computed using an algorithm that incorporates the discrete Fourier Transform.

B. ANALOG IMAGE SYNTHESIS

Any credible counter-targeting repeater decoy must synthesize the temporal lengthening and amplitude modulation caused by the many recessed and reflective surfaces, and must generate a realistic Doppler shift for each surface. Conventionally this has been done using analog systems that receive an interrogating signal and pass it through a length of cable having serial taps along its length, one tap per range-bin. Each tap modulates the signal in amplitude and/or frequency to synthesize the reflection from

16

the reflective surfaces within that range-bin. The delay time between taps is selected to correspond to the differing times of flight of the radar pulse to the respective range-bins. Finally, the signals from the taps are summed, and the synthesized signal is retransmitted. In this manner, the system returns what appears to be an echo from an object located within the selected range-bins having a signature indicative of the moving ship or aircraft object to being synthesized.

Unfortunately, analog systems have drawbacks that limit their usefulness as image synthesizers. They are inherently noisy and can hold an incoming signal only a short time for processing before the signal deteriorates below the noise. This limits the system bandwidth and permits effective synthesis of only small objects. Further, analog systems are costly and very bulky, the latter being a particular concern for military platforms, where space is extremely limited. Finally, analog systems cannot readily change operating parameters, such as relative delays among taps, or the amount of modulation in the various taps. This means that analog image synthesizers cannot switch among different simulated objects on the fly, but rather must typically be fabricated for one specific type of target.

C. DIGITAL IMAGE SYNTHESIS

The main advantage of the all-digital image synthesizer repeater is the increase in bandwidth provided to the tapped delay line processors of the kind above described. In addition, the capability to hold the received signals as long as necessary for a given application is provided. Due to the all-digital architecture, modulation of the target extent

(number of range-bins) and Doppler frequency of each resolution cell is also a capability. This results in a small, low-cost and flexible counter-targeting repeater decoy processor.

The digital image synthesizer uses a DRFM and an associated digital-processing circuit having a plurality of tapped delay lines, a summer in order to sum the output of the delay lines, and range-bin signal modulator in each of the delay lines. A DRFM is a semiconductor device that can rapidly and permanently record radio frequency information as digitized samples of the incoming signal, and read it back equally rapidly when needed. Because the DRFM can hold data indefinitely, the duration of the synthesized signal is not limited, as with analog systems, thus permitting (as in the example of Figure 8) simulation of larger objects by adding more taps to accommodate more range-bins. Because the associated circuitry is digital, and most especially because the circuitry can be dedicated to its processing task (rather than requiring extensive programming to perform its tasks), the speed of the synthesizer can be especially great.

In an optimum hardware configuration, the associated digital image synthesizer circuitry is made a part of the DRFM on the same monolithic chip in order to increase the synthesizer speed even more. This is in contrast to a computer, or programmable processor, which, in conjunction with a fast and permanent memory like a DRFM, could in principle do the necessary processing. But the time needed to execute the large number of programming instructions necessary to process data makes this far less desirable than the current design described in this report, and, for the specific problem of counter-targeting decoy repeaters, largely ineffective.

D. FUNCTIONAL DESCRIPTION OF THE DIGITAL IMAGE SYNTHESIZER

Figure 9 shows a block diagram of the digital image synthesizer [Ref. 3]. The antenna receives the radar pulse from a (possibly hostile) search radar. After the down conversion (not shown), a set of comparators digitizes the phase of the analog signal producing a stream of digital samples, which are stored in the DRFM. The phase samples are a digital representation of the phase only. Phase sampling DRFMs have fewer comparators and permit coherent reconstruction of the original signal using stored amplitude information [Ref. 4]. The digitized samples are read serially from the DRFM via the tapped delay line.
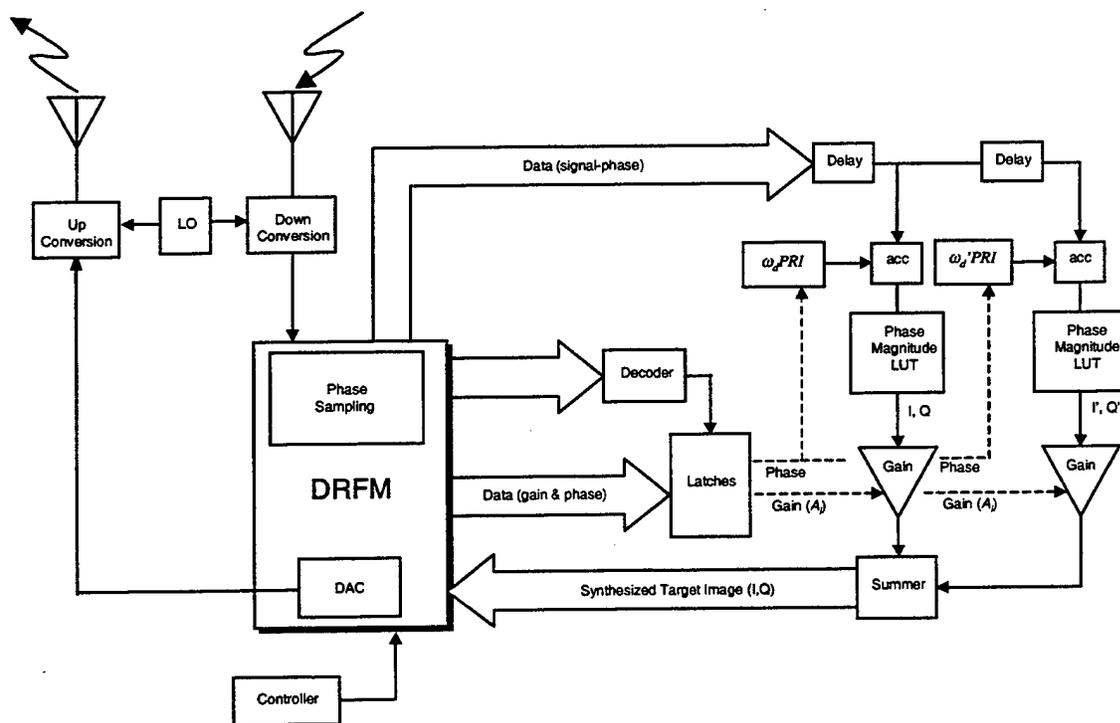
Figure 9. Block Diagram of the Digital Image Synthesizer (DIS) (From Ref. [3])

The circuit of Figure 9 shows two taps, but this is illustrative, and in principle the device contains the largest number of taps that the particular application dictates (the number of major reflective surfaces of the synthesized target). The digital-phase samples from the DRFM are sequentially read into the taps by clocking. The signals in the respective taps are delayed with respect to one another by pre-selected amounts dictated by the delays. For simplicity, the following discussion references the first tap leg only. However, the function of each leg is identical. The phase signals in the tap pass through a phase accumulator and an associated look-up-table (contains sine and cosine values for a 2π cycle used in constructing the I & Q components). Although the tap process could readily calculate $\cos(\phi_n)$ and $\sin(\phi_n)$, doing so is less computationally efficient than use of a look-up-table, and thus would reduce the overall system speed. At the output of the look-up-table, a selectable gain multiplies the signal by a pre-selected amount. Together, these blocks constitute a range-bin signal modulator.

The accumulator frequency modulates the signal, traversing the tap leg by phase-rotation (serrodyne modulation). The phase ϕ of any signal subjected to a linear frequency modulation, such as a Doppler shift is given by $\phi = (\omega + \omega_d)t$, where ω is signal angular frequency, ω_d is the change in frequency due to the modulation, and t is time. Thus at each point in time the difference in phase between the modulated and unmodulated signal is $\omega_d t$. For a digitally-sampled signal, the phase of the nth sample $\phi_n = n(\omega + \omega_d)PRI$, where n is an integer counter and PRI is the period at which the signal is sampled. The phase difference due to the Doppler frequency is $n\omega_d PRI$. Thus one can shift the frequency of a digitally-sampled signal by an amount ω_d by rotating each nth

phase sample by $n\omega_d PRI$. That is, the frequency of a digitally-sampled signal can be shifted by incrementing the phase $n\omega PRI$ of each nth sample by $n\omega_d PRI$.

In summary, the Doppler of a target is typically inferred by sampling target-echoes (within a single range-bin) at the pulse repetition rate and inspecting these samples for Doppler-induced phase differences between the echoes. One can simulate a Doppler shift of ω_d by repeating the pulses from a sensor, with each pulse phase shifted with respect to the next by an amount $\omega_d PRI$, where PRI is the pulse repetition interval. A unique property of the DIS is its ability to synthesize false targets using chirp signals of any duration. The number of tap stages is equal to the target range-extent desired for synthesis.

In operation, the phase accumulator sets nominal values of ω_d and ω_d' per instructions from the DRFM controller. A sensor sends a burst of N pulses having a pulse repetition period of PRI. The phase samples from the first pulse (stored in the DRFM) are piped to the first tap leg and the accumulator rotates the phase *of each sample* by an amount $\omega_d PRI$. The resultant phase samples are converted to I and Q components and scaled by a gain factor A_i. In the absence of output from the second tap leg shown, the complex signal is returned to the DRFM, and thereafter to the digital-to-analog converter that reconstructs the analog pulse for up-conversion and retransmission.

The waveform of the retransmitted pulse is identical to that of the received pulse, except that it is a phase rotated by $\omega_d PRI$. After processing this pulse, the DRFM changes the phase of the first tap accumulator to $2\omega_d PRI$, rotates each phase sample of the *second pulse* by $2\omega_d PRI$, and, again assuming no output from the second tap, retransmits the reconstructed pulse. This continues through the N pulses of the burst, with the phase

21

samples of each pulse rotated by an amount $n\omega_d PRI$, where n is pulse number, i.e., $n = 1$, $2, \ldots, N$. In the absence of output from the second tap, the result is a stream of analog pulses from the antenna that are different in phase *from one pulse to the next* by $\omega_d PRI$. A sensor detecting these echoes would interpret the constant pulse-to-pulse phase shift of $\omega_d PRI$ as a Doppler shift from a single reflector. The second tap accomplishes the same task by use of a different ω_d. The summer then combines the output of the first and second taps. The complex signal that the summer returns to the DRFM is the superposition of the signals exiting the first and second tap legs. This means that for each nth pulse of the N pulses, the summer's output will be the superposition of two copies of the nth pulse, delayed with respect to one another by the tap delay, scaled differently by the gains A_i, with one phase rotated by $n\omega_d PRI$, the other by $n\omega'_d PRI$. A sensor, which receives the corresponding N analog pulses, will interpret this as having come from two reflectors located in range-bins separated by the delay with reflective cross sections respectively proportional to the two gains. Because the pulse-to-pulse phase difference between these pulses is $\omega_d PRI$ for the range-bin corresponding to the first delay and $\omega'_d PRI$ for the bin corresponding to the second delay, the sensor will interpret that the reflectors in these two range-bins have Doppler frequencies of ω_d and ω'_d, respectively.

The decoder and latch shown in Figure 9 updates the phase-rotation and gain-coefficients for the tap legs. The controller is a process computer interfaced with the DRFM that permits an operator to change these parameters on the fly in real time. In addition to the phase and gain-coefficients, the number of taps utilized (target extent) can be changed. Alternatively, the controller can do this automatically. This is particularly important if ω_d in any tap leg varies with time. In the example of Figure 8, the aircraft

flies directly at the sensor at a constant speed and Doppler shifts the signal by a constant, positive, amount. The ship, on the other hand, could be rocking back and forth in the water along the line of sight and thus the Doppler shift corresponding to this motion would oscillate in time.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. ARCHITECTURE VARIATIONS AND SIMULATION

A. ARCHITECTURE VARIATIONS

Two different implementations of the DIS architecture have been studied. The major difference between the two implementations is the placement of the time-delay processor. Advantages and disadvantages of the two approaches are addressed in this Chapter and are mainly the result of the hardware technologies used. The two different implementations are referred to as the "original architecture" and the "modified architecture."

The "original architecture" described in Chapter III is illustrated in the block diagram shown in Figure 10. The intercepted chirp signal within the DRFM operating bandwidth is down converted into its I,Q components with a corresponding intermediate frequency that lies within the instantaneous bandwidth of the phase-sampling DRFM comparator technology. The phase-sampling DRFM digitizes the phase of the I,Q components with the sampling period (time between phase samples) corresponding to the range resolution of the DRFM. The DRFM-phase data is fed serially into the tapped delay processor with each delay corresponding to the range resolution of the image synthesizer. The phase data at each tap is processed in a pipelined range-bin signal processor in order to generate the selected scattering mechanism. As previously discussed, this is done by continuously rotating the phase $n\Delta\phi = n\omega_d PRI$, translating the phase into a complex signal I,Q that is amplitude modulated using A_i. When the complex I,Q data exits each tap, it is summed with available data from all the other tap processors

each clock cycle. The digital sum at each clock cycle is then converted to an analog signal for up conversion onto the carrier for retransmission.
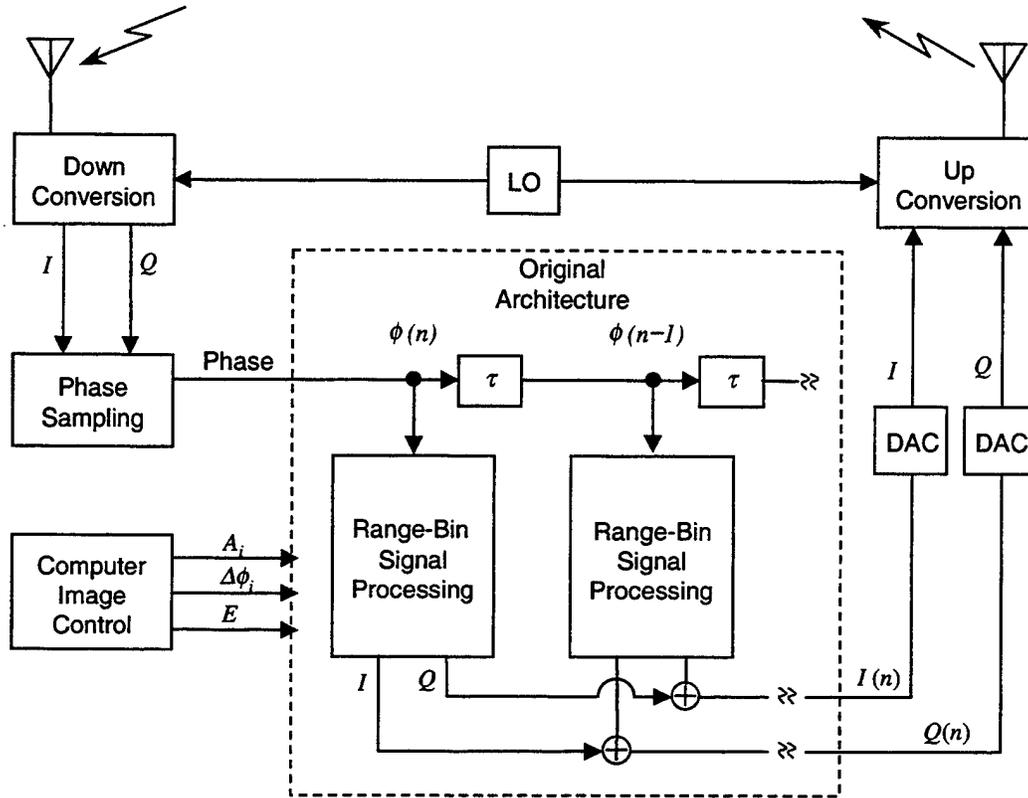
Figure 10. Block Diagram of the Original DIS Architecture

In order to show the equivalence of both architecture variations, we show the details of the original architecture for the in-phase processing in Figure 11 where E is the image extent, $\Delta\phi_i$ is the phase-increment value for the ith tap processor, and A_i is the amplitude modulation. The input phase is $\phi(n)$ and the output is:

$$I(n) = \sum_{i=0}^{E} A_i \cos(\phi(n-i) + \Delta\phi_i). \qquad (4.1)$$
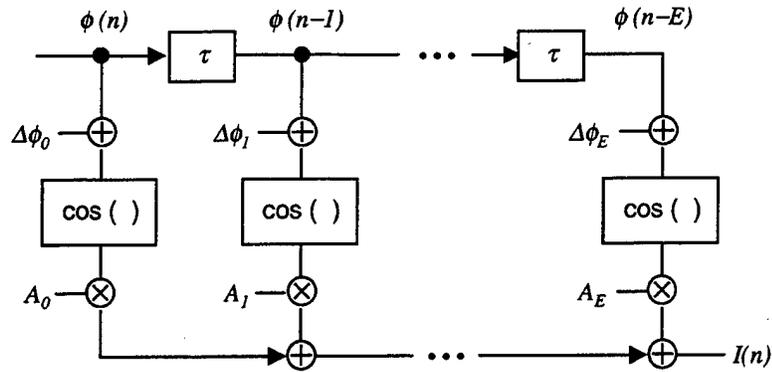
Figure 11. Original DIS Architecture for In-Phase Processing

The "modified architecture" was developed while investigating a move from field-programmable gate-array (FPGA) technology (Altera's Max+Plus II) to an application-specific integrated-circuit (ASIC). A block diagram of the modified architecture is illustrated in Figure 12. The modified algorithm enables loading all tap processors synchronously with the DRFM-phase data. The DRFM-phase data is processed in parallel in all tap processors in a pipelined fashion. The results from the taps are then added together by partial sums (serial summation) from one tap to another. The major difference between the original architecture and the modified architecture is that the time delay processor is embedded within the summation at the output. For both of the approaches described above, it is essential that the individual taps be sequentially enabled during the start-up or initial strobing of the phase data from DRFM into the tapped delay line. The taps must also be sequentially disabled during shutdown as the phase data leaves the DIS. This avoids the problem of erroneous data from entering into the summation during start-up and shutdown. More details concerning the change of technology is addressed in Chapter VII.

27
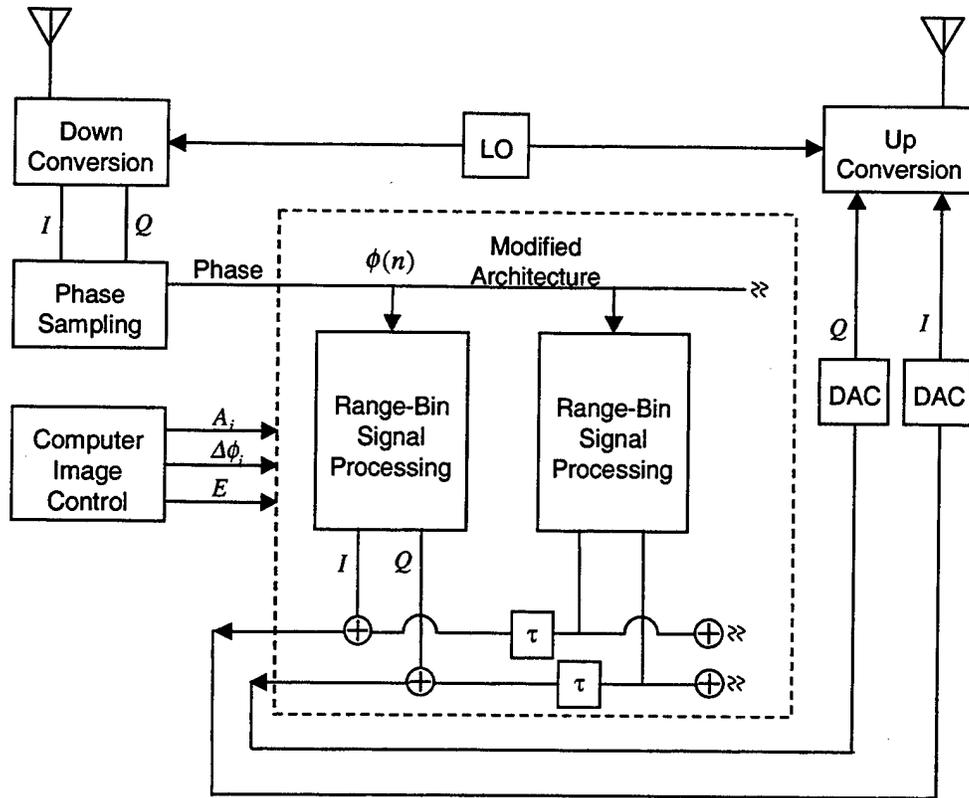
Figure 12. Block Diagram of the Modified DIS Architecture

The details for the modified DIS are shown in Figure 13.
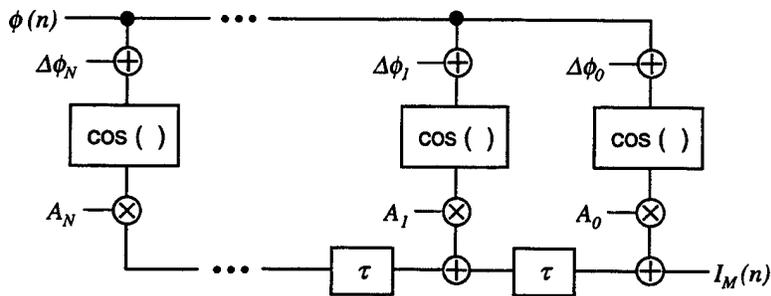
Figure 13. Modified DIS Architecture for In-Phase Processing

Proving the correctness of the modified algorithm relative to the original gives the following expressions:

$$I_M(n) = A_0 \cos(\phi(n) + \Delta\phi_0) + D^1[A_1 \cos(\phi(n) + \Delta\phi_1)] + \cdots + D^N[A_N \cos(\phi(n) + \Delta\phi_N)] \quad (4.2)$$

28

where D is a delay operator. Rewriting $I_M(n)$ gives:

$$I_M(n) = A_0 \cos(\phi(n) + \Delta\phi_0) + A_1 \cos(\phi(n-1) + \Delta\phi_1) + \cdots + A_N \cos(\phi(n-N) + \Delta\phi_N) \quad (4.3)$$

or

$$I_M(n) = \sum_{i=0}^{N} A_i \cos(\phi(n-i) + \Delta\phi_i) \quad (4.4)$$

which is exactly (4.1).

B. SIMULATION OVERVIEW

To evaluate the performance of the architecture and to compare the results of the hardware implementation, we constructed a Matlab simulation of both the DIS and an ISAR as shown in Figure 14. Some of the essential features of an ISAR are simulated including the wideband chirp pulse waveform that is intercepted by the DIS. The DRFM/DIS is also simulated. The complex outputs from the DRFM/DIS are presented to the ISAR signal processing for image generation. Matlab has also been used in several intermediate steps to compare simulation results with actual and simulated hardware design results.
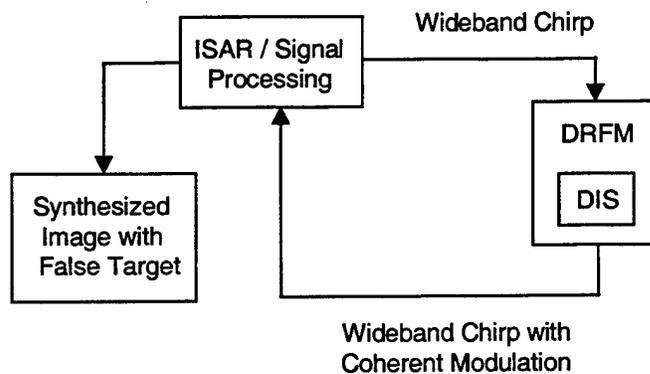
Figure 14. ISAR–DIS Simulation Configuration

29

Matlab is a product from the MathWorks, Inc., and it is an integrated-technical computing environment that combines numeric computation, advanced graphics and visualization, and a high-level programming language [Ref. 5]. Matlab includes several useful functions for:

- Data analysis and visualization

- Numeric and symbolic computation

- Engineering and scientific graphics

- Modeling, simulation, and prototyping

- Programming, application development, and graphical user interface (GUI) design.

Matlab can be used in a variety of application areas including signal and image processing, control system design, financial engineering, and medical research. It features a family of application-specific toolboxes, containing comprehensive collections of functions for solving particular classes of problems in areas, such as signal processing, image processing, control system design, neural networks, and more. The current version of Matlab used in this project is version 5.3.

In FY98, Siew-Yam Yeo developed the original set of codes during his thesis work at the Naval Postgraduate School [Ref. 6]. This set of codes has been modified to better serve the purpose of further development in the project. For example, the "original" code has been modified to deal with more then three taped delay lines. This set of codes all end with a "...v1.m" extension. Parallel to the development of the ASIC hardware design (modified DIS architecture), simulations were developed to emulate the new design. The new codes are used to verify that the newly-modified architecture is

giving the correct results. This set of codes all end with a "...v2.m" extension. Two

additional set of codes has been developed to deal with multiple scatterer per range-bin.

Version 3 ("...v3.m") varies phase modulation coefficients between radar pulses. Version

4 ("...v4.m") varies both phase and gain modulation coefficients.

C. SIMULATION DETAILS

The different steps of the simulation are easily identified by using numerous

comments within the set of simulation codes (m-files). A description of the steps,

together with some intermediate results is given below in order to visualize the

development process. The flowchart shown in Figure 15 together with Table 1

summarizes the different Matlab files used during the simulation. Important text files are

also listed.
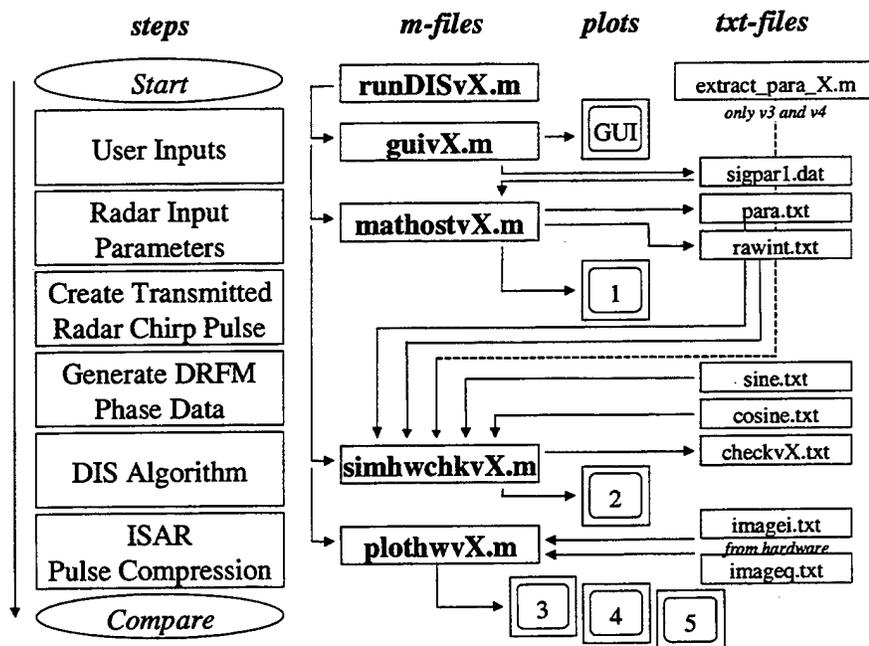
Figure 15. Matlab Simulation Flowchart

31

m-files	txt-files	Remarks
runDISvX.m		To execute the simulation
guivX.m		To get user inputs of the false target to be generated
	sigpar1.dat	Signal parameters of the false target to be generated
extract_para_X.m		Extracts parameters for multiple scatterer per range-bin
mathostvX.m		Simulates the ISAR transmitted pulse Simulates the DRFM at the DIS location
	para.txt	Number of range-bins of the ISAR Number radar pulses to be processed (integrated) Target extent Amplitude settings for each cell Phase values representing an increasing/decreasing Doppler shift
	rawint.txt	DRFM-phase data samples
simhwchkvX.m		Simulates the DIS algorithm
	cosine.txt	Cosine look-up table, 32 values for one period
	sine.txt	Sine look-up table, 32 values for one period
dec2two.m		Matlab function that converts decimal number to two's complement binary representation
two2dec.m		Matlab function that converts two's complement binary representation to decimal number
	checkvX.txt	Intermediate results through the DIS algorithm
	imagei.txt	Hardware/hardware simulation results (I-channel)
	imageq.txt	Hardware/hardware simulation results (Q-channel)
plothwvX.m		Pulse compresses the radar return of the false target generated by the DIS hardware Plots the final results for comparison

Table 1. Files Used during the Matlab Simulation

A selection of the m-files mentioned in the table above, and some other important m-files used in this thesis (referred to in later chapters), together with the cosine.txt and the sine.txt files are attached in Appendix A.

1. User Input

To run the simulation, the user executes the runDISv1.m or the runDISv2.m file depending on whether the original or the modified architecture is desired (afterward the files are referred to as "...vX.m"). The runDISvX program is a script file to execute other script files in a pre-defined order. The user is presented with a graphical user interface (GUI) of a Range/Doppler map–the Range-Doppler-Amplitude Map Entry Program guivX.m is shown in Figure 16 (runDISvX.m executes guivX.m).
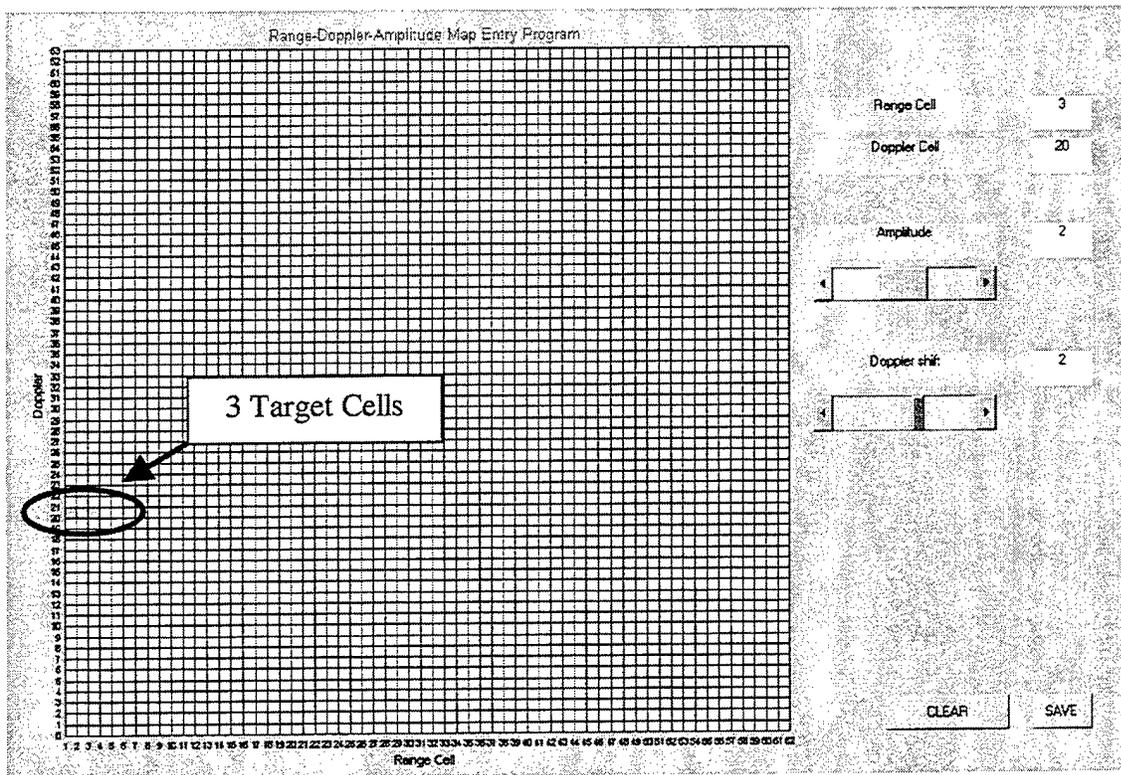
Figure 16. The Range-Doppler-Amplitude Map Entry Program

In this example the user has specified the following data to generate the false target using the DIS shown in Table 2.

Target Cell	Range Cell	Doppler Cell	Amplitude	Doppler Shift	Remark
1	1	20	2	0	Tap 0 – 1st Tap
2	2	20	2	1	Tap 1 – 2nd Tap
3	3	20	2	2	Tap 2 – 3rd Tap

Table 2. User Specified Inputs of the False Target

The values, called signal parameters of the false target, are written to an intermediate file that is called sigpar1.dat. Examining the sigpar1.dat file for this case will give the values shown in Table 3. The file only holds the numerical values. The header of the table has been applied later to explain what the different values relate to.

Range Cell	Doppler Cell	Amplitude	Doppler Shift
1.0000000e+000	2.0000000e+000	2.0000000e+000	0.0000000e+000
2.0000000e+000	2.0000000e+000	2.0000000e+000	1.0000000e+000
3.0000000e+000	2.0000000e+000	2.0000000e+000	2.0000000e+000

Table 3. Contents of the File sigpar1.dat

2. Defining the Radar Parameters

The next file to be executed by the runDISvX.m file is mathostvX.m. The mathostvX.m file represents both the ISAR while generating the transmitted chirp pulse and the DRFM on the platform where the DIS is located. The radar specific parameters of the ISAR are coded into this program. In this case the radar parameters used is shown in Table 4.

ISAR Theoretical Parameter	Value	Matlab Equivalent Variable	
		Version 1 and 2	Version 3 and 4
Uncompressed pulse width, τ	500 ns	pw	pw
Compressed pulse width, τ_c	8 ns	pwc	pwc
Pulse repetition frequency, PRF	2 kHz	prf	prf
Pulse repetition interval, PRI	500 μs	pri	pri
Bandwidth of the chirp pulse, BW	125 MHz	bw	$bw2$
Pulse compression rate, K	2.5×10^{14}	$mu=2\pi(bw/pw)$	k
Sampling frequency, f_s	125 MHz	fs	fs
Sampling time step, t_s	8 ns	Ts	Ts

Table 4. Defined Radar Parameters (file mathostvX.m)

3. Creation of the Intercepted Radar Signal

The signal parameters specified by using the GUI used to create the baseband complex signal represented by

$$S_b(t) = \text{rect}\left(\frac{t}{T}\right) e^{j2\pi\left(f_d PRI + K t^2/2\right)} \tag{4.5}$$

where f_d is the Doppler frequency of the DIS platform intercepting the chirp signal. Note that this expression is similar to (2.1) where the parameter K is the chirp slope-rate and T is the pulsewidth. The Doppler frequency f_d must be taken into consideration when building the received chirp waveform in the DIS simulation. An approximation is used that assumes a constant phase change due to Doppler within a chirp pulse. This assumption is valid since the Doppler shift is only tens of hertz compared to the MHz chirp bandwidth. The wideband intercepted signal is then phase sampled and the phase is quantized into 5-bits or 32 different values, representing a phase between 0 and 2π radians. The values used are 0 to 31 as a decimal representation of a 5-bit binary word

35

$(2^5 = 32)$. The DRFM-phase data is written to a text file (rawint.txt) that is read by simhwchkvX.m. An example of the DRFM-phase data matrix contained in the rawint.txt file is shown in Table 5. The file only holds the numerical values. The rows of the matrix represent radar pulses. The columns represent DRFM-phase data samples from a specific radar pulse at specific sampling times. The variable names used in Matlab are also shown.

Radar Pulse (batchCnt)	DRFM-phase Data (intraPulseCnt)													
	1	2	3	4	5	6	7	8	9	10	.	.	.	62
1	0	0	0	0	0	0	5	5	10	15	.	.	.	10
2	15	15	15	15	15	15	15	20	20	25	.	.	.	25
3	25	25	25	25	25	25	31	31	4	9	.	.	.	4
4	10	10	10	10	10	10	10	15	15	20	.	.	.	20
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
64	25	25	25	31	31	31	31	4	4	0				9

Table 5. Contents of the File rawint.txt

The impulse-response waveform used in the ISAR range-compression algorithm is also computed when executing this file. The amplitude and Doppler frequency-shift values for each range-Doppler cell are also obtained from the GUI and represent the gain- and phase-rotation values required for the DIS.

A number of different values are written to another text file (para.txt). The values are used for simulating the DIS both in Matlab and in the hardware design. The file only holds the numerical values. These values represent the following information (also exemplified in Table 6): number of range-bins of the ISAR, number radar pulses the ISAR is using for processing (integrating) received radar return signals, target extent (number of target cells/taps used), amplitude settings for each cell translated into a gain

value of 1, 2, 4 or 8, and set of phase values representing an increasing/decreasing Doppler shift due to the motion of the target cell relative to the ISAR

Value	Variable	Comment
62	nRangeCell	Number of Range Cells (range-bins of the ISAR)
64	nDopplerCell	Number of Doppler Cells (Doppler-bins of the ISAR)
3	targetExtent	Target Extent
2	gain(1)	Gain modulation coefficient, target cell 1
2	gain(2)	Gain modulation coefficient, target cell 2
2	gain(3)	Gain modulation coefficient, target cell 3
0	phi(1,batchCnt)	Doppler modulation coefficient, target cell 1, 1st radar pulse
0	phi(2,batchCnt)	Doppler modulation coefficient, target cell 2, 1st radar pulse
0	phi(3,batchCnt)	Doppler modulation coefficient, target cell 3, 1st radar pulse
.	.	.
0	phi(3,batchCnt)	Doppler modulation coefficient, target cell 1, 3rd radar pulse
1	phi(3,batchCnt)	Doppler modulation coefficient, target cell 2, 3rd radar pulse
2	phi(3,batchCnt)	Doppler modulation coefficient, target cell 3, 3rd radar pulse
.	.	.
0	phi(1,batchCnt)	Doppler modulation coefficient, target cell 1, 64th radar pulse
31	phi(2,batchCnt)	Doppler modulation coefficient, target cell 2, 64th radar pulse
63	phi(3,batchCnt)	Doppler modulation coefficient, target cell 3, 64th radar pulse

Table 6. Contents of the File para.txt

Remark: For Version 4 – Gain modulation coefficients are *gain(tap,batchCnt)*. That is, the gain modulation coeffiecients are individual for each tapline (tap) and radar pulse (batchCnt).

The range-Doppler image from the ISAR signal-processing simulation is plotted in Figure 17 to visualize the effect of the amplitude and the Doppler frequency-shift values shown in Figure 16.
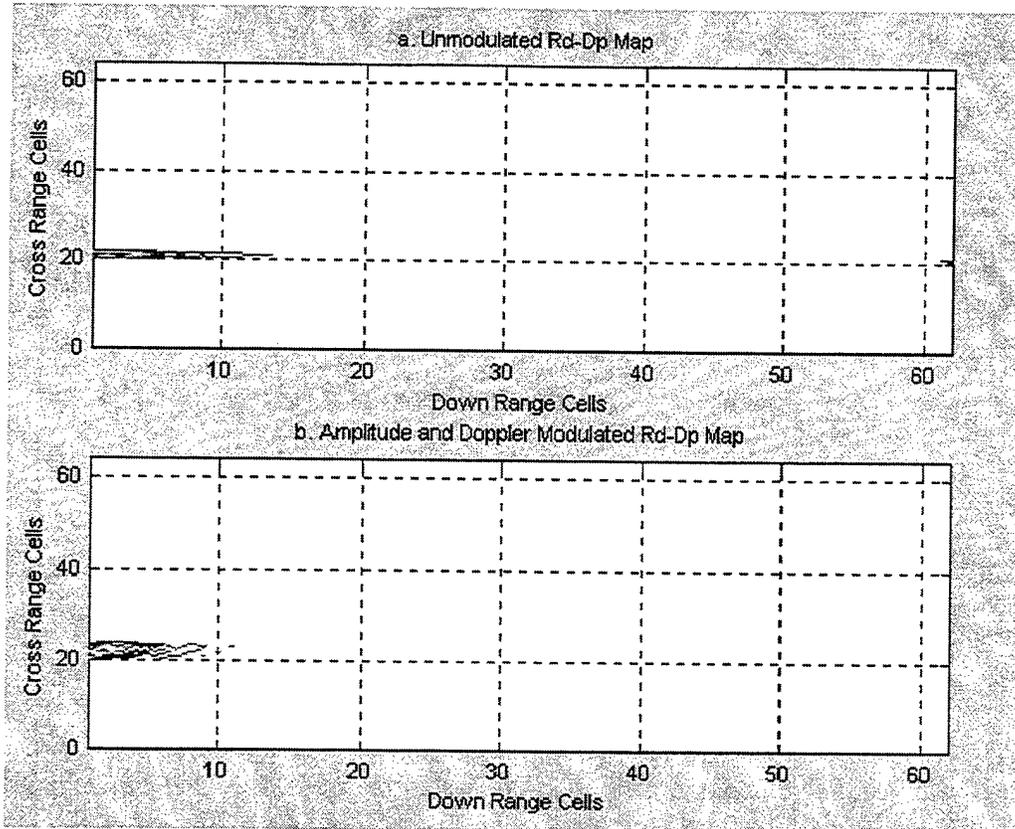
Figure 17. ISAR Range-Doppler Image with (a) No Amplitude or Doppler Frequency
Shift and (b) Amplitude and Doppler Frequency Shift as Shown in Table 2.

Figure 17 (a) represents the ISAR range-Doppler image but contains no amplitude
or Doppler frequency shift. Figure 17 (b) shows the ISAR range-Doppler image with
amplitude and Doppler frequency shift as shown in Table 2.

4. Simulation of the DIS (Original and Modified Architecture)

To simulate the DIS algorithm, the runDISvX program executes the
simhwchkvX.m file, which starts by reading in the values from para.txt. The number of
Doppler cells within the range-Doppler map is used as an index for an outer for-loop in
the program for processing phase data from one radar pulse to the next. The number of

range-bins within the range-Doppler map is used as an index for an inner (nested) for-loop and represents the number of clock pulses it takes to process the DRFM-phase data from one radar pulse to the next. The target extent represents the number of taps in the tap delay line. A target cell is also referred to as a tap in the DIS algorithm. The number of target cells specified in the GUI is therefore equivalent to the number of taps used to create a false target. The gain value selected for each tap along with the corresponding Doppler frequency shift are recorded and relate to the synthesized motion of each target cell.

Next the DRFM-phase data from the rawint.txt file is read. The program also loads data from cosine.txt and sine.txt. These files hold data used as the look-up table (LUT) and contain one period of a cosine waveform and a sine waveform (32 values) as shown in Figure 18.
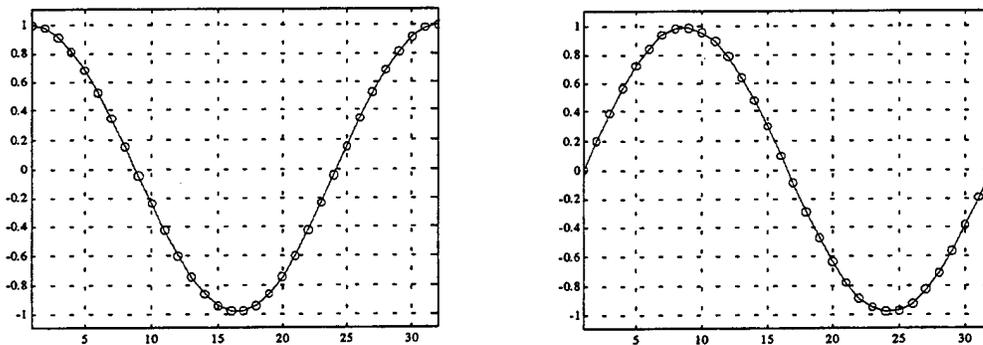
Figure 18. Cosine and Sine Look-Up Table (LUT)

Recall that the LUT translates the input phase (from the phase accumulator) into a complex signal. Using the for-loops, the DIS algorithm modulates the phase data to compute the signal that represents the return signal corresponding to the desired false

target. The original and the modified architecture calculate the modulation and perform the computation in different ways as described earlier.

In the original DIS architecture the DRFM-phase data propagates serially from tap to tap during one clock-pulse time delay. The phase data at each tap is then modulated and the results from all taps are summed together to form the output. In the modified DIS architecture, the DRFM-phase data is presented to all the taps synchronously. The phase data in this case, is processed in parallel in all taps. The delay is implemented during summation of the results from each tap. The individual taps are enabled during the start-up and disabled during shutdown according to the reasons described earlier.

In the Matlab simulation several sets of DRFM-phase data representing samples from a number of radar pulses are processed directly one after another. In an actual implementation, the set of DRFM-phase data will of course be separated in time by one PRI.

5. Range and Azimuth Compression

At the receiver side of the ISAR, as part of the signal processing, the radar return signals containing the generated false target are compressed both in range and azimuth.

First range compression is done. Range compression is based on correlating the received signal, $S(n)$ with a pre-stored reference waveform (also refer back to Chapter II, Digital range compression and (2.6)):

$$h(n) = \sqrt{\frac{j\Delta}{T}} e^{-j\pi K (n\Delta t)^2} . \qquad (4.6)$$

The FFT is performed on the received signal. The resulting spectrum is multiplied by the complex conjugate of the FFT of the reference waveform (4.6) created in the

mathostvX.m file. The procedure is shown in Figure 19. An inverse FFT (IFFT) is then performed to obtain the range-bin profiles for each PRI.
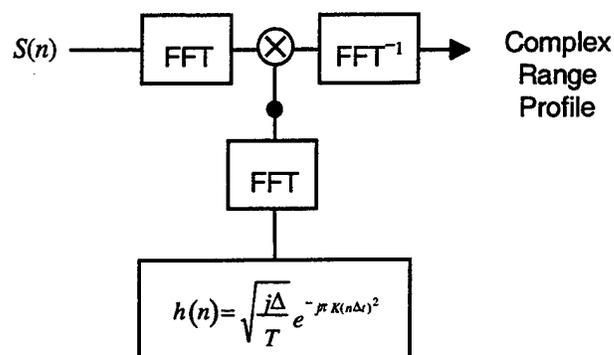
Figure 19. Range Compression

For the azimuth compression for a single range-bin, the complex range samples are taken from 2^n pulses and integrated into an FFT (also refer back to Chapter II, Azimuth compression process). The magnitude of the FFT output is the Doppler profile for that particular range-bin as shown in Figure 20.
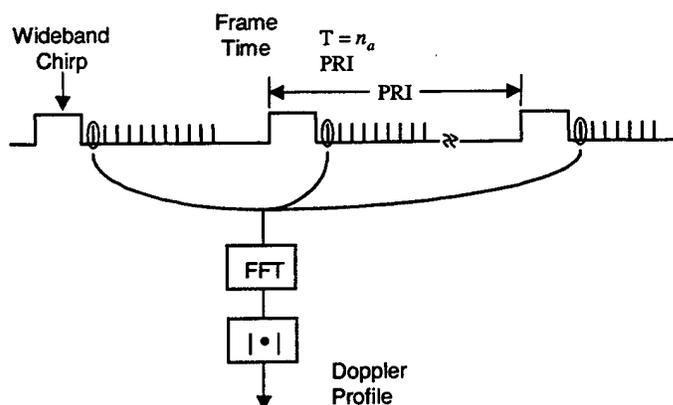
Figure 20. Azimuth Compression

The received signal after compression can be visualized as a contour plot as shown in Figure 21 and is referred to in the second sub-plot below as the Matlab Simulation plot (Amplitude and Doppler Modulated Range-Doppler Map).

41
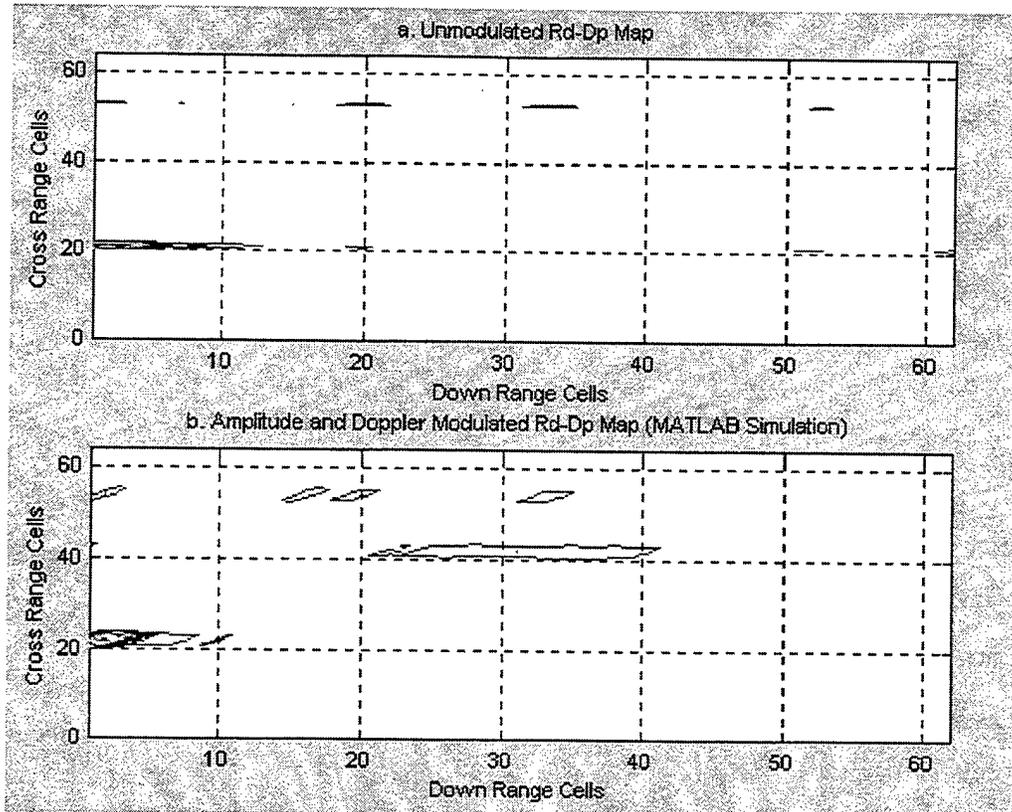
Figure 21. ISAR Range-Doppler Images Showing (a) the Unmodulated DIS Output

and (b) the Modulated DIS Output (Matlab Simulation)

6. Plot and Compare Results

The last file to be executed by the runDISvX program is the plothwvX.m. This file obtains the I- and Q-values of the hardware simulation from the imagei.txt and the imageq.txt file (written by Altera/Visual Basic FPGA hardware program). Range and azimuth pulse compression is performed using the same procedure as described for the Matlab simulation results. The results are plotted for comparison. The DIS simulation results are shown in the first sub-plot of Figure 22. In the second sub-plot, the hardware or hardware simulation result are shown when data is available. A full comparison is

42

shown in the following chapters when the different hardware implementation techniques are described.
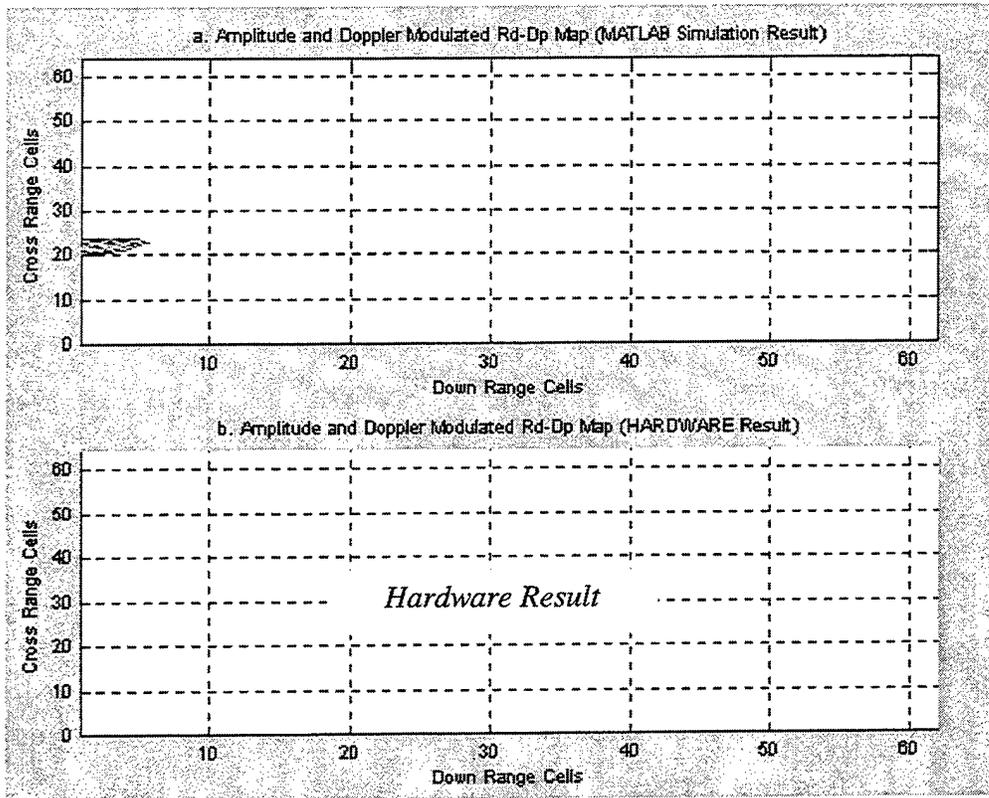
Figure 22. Matlab DIS Simulation vs. Hardware Result

To better visualize the image created by the DIS (the generated false target seen by the ISAR), Matlab uses the same data as before to construct a 3-D mesh surface plot, as shown in Figure 23. The first sub-plot shows the result of the Matlab DIS simulation. The second sub-plot shows the hardware (or hardware simulation) result. Finally, the third sub-plot shows the difference between the Matlab simulation and the hardware results.

43
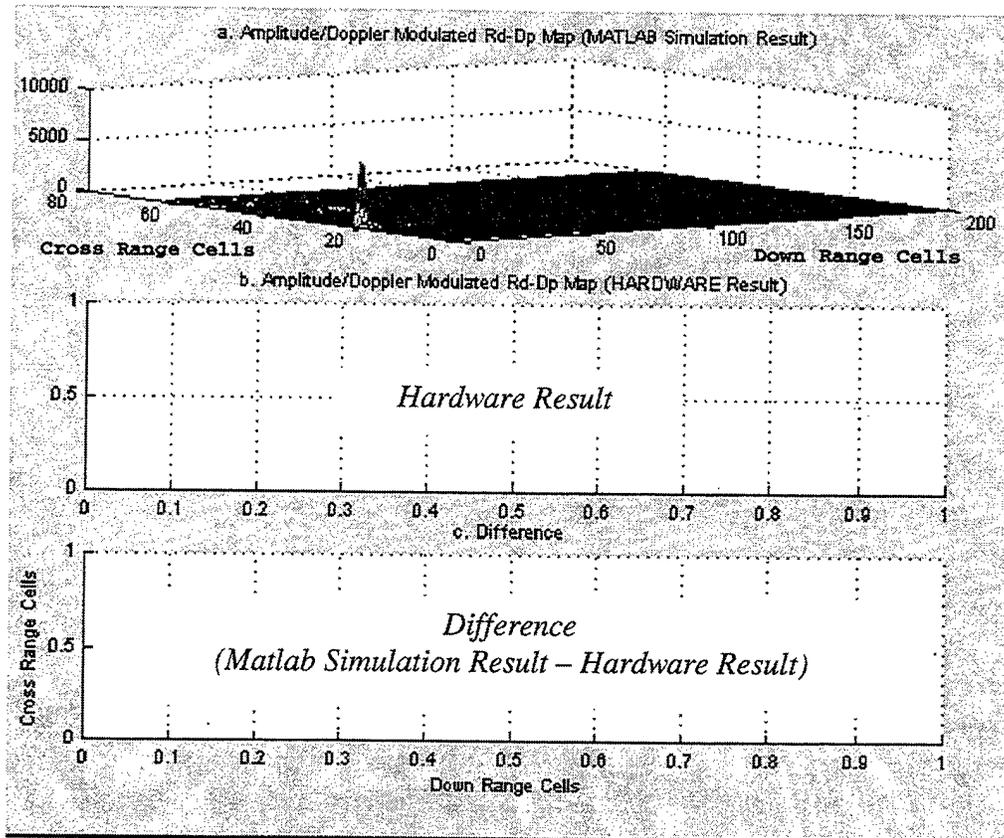
Figure 23. Matlab Simulation Result vs. Hardware Result and the Difference

To better study the results from the DIS simulation, the ISAR image of the false target is exposed, as shown in Figure 24. The user defined target cells, after DIS modulation and ISAR signal processing (range and azimuth compression) stand out clearly from the background in the plot.
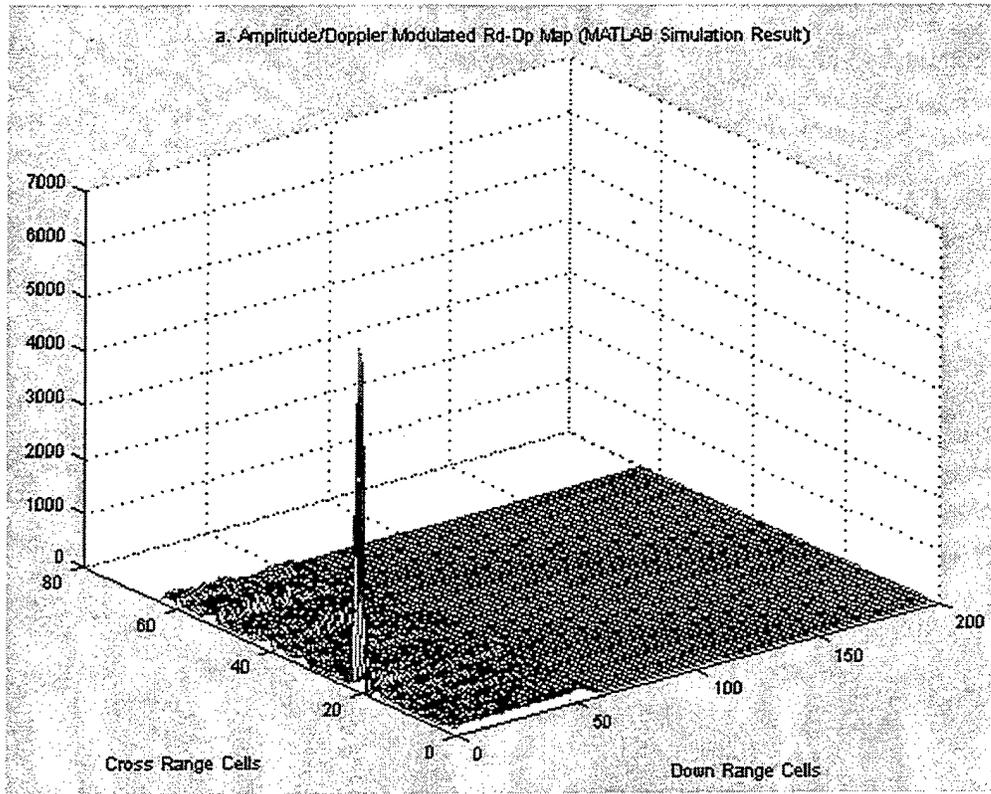
Figure 24. .Matlab Simulation Result (3-D Mesh Surface Plot)

7. Original and Modified DIS Comparison

To ensure that both the original and modified algorithms produce the same result, a series of comparisons for different test cases were conducted. The example below shows the ISAR output when using the different algorithms. It also shows the ability to modulate the extent of the false target using a large number of taps. In the test case below 32 taps are used. Figure 25 shows the input target entry. Table 7 shows the amplitude and Doppler offset values selected for the 32 range-bin false target to be synthesized.

45
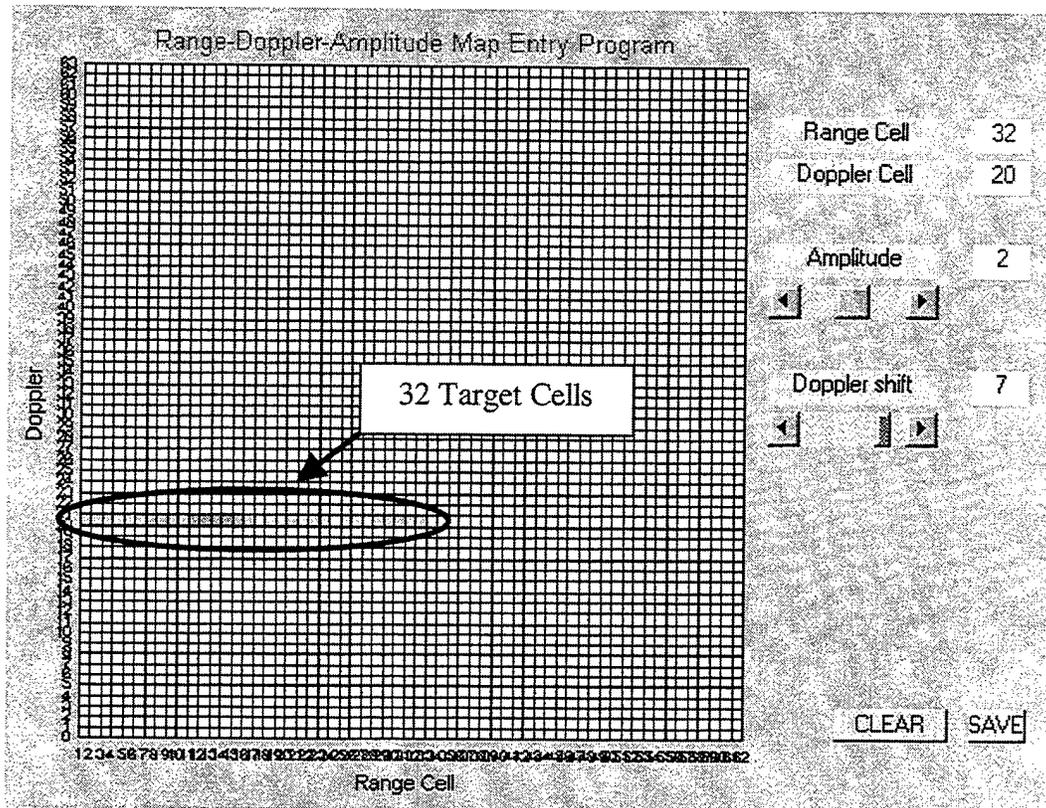
Figure 25. The Range-Doppler-Amplitude Map Entry Program

Target Cell	Range Cell	Doppler Cell	Amplitude	Doppler Shift	Remark
1	1	20	2	-8	Tap 0 – 1st Tap
2	2	20	2	-8	Tap 1 – 2nd Tap
3	3	20	2	-7	Tap 2 – 3rd Tap
4	4	20	2	-7	Tap 3 – 4th Tap
5	5	20	2	-6	Tap 4 – 5th Tap
6	6	20	2	-6	Tap 5 – 6th Tap
7	7	20	3	-5	Tap 6 – 7th Tap
8	8	20	3	-5	Tap 7 – 8th Tap
9	9	20	3	-4	Tap 8 – 9th Tap
10	10	20	3	-4	Tap 9 – 10th Tap

Target Cell	Range Cell	Doppler Cell	Amplitude	Doppler Shift	Remark
11	1	20	4	-3	Tap 10 – 11th Tap
12	12	20	4	-3	Tap 11 – 12th Tap
13	13	20	4	-2	Tap 12 – 13th Tap
14	14	20	4	-2	Tap 13 – 14th Tap
15	15	20	3	-1	Tap 14 – 15th Tap
16	16	20	3	-1	Tap 15 – 16th Tap
17	17	20	2	0	Tap 16 – 17th Tap
18	18	20	2	0	Tap 17 – 18th Tap
19	19	20	2	1	Tap 18 – 19th Tap
20	20	20	2	1	Tap 19 – 20th Tap
21	21	20	1	2	Tap 20 – 21st Tap
22	22	20	1	2	Tap 21 – 22nd Tap
23	23	20	1	3	Tap 22 – 23rd Tap
24	24	20	1	3	Tap 23 – 24th Tap
25	25	20	1	4	Tap 24 – 25th Tap
26	26	20	1	4	Tap 25 – 26th Tap
27	27	20	1	5	Tap 26 – 27th Tap
28	28	20	1	5	Tap 27 – 28th Tap
29	29	20	2	6	Tap 28 – 29th Tap
30	30	20	2	6	Tap 29 – 30th Tap
31	31	20	2	7	Tap 30 – 31st Tap
32	32	20	2	7	Tap 31 – 32nd Tap

Table 7. Amplitude and Doppler Offsets Selected for 32 Range-Bin False Target

As observed in Figure 26 and Figure 27, the two different algorithms perform the same result, which also has been proven in the previous chapter.
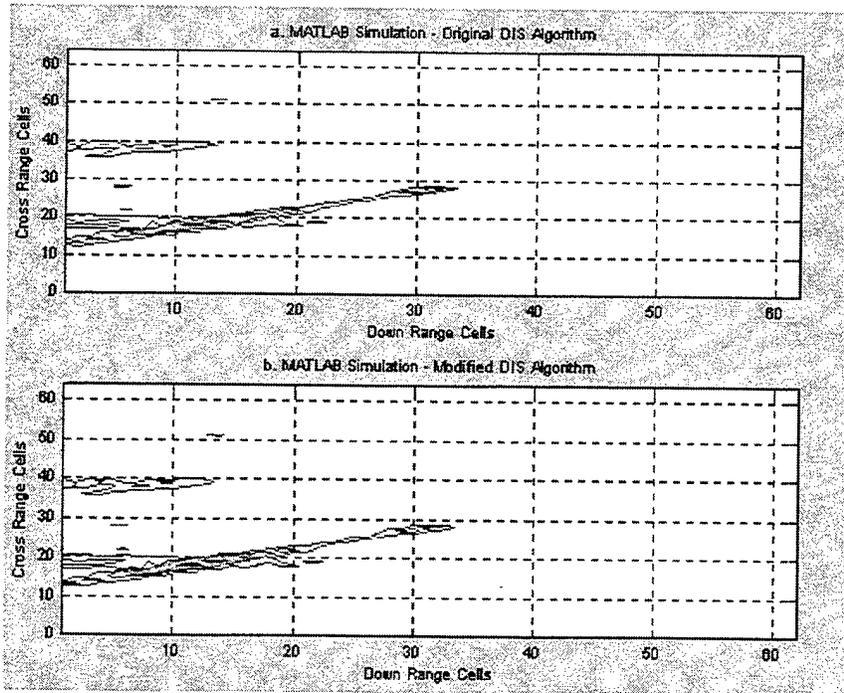
Figure 26. Original vs. Modified DIS Algorithm Simulation Results
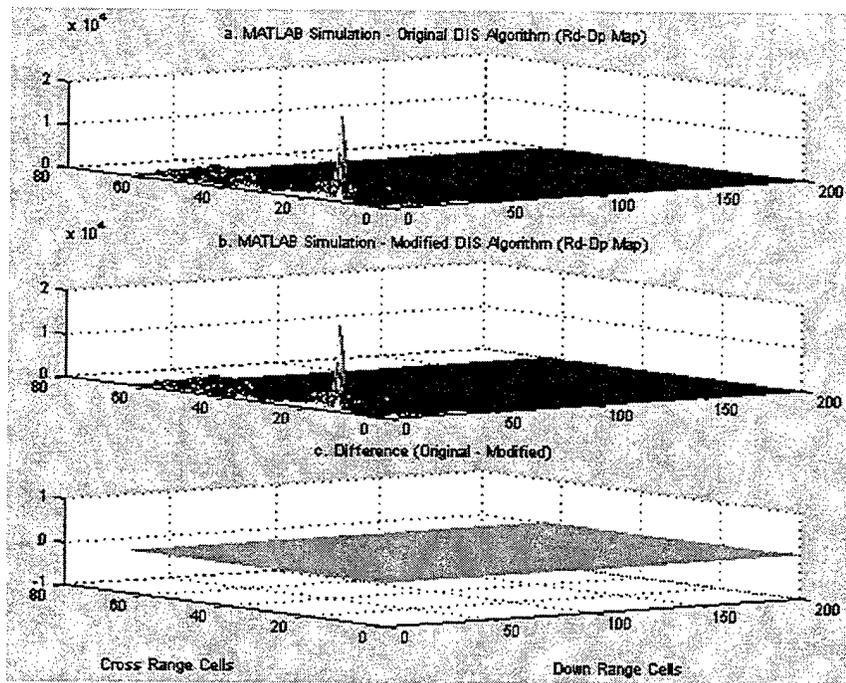
Figure 27. Original vs. Modified DIS Algorithm Simulation Results and the Difference

48

8. Multiple Scatterer Per Range-Bin

Up to this point, we have only considered one single scatterer per range-bin. The DIS mustbe able to deal with one more dimension. A true target will generate several radar returns from many different scatterers within the size of one single range-bin of the ISAR. The radar return in one range-bin can be treated as a sum of the individual scatterers radar-return signals due to superposition. The different scatterers will modulate the incident radar signal with a different gain and a different Doppler depending on factors such as shape, size, material, angle and relative motion.

Finding the corresponding amplitude values (gain modulation) and phase values (phase modulation) at a certain time interval and using these values to modulate the DRFM-phase samples in the DIS would then represent a combined radar-return signal for one entire range-bin. For this, the DIS must be able to process variable gain and phase modulation coefficients between radar pulses. This will also be addressed in further detail in later chapters where different hardware implementation techniques are discussed. The latest developed Matlab codes (version 3 and 4) can deal with this complex situation. The parameters (gain and phase modulation coefficients) must first be determined for the shape and motion of the false target to be generated. This has been done manually by mapping out the shape and specifying Doppler frequencies to each scatterer.

To illustrate the procedure, consider a simple "V" shape of a small set of scatterers. Each individual scatterer can be plotted in a range-Doppler map, where the different range-bins are on the x-axis, and the different Doppler-bins are on the y-axis. For this example, nine different scatterers were used, all with the same gain. The scatterers were assigned different initial Doppler frequencies representing differences in

relative motion to the ISAR. Some of the scatterers were also located in the same range-bin. Figure 28 shows to the left a sketch of the initial setup of the shape of the false target to be generated by the DIS. In the 1st range-bin there is only one scatterer located with zero Doppler (no relative motion to the ISAR). In the 2nd to the 4th range-bins, there are two scatterers per range-bin, each with different Doppler (negative or positive). Finally, for the 5th and the 6th range-bins, there are again only one scatterer per range-bin. The individual Doppler frequencies were specified in a Matlab script file (extract_para_VcaseX.m) and running the file produced a set of gain- and phase-modulation coefficients that were then used as inputs to the DIS. The Matlab simulation result can be seen to the right in Figure 28 below.

Setup   **Simulation Result**
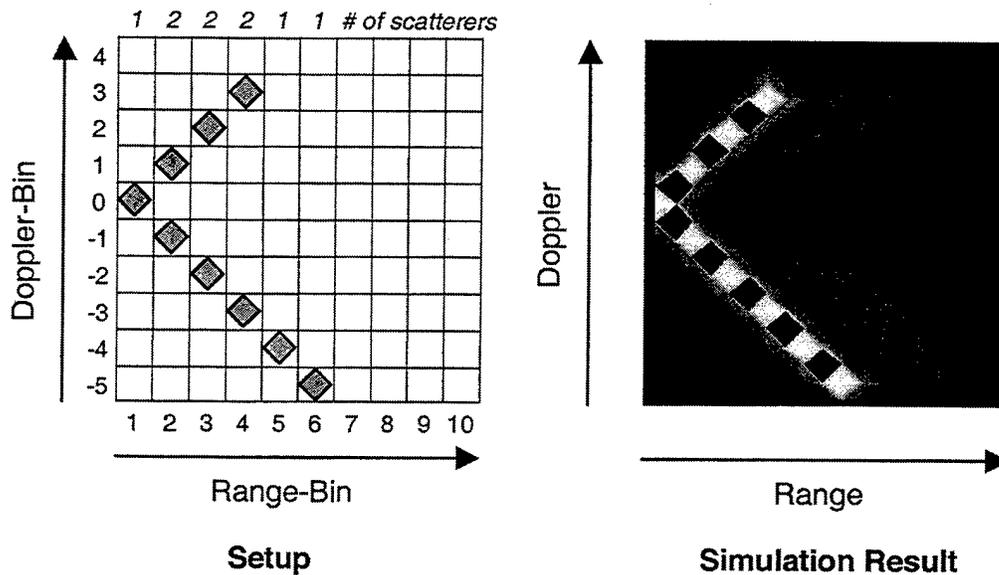
Figure 28. DIS V-Case: Setup and Simulation Result

The simulation results represent the modulated ISAR-return signals using 64 radar pulses, received by the ISAR and after performing range and azimuth pulse compression. The individual scatterers can clearly be identified (the bright spots) in the simulated

50

ISAR image of the false target. The following section presents simulation results of a more realistic false target, consisting of several scatterers per range-bins.

D. SIMULATION RESULTS

To verify correctness of the concept of the DIS algorithm, a set of larger scale simulations has also been conducted. The goal was to be able to produce a realistic ISAR image of a target, similar to a real ISAR image.

The ISAR Section of the Radar Analysis Branch at the Naval Research Laboratory (NRL) works on developing advanced algorithms and processing systems for ISAR [Ref. 7]. The picture shown below in Figure 29 is a radar image generated using ISAR imaging.
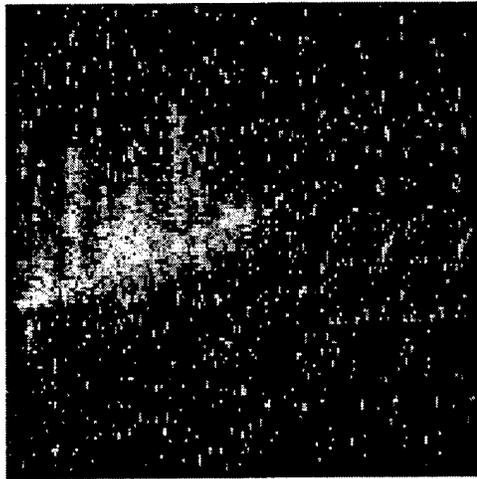
Figure 29. ISAR Image (From Ref. [7])

This image was taken by the P-3 aircraft (Figure 30). It is the image of the ship, USS Crockett, which is pictured in Figure 31.
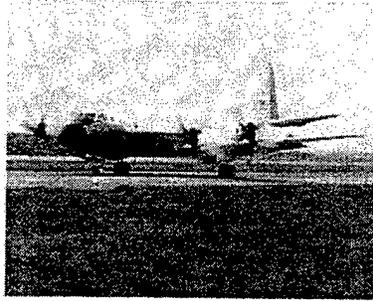
Figure 30. Photo of a P-3 Aircraft (From Ref.[7])
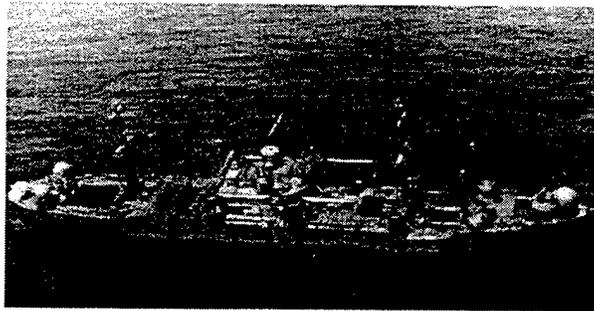
Figure 31. Photo of USS Crockett (From Ref. [7])

The ISAR image is a two-dimensional representation of the target, with the resolution in the horizontal dimension determined by the short pulse characteristic of the radar and the vertical dimension by the Doppler of the radar returns.

The radar used in the P-3 aircraft is an AN/APS-137. The APS-137 family of radars has consistently demonstrated outstanding performance in anti-submarine warfare (ASW) and anti-surface warfare (ASuW). Current operational capabilities include long-range surface search and target tracking, periscope detection in high sea states, ship imaging and classification using ISAR, and SAR for overland surveillance, ground mapping, and targeting. The radar system is produced by Raytheon and is shown in Figure 32 [Ref. 8].
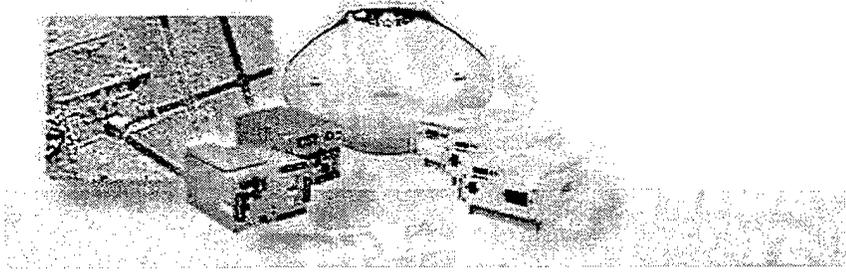
Figure 32. AN/APS-137B(V)5 Radar System (From Ref. [8])

In order to create a false target that looks similar to the NRL ISAR image of the USS Crockett, some simplifications had to be done, due to the complexity of a real target (i.e. the number of scatterers, RCS of each individual scatterer, and unknown exact radar parameters).

First, the necessary phase and gain modulation parameters of the false target had to be generated. A simplified range-Doppler map of the target is shown in Figure 33. The number of individual scatterers manually specified in range and Doppler was 182.
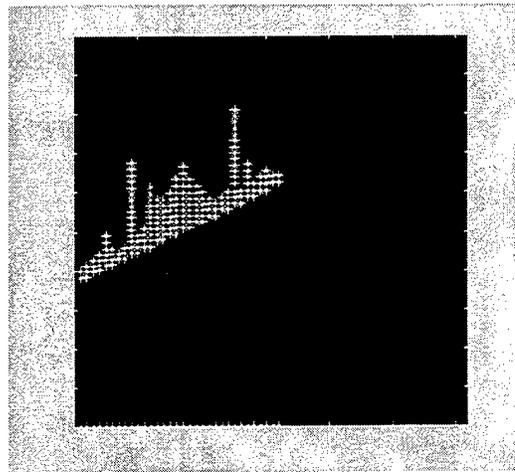
Figure 33. Ship Case–Simulation Setup in Matlab

Only 32 taps were used to represent the full length of the target, due to the limitations in the developing of the hardware equivalent circuit, and also to accelerate the simulation

time. The phase and gain-coefficients were thereafter extracted in a correct format by a Matlab script file (extract_para_ShipX.m). Most of the simulation parameters were kept the same as earlier, i.e. 62 DRFM-phase data were captured for each radar pulse. A series of seven different simulations was the conducted representing different Doppler resolutions of the counter-targeted ISAR (integration of 64, 128, 256, 512, 1024, 2048, and 4096 radar pulses respectively).

The results are shown in the following set of figures (Figure 34). The true ISAR image has of course a much higher resolution than the Matlab simulated images, which is best observed in range. The ISAR image is also a final image for the end user. That is, the radar-return signals have not only been signal processed as a radar signal, but various filtering and image enhancement techniques have also been applied. The Matlab simulation images result from only pure signal processing because of pulse compression. No additional filtering and image enhancements have been used.

Table 8 explains the differences of the nine sub-figures shown in Figure 34. The number of radar pulses referred to relates to how many radar pulses were used for the ISAR-image integration for that specific simulation. Seven different simulation results are shown.

True ISAR image	Simulation setup	64 radar pulses
128 radar pulses	256 radar pulses	512 radar pulses
1024 radar pulses	2048 radar pulses	4096 radar pulses

Table 8. True ISAR Image, Simulation Setup, and Seven Different Simulations

(a) True ISAR Image   (b) Simulation Setup   (c) 64 Radar Pulses

(d) 128 Radar Pulses   (e) 256 Radar Pulses   (f) 512 Radar Pulses

(g) 1024 Radar Pulses   (h) 2048 Radar Pulses   (i) 4096 Radar Pulses
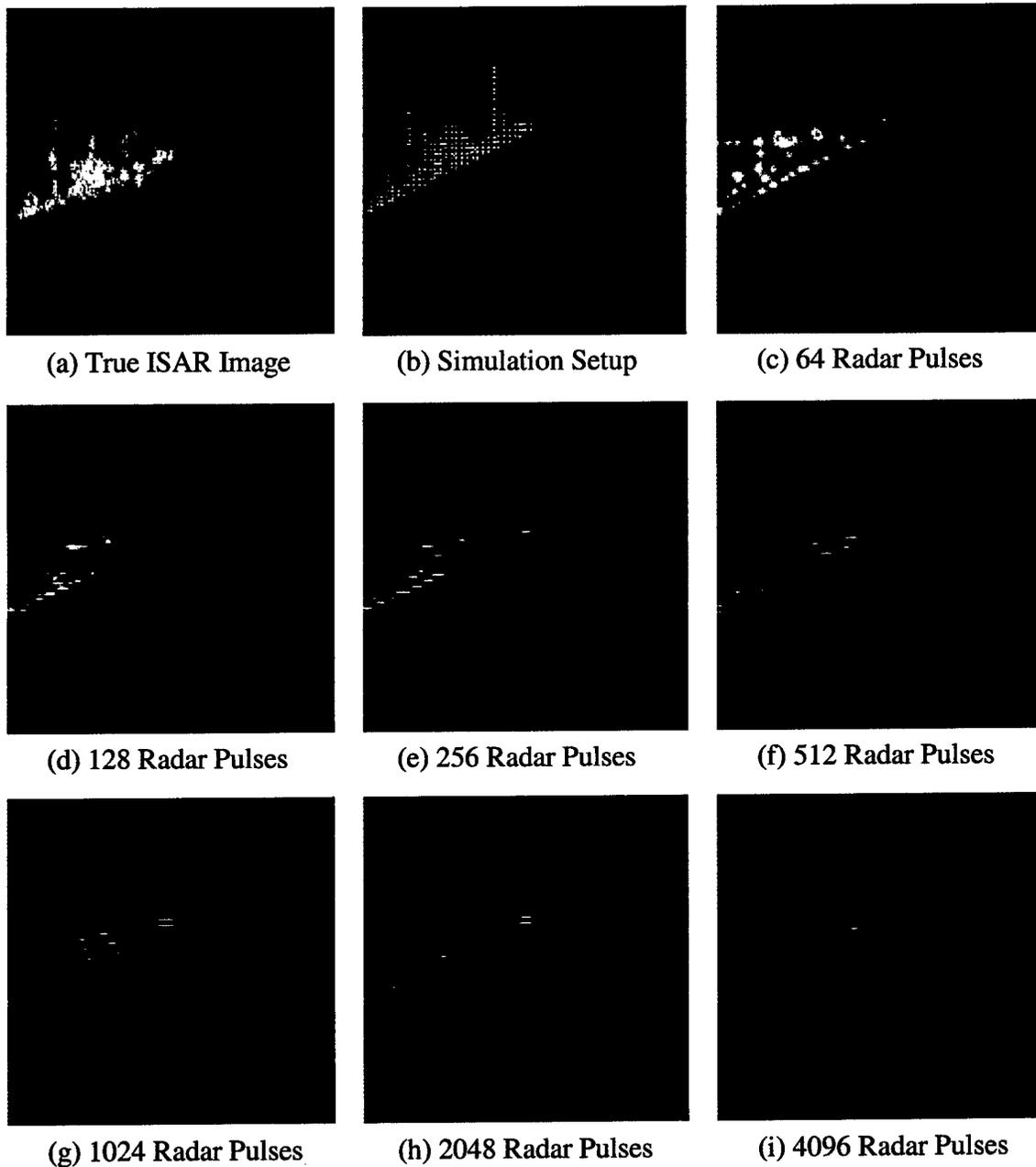
Figure 34. True ISAR Image, Simulation Setup, and Seven Different Simulations

This test case was developed to visualize the expected results of the DIS. The goal was to create a realistic image of a false target, though limiting the scatterers to a relatively small number. Other limitations were precision and dynamic range of the phase

55

and gain modulation coefficients used in the current design. In spite of that, the simulation results show that creating relative realistic false targets using a Digital Image Synthesizer as described is possible.

An additional remark is that the simulated DIS refers to a 5-bit phase sampling DRFM. The phase-modulation coefficients are 4-bit binary words, and the gain modulation coefficients are only 2-bit binary words. The final outputs (I- and Q-channel) consist of a 16-bit two's complement binary word respectively. A set of different tools (Matlab script files and function calls) has been developed for later use in order to further investigate tradeoffs in number of bits used throughout the architecture, especially for the final stages of adders and for representing the final output words. An example of how to use these files and function calls are presented in Appendix A.

V. DIS USING FIELD PROGRAMMABLE GATE ARRAYS

A. INTRODUCTION

This chapter discusses the hardware implementation of the DIS by using FPGA technology. The hardware design is captured using the Altera Multiple Array Matrix Programmable Logic User System or Max+Plus II software version 9.21 (the project was started in 1998 using version 8.3). Max+Plus II is the design environment for Altera Programmable Logic Devices (PLD). A brief description of the Max+Plus II software is given below followed by a short introduction to Field Programmable Logic Devices (FPLDs) [Ref. 9]. In particular, FPGAs, specifically the Altera 10K50 family is described. Later sections of this Chapter describe each of the modules of the DIS hardware design, starting from the top-level-hierarchy and progressing down. The final section addresses the FPGA results and the comparison to Matlab simulations.

B. THE ALTERA MAX+PLUS II ENVIRONMENT

The Max+Plus II software provides a multi-platform, architecture-independent design environment that easily adapts to specific design needs. The Max+Plus II development software is a fully-integrated programmable logic-design environment. This tool supports all Altera programmable device families and works in both PC and UNIX environments. The Max+Plus II allows seamless integration with industry-standard design entry, synthesis, and verification tools.

Figure 35 shows a block diagram of the Altera Max+Plus II environment. Max+Plus II both reads and writes:

- Altera Hardware Description Language (AHDL) files and standard EDIF netlist files

- Verilog HDL files

- VHDL files

- OrCAD schematic files
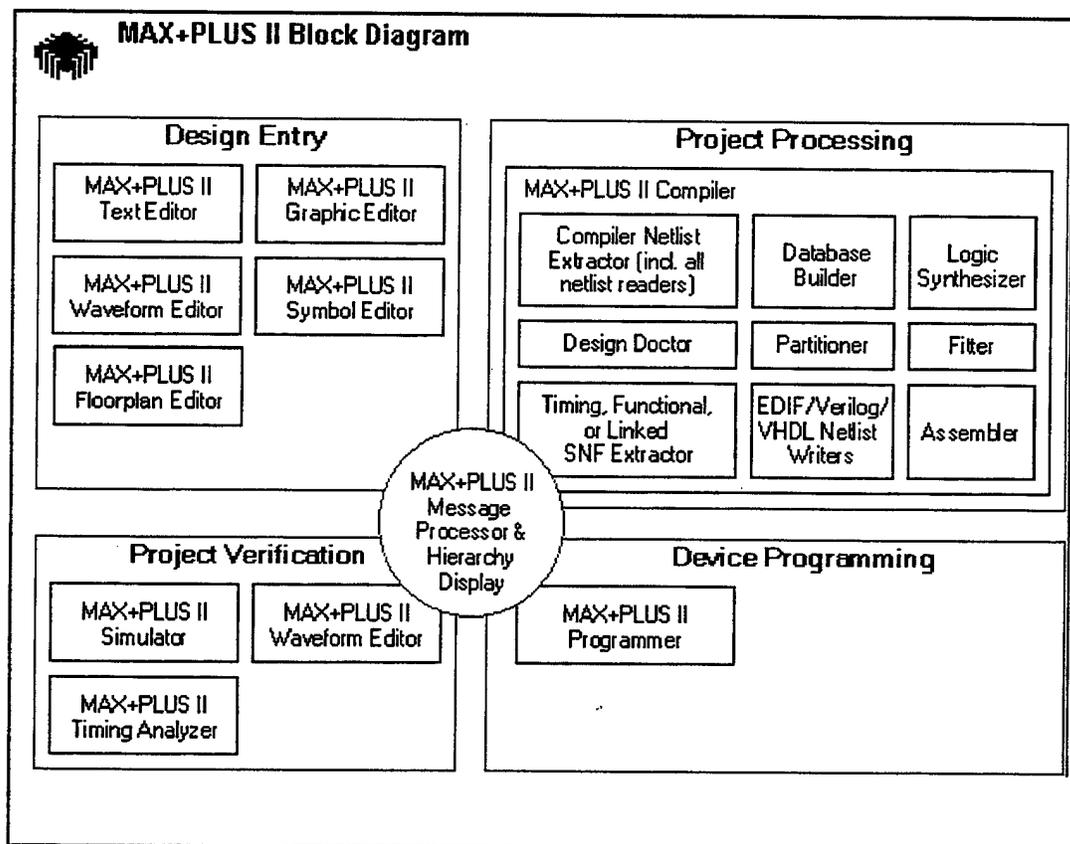
Figure 35. Altera Max+Plus II Environment (From Ref. [10])

In addition, Max+Plus II reads Xilinx netlist files and writes Standard Delay Format (SDF) files for interface to other industry-standard CAE software. The Max+Plus II message processor handles the different features like design entry, project processing,

58

project verification and device programming. An overview of the Max+Plus II compiler interface is shown in Figure 36. The hierarchy display is a convenient way to switch between the different parts of the program and shows a hierarchy tree with branches, that represents the sub-designs.

Design Verification

MAX+PLUS II Simulator
MAX+PLUS II Waveform Editor
MAX+PLUS II Timing Analyzer
Other Industry-Standard
    CAE Design Verification Tools

Design Entry

MAX+PLUS II Graphic Editor
MAX+PLUS II Symbol Editor
MAX+PLUS II Text Editor
MAX+PLUS II Waveform Editor
MAX+PLUS II Floorplan Editor
AHDL
VHDL
Verilog HDL
Other Industry-Standard
    CAE Design Entry Tools

MAX+PLUS II Compiler

Device Programming

MAX+PLUS II Programmer
Data I/O
Other Industry-Standard
    Programmers

Figure 36. Max+Plus II Design Environment (From Ref. [11])

The complete Max+Plus II system includes 11 fully integrated applications that take the designer through every step of creating a design. A logic design, including all sub-designs, is called a "project" in Max+Plus II. The main applications are summarized in Table 9.

Application	Function
Hierarchy Display	For displaying the current hierarchy of files as a hierarchy tree with branches that represents sub designs.
Graphic Editor	For entering a schematic logic design. Altera provides primitives, megafunctions, and macrofunctions, which serve as basic circuit-building blocks.
Symbol Editor	For adding existing symbol and creating new ones.
Text Editor	For creating and editing text-based logic design files written in hardware description language (AHDL, VHDL, Verilog HDL).
Application	Function
Waveform Editor	For entering test vectors and viewing simulation results.
Floor-Plan Editor	For assigning logic to physical device pins and logic cell resources in a graphic environment.
Compiler	For processing project, including checking for errors, synthesizing the logic, fitting the project into one or more Altera devices.
Simulator	For testing the logical operation and internal timing of logic circuits. The simulator supports functional simulations, timing simulations, and linked multi-device simulation.
Timing Analyzer	For analyzing the performance of the logic circuits after they have been synthesized and optimized by the compiler.
Programmer	For programming, configuring, verifying, examining and testing Altera's devices.
Message Processor	For displaying warning and information messages on the status of the project. It also locates the source of a message automatically in the original design files.

Table 9. Max+Plus II Suite of Applications and Functions (From Ref. [10])

C. FPGA TECHNOLOGY AND THE ALTERA 10K50

Different devices are available to capture the developed FPGA design file. The FLEX 10K50 chip (FLEX = flexible logic element matrix architecture) for example is a static random access memory (RAM) with typically 70,000 gates (logic & RAM). The Flex 10K50 device contains an embedded array and a logic array. The logic array performs the same function as a sea of gates in a gate array. The FLEX 10K50 is used to implement general logic, such as counters, adders, state machines, and multiplexers. The embedded array is used to implement memory and specialized logic functions. Table 10 describes the features and benefits of using FPGAs and Table 11 the features of the FLEX 10K50.

Feature	Benefit
200 MHz and above System Performance	Supports today's most demanding speed requirements
Density from 10,000 to over 1.5 Million Gates	Addresses 90% of all gate array design starts
Embedded Array Blocks	Efficient RAM, ROM, FIFO and other high-performance mega-functions
Multi-Volt I/O Operation	Ideal for mixed-voltage systems
5.0V, 3.3V, 2.5V, and 1.8V Device Options	Supports multiple operating voltages
PCI Compliance	Meets all specifications of the PCI local bus

Table 10. FLEX 10K Highlights (From Ref. [11])

The Altera FLEX 10K devices are configured at system power-up with data stored in an Altera serial configuration EPROM device or provided by a system controller. A picture of the FLEX 10K50 is shown in Figure 37.

61
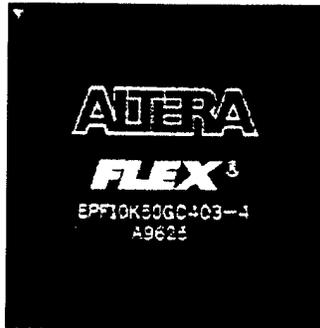
Figure 37. Altera FLEX 10K50 (From Ref. [11]).

A microprocessor interface permits the microprocessor to configure the FLEX 10K

devices serially, in parallel, and synchronously or asynchronously [Ref. 11].

The features of the FLEX 10K50 device are as shown in Table 11:

Features	FLEX 10K50
System Performance	115 MHz
Typical Gates (logic & RAM)	50,000
Logic Elements	2,880
Logic Array Blocks	360
Embedded Array Blocks	10
Total RAM Bits	20,480
Flip-Flops	3,184
Maximum User I/O Pins	310

Table 11. Altera FLEX 10K50 Device Features from [11]

D. DIS ARCHITECTURE USING FPGA

1. The Concept Demonstrator

A concept demonstrator of the DIS architecture has been developed in Field-Programmable Gate-Array (FPGA) technology. The concept demonstrator comprises three parts:

- Matlab simulations of the ISAR signal processing architecture (described in Chapter IV)

- Computer board containing hardware design using an Altera FPGA device (FLEX 10K50)

- A Visual Basic program (flextest.vbp) to access the Altera FPGA computer board and download the image-formation parameters and raw data and to upload from the board processed data. The data gathered from the board are stored in files that are in turn read by plothwvX.m for post-processing and display for comparison.

The DIS and its interface with the host computer are shown in Figure 38 as a block- and host-interface diagram. The host computer is an ordinary personal computer (PC). The DIS hardware is a FPGA (Altera 10K50 FPGA chip) mounted on a Naval Research Laboratory (NRL) custom designed PC I/O board. The various modules for the DIS are described below.
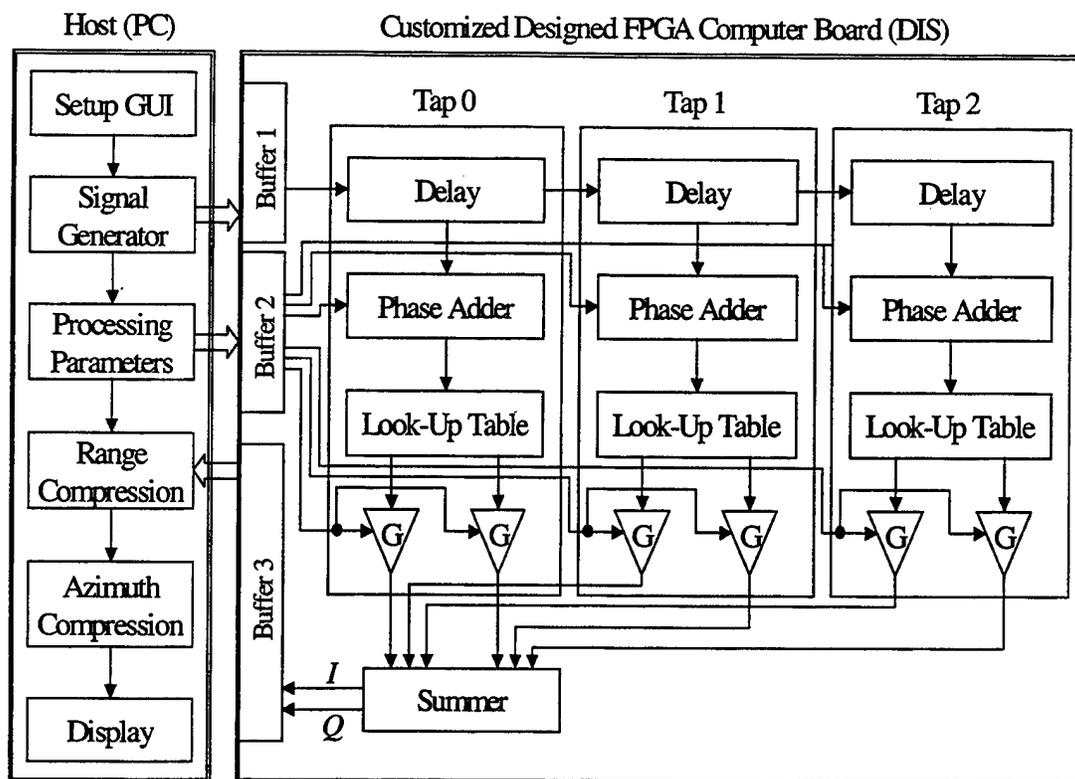
Figure 38. Block Diagram and Host-Interface Diagram of the DIS

a. Host (PC)

Setup GUI: The setup as most of the blocks of the host refers back to the Matlab code discussed in Chapter IV. In the GUI the user specifies the parameters for the false target to be generated.

Signal Generator: The DRFM-phase data samples are produced within this block and printed to a text file. This text file (rawint.txt) is used both in the Matlab simulation file and the Visual Basic program running the FPGA computer board.

Processing Parameters: The processing parameters of the specified false target consist of a phase-increment/decrement corresponding to the selected Doppler shift and the gain-coefficients representing the amplitude modulation.

Range and Azimuth compression: These parts represent basic signal processing functions in the ISAR. Pulse compression is performed on the radar return signal from the false target generated by the FPGA DIS.

Display: After processing, the signals will be presented to the user as an image. In this case, it will be done by a series of plots using Matlab (as described in Chapter 4).

b.   FPGA DIS

Buffers: "Buffer 1" is for storage of DRFM-phase data samples to be fed into the tapped delay lines. "Buffer 2" is for storage of modulation parameters, which are computed and updated by the host. These include parameters for target extent, amplitude modulation and Doppler shift. "Buffer 3" is for storing the outputs of the DIS (modulated signals).

Tap 0 to 2: Three tapped-delay lines have been implemented using the FPGA technology in order to study the trade-offs involved. Each tap consists of a delay element, implemented in hardware using a cascaded chain of flip-flops. The phase adder together with the look-up table provides a Doppler modulated complex signal. The gain modules provide amplitude modulation to the signal, represented by the triangular symbols connected to the outputs of the look-up tables.

Summer: The summer adds outputs from I- and Q-channels separately. The addition is accomplished by first taking a partial sum of the outputs from two last taps and then as an additional step, adding this result to the output of the first tap.

c. FPGA DIS Hardware

The hardware used for the DIS implementation and its interface with the host computer is shown in Figure 39 and Figure 40. Figure 39 shows a photo of the host computer, a PII 300 MHz with 128 MB RAM.
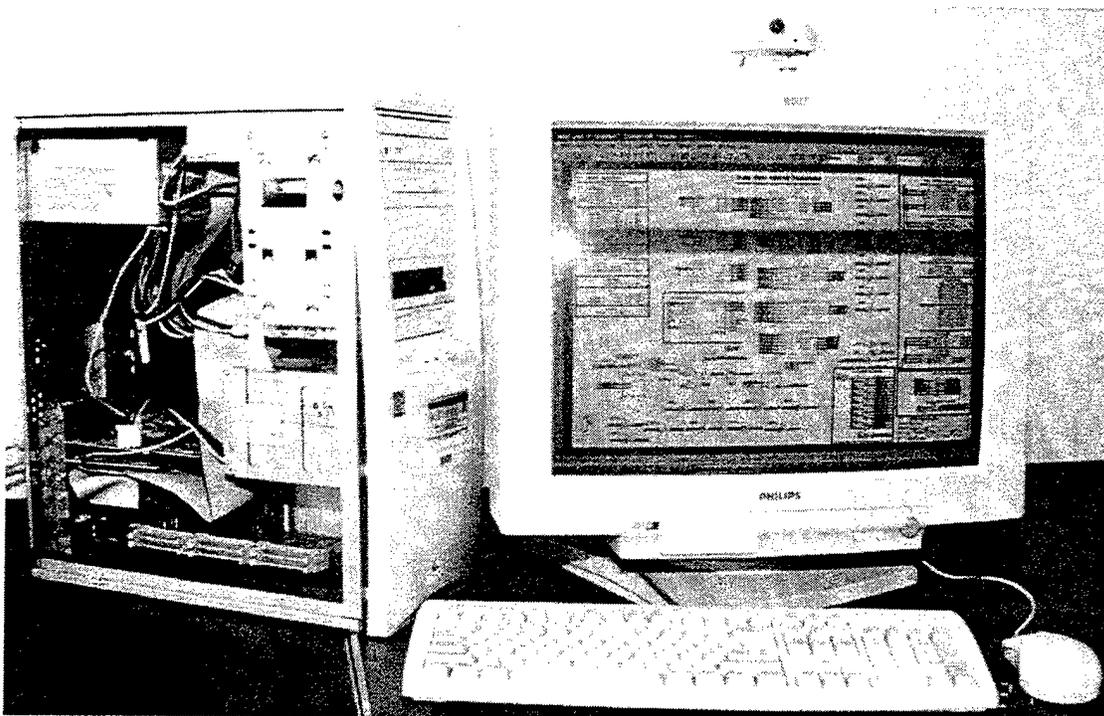
Figure 39. Picture of the Concept Demonstrator–Host (PC) with FPGA Board (DIS)

Figure 40 shows the DIS hardware consisting of a FPGA (Altera 10K50 FPGA chip) mounted on a Naval Research Laboratory (NRL) custom-designed computer board. It can be seen inserted in the lower slot of the computer. The Altera 10K50 FPGA chip is the large device in the center of the board.
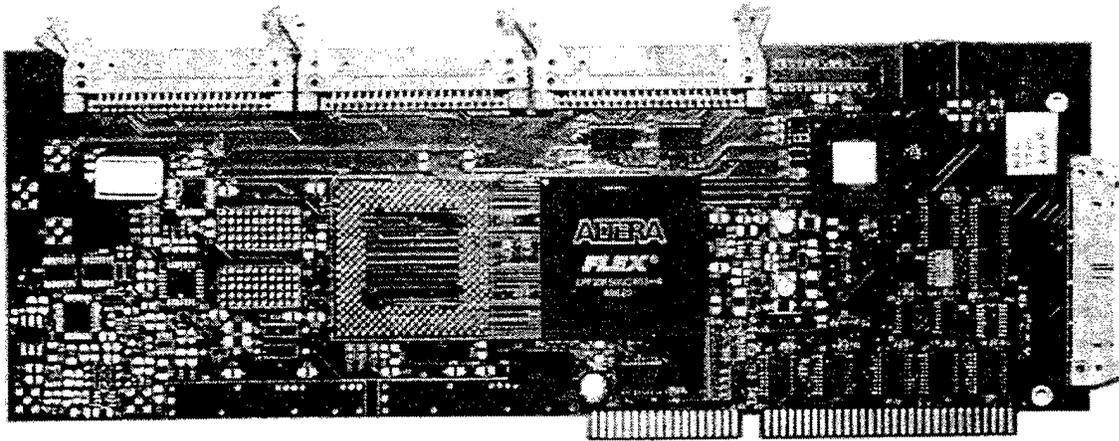
Figure 40. Picture of the Customized FPGA Board Used for the DIS Prototype

d. *Processing DRFM-Phase Data*

Processing of DRFM-phase data samples by the three-tap original DIS architecture can be visualized as follows. For each received radar chirp pulse, a set of phase samples will be provided by the DRFM. At startup, valid output data consists of only the output from the first tap (Tap 0). At the next clock cycle valid data will be the sum of processed data from Tap 0 and Tap 1. At the third clock cycle, the output will be the sum of processed data from all three taps. At the end of the pulse, the taps are shutdown in reverse order while the phase data is propagating through the delays. An example of 64 radar pulses and 62 DRFM-phase data samples (range-bins) per radar pulse are summarized in Table 12.

Radar Pulse	DRFM Data	Clk	Tap 0	Tap 1	Tap 2	Result
1	D_1	0	$P_n(D_1)$	0	0	$P_n(D_1)$
1	D_2	1	$P_n(D_2)$	$P_{n+1}(D_1)$	0	$P_n(D_2) + P_{n+1}(D_1)$
1	D_3	2	$P_n(D_3)$	$P_{n+1}(D_2)$	$P_{n+2}(D_1)$	$P_n(D_3) + P_{n+1}(D_2) + P_{n+2}(D_1)$
.	.	.	.	.	.	
1	D_{62}	61	$P_n(D_{62})$	$P_{n+1}(D_{61})$	$P_{n+2}(D_{60})$	$P_n(D_{62}) + P_{n+1}(D_{61}) + P_{n+2}(D_{60})$
1	-	62	0	$P_{n+1}(D_{62})$	$P_{n+2}(D_{61})$	$P_{n+1}(D_{62}) + P_{n+2}(D_{61})$
1	-	63	0	0	$P_{n+2}(D_{62})$	$P_{n+2}(D_{62})$
.	.	.	.	.	.	
2	D_1	64	$P_n(D_1)$	0	0	$P_n(D_1)$
2	D_2	65	$P_n(D_2)$	$P_{n+1}(D_1)$	0	$P_n(D_2) + P_{n+1}(D_1)$
2	D_3	66	$P_n(D_3)$	$P_{n+1}(D_2)$	$P_{n+2}(D_1)$	$P_n(D_3) + P_{n+1}(D_2) + P_{n+2}(D_1)$
.	.	.	.	.	.	
64	D_{62}	4093	$P_n(D_{62})$	$P_{n+1}(D_{61})$	$P_{n+2}(D_{60})$	$P_n(D_{62}) + P_{n+1}(D_{61}) + P_{n+2}(D_{60})$
64	-	4094	0	$P_{n+1}(D_{62})$	$P_{n+2}(D_{61})$	$P_{n+1}(D_{62}) + P_{n+2}(D_{61})$
64	-	4095	0	0	$P_{n+2}(D_{62})$	$P_{n+2}(D_{62})$

Table 12. Correct Processing of DRFM Samples (Original DIS Architecture)

Remarks for Table 12 (notations and descriptions):

- Radar Pulse – Represents one radar pulse. The number of radar pulses represents the number of Doppler cells for the ISAR, in this case 64.

- DRFM-phase Data – 62 DRFM-phase samples per radar pulse in this case

- Clk – Clock pulse for the DIS

- Tap n – Output of the nth tap

- Tap n+1 – Output of the (n+1) tap

- Tap n+2 – Output of the (n+2) tap (the last tap in this example)

- Result – The output from the DIS

- $P_{n+x}(D_x)$ – Processed phase data in a tap available as valid output

The processing of DRFM-phase data in the three taps that has been implemented using FPGA technology is shown in Table 13. Noted that the implementation of the DIS algorithm using FPGAs does not perform a correct startup and shutdown of the individual taps when a set of DRFM samples is processed. Instead a data value of zero is processed through the tap and produces an incorrect output due to the cosine look-up table ($\cos(0) = 1$). This adds an error at the beginning and trailing edges of the pulse compared with the Matlab simulation that follows the original DIS algorithm.

Radar Pulse	DRFM Data	Clk	Tap n	Tap n+1	Tap n+2	Result
1	D_1	0	$P_n(D_1)$	$P_{n+1}(0)$	$P_{n+2}(0)$	$P_n(D_1) + P_{n+1}(0) + P_{n+2}(0)$
1	D_2	1	$P_n(D_2)$	$P_{n+1}(D_1)$	$P_{n+2}(0)$	$P_n(D_2) + P_{n+1}(D_1) + P_{n+2}(0)$
1	D_3	2	$P_n(D_3)$	$P_{n+1}(D_2)$	$P_{n+2}(D_1)$	$P_n(D_3) + P_{n+1}(D_2) + P_{n+2}(D_1)$
.	.	.	.	.	.	
1	D_{62}	61	$P_n(D_{62})$	$P_{n+1}(D_{61})$	$P_{n+2}(D_{60})$	$P_n(D_{62}) + P_{n+1}(D_{61}) + P_{n+2}(D_{60})$
1	0	62	$P_n(0)$	$P_{n+1}(D_{62})$	$P_{n+2}(D_{61})$	$P_n(0) + P_{n+1}(D_{62}) + P_{n+2}(D_{61})$
1	0	63	$P_n(0)$	$P_{n+1}(0)$	$P_{n+2}(D_{62})$	$P_n(0) + P_{n+1}(0) + P_{n+2}(D_{62})$
1	0	64	$P_n(0)$	$P_{n+1}(0)$	$P_{n+2}(0)$	$P_n(0) + P_{n+1}(0) + P_{n+2}(0)$
.	.	.	.	.	.	
2	D_1	64	$P_n(D_1)$	$P_{n+1}(0)$	$P_{n+2}(0)$	$P_n(D_1) + P_{n+1}(0) + P_{n+2}(0)$
2	D_2	65	$P_n(D_2)$	$P_{n+1}(D_1)$	$P_{n+2}(0)$	$P_n(D_2) + P_{n+1}(D_1) + P_{n+2}(0)$
2	D_3	66	$P_n(D_3)$	$P_{n+1}(D_2)$	$P_{n+2}(D_1)$	$P_n(D_3) + P_{n+1}(D_2) + P_{n+2}(D_1)$
.	.	.	.	.	.	
64	D_{62}	4096	$P_n(D_{62})$	$P_{n+1}(D_{61})$	$P_{n+2}(D_{60})$	$P_n(D_{62}) + P_{n+1}(D_{61}) + P_{n+2}(D_{60})$
64	0	4097	$P_n(0)$	$P_{n+1}(D_{62})$	$P_{n+2}(D_{61})$	$P_n(0) + P_{n+1}(D_{62}) + P_{n+2}(D_{61})$
64	0	4098	$P_n(0)$	$P_{n+1}(0)$	$P_{n+2}(D_{62})$	$P_n(0) + P_{n+1}(0) + P_{n+2}(D_{62})$
64	0	4159	$P_n(0)$	$P_{n+1}(0)$	$P_{n+2}(0)$	$P_n(0) + P_{n+1}(0) + P_{n+2}(0)$

Table 13. Processing of DRFM Samples Using FPGAs (Original DIS Architecture)

Remarks for Table 13 (notations and descriptions):

- Radar Pulse – Represents one radar pulse. The number of radar pulses represents the number of Doppler cells for the ISAR, in this case 64.

- DRFM-phase Data – 62 DRFM-phase samples per radar pulse in this case

- Clk – Clock pulse for the DIS

- Tap n – Output of the nth tap

- Tap n+1 – Output of the (n+1) tap

- Tap n+2 – Output of the (n+2) tap (the last tap in this example)

- Result – The output from the DIS

- $P_{n+x}(D_x)$ – Processed phase data sample in a tap available as valid output

- $P_{n+x}(0)$ – Processed "0" in a tap available as output

2. FPGA DIS Schematic

a. Top-Level FPGA Hierarchy

The top-level hierarchy of the design using FPGAs is shown in Figure 41. The purpose of this figure is to visualize the Altera environment at the top-level of this architecture. The bottom left hand block is the I/O-decode and Built-in-Test (BIT) block. The purpose of the I/O decode block is to provide up to 256 addressable "internal" address spaces for reading and writing. The other blocks have direct correspondence to the other modules in the DIS:

- Tap-Delay Line (delay.gdf)

- Doppler Modulation Coefficient Latch (phi.gdf)

- Phase Summer (ph_acc.gdf)

- Look-Up-Table (lut.gdf)

- Gain Modulation Coefficient Latch (gain.gdf)

- Gain Modulator (newgain1.gdf, shift0.gdf, shift1.gdf, shift2.gdf, mux2.gdf)

- Output Summer (out_summer.gdf)

Each of these modules is described in further detail below.

Figure 41. Top-level FPGA Hierarchy of the DIS (simple.gdf)

b. Tap-Delay Line

The tap-delay line schematic with 3 tap delay lines is shown in Figure 42. The tap-delay lines are composed of a chain of D-flip-flops and occupy four internal addresses, 0x30, 0x31, 0x32 and 0x33. The meaning of the data values written to these locations is described in Table 14 below.

Internal Address (in hex)	Function
0x30	Write "1" to reset tap-delay line, "0" otherwise
0x31	Write any value to this address to cause a propagation of the values down the delay line
0x32	Write the new DRFM value to the first tap of the delay line
0x33	Unused

Table 14. Internal Address Usage in the Tap Delay Line

Updating the tap-delay line is a 2-stage process. This is accomplished by writing any value into address 0x31 (to effect propagation) followed by writing a new value into address 0x32 (to load in a new value at the first tap of the delay line).
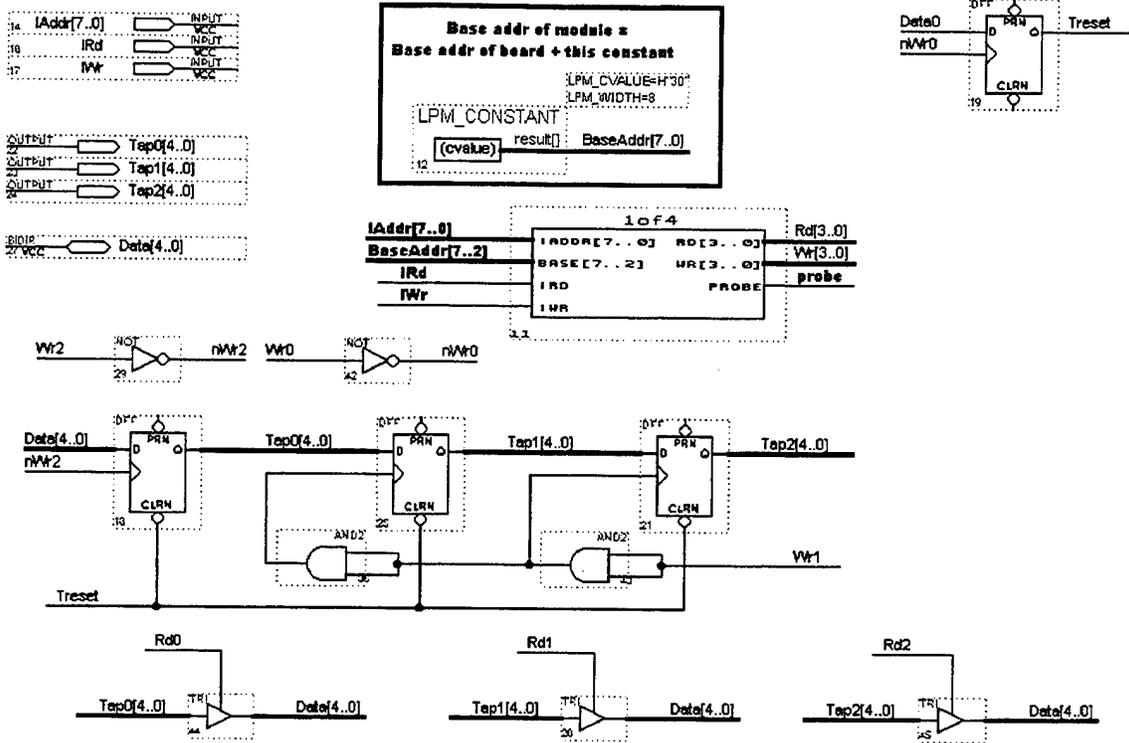
Figure 42. Schematic of the Tap-Delay Line (delay.gdf)

c. *Doppler Modulation Coefficient Latch*

The phase-coefficient latch (for Doppler modulation) is comprised of a 1-of-4 decoding block and a set of flip-flops as shown in Figure 43.
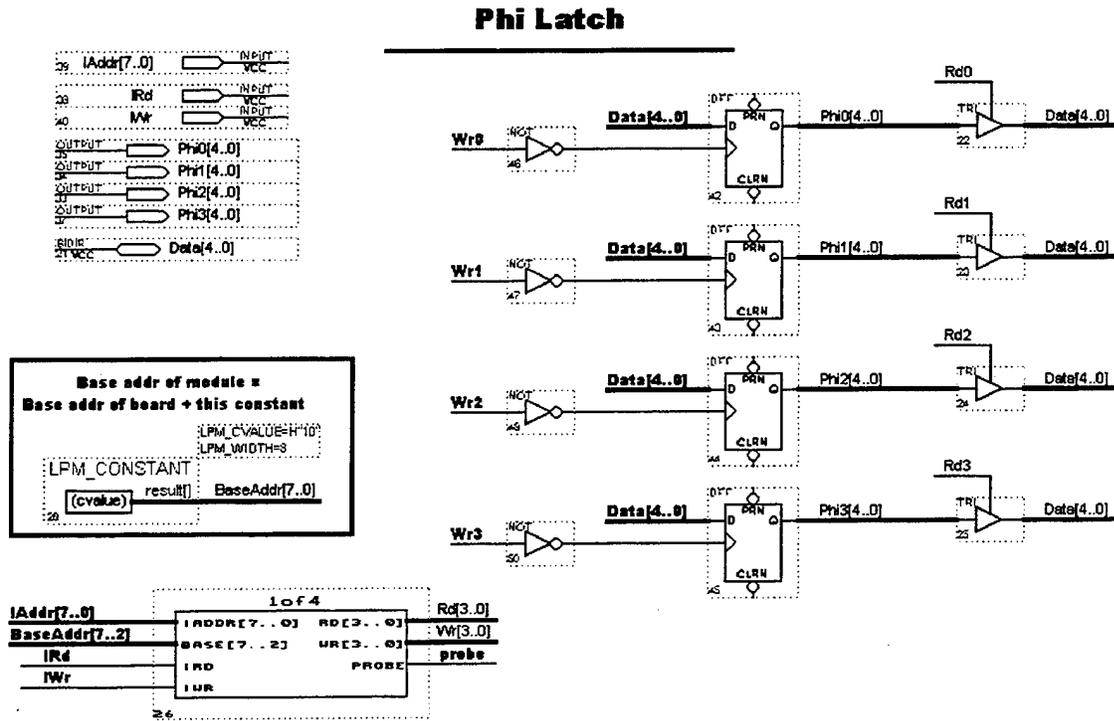
Figure 43. Schematic of the Phase-Coefficient Latch for Doppler Modulation (phi.gdf)

d. *Phase Accumulator*

The phase accumulator schematic is shown in Figure 44 (one for each tap). The inputs to the accumulator are the 5-bit DRFM-phase samples (the values from the tap-delay line) and the latched 5-bit phase coefficients. Furthermore, the output bit-width matches the input bit-width (the carry-bits are discarded) representing a modulus addition operation (which is desired). Due to truncation of values larger than five bits, the phase values above 2π are folded back into the principle range between zero and 2π. The LPM-ADD-SUB module available in the Library of Parameterized Module (LPM) is used to configure the adder.
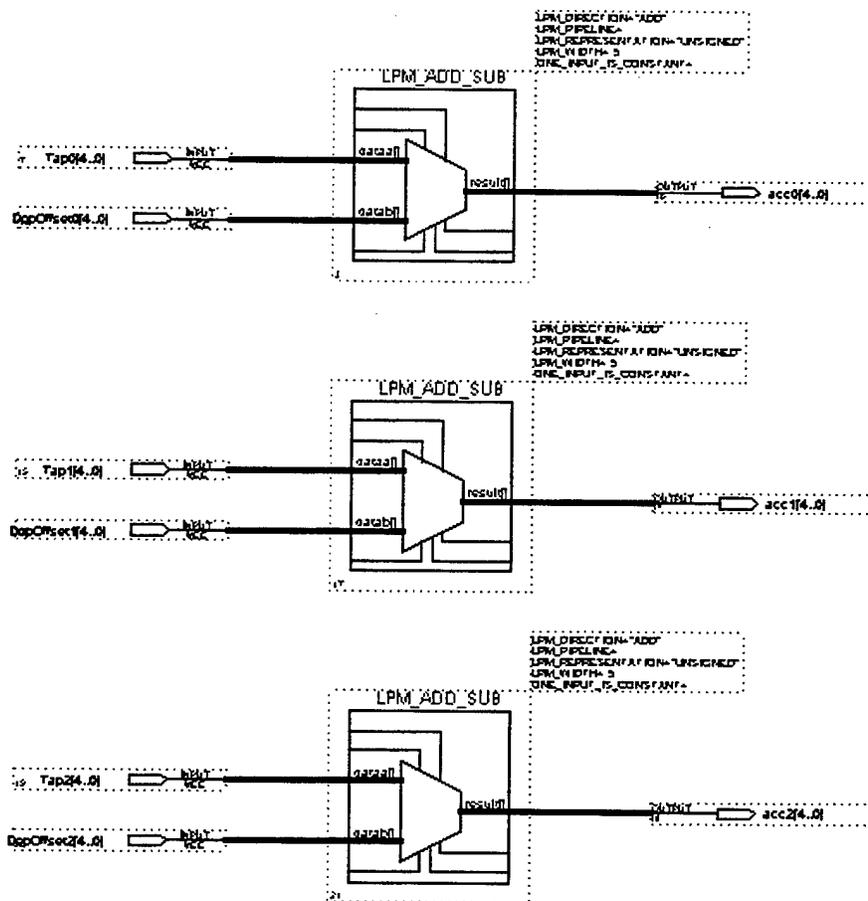
Figure 44. Schematic of the Phase Accumulator (ph_acc.gdf)

76

e. *Look-Up Table (LUT)*

The look-up table (LUT) is indexed by the output phase from the phase accumulator. This phase value is mapped to an 8-bit amplitude value stored in the LUT. Since the LUT output is a complex number, cosine and sine tables indexed by the same phase are required. The schematic diagram for the LUT is given in Figure 45. For the LUT configuration, a text file is associated with each LPM_ROM module. In Altera, this file is called a memory initialization file (.mif). A Matlab script file (genLUT.m), which is capable of automatically generating the text-file, based on the width and depth of the LUT desired, has been used. This file generates the memory initialization file for the LPM_ROM module (cos.mif and sin.mif). It calls two Matlab function files (genfixptv0.m and genfloat.m). These two programs perform the fix and floating-point conversions. The Matlab files [Ref. 6] are included in Appendix A.
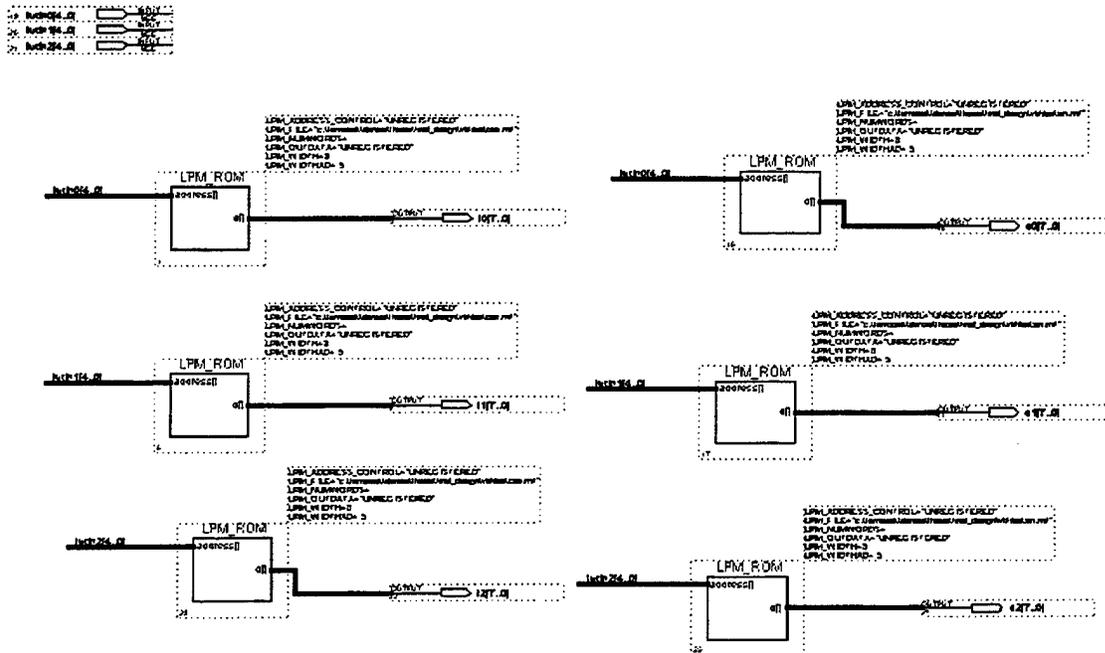
Figure 45. Schematic Diagram of the Look-Up Table (LUT) (lut.gdf)

77

f. Gain Modulation Coefficient Latch

The latch for the gain modulation coefficient comprises a 1-of-4 decoding block and a set of flip-flops as shown in Figure 46. Although four DQ flip-flops are shown, only three of them are used (one for each tap).
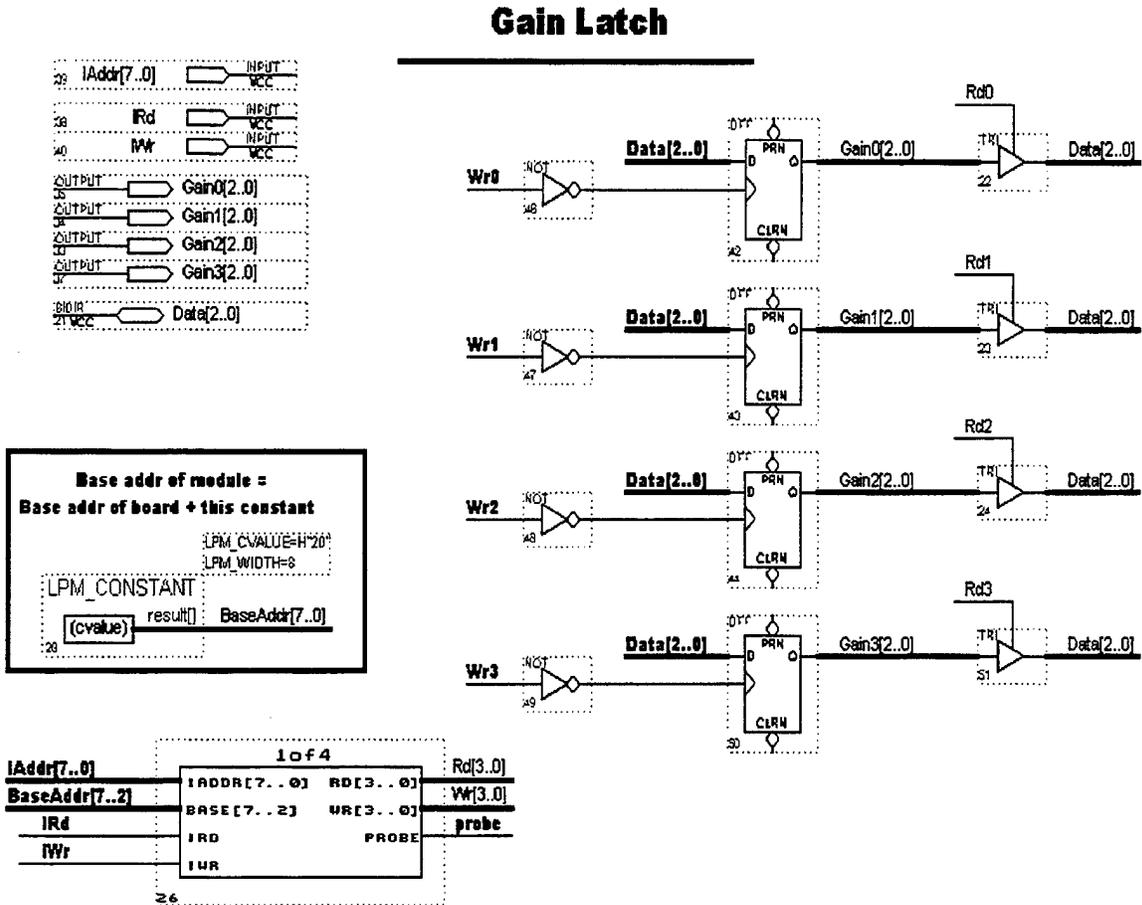
Figure 46. Schematic of the Gain Modulation Coefficient Latch (gain.gdf)

g. *Gain Modulator*

The gain modulator applies a gain to the binary signal from the LUT by shifting the binary word toward the most significant bit position and pads zeros at the least significant bit position. The gain modulator is shown in Figure 47.
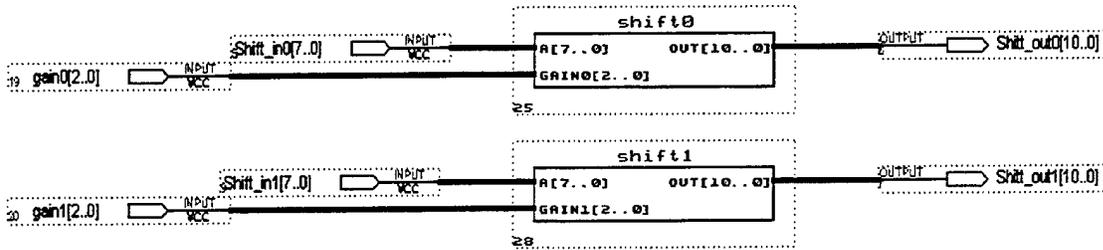
Figure 47. Schematic of the Gain Modulation (newgain1.gdf)

The original amplitude values, as set by the user in the Matlab GUI (the Range-Doppler-Amplitude Map Entry Program), are "translated" into a corresponding number of positions for the shift according to Table 15.

GUI Amplitude Value	# of Shift Left Steps	Represents Decimal Multiplication by
1	0	1
2	1	2
3	2	4
4	3	8

Table 15. Translation of Gain Values

Figure 48 exemplifies the results of applying different gain modulation coefficients. In the first sub-plot, a GUI Amplitude value of "1" was applied, representing a decimal gain value of "1," for a 3-target cell long target. In the next three sub-plot the GUI Amplitude value was increased to "2," "3," and "4" respectively, representing a decimal gain value of "2," "4," and "8."
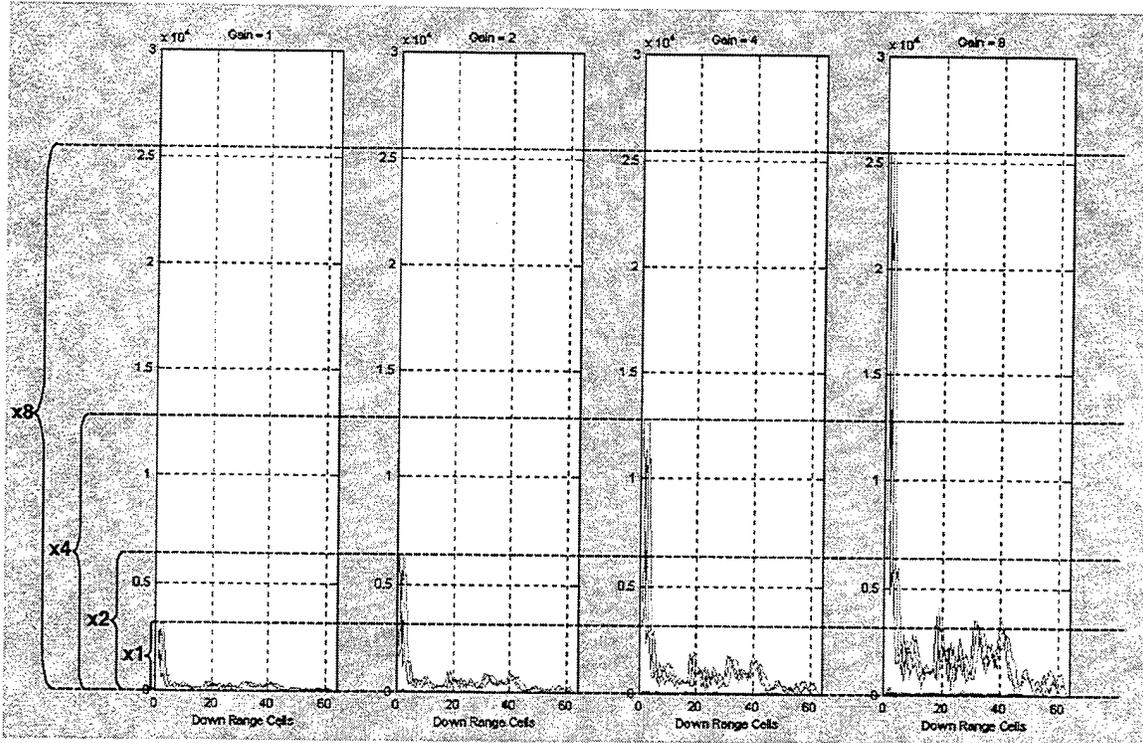
Figure 48. A 3-Target Cell Long Target with Different Gain Modulation Coefficients

Using only a 2-bit word for representing the gain modulation coefficient will limit the dynamic range to 18.1dB when using shift modules. Another limitation is that only four discrete amplitude levels can be used. Increasing the word size, i.e. to three (or four) bits will give 42.1dB (90.3dB) dynamic range and 8 (16) different amplitude levels.

# of bits	Shift by	Multiplication by	Dynamic Range [dB]
n	0 to 2^n-1	1, ..., $2^{2^{n-1}}$	$20\log_{10}(2^{2^{n-1}}/1)$ $20\log_{10}(V_{max}/V_{min})$

Table 16. Number of Bits vs. Dynamic Range

The schematic diagram of the shift primitive is shown in Figure 49 and the schematic diagram of the MUX2 building block is provided in Figure 50.
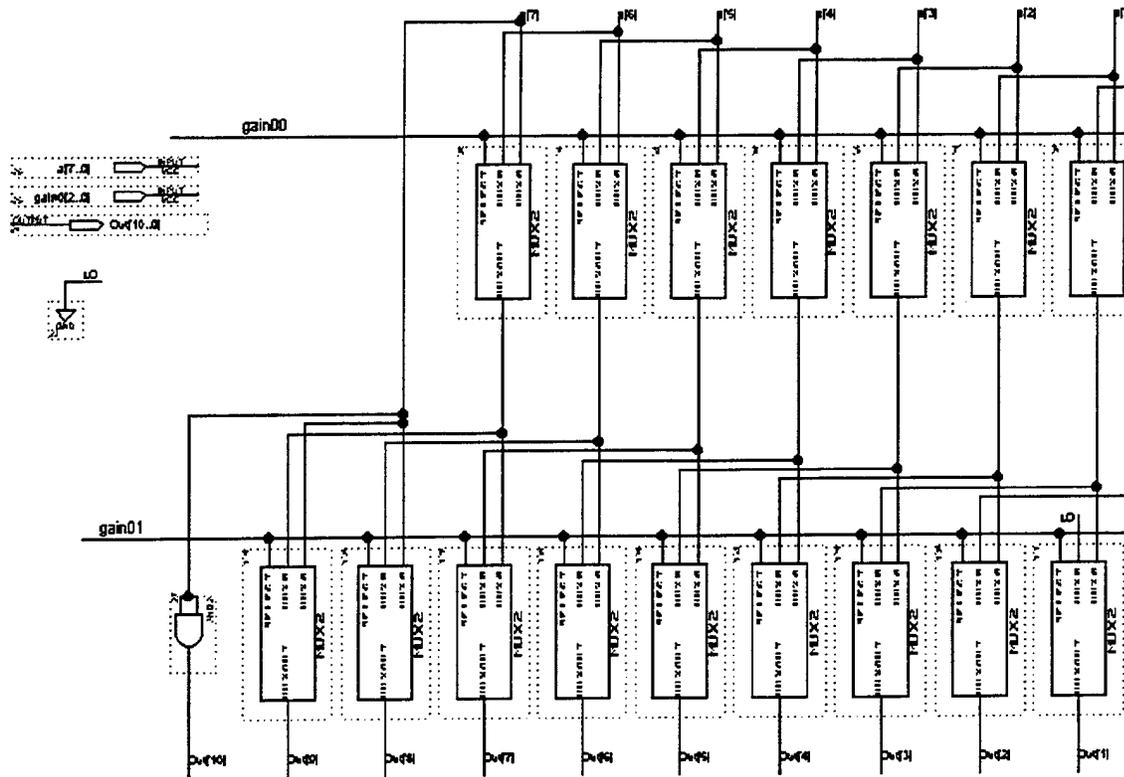
Figure 49. Schematic of the Shift Primitive in the Gain Modulation Block (shift0.gdf)
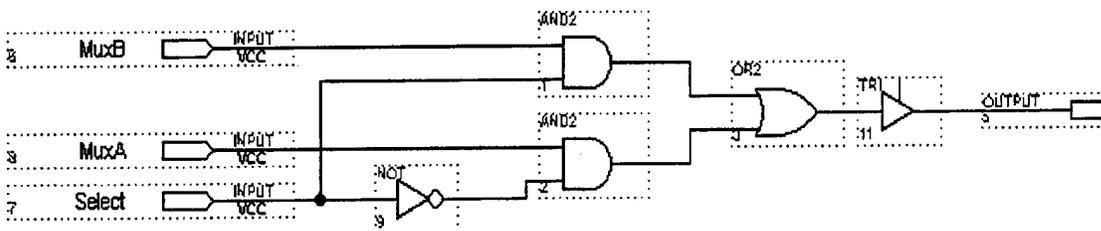
Figure 50. Schematic of the MUX2 in the Gain Modulation Block (mux2.gdf)

81

h.   Final Summer

The schematic of the final summer is given in Figure 51. This circuit implements the addition of the tap outputs in two's complement. The addition in two's complement involves sign-extension of the numbers to be added and discarding the carry-out bit. The LPM-ADD-SUB module, available in the LPM, is used to configure the summer.

Figure 51. Schematic of the Final Summer (out_summer.gdf)

E. SIMULATION RESULTS

1. Simulation Setup

Several simulations have been done to verify the expected results. Below is one example of a simulation run to illustrate the steps and to visualize the results. In this case, the false target to be generated has the same parameters given in the example above (Matlab simulation) as is shown again in Figure 52.
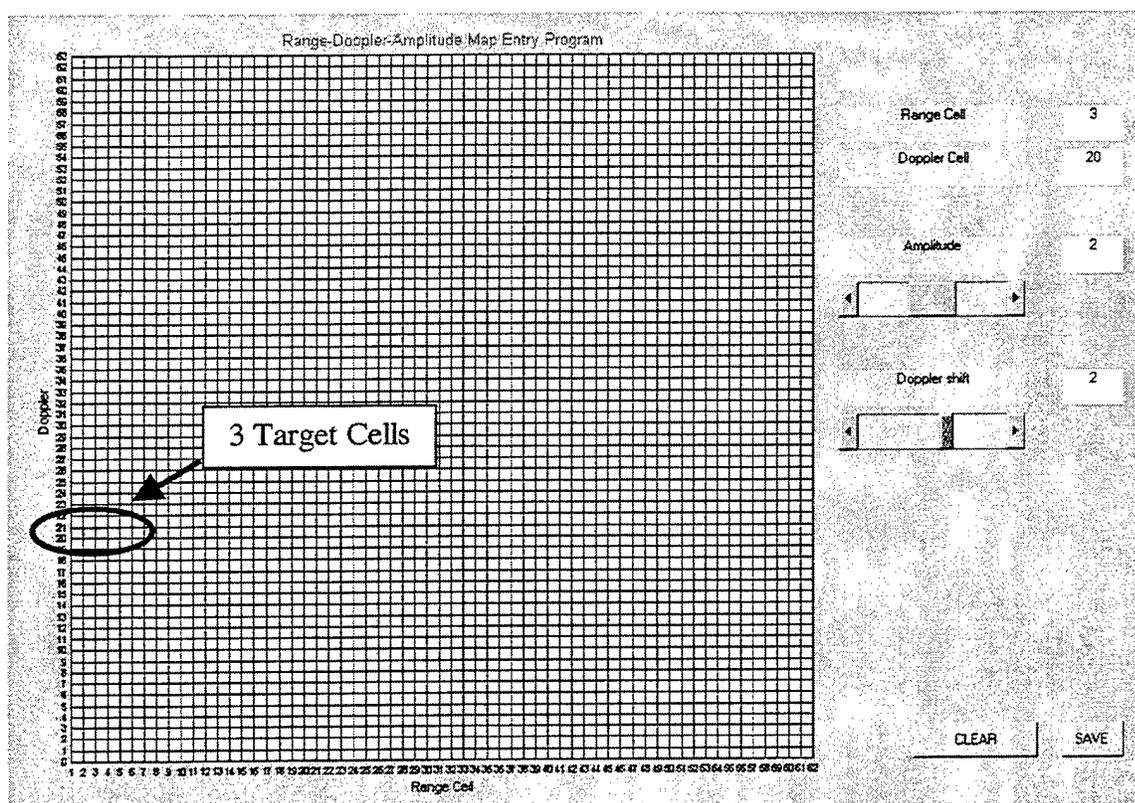
Figure 52. The Range-Doppler-Amplitude Map Entry Program

In this example the user has specified the data in Table 17 for the false target to be created by the DIS.

Target Cell	Range Cell	Doppler Cell	Amplitude Value	Doppler Shift	Remark
1	1	20	2	0	Tap 0 – 1st Tap
2	2	20	2	1	Tap 1 – 2nd Tap
3	3	20	2	2	Tap 2 – 3rd Tap

Table 17. User Specified Inputs of the False Target

In order to make the comparison between the Matlab simulation and the DIS implemented using FPGA technology, an intermediate step was added in the simulation flow as described in Chapter 4. After the Matlab file mathostvX.m has been executed, all necessary inputs are available in text files to run the hardware implementation of the DIS. The interface with the FPGA computer board is a set of Visual Basic files composed into a Visual Basic project called FlexTest (flextest.vbp). To compile and run the project and the board properly, the necessary files must be located in a file structure with the following path: c:\temasek\denise\thesis\final_design\vbfiles.

To run the Visual Basic project, FlexTest, the user must open the project, open the the_isar.bas file, and then run the file. Another GUI will show up on the computer display to visualize the signal processing taking place in the taps of the DIS.

2. Simulation Results

The 2-D contour plots in Figure 53 show the results from the Matlab simulation and the results from running the DIS implemented on the Altera FPGA device. The Matlab simulation results are shown in the upper sub-plot. Both sub-plots are presented in a range-Doppler map. That is, Down Range (range) versus Cross Range (Doppler). The two results look quite similar but will be examined closer to verify if any differences are present or not.
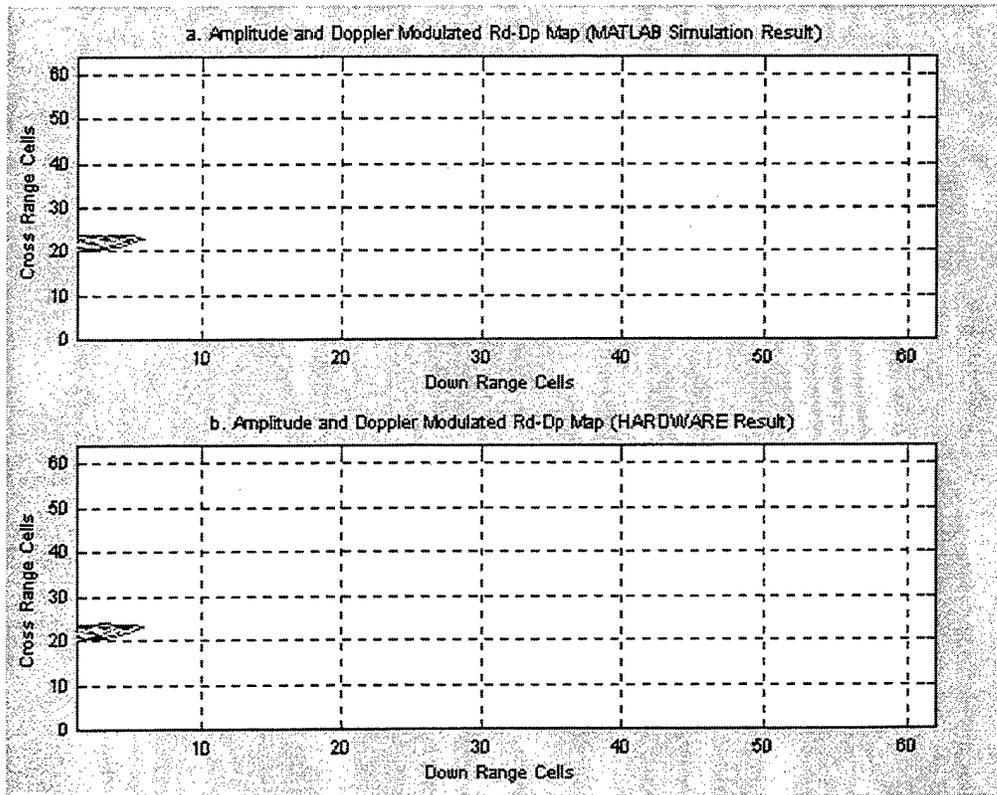
Figure 53. Matlab DIS Simulation vs. FPGA Hardware Results

Figure 54 shows the 3-D mesh surface plots. The first sub-plot shows the results from the Matlab DIS simulation. The second shows the result from the FPGA hardware. Finally, the third sub-plot shows the difference between the Matlab simulation and the FPGA hardware results. As expected, a slight difference between the Matlab simulation and the hardware results can be observed (note the scale on the amplitude axis of the three sub-plots).
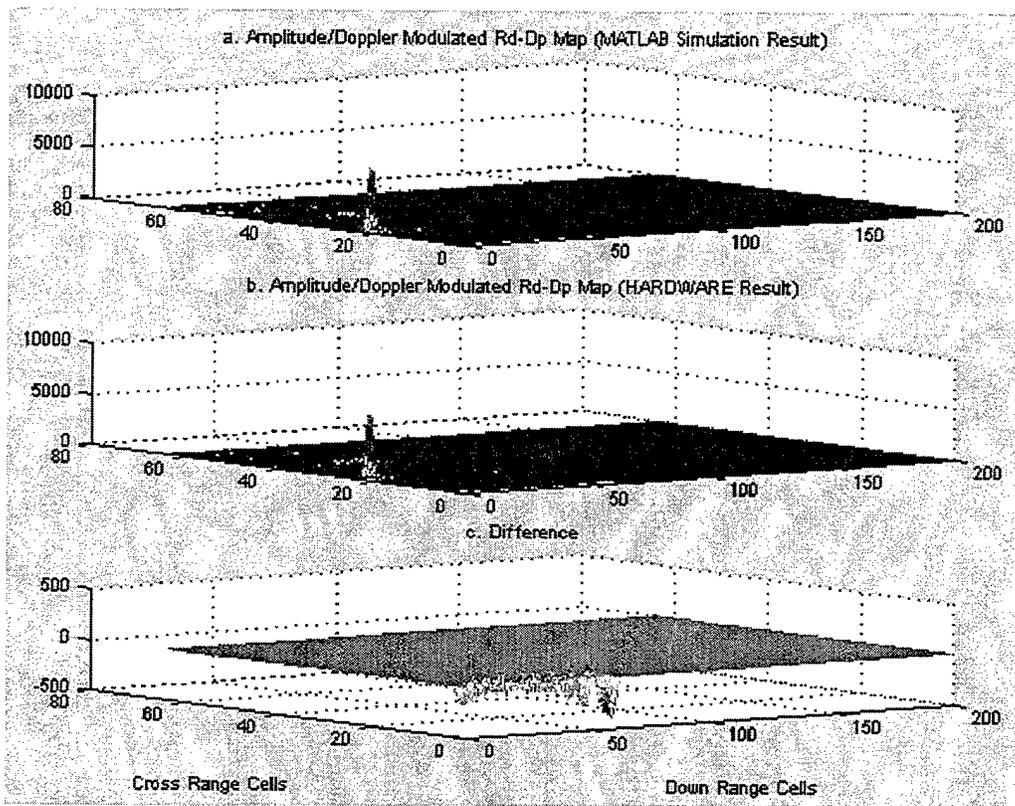
Figure 54. Matlab Simulation Result vs. FPGA Hardware Result and Their Difference

These differences are because the implementation of the DIS algorithm using FPGAs does not consider a correct startup and shutdown of the individual taps when a set of DRFM-phase data from one radar pulse is processed (as discussed earlier). This

contributes to a slight error compared to the Matlab simulation, which strictly follows the original DIS algorithm.

To verify that the errors are actually due to the difference in startup and shutdown sequences, the Matlab simulation code was adjusted to process the phase data in the same manner as the FPGA hardware. The results of the modified test case are shown in Figure 55.
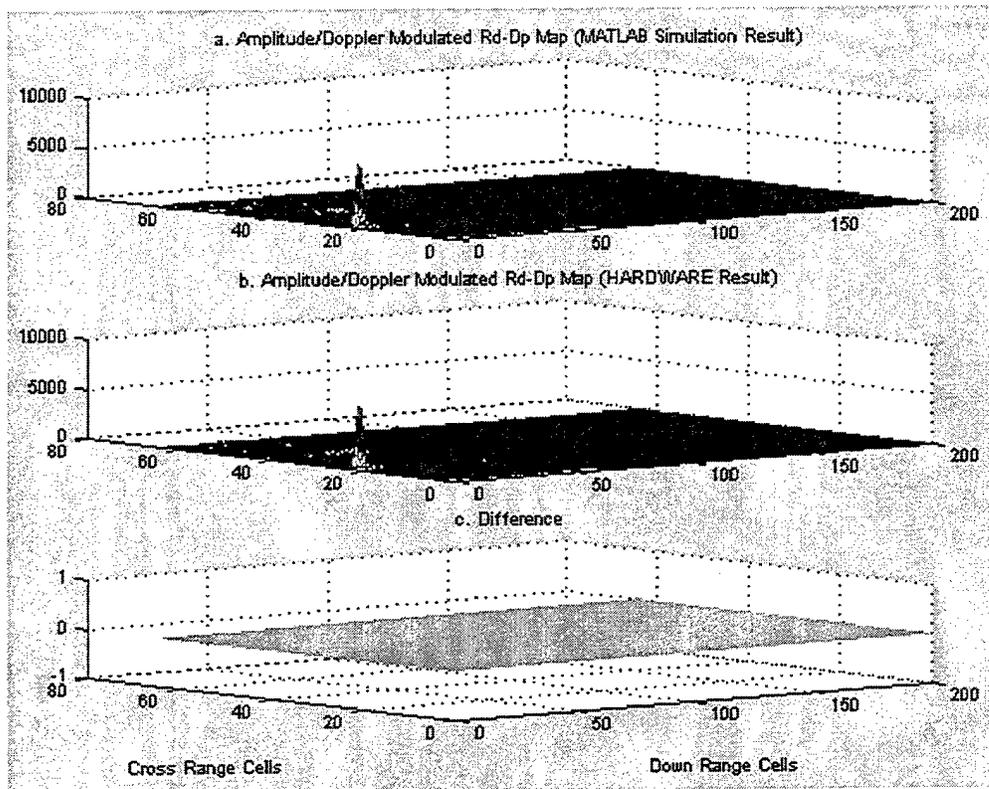
Figure 55. Matlab Simulation Result vs. FPGA Hardware Result and Their Difference

As expected, there are now no differences between the Matlab simulation results and the hardware FPGA results. The Concept Demonstrator has therefore been proven to work with its known limitations.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. FPGA-TO-ASIC CONVERSION

Since the first design was captured using the Altera Max+Plus II software, which targets the Altera Programmable Logic Device (PLD), several methods were investigated to convert this design using third-party tools. This chapter discusses the different methods of converting the existing design in a format for tools that target an Application Specific Integrated Circuit (ASIC), instead of a PLD and their related problems. This chapter further concludes with a summary of problems encountered and the reasons behind choosing the Tanner Tools environment instead of one of the discussed converting tools.

A. FPGA LIMITATIONS

After analyzing the original, nearly complete implementation of the original architecture, we realized that several limitations were being imposed on the design, solely because the implementation employed FPGA technology. First and foremost was the speed limitation. The target clock speed for the design is 500MHz (2ns clock rate). This is an aggressive goal for any new chip design, and although it might eventually be possible to meet this target with an FPGA design, in the foreseeable future, a full-custom IC has a higher probability of meeting this speed requirement. A second major contributing factor was the physical size of the implementation. The initial, proof-of-concept design does not require a large number of taps. However, even with a small number of taps the design could not be implemented into a single FPGA. One of the goals for this initial, proof-of-concept design was to create a devise easily extendable to more taps. Extending the FPGA implementation to more taps would require a significant

increase in the number of FPGAs. This was considered a major drawback of the FPGA implementation.

After realizing the limitations of the FPGA implementation, we decided to convert the FPGA design to an ASIC design. Several FPGA-to-ASIC conversion techniques were investigated.

1. Altera-to-MOSIS Process Flow

The Altera-to-MOSIS conversion process investigates attempts to translate the design from Altera's Max+Plus II implementation to a high speed ASIC fabricated by MOSIS. It will be shown that the conversion is highly complex, and that parts of the conversion process are unpredictable, since some tools do not have a common interface.

a. Altera to MOSIS Link Overview

The flowchart shown in Figure 56 shows the complete conversion path from the current FPGA design in Max+Plus II to an ASIC fabrication at MOSIS using several tools in different stages of the process. Statecad in conjunction with Statebench provides an add-in state machine into the existing FPGA design so that the resulting project file can be compiled in the Max+Plus II compiler. SimGen converts the compiler output file (.EDO) into a .MAC file, which can be read by Nettran. Nettran is a program of the Tanner Tool environment and converts different formats into useable input files for other Tanner Tool programs, e.g., the layout editor L-Edit. L-Edit uses the resulting .TPR file as input and creates a physical layout based on its library elements. The layout file needs to be compared with the original input files to ensure that the circuit representation is the same as the compiler output file. After verification and post-layout simulation, the layout can be sent to Mosis in form of a .CIF file for fabrication.
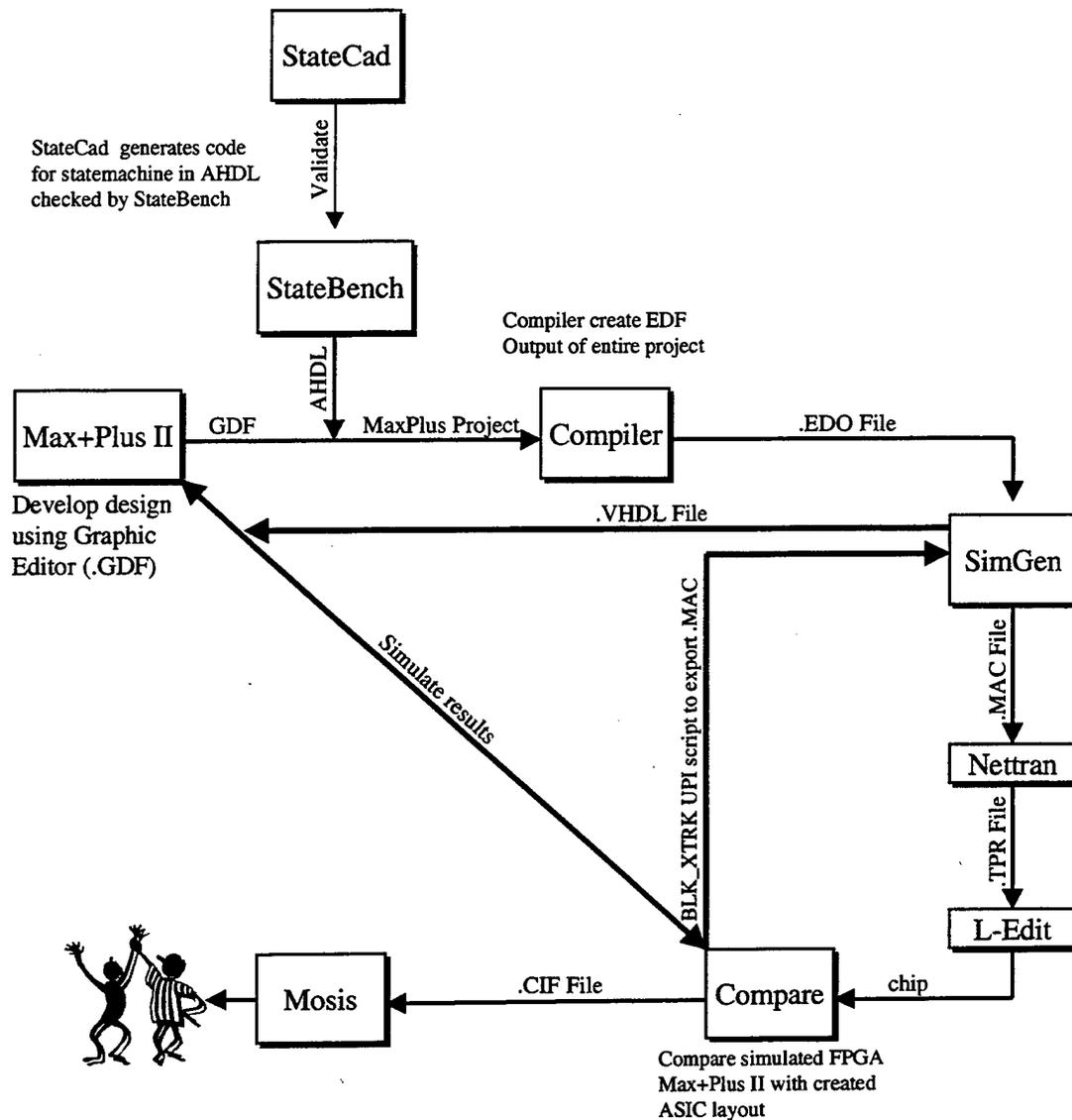
Figure 56. Flowchart–Altera to MOSIS Link

Remarks for Figure 6.1

- GDF is Graphic Design File and is the file format of the Graphic Editor in Max+Plus II

- AHDL is Altera Hardware Description Language

- EDO is EDIF output file

91

- MAC is Macro file, to use between SimGen and Nettran

- TPR is Tanner Tools file type

- CIF is Chip fabrication format for the final layout

- VHDL is VHSIC Hardware Description Language

Controling the data flow by a state machine was considered, however, due to a different approach in the later ASIC design, a state machine implementation was not deemed necessary. Furthermore, Nettran and L-Edit are not part of the following program description. They are explained in detail in the following chapter to avoid unnecessary redundancy.

b. Statecad and Statebench

Statecad is a powerful tool to create state machines of all kinds easily. It is a graphical entry tool that allows the user to express ideas as state diagrams. Statecad has been designed for simplicity in use as a tool for digital design, documentation, and error analysis. The Statecad GUI is shown in Figure 57 to illustrate the graphical concept of the tool. After validating a diagram, the program generates, directly from the diagram, hardware description language (HDL) code that can be simulated and synthesized. The HDL is valid, consistent, maintainable, and implements the graphical diagram. The HDL can be VHDL-1076, Verilog, ABEL-HDL, AHDL or ANSI-C. Interactive dialog boxes provide an environment for intuitive work and help to eliminate syntax errors and incomplete portions of state diagrams. [Ref. 12]

Once a design is completed in Statecad, it can be verified in the add-on software Statebench. After verification, a timing test bench can be written automatically. The test bench can be used for post synthesis timing verification.
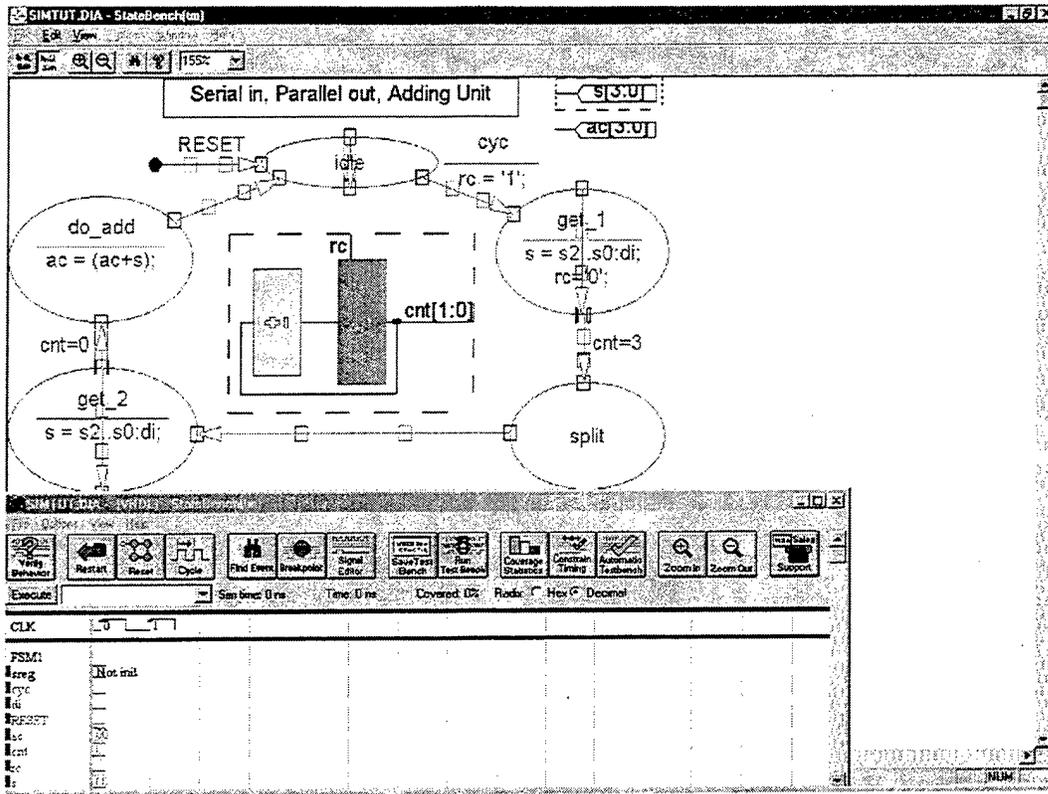
Figure 57. Statecad Screenshot (From Ref. [13])

Statebench is an add-on program to Statecad and automates behavioral verification and VHDL/Verilog test bench generation of any state diagram created in Statecad. Automatic test benches attempt to exercise every input, output, transition, and logic equation in a design. For verification the program can usually check approximately 80% of the design automatically. The remaining 20% requires minor user inputs to complete the validation of the design. Statebench can generate VHDL or Verilog test benches for post-synthesis verification by adding time constraints that can be imported in third-party test programs for further validation.

c.   SimGen

SimGen is an EDIF to Nettran and a FPGA to ASIC conversion utility for Tanner Tools EDA that improves routine operation designing within the Tanner environment. SimGen uses EDIF or MAC files as input and can generate VHDL files from a chip layout to support verifications in VHDL design flows. It automatically creates simulation files (.SIM and .VEC) for GateSim. For these types of files SimGen sets up template files with input/output lists and restores true port names. Due to its ability to create .MAC files, SimGen supports file import into Tanner's Nettran software. Since the conversion between different file formats is not unproblematic, it attempts to clean up and repair netlists so that they can work as expected when going from one tool to another. SimGen provides a Windows control shell to activate, coordinate, and generate command files for Tanner's remaining DOS tools, e.g., GateSim, as well as file editing functions and waveform viewing functions [Ref. 14].

The conversion process from Max+Plus II to an ASIC in form of the Altera-to-Mosis Link is long and in parts unpredictable. One of the major drawbacks is that SimGen has no direct supported interface to Max+Plus II. Another major drawback is the incompatibility between the library cells used in the FPGA design and the required library cells for an ASIC design. Therefore a significant amount of hand conversion of library cells is required, which is time consuming and potentially error prone. Other problems with this conversion approach include the efficiency of the conversion process with respect to speed, layout area, and power consumption of the final IC design. Furthermore a future chip expansions or even minor changes to the design require

working through the entire link again, resulting in a considerable amount of time and new sources of conversion errors.

B. LEONARDO SPECTRUM

An alternative program to create an ASIC can be found in Spectrum's Leonardo software [Ref. 15]. Leonardo was not chosen because of the very complicated workspace creation process between Max+Plus II and Spectrum's software. Also Leonardo lacks the capability to directly import file types, which are generated by Max+Plus II. Nevertheless, Leonardo has the capability to target an entered or imported design either as an ASIC or as a FPGA. It includes several wizards to optimize, re-target and improve the design. Spectrum and Altera offer the possibility to create a working environment between MAX+PLUS II and Leonardo, which is illustrated in Table 18 and Figure 58. The described data flow was never investigated in detail since the involved programs had to be bought and were not available for testing. Nevertheless this was not desired either owing to the extremly long and complicated data flow to generate the workspace between the two endpoints. The MAX+PLUS II read.me file provides more information about which versions of Mentor Graphics applications are supported by the current version of MAX+PLUS II. It also provides information on installation and operating requirements that are not mentioned in this report.

Max+Plus II/Mentor Graphics Software Requirements			
The following products are used to generate, process, synthesis, and verify a project with the Max+Plus II and Mentor Graphics Leonardo software:			
Mentor Graphics		Exemplar	Altera
System_1076 Compiler QuickSim II Design Architect ENRead ENWrite GEN_LIB library	Quick HDL Quick HDL Pro Quick Path LS_Lib library DVE	Galileo Extreme V4.1.1 Leonardo V4.1.3	Max+Plus II V.9.2

Table 18. Workspace between Max+Plus II and Leonardo

In more general terms, the flexibility of programs like Max+Plus II, Leonardo, Statecad, etc. is determined by their ability to import files of different types. The most common file types are EDIF, Verilog, and VDHL files. One has to strictly differentiate between input and output files. Output files from Altera's MAX+PLUS II software are not compatible to input files with the same file extension, so there is a need to examine the differences in more detail. The following example is based on the MAX+PLUS II software, but is transferable to the other above-mentioned programs: the input file types are VHDL, Verilog, AHDL, GDF, SCH (schematic files from ORCAD) and the EDIF files from third party synthesis tools. The output files produced by the Max+Plus II compiler are VO (Verilog output netlist file), VHO (VHDL output netlist file),
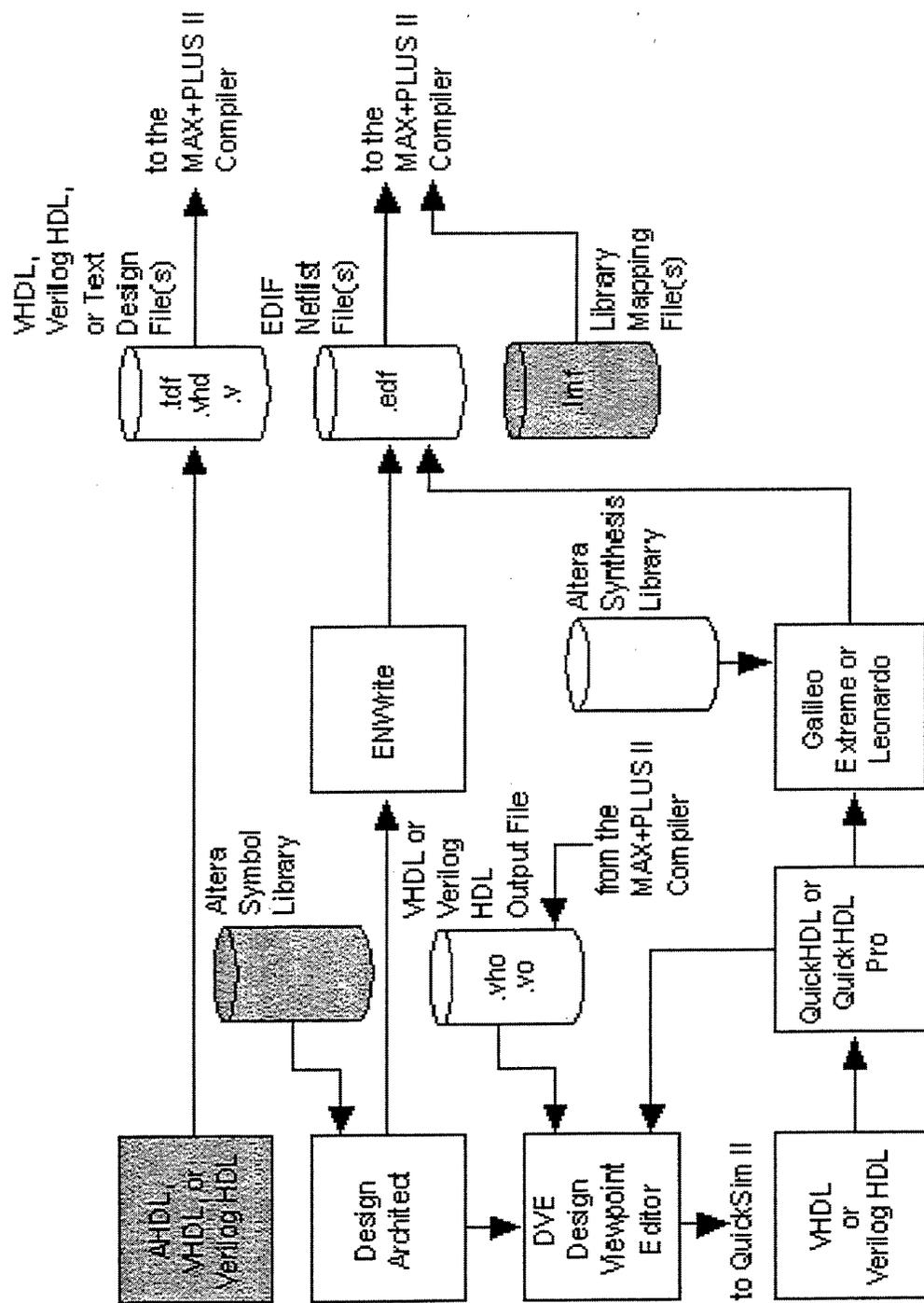
Figure 58. Block Diagram for the MAX+PLUS II/Leonardo Workspace (From Ref. [11])

TDO (AHDL output netlist file), and EDO (EDIF output netlist file). Now it should be obvious that output files cannot easily be used as import files for other programs. Hence a lot of other third-party tools are required to establish a conversion path in order to create a link between the two main programs.

The only VHDL, Verilog or EDIF files that can be generated by the Max+Plus II compiler after synthesis are post place and route netlist files. These files are normally used as either input to third party simulation tools like e.g., Verilog-XL from Cadence, Modelsim from Modeltech etc. or as input for static timing analysis tools, like Primetime & Motive from Synopsys. These netlists contain a gate-level description of the design and the timing delays, where Max+Plus II's EDIF input file is a synthesized netlist. Therefore extracting an input file from an output file involves an extraordinary number of steps since the output files are place and route netlists.

C. AMERICAN MICROSYSTEMS INC.

Another alternative to building an ASIC from an FPGA is to contract with a company that specializes in FPGA-to-ASIC conversions, e.g., American Microsystems Inc. (AMI). For this approach, the entire design has to be done in FPGA-oriented software like Max+Plus II and sent to AMI for the conversion process. AMI also provides customers with their software in a light version, warning, however, that they cannot recommend this method since the tools are very complicated and require much experience [Ref. 16].

This approach was not selected and not further investigated for several reasons. First, the design conversion process yields an ASIC design that is readable by a computer

and cannot be read, manipulated, and modified easily by a human, even with the appropriate CAD tools. Therefore, when the initial DIS design is eventually expanded to include more taps, the expanded design would have to be accomplished using the FPGA tools and then another design conversion would have to be performed by the contractor and paid for. Another drawback is the efficiency of the design conversion with respect to speed, layout area, and power consumption of the final IC. Although great strides have been made in automated optimization for design conversion, much work still needs to be done in this area. Moreover, chip designs that start life as an ASIC design usually wind up being faster, smaller, and consume less power. Finally, one of the goals at the NPS *Center for Joint Services Electronic Warfare* is to offer students the chance to research and to create projects while working toward a Master's degree. Hiring an outside firm to perform the design conversion would eliminate this opportunity in addition to being costly and ending up with an imperfect design.

D. MIGRATION TO TANNER

All the above-described processes were investigated to convert the existing FPGA design into an ASIC in order to achieve two goals. Most importantly, the high-level DIS architecture had to be fast, both with respect to high throughput and short latency. Second, the design had to be extensible, allowing an inexpensive prototype with fewer taps to be easily turned into a more finished product by just increasing the number of taps. After analyzing the original architecture, we realized that there were several limitations being imposed on the design solely because the implementation was done using FPGAs.

The major concern is the speed limitation, in view of the fact that the clock speed for the design should be close to 500MHz. Although it might eventually be possible to meet this target with an FPGA design, in the foreseeable future, a full-custom IC will have a higher probability of meeting this speed goal.

A second major contributing factor is the physical size of the implementation. The initial, proof-of-concept design did not require a large number of taps. However, if more taplines are desired to build a full operational prototype, the taplines would not fit into a single FPGA. Extending the FPGA implementation to more taplines would require a significant increase in the number of FPGAs. This is considered a major drawback of the FPGA implementation. Furthermore, additional taps could not just be added to the design since the adder tree used to sum the outputs of the taps for the final output would have to be redesigned. Beyond this, as the number of taps increases, either the clock speed must be slowed down (reduced throughput) or the number of pipeline stages must be increased (increase in the total latency) to accommodate the extra delay in the additional adders in the adder tree. The total latency is the sum of the latency in the tap and the latency in the adder tree, which increases as the number of taps increases.

After considering the various different alternatives for design conversion, we realized that a dedicated ASIC design using the Tanner Tools would be the most efficient approach. The original architecture and FPGA design allows, however, an in-depth analysis of the behavior of the algorithms to be implemented in the ASIC and also allows the investigation of future design concepts (for example, to counter stepped-frequency waveforms).

VII. ASIC DESIGN: SCHEMATIC

This chapter gives an overview over the Tanner Tool environment with emphasis on the programs used to construct the DIS architecture. The second section discusses the DIS architecture in more detail as it is modified and completed based on the FPGA architecture. The last section melts the previous section together and presents the detailed design implementations in Tanner's schematic capture tool S-Edit. Additionally, ideas and already created circuit improvements for future development will be addressed briefly. These improvements reflect circuit simplifications in terms of less transistors used in certain modules, or redesign issues for a higher clock speed.

A. INTRODUCTION TO TANNER TOOLS

The Tanner Tool environment consists of five major integrated modules: S-Edit, T-Spice, W-Edit, L-Edit, and Nettran. The following list presents a short overview of the complete Tanner environment [Ref. 17]:

Simulation Tools:

- T-Spice–an analog/digital circuit simulator

- GateSim–a gate-level simulator

- W-Edit–a waveform viewer

- L-Edit/Therm–a 3-D finite-element thermal analyzer

Front End and Netlist Tools:

- S-Edit–a schematic editor

- LVS–a layout-versus-schematic netlist comparator

Mask-Level-Tools:

- L-Edit–a layout editor

- L-Edit/SPR–an automatic standard cell placement and routing package

- L-Edit/Extract–a layout extractor

- L-Edit/DRC–a design rule checker

The ordered Tanner Tool package consists of:

- L-Edit with Design Rule Checker (DRC), Extract, and Standard Place and Route (SPR)

- S-Edit (Schematic Editor)

- LVS (Layout vs. Schematic)

- T-Spice Pro with Advanced Model Library

- W-Edit (Waveform Viewer)

- Tanner Tools Pro Manuals

Figure 59 [Ref. 17] illustrates at first a schematic overview of the Tanner environment and at second the data flow between the different programs of the package. The main environment consists of the programs S-Edit, LVS and L-Edit, where L-Edit finally saves the layout in a GDSII or CIF file that will be send to MOSIS for chip fabrication. The other components may not be used but are shown for completeness.
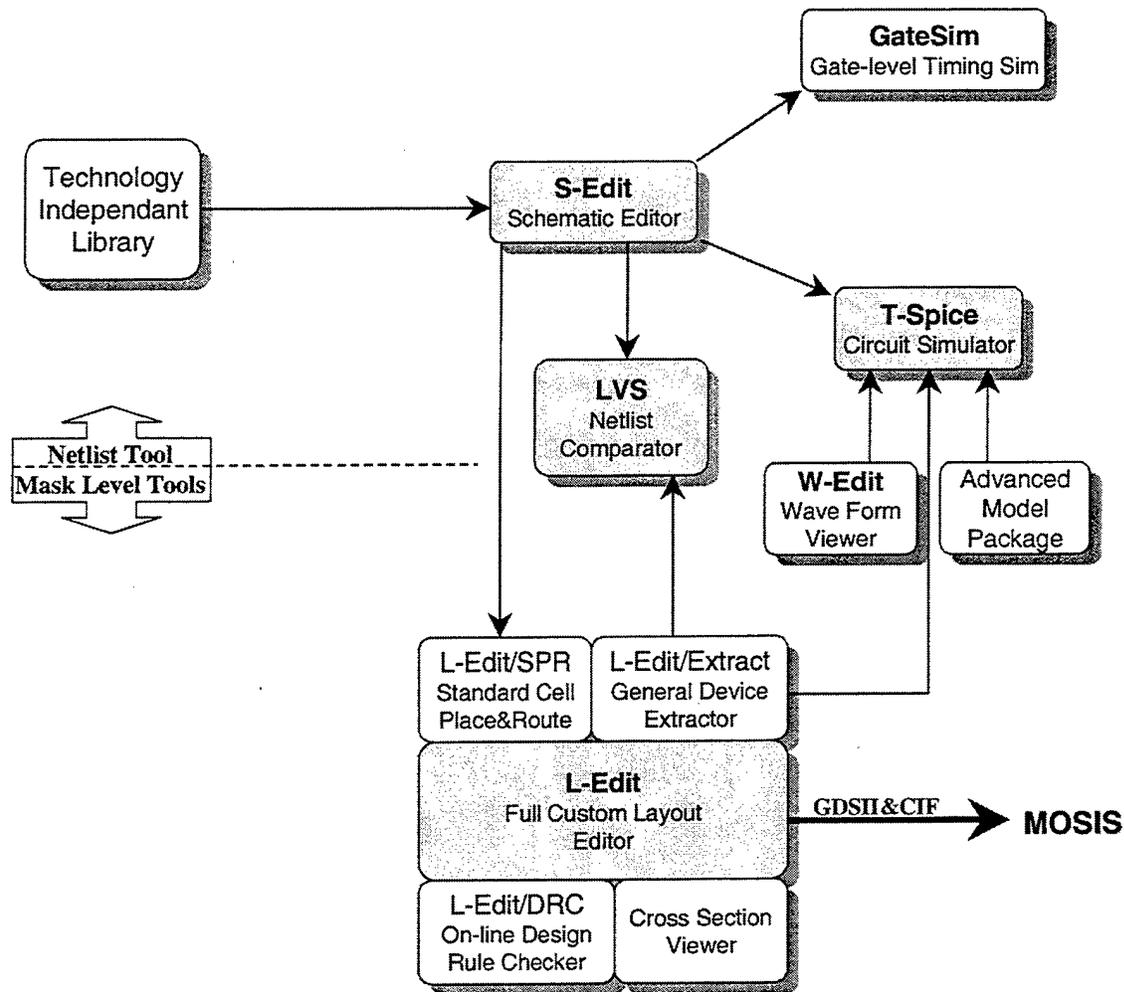
Figure 59. Tanner Tools Block Diagram (From Ref. [17])

1. Nettran

Nettran is a tool within the Tanner environment that has routines and libraries to import different file types and convert them into other Tanner programs readable formats. It is used as a netlist translation application to ensure file exchange between the different tools and other applications. Figure 60 [Ref. 17], shown below, illustrates how Nettran fits into the Tanner Tools environment. The use of Nettran is required to translate S-Edit

files into an appropriate format for the use in the logical simulator GateSim or third party programs.
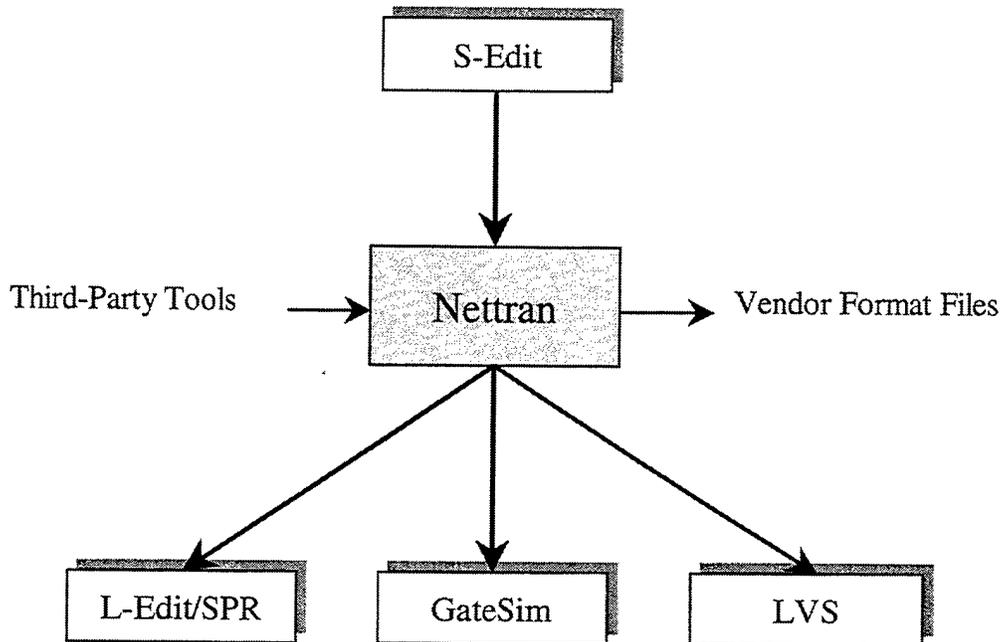
Figure 60. Nettran Function Block Diagram (From Ref. [17])

Nettran can to translate either wirelist, netlist, or EDIF files from third party tools like OrCad to standard Spice format, GateSim, or L-Edit netlist formats. Due to extended capabilities, S-Edit is now able to export a Tanner Data Base file (.TDB) that can be directly imported into L-Edit so that a conversion with Nettran is not required in this step.

2. L-Edit

L-Edit is a physical design layout editor that creates the device level fabrication files necessary to realize the integrated circuit. The user has different possibilities to enter a layout or project into L-Edit. The first is the layout by hand. Like in CAD tools, the elements of transistor components, e.g., P-Well, N-Well, or different metal layers are drawn in the editor window. Since this process is quite time consuming, the vendor

104

provides various libraries for different layout processes. These libraries contain layouts for digital design components, which are called L-Edit's wizards. Using the wizards is the second way to generate a layout in L-Edit. Besides the Block Place and Route tool (L-Edit/BPR), the Standard Place and Route tool (L-Edit/SPR) is the most important wizard. SPR and BPR generally perform the same task, where SPR is more specialized, provides more sophisticated functionality, and has more constraints.

The Standard Place and Route module generates layouts for standard cell design and can automatically construct entire chips. It includes cell placement and routing, pad frame generation, and pad routing. SPR reads netlist files produced by S-Edit and creates masks useable as a basis for fabrication. Nevertheless, this automatically generated layout needs at least to be verified with the DRC. Tests have shown that the SPR module is working only to a certain degree of satisfaction. Even with specified design rules for the target process, e.g., 0.5 micron, it can produce faulty designs. The automatic layout process is adjustable in many ways. The most important adjustments are the placement optimization factor and the routing optimization. The placement optimization factor determines the effort of the algorithm to reduce the size of the layout and therefore the size of the entire chip. Factors between 00.0 (no optimization) and 10.0 (highest level of optimization) can be specified. With higher factor the computation time will increase exponential with decreasing effect. Furthermore, tests have shown that the results with different factors are variable and that a higher factor does not necessarily produce better results. Unfortunately the only way to determine the best factor settings is through trial and error. The best result gives a trade off between the least possible DRC violations and the smallest layout area.

105

The Design Rule Checker (L-Edit/DRC) performs a design rule check for the intended fabrication process and can optimize the place and route. It verifies the generated layout with pre-defined rules, which can be edited or extended. Even with the automatic function of place and route with SPR or BPR, it is absolutely necessary to run DRC on a layout generated by these tools. Additionally, a design that passed the DRC is not assumed to be free from errors. A post-layout simulation in the circuit-simulator tool or a netlist comparison with the netlist comparison tool is crucial.

L-Edit/Extract creates SPICE-compatible circuit netlists from L-Edit layouts. The output can be exported in either GDSII or CIF file format for fabrication. The extract tool is the way to produce a Spice code for post-layout simulation or netlist comparison. For netlist comparison, see the description for LVS.

3. S-Edit

S-Edit is a schematic capture tool to enter the electronic layout of a circuit. For research and prototype devices intended for MOSIS fabrication, S-Edit contains a complete MOSIS library of components for each of the different scheduled runs, e.g., 0.5 micron or 0.35micron. S-Edit can directly generate netlists that are usable in the circuit simulator, where a direct link writes a complete schematic directly into T-Spice. S-Edit holds the complete DIS architecture in form of a schematic representation. To simplify the circuit creation, the program can use different design levels. In other words, repeatedly used circuits, such as a tapline in the DIS design, are assigned to a symbol. To accomplish this, S-Edit has two main workspaces, the schematic editor and the symbol editor. Besides the creation of electronic circuits in the schematic editor, the user can create a symbol for a circuit of any size in the symbol editor. Due to this possibility, S-

Edit is able to handle different levels of a project and can use custom-made circuits in the same manner as its own library elements on every level of the design. For the DIS project, S-Edit is used to construct a hierarchy consisting of five levels, where lower levels provide higher levels with building blocks to create more complex circuits.

4. Layout Versus Schematic (LVS)

LVS is a layout-versus-schematic netlist comparator. It compares the exported netlist from S-Edit and the extracted netlist from L-Edit/Extract. It can also compare the layout with any other SPICE compatible netlist and ensures that both netlists represent the same circuit. LVS is working on the logic gate level. It uses the pre-defined library element for comparison and is not able to compare on the transistor level. Therefore custom-made layouts on the transistor level are a potential problem.

The goal is to compare the layout mask generated by SPR with the schematic circuit in S-Edit. LVS is used to compare the netlists of both representations. This will guarantee the equality of the layout with the tested circuit before the design is sent to fabrication. As shown in Figure 61, the differences after netlist comparison are used for editing the compared files by hand. Finally this procedure will ensure the equality of the circuits.
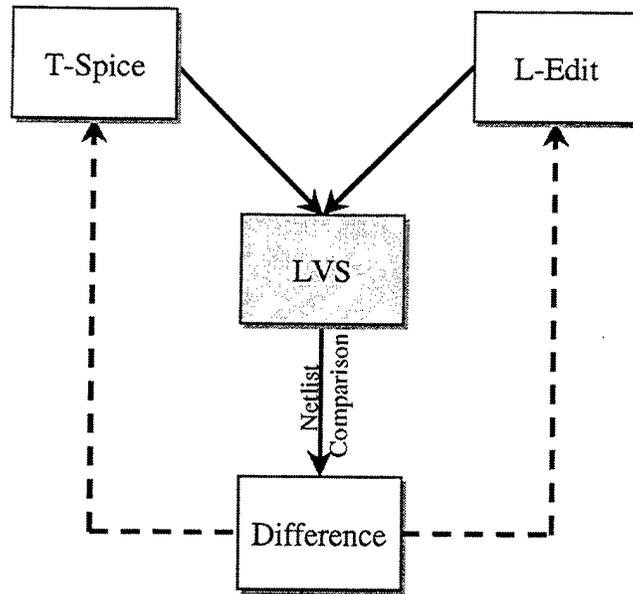
Figure 61. Flow for Netlist Comparison in LVS

5. The Circuit Simulator T-Spice Pro

T-Spice Pro is a complete circuit design and analysis system, which includes T-Spice, the Advanced Model Package, the waveform editor W-Edit, and S-Edit. T-Spice is a circuit simulator using SPICE as input language. The Advanced Model Package consists of the latest transmission and semiconductor device models to achieve more realistic simulation results that are closer to real world behavior. The Tanner Company claims that T-Spice simulates a circuit design with more than 300,000 elements. With extremely large circuits, the simulator requires an exponential increase of computer resources. To some extent T-Spice can handle very large circuits consisting of linear elements like switches or resistors. However, transistors are not linear and are approximated by polynomial functions. Because transistors are exclusively used in digital designs, the program is not capable of handling a large digital circuit with approximate 300,000 elements. The DIS design with 32 taplines consists of almost 290,000 transistors

108

and is not simulateable with T-Spice using a transistor model. In order to test the design, a switch model can be used to replace the transistor model. This approach will be discussed in greater detail in Chapter 9.

S-Edit (described above) provides a direct link to T-Spice, which makes the translation of the schematic design into a SPICE file easy. By adding parameters and bit pattern test vectors, the circuit logic can be tested before layout. T-Spice offers only a semi-usable algorithm for binary testing. The input data to the circuit are digital and coded in a binary form (0=0V, 1=5V), but the output will be in real voltages instead of binary words. Therefore the output is of limited use.

6. The Waveform Viewer W-Edit

W-Edit is a waveform editor acting primarily as a back-end data processor for the data generated in T-Spice. It is designed to display T-Spice simulation output waveforms. W-Edit is used to verify the functionality of small circuits like a register cell or a 2 input NAND gate. It is not useful for larger circuits.

B. DIGITAL IMAGE SYNTHESIZER ARCHITECTURE

This section focuses on the new DIS design (ASIC architecture) and discusses its implementation in detail. The ASIC architecture is based on a modified FPGA concept, where a tap and its associated range-bin processing is now called a *tapline* to distinguish between the two implementations. The general data flow within a tapline is shown in Figure 62. The main differences between the original architecture and the modified architecture as implemented in the ASIC are summarized as follows:

109

1. Parallel DRFM-phase data input into all (32) taplines simultaneously instead of serial inputs through a tap delay line

2. Implementation of registers in the data flow of a tapline (pipelining

3. Serial summation of the tapline data output in order to achieve the necessary delay and to add the output data in correct sequence for the final output

4. Built-in *scan-path test* capability
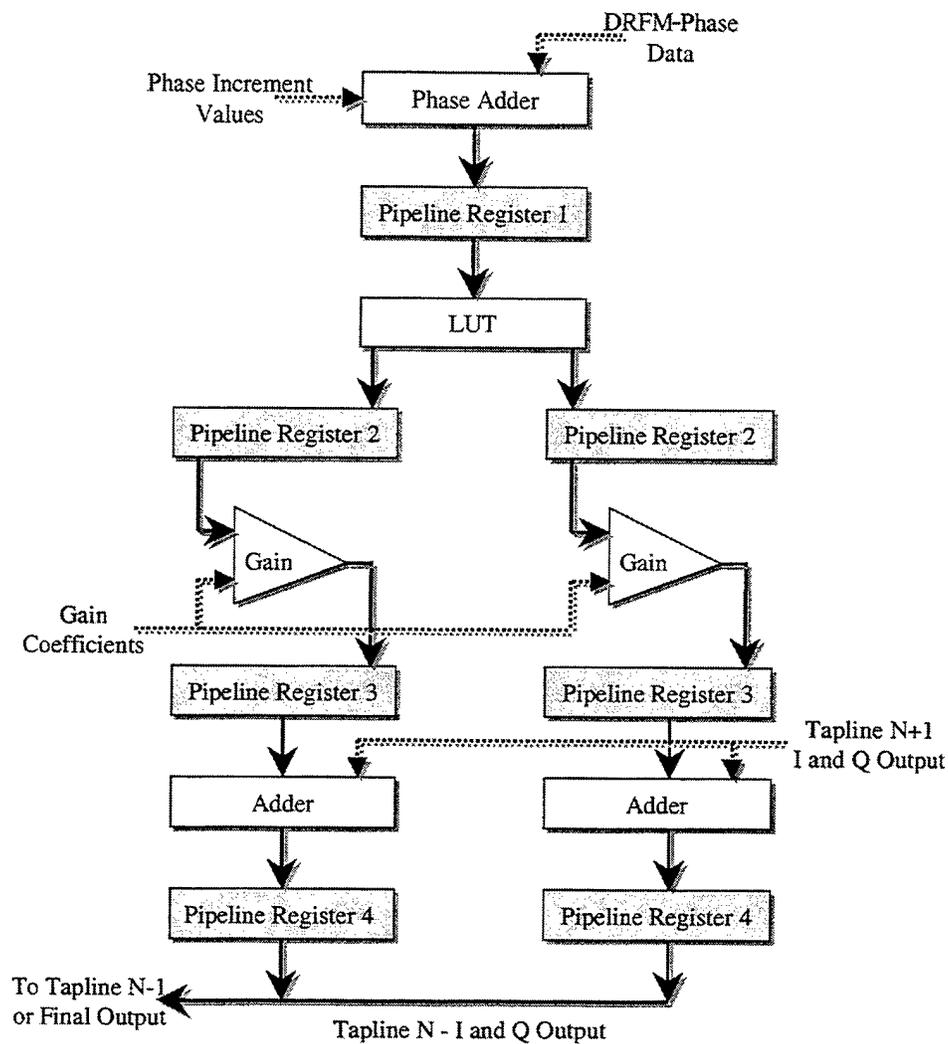
Figure 62. Tapline in ASIC Architecture

As illustrated in Figure 62, the DRFM-phase-increment values ($\phi(n)$) are one of the four inputs for a tapline. So far, the integrated circuit is composed of 32 taplines, which synchronously receive (no delay) the same clocked DRFM-phase data for each tapline as input. The Phase Adder combines the DRFM-phase data and the phase-increment value ($\Delta\phi_n$). The phase-increment consists of the phase-rotation of several backscatters and is generated off chip for the most recent tapline version. The result of the phase addition in the phase adder continues to propagate into Pipeline Register 1, where it is available for the LUT (Look Up Table) after the first clock cycle. The LUT uses this input as a pointer to an address space in the LUT-ROM and stores the resulting I and Q values into Pipeline Register 2. After the second clock cycle, the values can penetrate the gain block, where the appropriate gain (A_n) is applied. After the third clock cycle the values can enter the second adder. The adder's function is to combine the phase data with the phase data from the next higher tapline. Figure 63 illustrates this concept of the summation in a simplified way. To compensate for not having delay in the input DRFM-phase data, delay
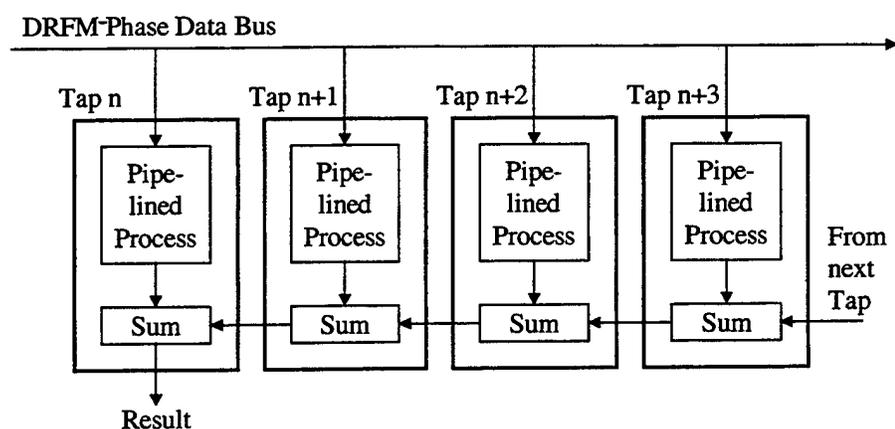
Figure 63. Simplified Data Flow in the ASIC Architecture

is automatically achieved during the second addition ("Sum" in Figure 63) by using a pipelined adder chain instead of an adder tree. Recall that the processed DRFM-phase data resulting in a tapline output are of the form $P_n(D_3) + P_{n+1}(D_2) + P_{n+2}(D_1)$. In the original architecture, this form is achieved by delaying the DRFM-input data at the input because it is propagating through all available taps. The parallel summation at the end of the process gives the above-mentioned form. In the ASIC arcitecture, the adder chain not only creates the required delay, it also eliminates the two most significant problems of the original design. First, adder chains are easily extensible since additional adders can be chained together connecting output to input, as long as the adders do not overflow. If the adder overflows, it is a simple process to increase the number of bits for the second adder in the VLSI library. Second, in a pipelined ASIC architecture, the total pipeline latency from the first input-data to the first output-data is the pipeline latency in the tapline plus the pipeline delay of only one adder. Thus, as the number of taplines increases, the latency stays the same instead of increasing as with a growing adder tree. Of course, the latency from the last phase-data input sample to the last output result does increase, but this is inherent in the algorithm being used and occurs in both designs.

As shown in Figure 62 and Figure 63, every tapline combines its processed data with the output data of the next higher tapline. The result is a chain of data between the first and the last used tapline. Table 19 illustrates this concept. The clock cycles used to describe the data flow in the tapline ignore the time necessary to load the inputs into the IC. The method of describing the data in Table 19 simplifies this even more and assumes, that no time is needed to process the data within a tapline. In this example tapline n (T_n) is the first one in a row of three taplines. The output of T_n is the final output, consisting of

I and Q values. After clock one, every tapline produces an output with the same DRFM-phase data (D_1) as input. With clock two, the output of T_{n+2} gets added to the processed data (D_2) in T_{n+1} and the output of T_{n+1} gets added to the processed data (D_2) in T_n. Continuing with this concept, the final output is the same as the previously proven FPGA architecture.

Radar Pulse	DRFM Data	CLK	tapline n	tapline n+1	tapline n+2
1	D_1	0	$P_n(D_1) + 0 + 0$	$P_{n+1}(D_1) + 0$	$P_{n+2}(D_1)$
1	D_2	1	$P_n(D_2) + P_{n+1}(D_1) + 0$	$P_{n+1}(D_2) + P_{n+2}(D_1)$	$P_{n+2}(D_2)$
1	D_3	2	$P_n(D_3) + P_{n+1}(D_2) + P_{n+2}(D_1)$	$P_{n+1}(D_3) + P_{n+2}(D_2)$	$P_{n+2}(D_3)$
1	D_4	3	$P_n(D_4) + P_{n+1}(D_3) + P_{n+2}(D_2)$	$P_{n+1}(D_4) + P_{n+2}(D_3)$	$P_{n+2}(D_4)$
.	.	.	.	.	.
.	.	.	.	.	.
1	D_{62}	61	$P_n(D_{62}) + P_{n+1}(D_{61}) + P_{n+2}(D_{60})$	$P_{n+1}(D_{62}) + P_{n+2}(D_{61})$	$P_{n+2}(D_{62})$
1	-	62	$0 + P_{n+1}(D_{62}) + P_{n+2}(D_{61})$	$0 + P_{n+2}(D_{62})$	0
1	-	63	$0 + 0 + P_{n+2}(D_{62})$	$0 + 0$	0
1	-	64	0	$0 + 0$	0
.	.	.	.	.	.
2	D_1	65	$P_n(D_1) + 0 + 0$	$P_{n+1}(D_1) + 0$	$P_{n+2}(D_1)$
2	D_2	66	$P_n(D_2) + P_{n+1}(D_1) + 0$	$P_{n+1}(D_2) + P_{n+2}(D_1)$	$P_{n+2}(D_2)$
2	D_3	67	$P_n(D_3) + P_{n+1}(D_2) + P_{n+2}(D_1)$	$P_{n+1}(D_3) + P_{n+2}(D_2)$	$P_{n+2}(D_3)$
2	D_4	68	$P_n(D_4) + P_{n+1}(D_3) + P_{n+2}(D_2)$	$P_{n+1}(D_4) + P_{n+2}(D_3)$	$P_{n+2}(D_4)$
.	.	.	.	.	.
.	.	.	.	.	.
64	D_{62}	4093	$P_n(D_{62}) + P_{n+1}(D_{61}) + P_{n+2}(D_{60})$	$P_{n+1}(D_{62}) + P_{n+2}(D_{61})$	$P_{n+2}(D_{62})$
64	-	4094	$0 + P_{n+1}(D_{62}) + P_{n+2}(D_{61})$	$0 + P_{n+2}(D_{62})$	0
64	-	4095	$0 + 0 + P_{n+2}(D_{62})$	$0 + 0$	0
64	-	4096	0	$0 + 0$	0

Table 19. Tapline Outputs with Three Taplines

Remarks for Table 19

1. A radar pulse consists of 62 samples, where a sample is the DRFM-phase data

2. Tapline n, n+1, n+2 are the outputs of the three taplines.

3. $P_{n+x}(D_y)$ represents the processed phase-data sample available at the designated tapline output.

Table 19 ignores the time that is needed to process data within a tapline. To complete this discussion, Table 20 summarizes the clock cycles needed to process the data within a tapline.

Clock Cycle	Output available at:
0	Phase Accumulator
1	Output of Pipeline Register 1 (5-bit)
2	Output of Pipeline Register 2 (8-bit)
3	Output of Pipeline Register 3 (11-bit)
4	Output of Pipeline Register 4 (16-bit), end of tapline

Table 20. Clock Cycles within a Tapline

Before data can be processed in a tapline, it must be loaded into the chip. Even though the loading process requires a certain number of clocks, this section considers only the general concept of a tapline. The loading cycles for the chip are addressed later.

Due to the adaptation of registers between the building blocks of a tapline, a test path is installed to improve the testability and functionality for the entire IC. This scan-path test capability can be used to strobe values into the registers to produce results for special test cases. The test vectors within the registers can then be processed for a desirable number of clock cycles before they are read out again. The implemented scan path is also part of later discussions.

C. SCHEMATIC DESIGN IMPLEMENTATION

The following section provides information about how the DIS concept is implemented in the schematic capture tool S-Edit. Since the program supports design hierarchy, the DIS architecture is divided into five design levels. In order to increase the signal flow control and the functionality, several control signals are introduced. These control signals are also used to indicate the states (valid/not valid) of the output data. Furthermore a scan-path test capability is installed to enhance the testability for sub-levels during the test phase and to verify the correct operation for the entire IC.

1. General Design Hierarchy

S-Edit is a schematic editor to enter an electronic layout or schematic of a circuit. It is capable of creating hierarchical circuits by using library elements or self-created modules. Tanner provides the customer with a great variety of modules and building blocks, but only a few of them are used to build blocks on the lower levels for the use on higher design levels.

The DIS design in S-Edit consists of five levels. The first level uses transistors or low-level building blocks like logic gates. Using blocks from lower hierarchy levels allows creating higher levels in order to increase the complexity stepwise. This concept provides two main advantages:

1. The logic of the design is more obvious, easier to understand, and easier to verify.

2. The layout editor L-Edit can use the same hierarchy to synthesize the layout stepwise. Since the hierarchical layout process allows a slow increase in the complexity, the layout editing is easier, more reliable, more efficient and faster.

116

The following hierarchy tree illustrates the structure of the architecture in S-Edit:

1. Level 1 elementary building elements from existing libraries or modified library elements. This includes for example, a register cell, an adder cell, a Mux2, transistors and all logic gates.

2. Level 2 builds on elements from Level 1 to create: 5-to-32-bit decoder part 1, 5-to-32-bit decoder part 2, the LUT-ROM, Gain Shift, N-bit register, and N-bit adder.

3. Level 3 makes use of the elements from Level 2 and 1 to build the tapline.

4. Level 4 holds the Supertap and the Supertap Mirror consisting of Level 3 and level 1 components.

5. Level 5 consists of a 5-to32-Bit decoder and extends the concept of Level 4 components to create the top level circuit with a data input bus, input pads, and output pads.

A complete graphical representation of each building block can be found in the Appendix. Also listed are the symbols and schematics for all sub-circuits used for the design implementation.

2. Architecture Circuit Description in Level 1

a.   Basis Elements

The very basic elements in a digital design are the P-FET and the N-FET transistors. These types of transistors are represented only by a symbolic appearance, which is specified by a SPICE output statement, as shown in Figure 64. The SPICE

[A S = 56 *1*1]

[PS = 24 *1]

W = 28 *1

L = 7 *1

[M = 1]

[A D = 56 *1*1]

[PD = 24 *1]
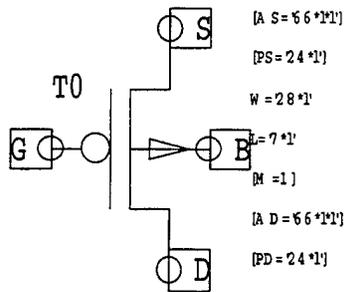
Figure 64. P-FET Transistor

output defines the transistor in ports, gate length and width. Additionally parameters are also defined that are not relevant for the current design and therefore are not mentioned here. The important parameters are multiples of the technology specific variable lambda (l) so that the transistor is scaleable and can be used for different layout processes. Due to monetary reasons and availability of certain process runs at MOSIS, we decided to target on HP 0.5μm process. Nevertheless, if the concept is demonstrated and the IC is fully operational, the target process can be easily changed to a smaller (faster) process without any changes to the existing design in S-Edit.

b.   Adder Cell

The adder cell and register cells, as shown in Figure 65 and Figure 66, are building blocks to create n-bit adders or n-bit registers. The adder cell can add two 1-bit binary input words. The input pads are labeled A and B, where the third input pad, Ci the carry-in bit is used to connect two or more adder cells. The carry output pad, Co and the output pad S define the 2-bit output word. The function of the cell is described by the following two equations:

118

$$S = invB * invCi + invA * B * invCi + invA * invB * Ci + A * B * Ci \qquad (7.1)$$

$$Co = A * B + B * Ci + A * Ci \qquad (7.2)$$
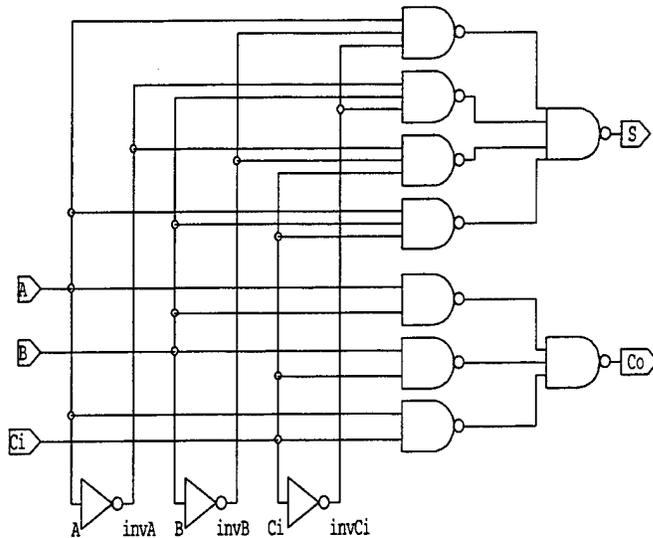
Figure 65. Adder Cell

To build an n-bit adder, Co of cell N gets connected to Ci of cell N+1. A 5-bit adder and a 16-bit adder are part of Level 2 in the hierarchy.

c.   Register Cell

The register cell, as shown in Figure 66, implements the scan-path test and introduces control over the data flow in the tapline logic. The control logic consists of hold, load, clock, and the scan path pads. Besides the clock, only one of the control signals is allowed to become high at the same time. If load goes high, the register performs normal operations and "clocks" the input to the output. A logical high for "hold" freezes the last processed value and ignores new input data. If all control pads are low at the same time, the register is forced to perform a synchronous clear (all outputs become low). To construct an n-bit register, Q of cell N must be connected to SRDi of cell N+1 and Q of cell N must be connected to SLDi of cell N-1, where the register

119

control pads are connected in parallel. A 2-bit register, a 4-bit register, a 5-bit register, an 8-bit register, an 11-bit register, and a 16-bit register are part of Level 2 of the design hierarchy.
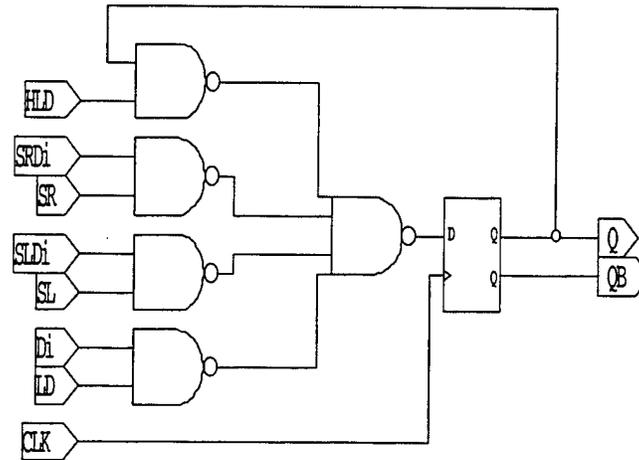
Figure 66. Register Cell

The register cell in Figure 66 consists of 48 transistors for the logic gates and the D-Flip-Flop. By using library elements throughout a module, L-Edit can perform the layout process without any complication. Nevertheless, this construct has three levels of delay and cannot be driven at very high clock speeds. Furthermore, in a tapline there are 94 register cells. Depending on the number of taplines used for one chip, the layout area could be reduced to a more compact design. ASIC designers use Transmission Gates (TG) for these purposes. Transmission Gates are basically one P-FET and one N-FET transistor connected via drain and source controlled through their gate ports. The control signal is spit to provide two signals, the signal itself and its complement. If the control signal is high, the TG lets the input data pass. If the control signal is low, the TG blocks the input data. The described behavior is identical to that of a simple switch controlled by

voltage. Figure 67 shows a register cell using TGs. The control signals in this cell are the same as for the register cell in Figure 66 plus their complements needed to drive the gates. The controls drive the column of four TG (counting from the bottom) and implement the basic register functions load, hold, and the scan path features. At the top of the figure, the row of four TGs is responsible for the data output and is basically formed out of two latches. The marked point "M" separates the master (left) from the slave (right). The clock signal and its complement with a combinational logic of inverters drive the gates. Due to the implementation of the clock signal, the master together with its slave form a positive triggered flip-flop so that its behavior is quite different from a latch. This means that at clock low the master gets loaded with the resulting value out of the TG column. If the clock switches from low to high, the loaded value gets stored in the master's second TG and feeds the slave, which is now in loading mode. When the clock changes again, the master can load new data, where the slave is in store mode and feeds the output.

The D-Register cell consists of 26 transistors. This is a reduction of 45% in comparison to the current used register cells. For a chip with 32 taplines and 95 registers per tapline, 66,880 transistors could be saved. Moreover, TGs are used for high-clock rate circuits and process data faster. In spite of these advantages, the critical factor is the coherence of the control signals and its complement. The phase difference for the clock in particular must be held at a minimum to ensure correct function at high clock speeds.
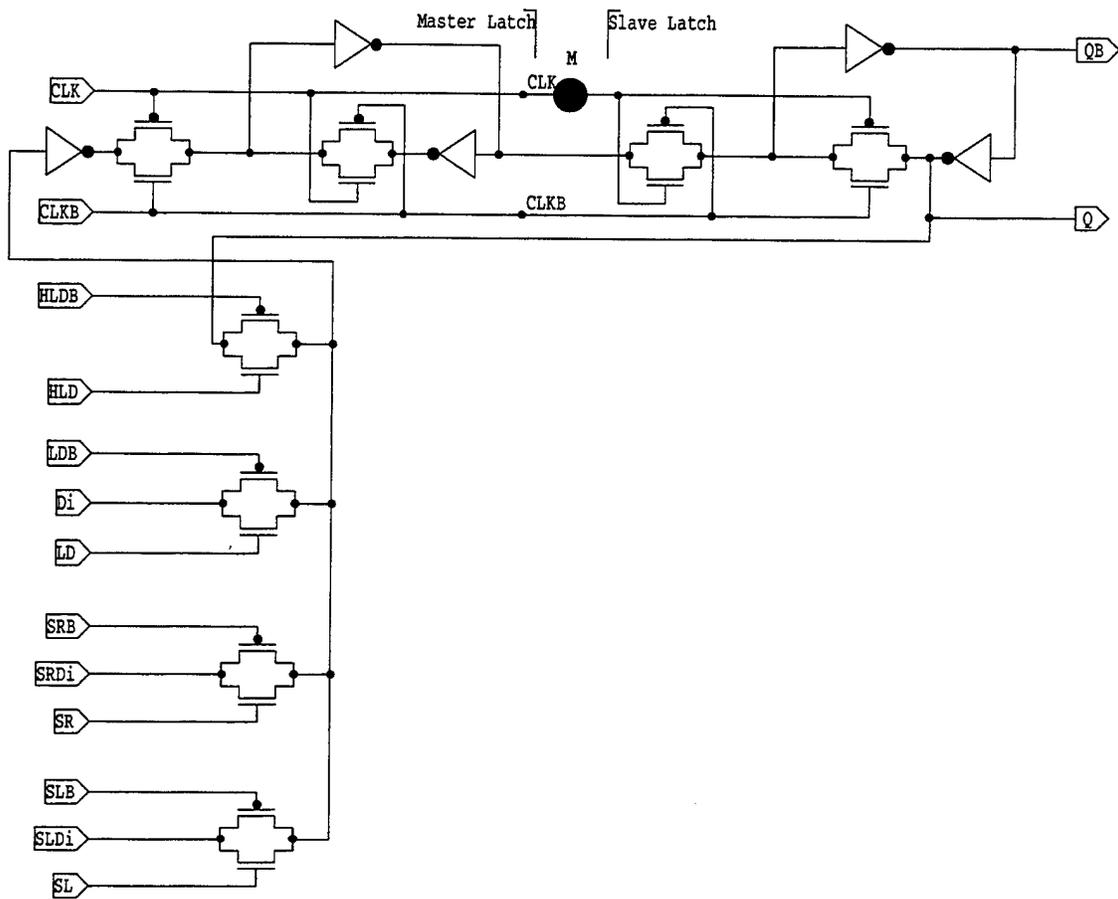
Figure 67. D-Register Cell

3. Architecture Circuit Description in Level 2

Level 2 elements are the building blocks for a tapline.

a. Look-Up Table

The Look-up-Table (LUT), as shown in Figure 68, is a composition of three sub-building blocks that are listed in the Appendix: 5-to-32 bit decoder part 1, 5-to-32 bit decoder part 2, and the LUT-ROM. For simplicity during circuit creation, the 5-to-32 bit decoder was split into two parts. The two parts together use a five bit binary input and convert it into an address space used as input for the Look-Up-Table. A five bit

122

binary number represents 32 decimal numbers. Each of these numbers corresponds to two lines in the LUT module. Therefore the five-bit input triggers the corresponding address line in the LUT, where the LUT makes the stored value available at the output. Figure 69 shows only a small part of the LUT-ROM to illustrate the general structure.
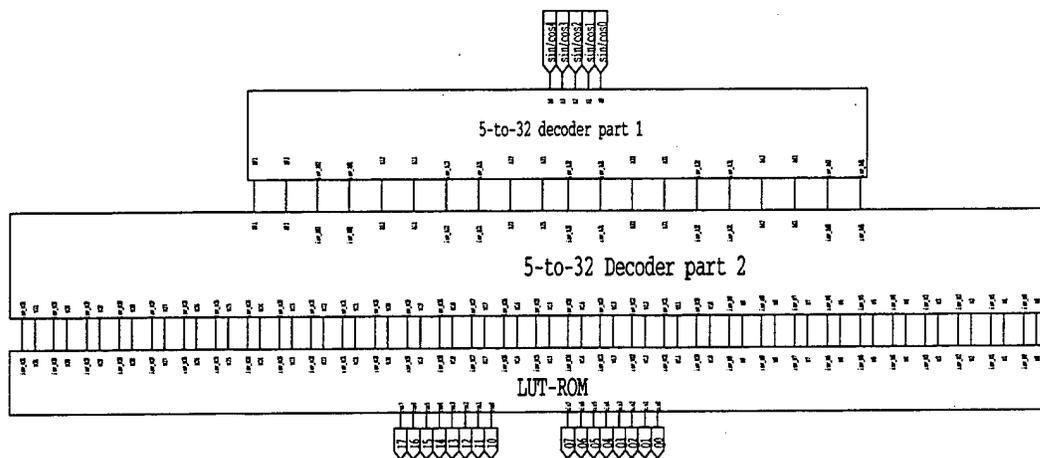
Figure 68. Look-Up-Table (LUT) Module

The entire block is shown in the Appendix. The ROM consists of 32 double rows of transistors with a length of 16 transistors per row, each divided into two columns. Every double row represents a 2 * 8 bit value, the cos (I) and sin (Q) outputs. Placing P-FET and N-FET transistors at the wire crossing of row and output pad programs the desired output value as shown in Table 21. Recall that this is the same concept as for the FPGA architecture. The difference here is that the ASIC architecture combines the table look-up for the I and Q phase values in only one table, using the fact that I and Q are always in phase quadrature.

123
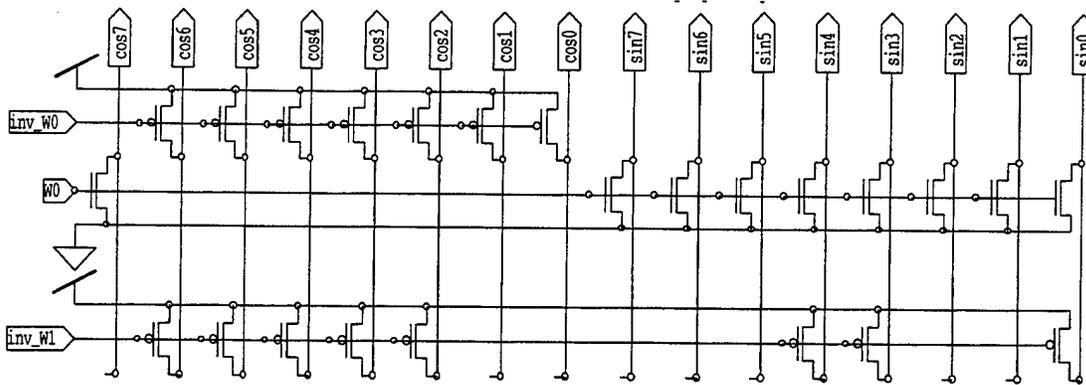
Figure 69. Part of the LUT-ROM

Program a "0"	Program a "1"
Set N-FET transistor in row for input pad W_n	Set P-FET transistor in row for input pad inv_W_n

Table 21. LUT Programming

b. *Gain Shifter*

The Gain Shifter "multiplies" the input by performing a shift. The binary input pads Gain0 and Gain1, as shown in Figure 70, determine the amount of shift or gain applied to the two's complement input word. Two rows of Mux2 perform the shift of the input. The first row connects its "select" input to Gain0 and the second one to Gain1. If select gets high, the Mux2 uses Port A as input otherwise Port B. Since Port A is connected to the input of the next lower bit, the row performs a shift of one digit to the left. The Mux2s of the second row connect their Port A to the next but one input bit performing a shift by two. The equation for a Mux2 is as follows:

$$Out = MuxA * Sel + MuxB * not\_Sel \qquad (7.3)$$

Table 22 illustrates the gain effects on the input and summarizes the shift discussion.

124

Binary Input Gain1 Gain0	Gain Factor	Multiplication Factor	Effect on binary input word
0 0	0	1	No effect on input word; input = output
0 1	1	2	Input word gets shifted by one digit to the left, For example: Input = 1101 Output = 11010
1 0	2	4	Input word gets shifted by two digit to the left, For example: Input = 1101 Output = 110100
1 1	3	8	Input word gets shifted by three digit to the left, For example: Input = 1101 Output = 1101000

Table 22. Gain Shift

The gain factor in Table 22 is the integer representation of the two gain inputs. They are related to a multiplication factor as specified in the Matlab m-file Range-Doppler-Amplitude Map Entry Program described in previous chapters.

Figure 70 illustrates the concept of the Gain Shift block. It shows the logic that leads to the shift of the two's complement binary input. The Gain Shift block or Gain Modulator requires an 8-bit two's complement input word and two gain-coefficients (Gain0, Gain1). Due to the largest possible shift of three positions with gain-coefficients of Gain0=1 and Gain1=1, the output word can be an 11-bit two's complement binary number.
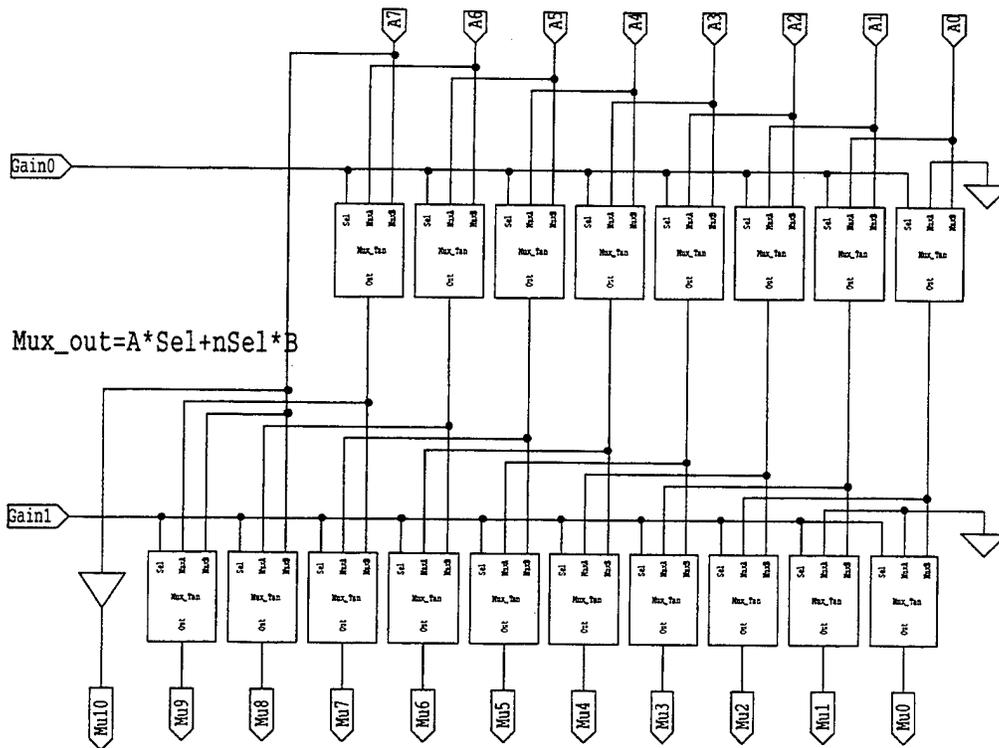
Figure 70. Gain-Shift Block

The dynamic range of the Gain Shifter can be calculated as follows:

$$DR = 20 * \log(\text{max multiplication}) \tag{7.4}$$

$$DR = 20 * \log(8) = 18 \text{ dB} \tag{7.5}$$

Since a dynamic range of 18dB is not sufficient to counter a sophisticated ISAR, a higher dynamic range might be desired for future design implementation. Adding another row of Mux2s performing a shift of four easily does the extension. The highest multiplication factor for a shift of seven bits is 128, which would increase the dynamic range to 42dB. Extending the current gain shifter by even two more rows would increase the dynamic range to 90dB.

4. Architecture Circuit Description in Level 3

Design architecture level 3 holds only one, but the most important module, the tapline. The tapline combines the modules of Level 2 and Level 1 to form the data pipeline for processing as descried earlier. Three different taplines have been created, where only two are described here. The third tapline is a realization of the D-Register cell implementation and was not tested.

a.   *Tapline with Phase-Rotation*

A tapline as shown in Figure 71 is the central building block of the DIS architecture since every other block in higher design levels is a multiple of this module. The chip capabilities are directly related to the number of taplines implemented in the chip. Every additional tapline extends the possible size of a false target. In reference to the Range-Doppler Map Entry, one tapline in the hardware represents a single cell in the Range-Doppler Map. Currently the chip design contains 32 taplines. Therefore the target extent is 32 cells in the "Range-Doppler-Amplitude Map Entry program," which can be related to a physical false target extent of:

$$1.2\text{m (for each cell)} * 32 \text{ taplines} = 38.4\text{m} \tag{7.6}$$

The tapline module allows an unproblematic interconnect of several taplines so that a greater target extend can be achieved by adding taplines. A constraint for more than 32 taplines is the size of the second adder, as discussed in upcoming chapters.

Figure 71. Tapline with On-Board Phase-Increment

Due to the complexity of the tapline, it is split into three sub-blocks for illustration purposes. The control signals and the scan-path test are not part of the discussion and are excluded for now. The first block is called the phase-incrementer and is shown in Figure 72. It consists of a 4-bit register, a 5-bit register and a 5-bit adder and requires a 4-bit binary input word. The two's-complement binary input is the desired phase-increment value ωPRI that has to be added to the DRFM-phase data. The phase-incrementer

128

supplies integer multiples of the desired phase-increment ($n\omega PRI$) to the phase data on a pulse-to-pulse basis. That is, due to the phase-rotation requirement, the output of the phase-incrementer must increase or decrease by the amount of the phase-increment value for every new pulse. The increment value supplied to the tapline may be constant over several radar pulses (constant Doppler frequency). To achieve this "constant," the increment value is added as the first input for the 5-bit adder, where the connection
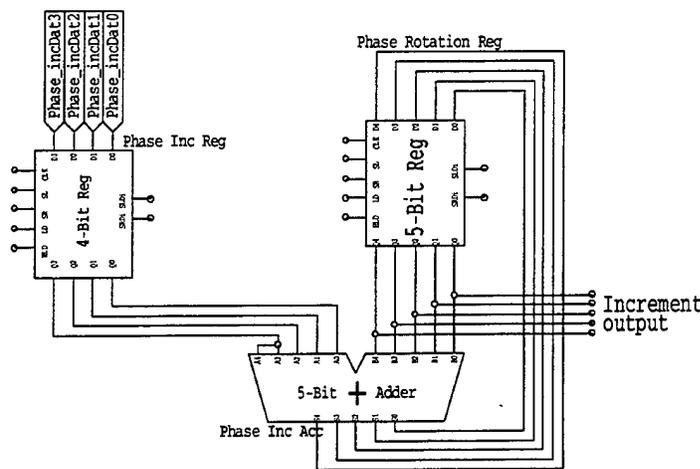
Figure 72. Phase-Increment Block

between bit four and five is the sign extension. The adder output goes into a 5-bit register that is again connected to the adder in a loop. Due to this construction, the output of the 5-bit adder is always a n-multiple ($n = 1, 2, ...$) of the original input. The phase-rotation can be adjusted by control signals, which control the registers in this block. The master clock controls the overall behavior of the registers. Since a register needs one clock cycle to produce a valid output, the Phase-incrementer has a requirement of at least one clock cycle before a valid result is present at the output. Therefore the phase-increment in the 5-bit register needs to be activated exactly one clock cycle before a new pulse with new DRFM-phase data can be processed to produce a valid output. The output is a two's

129

complement binary word, which is the input to the next logical block, the LUT-Module. An alternative phase-increment design is described in Section B that allows even further flexibility in false target generation.

The LUT-Module, shown in Figure 73, is the second sub-block of a tapline. It consists of a 5-bit adder, a 5-bit register, a LUT for I and Q values, and a 8-bit register. The adder takes the DRFM-phase data inputs and the Phase-incrementer outputs and adds them. Note that the addition of two 5-bit binary words could result in a 6-bit word. This fact can be ignored, since the Phase-incrementer output is a phase value repeating over a period of 2π. Therefore the adder output, in conjunction with the cosine and sine LUT, can also be shown to be periodic over five bits.
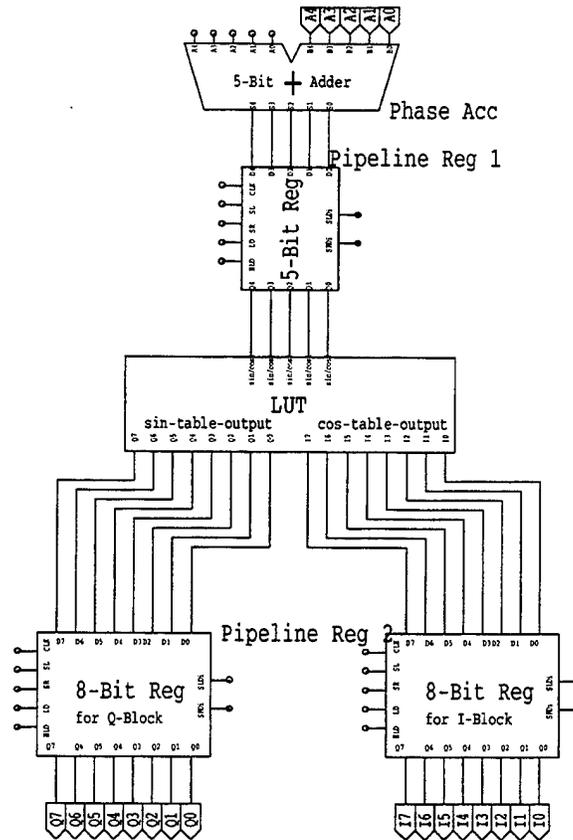
Figure 73. Tapline LUT Module

130

In continuation with the data flow, the adder output leads into a 5-bit register (Pipeline Reg 1), where it is available at the output after one clock cycle. The LUT block takes the 5-bit input word and uses it as an address to determine the corresponding I and Q values in the sine/cosine LUT as described earlier. From this point the signal flow is divided into two data channels, which are the in-phase and quadrature values (I and Q). However the operations performed on data within the I and Q channel will be the same up to the final output at the end of the first tapline. The LUT output data are inputs to an 8-bit register (Pipeline Reg 2) and become available at the output after the next clock cycle leads into the last module, the gain and adder block

The "gain and adder block" is shown in Figure 74 and consists of a gain shift block with a gain register, a 16-bit adder, and a 16-bit register each for the I and Q channels (compare also with Figure 71). The input to the gain shift is the output of the LUT. As described earlier, the gain shift performs a shift of the input data in accordance with the specified gain-coefficients. The output results in a two's complement 11-bit word for each channel. After one clock cycle, the values are present at the output of the following 11-bit register (Pipeline Reg 3). The following 16-bit adder takes two inputs. One is the output of the Gain Shift block, which again requires a sign extension to the most significant bit achieved by the interconnection of bits 11 to 15. The second input is the output of the tapline that is the next higher in a cascade of 32 taplines. This illustrates the concept mentioned at the beginning of this chapter. To recall, imagine that the considered tapline is tapline #1. The next higher tapline is #2. After four clock cycles, the first outputs at both taplines are available at the gain and adder block output (see Table 20), where tapline #2 presents its values to the 16-bit adder of tapline #1. With the next

131

clock cycle, tapline #2's incoming data is added to tapline #1's data coming out of the 11-bit register after the gain shift. After the addition in the 16-bit adder, the values are presented to the input of the last register (Pipeline Reg 4) in the tapline. After one more clock cycle, the values in the form of 16-bit two's complement words are available at the tapline output. The size of the last adder determines the numbers of taplines because all outputs are added at the adder inside the first tapline. A 16-bit adder can be used with 32 taplines without achieving an overload at the end of the chain. Every multiple of the current 32 taplines requires increasing the number of adder bits by one bit.
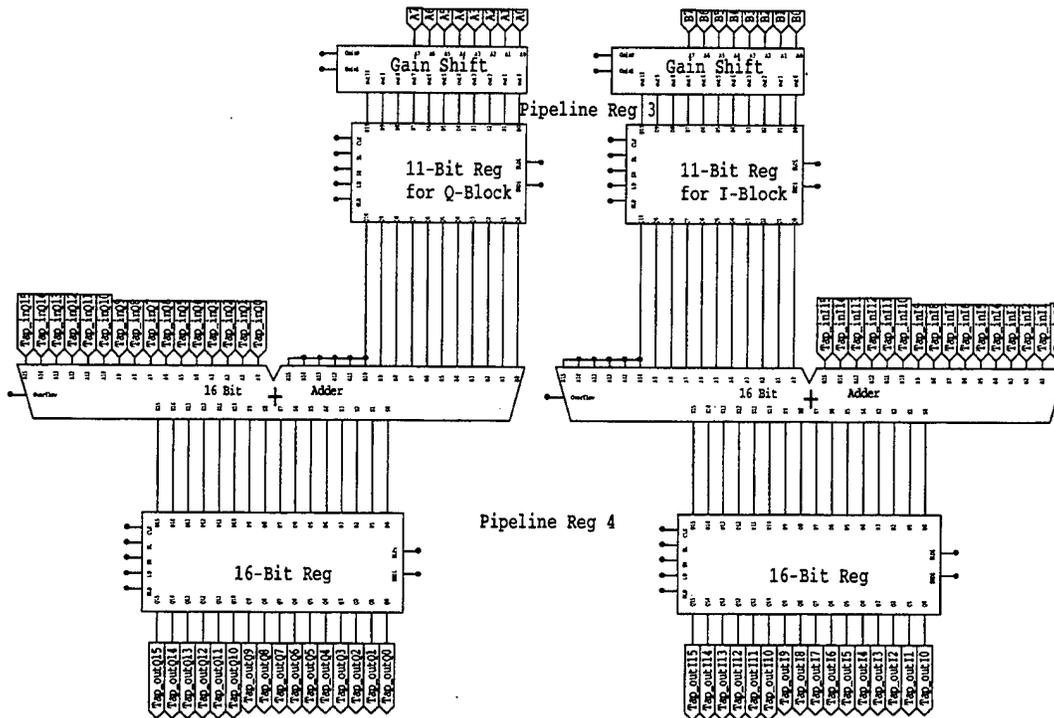
Figure 74. Tapline Gain and Adder Block

All of the three sub-blocks that have been discussed include control and test signals managing the data flow to achieve the physical requirements for the DIS. These signals are not part of this section but are discussed in detail later on. Furthermore, there are four

132

1-bit registers on the right hand side shown in Figure 71, that were not part of the discussion either. These registers are used to allow particular control signals to penetrate the tapline synchronously with the clock.

b. Tapline with Double Buffering

Figure 75 shows a modified version of a tapline with double-buffered phase data and gain-coefficients. The main body for DRFM-phase data treatment is still the same, but the application of the phase data and the gain-coefficients is different. The Phase-incrementer is replaced with a double register buffer consisting of two 4-bit registers as shown in Figure 76 (compare with Figure 72). The generation of the phase-increment is done off chip and can be applied on a PRI to PRI basis. Therefore the flexibility is increased, since generation of multiple scatterers (superposition of several Doppler phases) are possible within one Range-Doppler-cell.

For this process, the phase data is stored in the second register while new data is loaded into the first register. The phase-increment is normally fixed for one radar pulse so that the loading requirements will not reduce the speed of the data processing. The only trade off in this design is the resolution in Doppler due to the supplied phase data. The phase-rotation in the other tapline design allows a controller to produce multiples of the original 4-bit input phase, which could result in a 5-bit output going into the phase accumulator. With phase buffering the input is four bits and no phase-rotation is done on chip. The phase data must propagate through the registers before being added to the DRFM-phase data in the phase accumulator. The phase accumulator requires two 5-bit inputs to produce the different Doppler phase values. In order to avoid limitations,

133
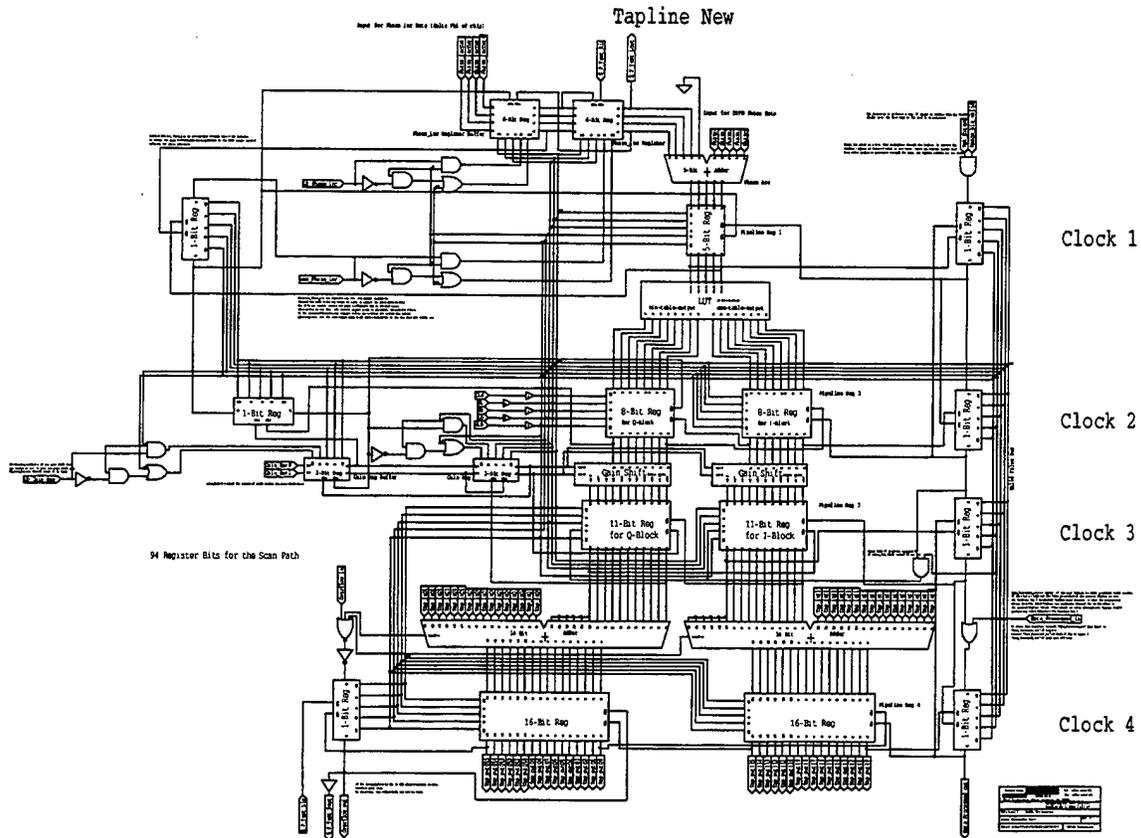
Figure 75. Tapline with Double-Buffered Phase and Gain
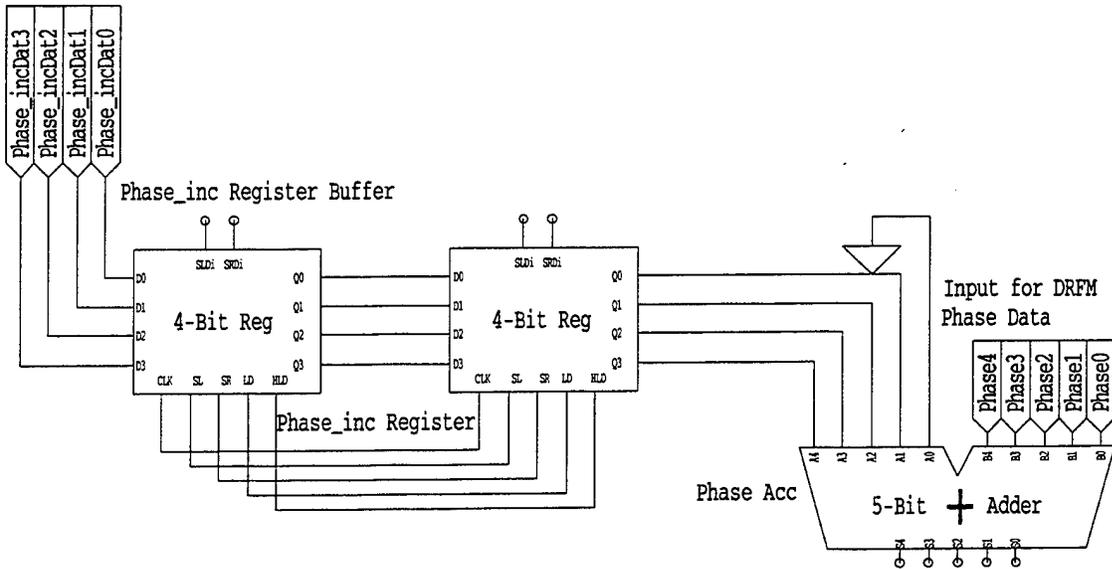
Figure 76. Phase Buffering

the least significant bit for the phase-increment data leading into the phase accumulator is fixed at zero (grounded). This overcomes a reduction in the range of possible phase-increment values since the four input bits are equally spread over a range of five bits. Nevertheless the phase-increment data are limited to even decimal numbers loaded as binary words. This is considered acceptable for a concept demonstrator, since otherwise a major redesign of Level 4 and Level 5 would be required and would postpone the fabrication considerably.

Figure 77 shows the gain-coefficients that are also double-buffered. This provides increased flexibility in terms of loading the chip. The gain-coefficients normally change together with the phase data with every radar pulse. A logic is required to change the new gain at exactly the same time the processed data is present at the gain shifter. This logic is not shown in Figure 77, but can be observed in Figure 75. To change the gain-coefficients, the control signal activating the phase-increment data penetrates two

1-bit control registers. Thus, the control signal is delayed by two clock cycles, which represent the time the data needs to reach the gain shifter. Once the control signal is present at the second gain register, the register loads the new gain-coefficient from the gain register buffer and the gain shifter uses the new gain to perform the shift of its input data.
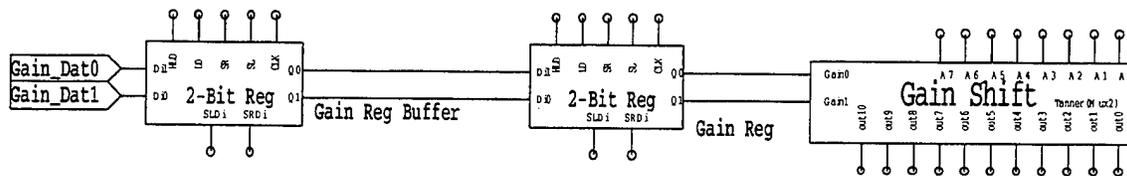
Figure 77. Gain-Coefficients Double Buffer

5. Architecture Circuit Description in Level 4

Level 4 of the design construct in S-Edit pursues the tapline. Figure 78 shows a Supertap that consists of 8-taplines connecting the data pipleline in series. The Supertap design is not affected by the two different tapline designs mentioned in the previous section. The only difference is the name of a control signal that reflects the tapline used to construct the Supertap. Form now on, we will consider only the double-buffered tapline, as it is used to generate the layout in L-Edit.

A Supertap has the same control signals and output pads as a tapline. For example, the gain-coefficients, the phase-increment values, and the target-extent control signals are different for the individual taplines. The target extent signal is discussed under the control signal section and will be disregarded for now. The gain-coefficients and phase-increment values are important data for the false target generation, as described earlier.

136
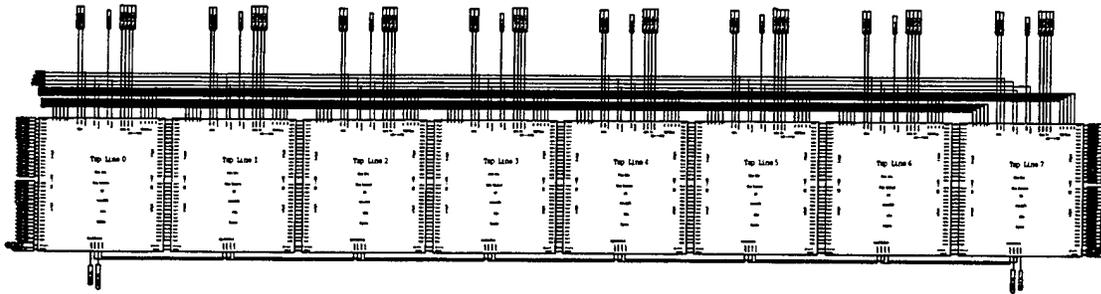
Figure 78. Supertap Schematics

Due to the serial addition in the 16-bit adder for the new architecture (see Figure 74), the same DRFM-phase samples serve as inputs for all taplines simultaneously, as illustrated earlier in this Chapter. The 2*16-bit input pads on the right hand side of Figure 78 are used to cascade another Supertap. Due to this concept of cascading Supertaps, in theory any number of Supertaps could be easily chained together almost without any need for modifying the existing design. The number of Supertaps is therefore limited only to the size of the chip and the current available technology for mask layout. If the desired false-target-extent is larger than the available Supertaps that could fit into an IC, several ICs can be "daisy-chained" together to increase the possibilities for false target generation. For daisy-chaining more than four Supertaps together, the 16-bit adder within a tapline needs to expand by one more bit. Every doubling of the used 32 taplines or four Supertaps requires one more bit for the adder.

6. Architecture Circuit Description in Level 5

Level 5 is the highest level of the current design. As shown in Figure 79, four Supertaps are connected to get an overall number of 32 taplines (8-taplines per Supertap). A 5-to-32-Decoder is used to control the target-extent control signal in the form of a truth table. The required target size can be smaller than the available number of taplines. The

decoder generates control signals to "turn on" the needed number of taplines. To illustrate, a five bit input word corresponds to decimal numbers between 0 and 31. These numbers are directly related to the tapline enumeration as shown in Figure 78. If the generation of a false target requires five taplines, the binary input to the decoder is 00110, which activates tapline zero to tapline four in Supertap A (lower left corner in Figure 79). The schematics of the decoder can be found in the Appendix.
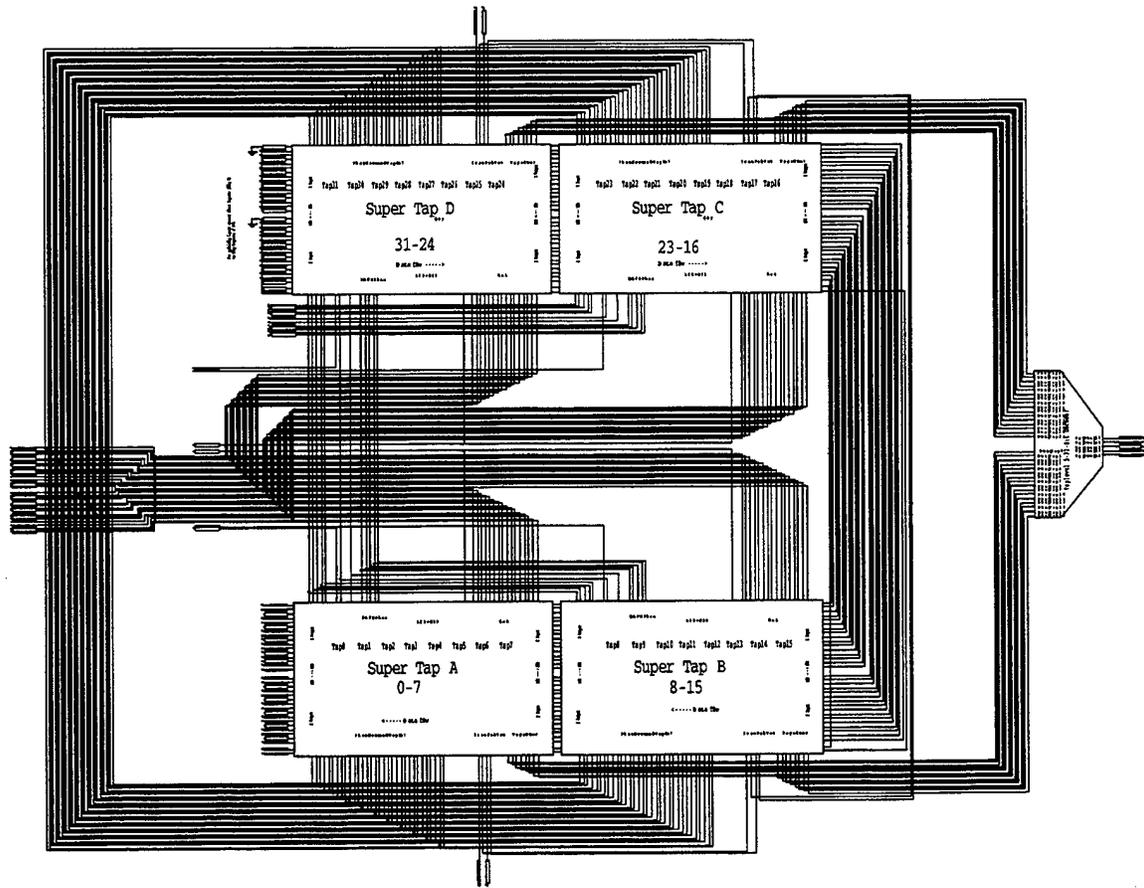
Figure 79. Toplevel Consisting of Four Supertaps

The input and output pads used in Level 5 are summarized as follows:

Inputs:

- Supertap D has two rows of 16 input pads each for I and Q values in order to connect Supertaps in series. For this layout, they are connected to ground to exclude possible side effects based on pending or floating nodes.

- 5-bit input for the 5-to-32-Decoder to control the target size.

- A master clock, which is connected in parallel to clock input pads throughout all five levels.

- Control signals for the scan-path test (SR, SL, S_P_Test_Rin, S_P_Test_Lin).

- 2 * 32 = 64 gain-coefficients (Gain0/Gain1 for each tapline).

- 4 * 32 = 128 phase-increment values (phase_inc0/1/2/3 for each tapline).

- 5-bit DRFM-input data (same input for all tapline).

- Several 1-bit control inputs to control the data flow in the chip (load_phase, delta_Phase_increment, Range_bin_valid, Load_Gain_Reg and overflow_in). These signals are discussed in the next section.

Outputs:

- Final output for I and Q channel in Supertap A representing the data for the false target. This data is imported into Matlab for verification.

- Control signals for the scan-path test (S_P_Test_Rout, S_P_Test_Lout).

- Two 1-bit control outputs to verify the output result of the chip (Overflow_out and Data_Processed_out)). These signals are discussed in the next section.

With a total of two gain-coefficients, four phase-increment inputs per tapline, and 32 taplines for the current design, a total of 192 input pins are required. Adding this many

139

pins to the number of high-speed input and output pins would greatly increase the cost of IC fabrication and the complexity and cost of using the finished IC in a system. Furthermore, if the number of taplines is increased in the future, this problem would become even worse. However, the gain-coefficients and the phase-increment values change only at the beginning of a new radar pulse, not on every sample within a radar pulse. Therefore, the gain-coefficient and phase-increment inputs are of relatively low bandwidth and can be bussed together. To maintain compatibility with off-the-shelf, digital signal processing microprocessors and components, a 32-bit input bus has been designed for the top-level design. The 64 gain-coefficient inputs for 32 taplines (two per tap) are loaded in two bus cycles. The 128 inputs for 32 taplines (four per tap) for the phase-increment are loaded in four bus cycles. Table 23 lists the bus cycles and the control signals and represents an example for how the inputs could be loaded into the IC.

Bus-CLK	Control Signal	Function
1	Load gain Reg Tap 0-15	Loads the gain-coefficients for tapline 0-15
2	Load gain Reg Tap 16-31	Loads the gain-coefficients for tapline 16-31
3	Load Phase Inc Supertap A	Loads the 4-bit phase-increment value into taplines 0-7
4	Load Phase Inc Supertap B	Loads the 4-bit phase-increment value into taplines 8-15
5	Load Phase Inc Supertap C	Loads the 4-bit phase-increment value into taplines 16-23
6	Load Phase Inc Supertap D	Loads the 4-bit phase-increment value into taplines 24-31

Table 23. Loading Example for the Bussed Inputs

The current design can be easily expanded to include more than 32 taplines without a further increase in on-chip hardware related to the bus or the number of I/O pins. To extend the current design, the gain-coefficient inputs and phase-increment inputs from each additional Supertap need to be connected to the bus. Every additional Supertap requires its own loading cycle so that the number of bus cycles will increase.

In Level 5 all inputs and outputs are attached to Pad cells. A Pad consists of a Buf4 and a PadIn or PadOut for an input or output respectively; Figure 80 shows an output pad. The Pads provide the interface between the chip and the outside world. Their primary element is a piece of metal that connects to the pins of the chip via the pad frame. Another important element within a pad is a buffer (Buf4). A Buf4 is a cell that does not perform any logic function but does provide buffering of logic signals (triangular symbol with number four inside in Figure 80). A Buf4 can be driven at high speed by a minimum-sized logic gate. It is capable of sinking and sourcing four times the amount of current that a minimum-sized logic gate can sink or source. Therefore it is very good for driving networks that have a high fan out and have large capacitive loads, such as clock and control signals and is used throughout the design.
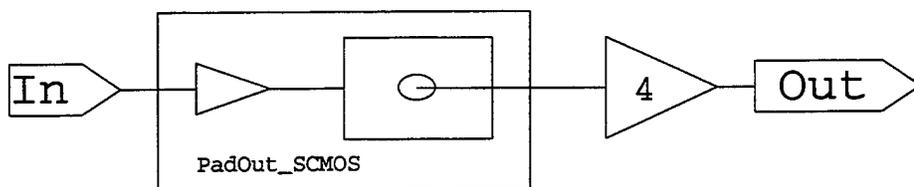
Figure 80. Output Pad

The two types of Pads are distinguishable by their function. PadIn is used to connect signals from outside the IC to the on-chip inputs. It provides a bond pad site for a wire bond, a static-discharge protection circuit, and logic signal buffering to drive high-

141

fanout and high-capacitate on-chip networks. PadOut is used to connect the IC outputs to the off-chip networks. It provides a high-power driver circuit, a static discharge protection circuit, and a bond pad site for the wire bond.

VIII. ASIC DESIGN: TIMING & CONTROL

This chapter provides a detailed description of inputs controlling the IC and the signal-flow. Furthermore, timing control diagrams will illustrate the signal flow within the IC and demonstrate the use of the control signals to achieve a correct data output. Moreover, the concept of *scan-path testing* is discussed and illustrated as an example for the tapline implementation.

A. CONTROL SIGNALS

At this point it should be mentioned that the master clock exclusively controls all control signals. Due to this setup, the data processing can be controlled precisely in the data pipeline, where a pipeline is the data flow within a tapline.

The Scan-path test consists of several inputs and outputs and will be discussed in more detail later. The associated ports are: Scan-path test Left Out (S_P_Test_Lout), Scan-path test Right Out (S_P_Test_Rout), Scan-path test Left In (S_P_Test_Lin), Scan-path test Right In (S_P_Test_Rin), Shift Right Data In (SRDi), Shift Left Data In (SLDi), Shift Right (SR), and Shift Left (SL). They are mentioned here briefly for introduction purposes because they are used in the following description.

1. Clock

The clock (CLK) is the most important control signal throughout the IC. Every D-Flip-Flop used passes its input data to the output when the clock signal is high. This concept is called "positive edge" clocking. Since the clock driven register cell is the basic element of a n-bit register, the entire data flow is clock controlled. With the clock

143

changing from low to high, the data transports one step further down in the tapline pipeline and allows total control over the internal control signals and the data processing.

2. Load

Load (LD) is a signal to control the behavior of the registers. Due to the structure of the register cell, the signals Shift Right (SR), Shift Left (SL) and Hold (HLD) have to be low when LD is high. Otherwise the register cell is in an undefined state and will produce erroneous results. Load is the mode for normal operation. If LD is high, the data can penetrate the pipeline controlled by the clock. If LD is low, the chip is principally in a special mode, where the other control signals, such as, Hold or Shift Right can be used.

3. Hold

Hold (HD) is one of the register cell signals that can be exclusively high for a certain performance within a register. If HD is high, SR, SL and LD have to be low. Hold is used to store or hold a value within a register that should not change over the clock period. The input data bits to a register with HLD high are simply ignored and the last data within the register are retained. This concept is used to achieve buffering for the gain-coefficients and the phase-increment data. Since these data bits are used within a period of a radar pulse, they are stored in registers and made available for subsequent processing using the clocked DRFM-phase data.

Hold, Load, Shift Right and Shift Left are the key control signals for every operation. As mentioned earlier only one of these signals is permitted to be high within a clock cycle, or the IC will be in an undefined state. However, if all control signals are low at the same time, the register performs a special function, the synchronous clear. A synchronous clear forces the register to reset and sets all output bits to low (zero). This

function is used as initialization before data is loaded into the IC and for the two 11-bit registers leading its output into the 16-bit adder. The 11-bit register's LD signal is connected to a two-input AND gate. The gate inputs are connected to control signals in such a way that they signal if data is present or if data is not present at the gain shifter. If data is present, the register will perform a normal load. If no data is present, it will perform a synchronous clear and zero the output. The requirement for this operation is due to the way the data is summed in the 16-bit adder. The DRFM-phase data penetrate all taplines simultaneously. Therefore tapline 2 to 32 will still have valid data in the adder chain, where as tapline 1 is already finished with its data processing. The rest of the data is clocked through the adder chain, consisting of the 16-bit adders within the single taplines. No data is allowed to influence the continuing data transport at this stage. Therefore it is necessary for the idle tapline not to have any undefined data present at its 16-bit adders. The synchronous clear function will guarantee that this input (A input row of the 16-bit adder) is zero.

4. Load Phase Increment

Load Phase Increment (LD Phase Inc) is a control bit that affects the two registers in the phase-increment block in order to signal a change for the phase-increment value. The signal performs the same operation for both described taplines. If LD Phase Inc and LD are high, a new phase value gets loaded into the first phase-increment register. For the rotation phase tapline, the second register performs a synchronous clear to reset the phase-rotation, whereas the second register in the double-buffered tapline design remains unchanged. If LD Phase Inc is low and LD is high, the Phase Inc Reg in both designs is

in a hold mode, in order to keep the phase-increment value constant over the duration of a radar pulse.

The gain-coefficients and the phase-increment values are transported on a 32-bit bus. There are four phase-increment bits per tapline and 32 bits for a Supertap, therefore the chip needs to load a total of 128 phase-increment values to be able to process DRFM-phase data. Since the bus has a length of 32-bits, four bus cycles are needed to load the gain-coefficients, controlled by the signals "Load Phase inc Supertap A-D." These signals are equivalent to the controls to LD Phase Inc on the top level of the chip. They perform the same operation and trigger the tapline controls.

5. Delta Phase Increment

Delta Phase Increment (Delta Phase Inc) is a control signal for the Phase-Rotation Register (Phase Rot Reg) and is used only in the tapline with on-chip phase-rotation instead of double-buffered coefficients. Since a CHIRP pulse or radar pulse is divided into samples and the phase should only rotate once for every pulse, the phase-rotation value has to be incremented between pulses. If Delta Phase Inc and LD are high, the phase-increment can "rotate" under control by the clock. The resulting value will be added to the DRFM-input data. If Delta Phase Inc and LD Phase Inc are low, the Phase Rot Reg is in a hold mode and the phase-increment value (input for the Phase Accumulator) is fixed. Before processing a new Radar pulse, the phase has to rotate once to produce the new phase value.

6. Use Phase Increment

Use Phase Increment (Use Phase Inc) is the substitute for Delta Phase Inc in the double-buffered version of the tapline. In this case the phase-increment data has to pass

146

through two registers in order to arrive at the phase accumulator. With Use Phase Inc and LD high, the data can flow from the buffer register into the phase-increment register. Furthermore the same control signal propagates through two more control registers to adjust the loading of the gain-coefficients at the proper time, since gain and phase usually change collectively. Thus the controller is free from initiating the gain change to correspond to the phase change.

7. Load Gain Register

Load Gain Register (LD Gain Reg) affects the behavior of the gain register (the buffer for the double-buffered tapline) in order to signal a change of gain data within a tapline. To load new gain data, LD Gain Reg and LD must be high at the same time. If LD Gain Register is low and LD is high, the gain register is in a hold mode.

The gain-coefficients and the phase-increment values are bused on a 32-bit bus. There are two gain-coefficients per tapline and 16 coefficients per Supertap. Therefore the chip needs to load 64 gain-coefficients values to be able to process DRFM-phase data. Since the bus has a length of 32-bits, two bus cycles are needed to load the gain-coefficients controlled by "Load Gain SupTap AB" and "Load Gain SupTap CD." Load Gain SupTap AB/CD are the equivalent controls on the top level of the chip. They perform the same operation and trigger the tapline controls.

8. Target Extent

The *Target Extent* (Tgt Extent) control is used to activate or deactivate taplines in accordance with the appropriate size of the false target. The current design is able to handle a false target up to 32 cells corresponding to the in Matlab constructed Range-Doppler map. For a small false target, less taplines are needed to create the target. In

design level 5 a 5-to-32-bit decoder uses a truth table to adjust the required taplines in dependency of the target size. If the target generation requires, for example only 12 taplines, the Tgt Extent for the first 12 taplines is high. The Tgt Extent for the other taplines is low and the output values are ignored.

9. Range Bin Valid

A tapline needs four clock cycles to produce a valid output. *Range bin valid* goes high when new DRFM-phase data are presented to the input of a tapline. The bit penetrates through 1-bit register cells to the Data Processed Out pad. If Data Processed Out goes high, the output from the tapline is fully processed and the output is valid. As long as Data Processed Out is low, the clocked output must be ignored.

Besides the normal function, there is an interaction between two controls at this point. The Range bin valid control string feeds the two input AND gate for the Pipeline Register 3 as mentioned in VIII.A.3. If the Range bin valid is low, the register after the gain-shift block gets cleared with every clock cycle. Recall, that a higher tapline can produce valid results, even if a lower one cannot. Consequently the lower tapline is not allowed to add undefined data to the valid output of a higher tapline and must be cleared.

10. Valid Result In

Valid Result In performs a similar operation as Range bin Valid. It connects to the Valid Result Out port of the next higher tapline. If the next higher tapline produces a valid output that leads into the lower tapline, Valid Result In is high and the next lower tapline produces a valid output with the following clock, although it may not produce any valid data within its own gain-shift block.

11. Overflow In/Out

Overflow In is an error-checking signal from the next higher tapline. If a higher tapline produces an invalid output due to an overflow in the 16-Bit-Adder, the entire chip output will become invalid. Overflow Out is the pipelined output to flag data produced by a 16-bit adder overflow.

B. TIMING CONTROL

The clock controls the normal mode of operation. Several control signals have to interact in order to ensure correct DRFM-input data treatment. In other words, the operator needs to know the timing relationship for functions, such as bus loading, DRFM-phase data input, and data read out. The best method to demonstrate the complicated timing control is with an example. The example in Figure 81 shows the timing diagram for the initial loading phase and Figure 82 shows the timing constraints in terms of clocks for the initial loading phase and the time between two radar pulses. In the diagrams, the clock is set to 5nsec low and 5nsec high so that one clock cycle is 10nsec. Moreover, for illustration purposes, multiple input and output bits are collapsed into a single bit.

1. Initial Loading Phase

Before data processing can begin, the IC should be initialized, clock 0-10nsec, as shown in Figure 81. This will clear all registers with a synchronous clear and set the control bits to a defined state. The next six clock cycles are reserved to load the gain-coefficients and the phase-increment values through the 32-bit bus. From these six clock

cycles the first four are required to load the phase-increments for all four Supertaps. This involves interacting with the controls and is summarized as follows:

1. Supertap A loads its phase-increment data during the first of the six clocks. For this purpose the corresponding data are presented to the 32-bit input bus and the control "LD Phase SuptapA" is high for this particular clock.

2. Supertap B loads its phase-increment data during the second of the six clocks. For this purpose the corresponding data are presented to the 32-bit input bus and the control "LD Phase SuptapB" is high for this particular clock.

3. Supertap C loads its phase-increment data during the third of the six clocks. For this purpose the corresponding data are presented to the 32-bit input bus and the control "LD Phase SuptapC" is high for this particular clock.

4. Supertap D loads its phase-increment data during the fourth of the six clocks. For this purpose the corresponding data are presented to the 32-bit input bus and the control "LD Phase SuptapD" is high for this particular clock.

The following last two clock cycles are used to load the gain-coefficients for the Supertaps and move the buffered phase-increment data into the next register:

5. Supertap A and Supertap B loads its gain-coefficients during the fifth of the six clocks. For this purpose the bus pads "Bus0" to "Bus15" are the inputs for Supertap A and "Bus16" to "Bus32" are the inputs for Supertap B. Additionally, the controls "LD_Gain_SupTap_AB" and "Use Phase Inc" are required to be high for this particular clock, where "use Phase Inc" moves the buffered data into the registers for data treatment. Due to the delay

propagation of the Phase Inc control, the buffered gain-coefficients move after two more clock cycles into the register for data treatment.

6. Supertap C and Supertap D loads its gain-coefficients during the sixth of the six clocks. For this purpose the bus pads "Bus0" to "Bus15" are the inputs for Supertap C and "Bus16" to "Bus32" are the inputs for Supertap D. Additionally, the control "LD_Gain_SupTap_CD" is required to be high for this particular clock.

Thus, seven clock cycles are needed before the first DRFM-phase data can be read into the taplines. Due to the pipelined structure of a tapline, the data treatment demands four more clock cycles before the first valid output is present at the first tapline (clock 70-100nsec). During these four clocks, the phase-increment and the gain are applied to the DRFM-phase data and the resulting data adds up with the output data from other taplines in the 16-bit adder. In summary, eleven clock cycles are required before the first valid output is observable at the output. The "Data Processed Out" control signal is an indicator for valid results. As long as this control signal has a high output, the corresponding output for the I and Q channel are valid and can be used for false target generation. After initialization the DRFM-phase data for the first radar pulse can be processed within the taplines. The time between two radar pulses requires some attention again and is discussed in the next sub-section. The initialization for the second Radar pulse is the same, as it concerns the loading phase with phase-increment data and gain-coefficients. Nevertheless the register initialization with a synchronous clear is not required.

151

During the time of loading and data processing, LD is high except for the synchronous clear at the very beginning. All other register control signals like the scan-path controls and HLD are low, since this is the defined state for normal operation. Moreover, the target extent as defined through level 5's 5-to-32-bit decoder activates the taplines. The example in Figure 81 assumes all taplines active and shows the corresponding decoder inputs (Tgt Extent In 0-4) as high.
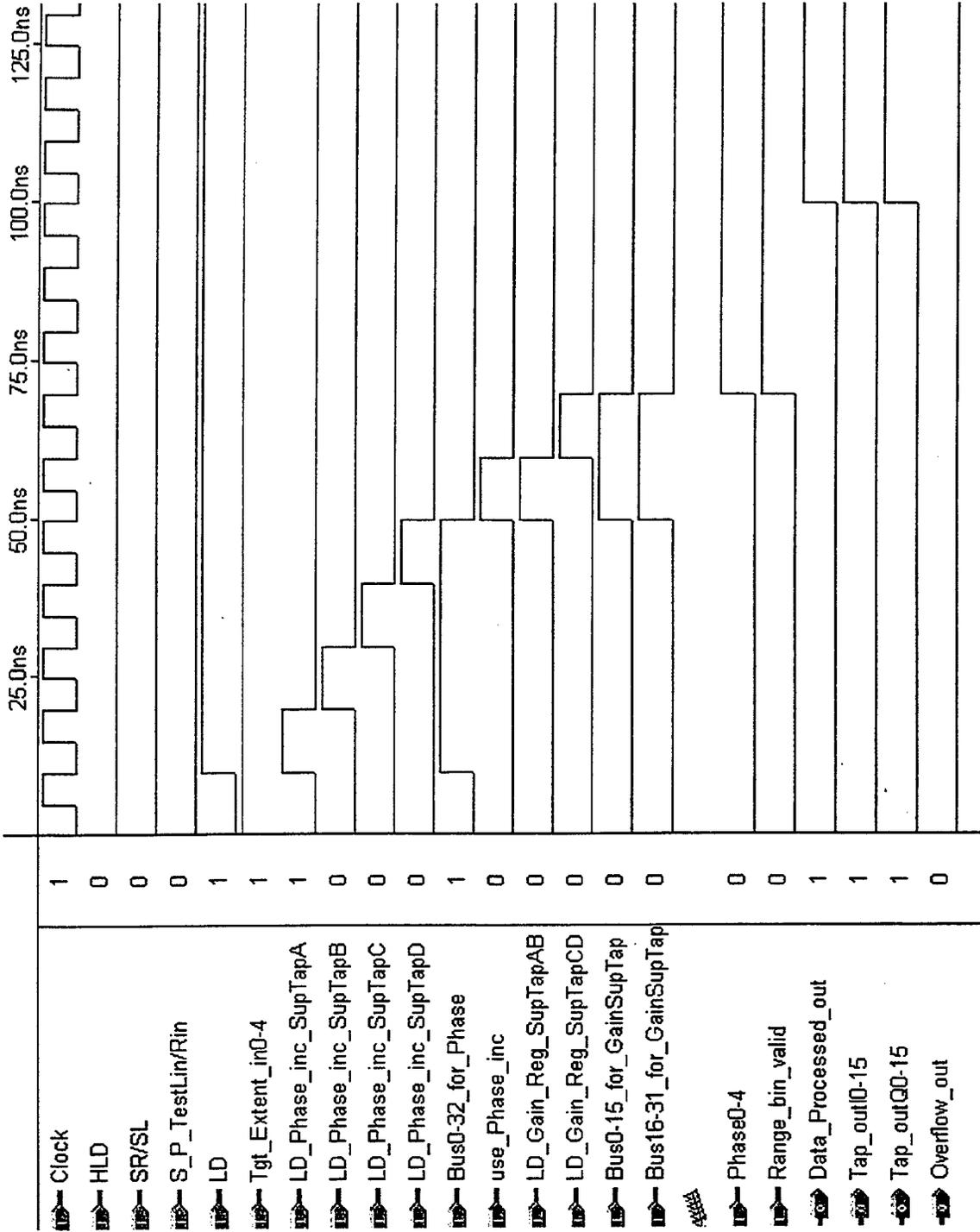
Figure 81. Timing Diagram for the Initial Loading Phase

153

2. Timing between Radar Pulses

After processing all DRFM inputs for one Radar pulse, there are still valid results in the adder chain propagating in direction of the first tapline in the row. The time between two radar pulses is much greater than the time between the samples. Therefore it is desired to read out the rest of the data. For example, assume 62 DRFM-phase samples for one radar pulse and a requirement for 32 taplines, where the 62nd DRFM input is processed parallel in all taplines and now reside in the corresponding 16-bit registers. The 16-bit register of the first tapline is the chip output. Therefore its output is already read out, but 31 outputs need to be clocked through the chain. Consequently, 31 clock cycles are required to read the rest of the remaining data. Figure 82 shows that the DRFM input is low during this time and does not contain any data for processing. As mentioned earlier, the 11-bit register within a tapline not containing any valid data needs to be cleared since it leads into the 16-bit adder. Therefore the control Range-bin valid needs to change from high to low after the last DRFM-phase data gets loaded into the taplines. In view of the fact that the taplines are basically idle during the read out of the processed data, the time is used to load the phase-increment data and gain-coefficients for the next radar pulse.
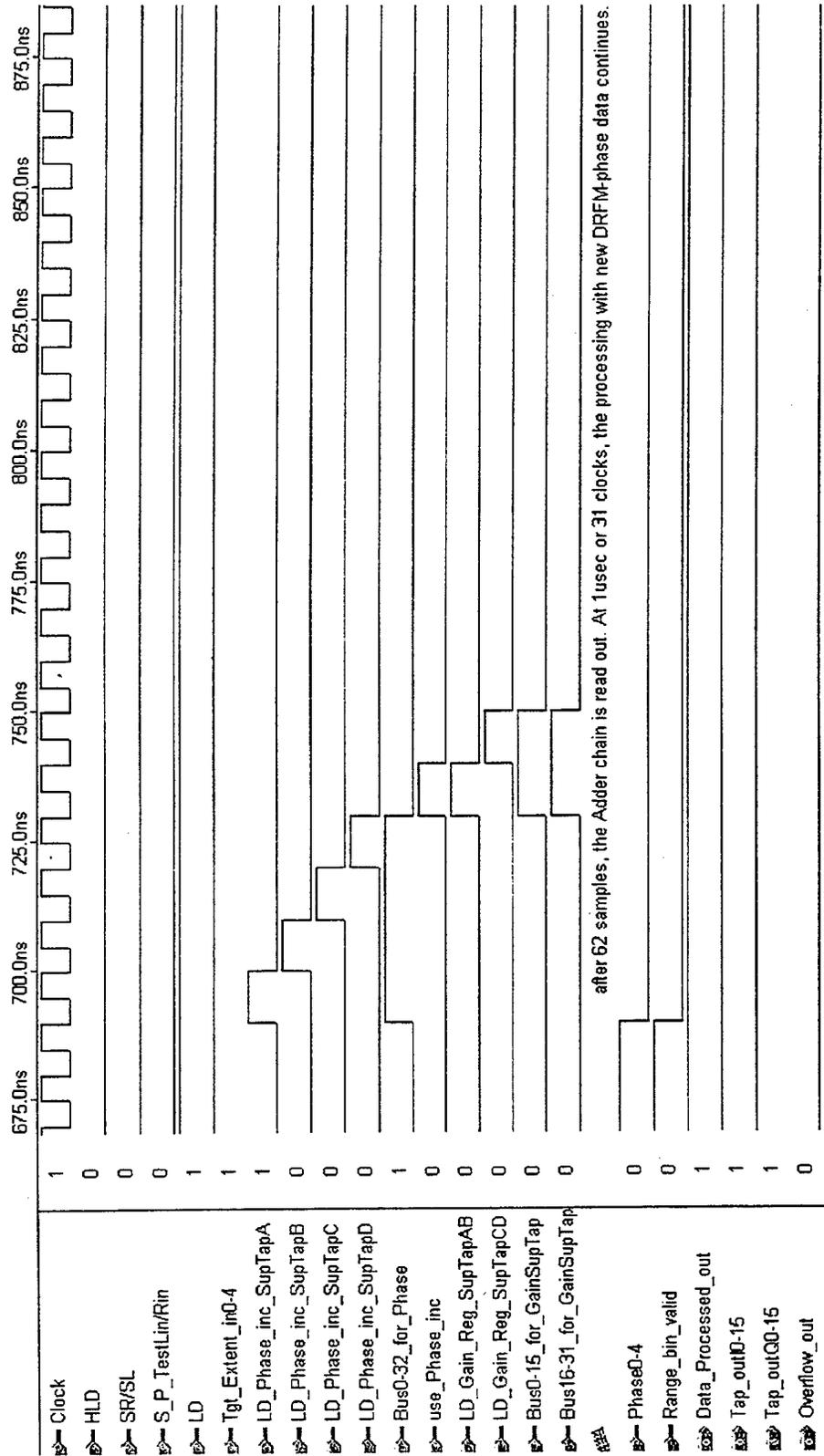
Figure 82. Timing Diagram between Two Radar Pulses

155

C. SCAN-PATH TESTING

The design of a workable system solution for a given problem is only half of the work. Furthermore one must also be able to test the system to a degree, where it can be ensured that the system is fully functional with a high confidence level. In very small-scale digital systems, tests can be performed exhaustively, where the system is exercised over its full range of operating conditions. This method is not an economical or useful approach to verify the functionality. Therefore other strategies are necessary to perform proper testing. The scan-path methodology is probably the most widely used technique for testing those parts of a integrated circuit that are constructed of clocked flip-flops interconnected by combinational logic. As illustrated in Figure 83, the scan path can be implemented into a simple circuit very easily. When the circuit is put into test mode, one can shift an arbitrary test pattern into the register. By returning the circuit to normal mode for one clock period, the contents of the scan register and primary input signals act as inputs to the attached combinational circuitry and new values are stored in the register. If the circuit is then placed into test mode again, the controller can shift out the contents of the scan register for comparison with the correct response.

By using test points, one can easily enhance the absorbability and controllability of a circuit. The scan-path register effectively provides such test points, whereas in FPGA design the implementation of Tristate-buffers is necessary. To control the test points in a scan-path test several control signals are implemented to adjust the mode of operation. Table 24 lists the signals used for the scan-path test in the new DIS design.
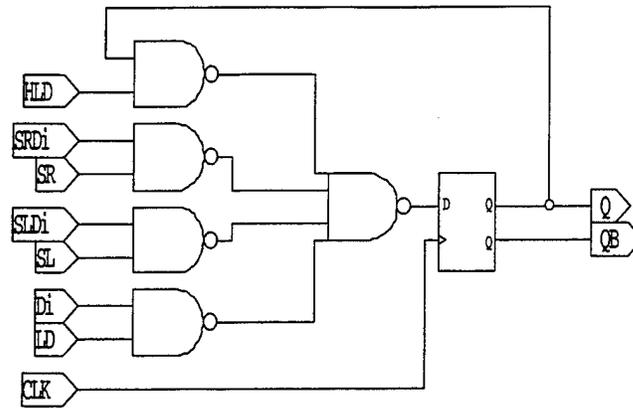
Figure 83. Register Cell

Padname	Function
SR (Shift Right)	Input pad to control function of register. If high, the data within the register will be shifted to the right with every clock cycle. All other control signals have to be low (HLD, LD, SL)
SL (Shift Left)	Input pad to control function of register. If high, the data within the register will be shifted to the left with every clock cycle. All other control signals have to be low (HLD, LD, SR)
SRDi (Shift Right Data in)	Test data input pad from right front end of scan-path test
SLDi (Shift Left Data in)	Test data input pad from left front end of scan-path test
SRDo (Shift Right Data out)	Test data output pad for a right shifted output
SLDo (Shift Left Data out)	Test data output pad for a left shifted output

Table 24.   Scan-Path Test Control Signals

157

A scan path register is a serial cascade of scan path register cells whose inputs and outputs are connected to the internal logic of a chip as illustrated in Figure 84. During normal operation, the LD signal is asserted and the logic value at the inputs D0 and D1 reach the outputs Q0 and Q1 after one clock cycle. When on the other hand the SL is asserted during test mode, the logic value at SLDi arrives at Q1 one clock cycle later and continues propagating to Q0 with the following clock cycle. When SR is asserted during a test, the logic value at SRDi arrives at Q0 one clock cycle later and continues propagating to Q1 with the following clock cycle. If the chip is still in test mode, the values keep propagating in the forced direction through all the connected registers in the scan path.
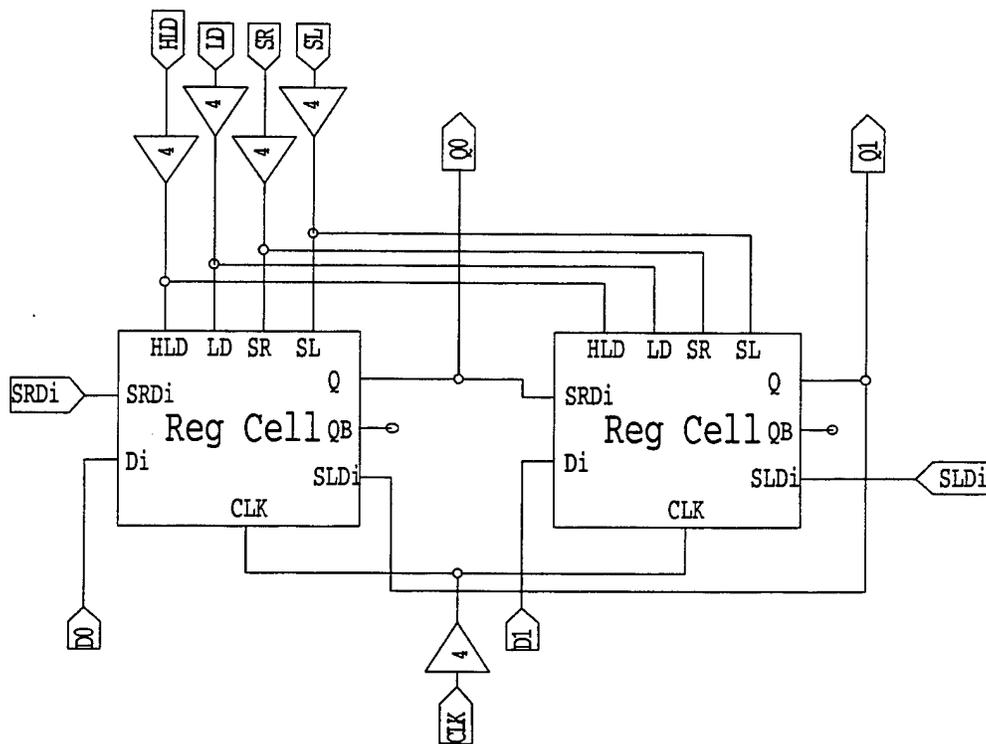
Figure 84. Schematics of a 2-Bit Register

In summary, the scan-path test can be used for two valuable testing functions. First, a certain test setup can be tested, where test data is placed in every register of the chip. After loading the test data via the scan path, the chip is put into normal operation mode and the resulting outputs can be observed at the output for examination. Second, after normal operation all stored data in the registers can be read out by using the scan-path shift option to move the register data to the left or to the right. The results can be examined by comparing them with calculated values. The scan-path test implementation for a tapline with phase-rotation on-chip is shown in Figure 85. The path between the taplines within a Supertap and beyond is simply realized by a serial connection of inputs and outputs. The scan chain from the toplevel point of view connects tapline 0 to 31 in a long row of registers. To give an overview about the number of bits penetrating through the scan chain, imagine the following calculation: 90 bits are used in the registers of one tapline, where 32 taplines are implemented in the chip. This will result in 90 * 32 = 2,880 values to read out for the complete scan path. A double-buffered tapline has 94 register bits. Therefore the length of the scan path on the toplevel is even higher with 3,040 values.
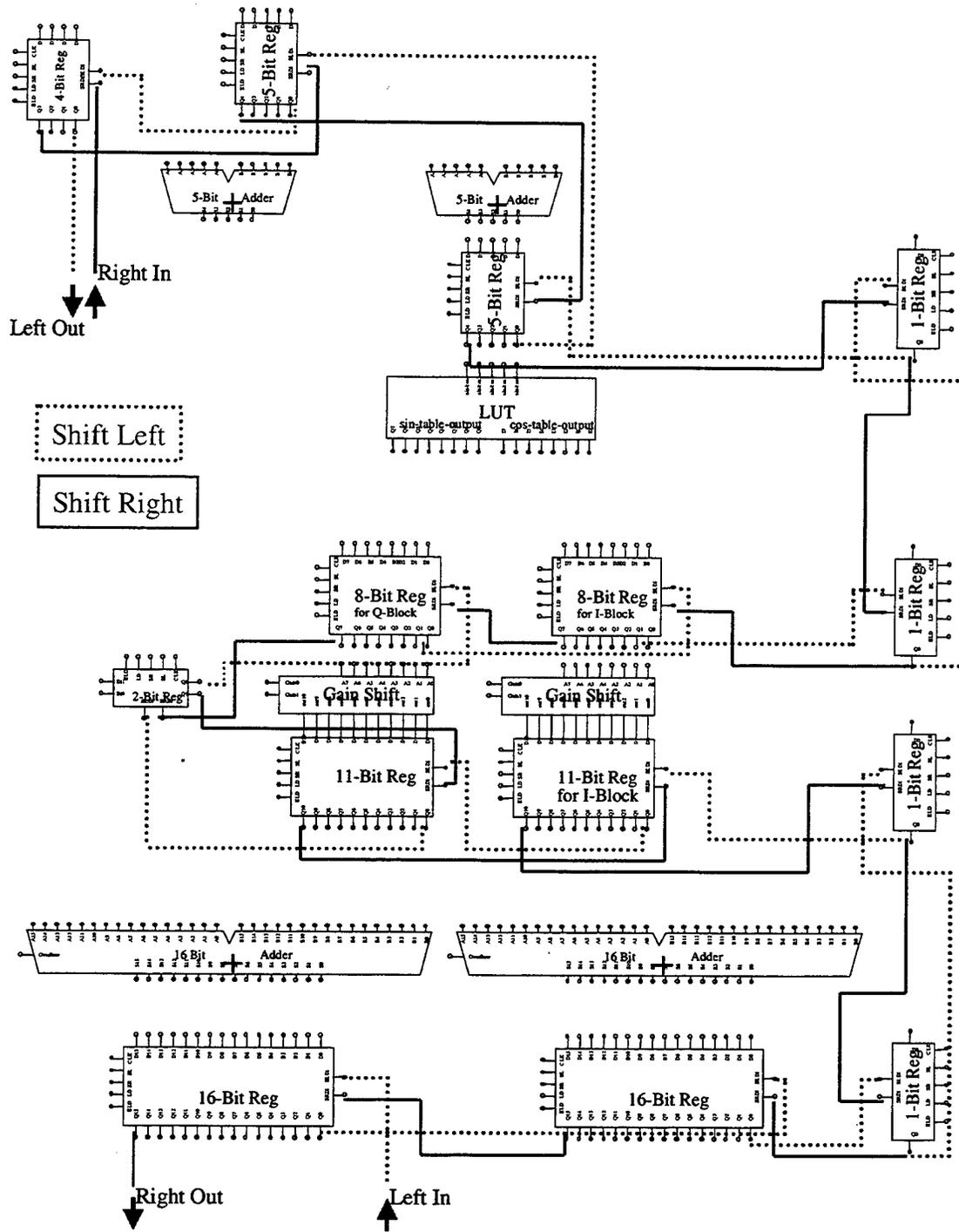
Figure 85. Scan Path in a Tapline with Phase-Rotation On-Board

160

IX. ASIC DESIGN: SIMULATION

This chapter is dedicated to circuit verification with simulation using the circuit simulator T-Spice. It provides useful information about simulation parameters and semi-digital simulation. Two examples are used to illustrate the testing concept. The first example is the two-tapline test case, where the regular transistor model is used to perform the simulation. The second example is the 32-tapline chip where a switch model replaces the transistor model to reduce simulation time and complexity.

A. T-SPICE SIMULATIONS

Two goals are established by doing the simulation in T-Spice. First and foremost, the correct logical implementation needs to be verified, which includes the check of each connection between elements (wire connections). The second goal is to prove the proper implementation of the developed algorithm within the circuit. This section describes how the simulation is done in T-Spice. Simulating a smaller part of the entire circuit design and comparing the results to the Matlab simulation achieves the verification of the circuit functionality.

S-Edit supports a direct export of a schematic layout into a T-Spice readable SPICE format. The exported SPICE file contains only circuit information, but does not contain test-commands or test-vectors. Therefore several lines of code have to be added to create a valid simulation file that can be used in T-Spice. To illustrate the test concept in T-Spice a 2-bit register is used as example. Table 25 contains parts of the 2-bit-register SPICE file that are used for simulation.

T-Spice is not a logical circuit simulator, but can perform various analog simulations like DC-analysis and frequency sweeps. Nevertheless, T-Spice can make use of the "bit" command to push binary inputs into the input pads of the circuit representation. The voltages are 0V for a logical zero and 5V for a logical one. By defining the inputs as voltage sources, T-Spice analyses the input vectors, calculates a DC operating point, and calculates the defined output pads in form of voltages.

T-Spice Code	Meaning
Vdd Vdd Gnd DC 5	Defines the voltages between 0V (Ground) and +5V DC
.include "D:\Chris\Thesis\schematics\testfiles\ Register\2Bit\input_table2Reg.md"	Reads the file input_table2Reg.md, which is an text file containing all input used during simulation
.options prtdel=80n	The option command customizes the simulation. PRTDEL sets the reading for output pads to exact every 80nsec
.tran 10n 800n start=70n	Performs a transient analysis with a maximum step size for calculations of 10nsec, a simulation stop time of 800nsec and an offset for the first output reading of 70nesc
.print tran "D:\Chris\Thesis\Schematics\testfiles\ Register\2Bit\Inputs.out" V(CLK) V(SL) V(SR) V(SLDi) V(SRDi) V(LD) V(HLD) V(D0) V(D1) .print tran "D:\Chris\Thesis\Schematics\testfiles\ Register\2Bit\Outputs.out" V(Q0) V(Q1)	The print tran command is used to define the monitored output pads and the file in which the records are saved. The file "inputs.out" records all control signals and the inputs of the register, whereas the file "Outputs.out records only the outputs of the register.
.param l=0.05u	Specifies the wavelength as 0.05μm
.include "D:\Chris\Thesis\ModelParammod.md"	Includes the transistor parameters for the target process (MOSIS – HP 0.5μm) used for the simulation.

Table 25. T-Spice Simulation Commands

163

Below is an example of the input vectors for the 2-bit-Register simulation.

VinD0 D0 Gnd bit   ({0010111111} on=5.0 off=0.0 pw=80n rt=0.1n ft=0.1n

VinD1 D1 Gnd bit   ({0001011111} on=5.0 off=0.0 pw=80n rt=0.1n ft=0.1n

VinSRDi SRDi Gnd bit   ({0000000000} on=5.0 off=0.0 pw=80n rt=0.1n ft=0.1n

VinSLDi SLDi Gnd bit   ({0000000011} on=5.0 off=0.0 pw=80n rt=0.1n ft=0.1n

VinHLD HLD Gnd bit   ({0000100000} on=5.0 off=0.0 pw=80n rt=0.1n ft=0.1n

VinLD LD Gnd bit   ({0111010000} on=5.0 off=0.0 pw=80n rt=0.1n ft=0.1n

VinSR SR Gnd bit   ({0000001100} on=5.0 off=0.0 pw=80n rt=0.1n ft=0.1n

VinSL SL Gnd bit   ({0000000011} on=5.0 off=0.0 pw=80n rt=0.1n ft=0.1n

VinCLK CLK Gnd bit   ({01} on=5.0 off=0.0 pw=40n rt=0.1n ft=0.1n

The input vectors are defined by name for the voltage source (input pad) against ground, bit pattern used as inputs for the voltages sources, the definition of zero and one, pulse width (pw) of the signal, rise time (rt), and fall time (ft) in nano seconds. The registers are constructed for using a positive edge triggered clock. This means that all signals have to be changed and stable before the clock switches from low to high. A change of a value after the clock goes high cannot be processed properly. As an illustrated example in Figure 86, the clock starts low (zero Volts) for a time of 40nsec and switches to high (five Volts) afterwards. Consequently, the entire clock cycle is 80nsec, which corresponds to the pulse width of the input signals.

The above-mentioned input values are used to test the behavior of the 2-bit-Register under normal and test-mode conditions. Table 26 illustrates the basic test concept and the relation between the control signals in T-Spice.
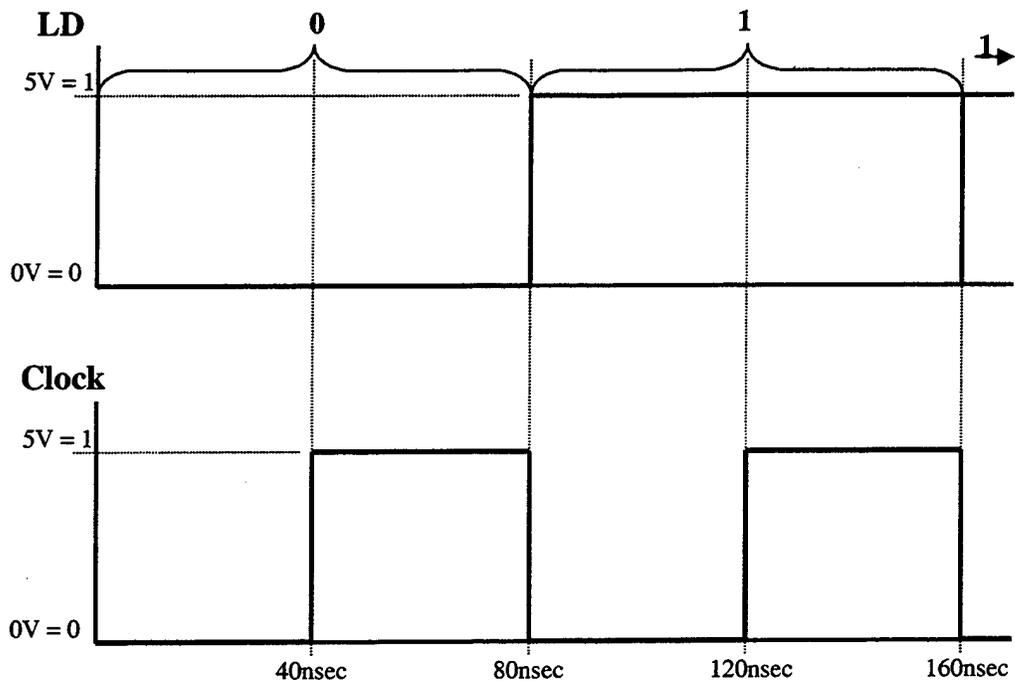
Figure 86. Positive Edge Triggered Clock

D0	SRDi	SLDi	HLD	LD	SR	SL	CLK	Q0	Remark
0/1			0	0	0	0	0→1	0	Synchronous Clear
0/1			0	1	0	0	0→1	0/1	Data (D0) – normal operation
0/1			1	0	0	0	0→1	0/1	Previous data, "do nothing"
	0/1		0	0	1	0	0→1	0/1	Right data (SRDi)
		0/1	0	0	0	1	0→1	0/1	Left Data (SLDi)

Table 26. Test Concept of a 2-Bit-Register

The test mode signals are in direct relationship to each other because only one input of

SR, SL, HLD, and LD can be high at the same time to perform a legal operation in test

mode.

During the transient analysis, T-Spice uses the input vectors to determine the voltage values for the outputs. The determined values are saved in the predefined files and the transient analysis results are automatically stored in a separate file. These transient results can be used to probe at circuit nodes of the schematic layout in S-Edit. Probing calls W-Edit automatically and creates a graphical output of the voltage versus time for the probed node. The user defined output files, which contains the simulation results, hold exact voltage values in the region between 0V and 5V, as shown below in Table 27. The analog output values distinguish between one and zero. The output values are analog voltages and the results must be sent through a hard limiter to get a binary output table in order to compare the results with the correct binary output pattern produced by Matlab.

Time (sec)	V (Q0) (Volts)	V (Q1) (Volts)
7.0000E-08	1.2408E-07	1.2414E-07
1.5000E-07	1.4298E-07	1.4302E-07
2.3000E-07	5.0000E+00	6.3223E-08
3.1000E-07	8.6684E-08	5.0000E+00
3.9000E-07	1.9326E-07	5.0000E+00
4.7000E-07	5.0000E+00	5.0000E+00
5.5000E-07	2.8934E-07	5.0000E+00
6.3000E-07	1.2812E-07	-2.4128E-07
7.1000E-07	4.3775E-08	5.0000E+00
7.9000E-07	5.0000E+00	5.0000E+00
8.0000E-07	5.0000E+00	5.0000E+00

Table 27. Output Table for the Transient Analysis of a 2-Bit-Register

In very small-scale digital systems, tests can be performed exhaustively, where the system is exercised over its full range of operating conditions. This kind of test was used to simulate the elements of Level 1 and 2 of the design hierarchy, as exercised for the 2-bit-Register test case. However, for higher-level elements the input values were chosen more carefully to simulate only critical cases. By increasing the number of sub-circuits, the simulation time and the amount of required computing power increases in an almost exponential manner. For a Supertap simulation (level 4), a PC with Pentium III processor and 768MB RAM could not satisfy the need for resources by the T-Spice simulation.

B. 2-TAPLINE SIMULATION

Due to the limitations in available computer resources, we decided to prove the algorithm with a 2-tapline circuit using a transistor model. The setup is shown in Figure 87. The Matlab programs discussed earlier produces a set of input DRFM data for 10 radar pulses with 15 samples each. After simulation, the "hard_limiter2Taps.m" Matlab program converts the results into a binary form. Then, the data is translated into decimals and plotted in Matlab. Matlab also performs the same simulation so that the outputs of both simulations are comparable and so the T-Spice simulation can be verified. The following figures and tables give an overview about the setup and the obtained results.

This particular simulation needs a special setup in the Matlab test environment. The range-Doppler-amplitude map entry program is modified for the 2-tapline test case so that only 10 radar pulses with 15 samples per pulse (150 DRFM data) are used to decrease the simulation time in T-Spice. Since two taplines are used, only two cells in the

167

range-Doppler map are defined for the false target generation, as shown in Figure 88. The setup for the amplitude (gain-coefficients) and the Doppler Shift (phase-increment data) are summarized in Table 28.
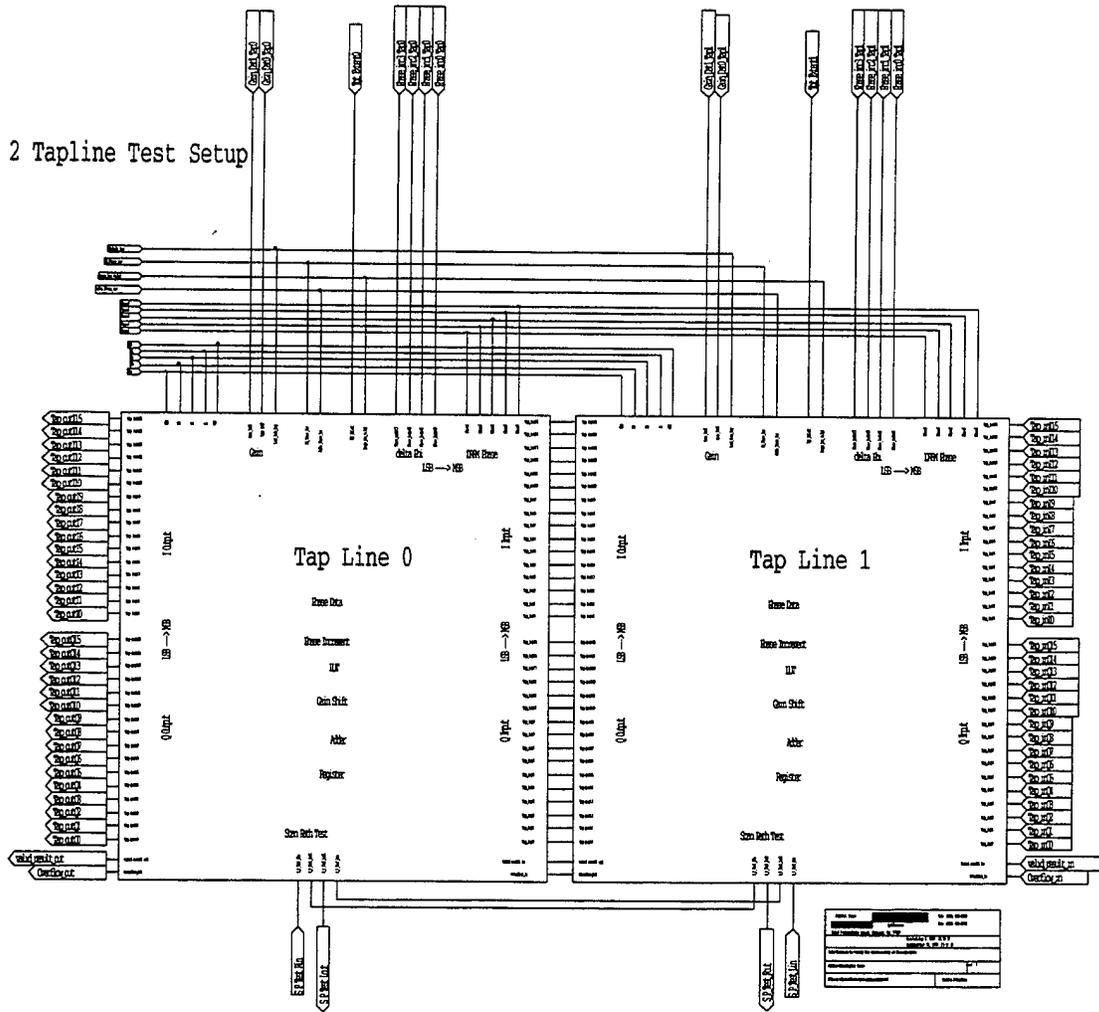
Figure 87. 2-Tapline Test Case

Target Cell	Range Cell	Doppler Cell	Amplitude	Doppler Shift	Remark
1	1	1	2	2	Tap 0 – 1st Tap
2	2	1	3	4	Tap 1 – 2nd Tap

Table 28. Matlab Inputs into the Range-Doppler Map

168
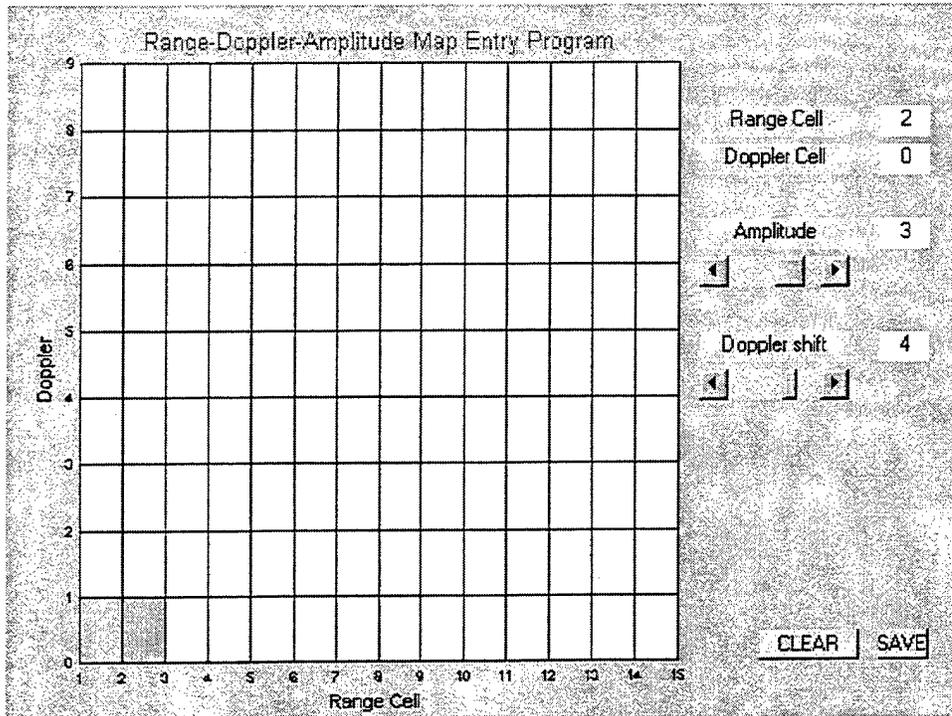
Figure 88. Modified Range-Amplitude Entry Map

The rest of the Matlab simulation follows the same path as described in Chapter 4. The values in Table 28 can be translated into T-Spice test vectors, which specify the input parameters for the gain-coefficients and the phase-increment values to the taplines. The corresponding values are listed in Table 29.

Tapline	Range Cell	Doppler Cell	Gain Coefficients	Phase Increment	Remark
0	-	-	01	0010	Binary Inputs
1	-	-	10	0100	Binary Inputs

Table 29. T-Spice Inputs for Gain and Phase-Increment

Table 30 shows only a small part of the input vectors used for this test case, but it explains the interaction between input control signals, input vectors and the output

169

values. Note that Phase 0-4 represents the 5-bit DRFM-phase data input to both taplines. Also, the first input performs a synchronous clear to zero all registers. A radar pulse consists of 15 samples, where sample by sample is read in controlled by the clock. Note also that the gap between the radar pulses is manually set and not part of the DRFM-phase data. This gap is required to read out the processed value in tapline 2 that is still in the adder chain. Therefore one clock cycle between the radar pulses is required to read out the last final output. Setting range-bin valid to low between the pulses clears the 11-bit register in the taplines to ensure that no undefined data gets added to the last valid output. The delay of one output produces the 16th output value for only 15 input samples. After the last processed phase sample in tapline 1 reaches the final output through tapline 0, the first fully-processed sample from the next radar pulse is already present for output.

Input Pad Name	Sync CLR	Radar Pulse 1		Radar Pulse 2		Radar Pulse 3
Phase0	0	001101010000110	0	001101010000110	0	001101010000110
Phase1	0	000100000011111	0	000100000011111	0	000100000011111
Phase2	0	001110100100110	0	001110100100110	0	001110100100110
Phase3	0	000101010011111	0	000101010011111	0	000101010011111
Phase4	0	000001110100010	0	000001110100010	0	000001110100010
Delta Phase inc	0	000000000000000	1	000000000000000	1	000000000000000
LD Phase inc	0	100000000000000	0	000000000000000	0	000000000000000
LD Gain	0	100000000000000	0	100000000000000	0	100000000000000
Range-bin valid		111111111111111	0	111111111111111	0	111111111111111

Table 30. Input Data for the Three Radar Pulses as used in the 2-Tapline Test

Table 31 shows only the first 22 clock cycles out of 171. The I and Q values are listed in form of most significant bit to least significant bit. The first five clock cycles are needed to process the first input, where the first output is a synchronous clear. Bus cycles are not required for a 2-tapline-test, but have to be included for a 32-tapline-test. Clock 19 and 20 are the last outputs from radar pulse 1. Due to the delay just described, sample 15 will produce two outputs. The phase outputs for both channels are 16-bit two's complement words. To verify the results, the outputs are converted into decimal numbers, in order to plot them in Matlab. The Matlab simulation produces similar results to compare both simulation output against each other.

171

CLK	Valid Result	I values	Q values	Pulse#	Sample#
1	0	0000000000000000	0000000000000000	Sync	Clear
2	0	0000000000000000	0000000000000000		
3	0	0000000000000000	0000000000000000		
4	0	0000000000000000	0000000000000000		
5	1	0000000011111110	0000000000000000	1	1
6	1	0000001011111010	0000000000000000	1	2
7	1	0000001010000010	0000000011010110	1	3
8	1	0000000000010000	0000000111000100	1	4
9	1	1111111010110110	0000000011101000	1	5
10	1	0000000110110100	0000000010000010	1	6
11	1	0000000000010110	1111110101011100	1	7
12	1	1111111100100100	1111110110000010	1	8
13	1	0000000110101110	1111111000100100	1	9
14	1	0000000101100010	1111111100111000	1	10
15	1	1111111001011110	1111111101010100	1	11
16	1	1111111010110110	0000001010101100	1	12
17	1	1111111000101000	0000000111100000	1	13
18	1	1111111100000110	0000000000110000	1	14
19	1	0000000110001110	0000000011100100	1	15
20	1	1111111100100100	0000000111001000	1	15
21	1	0000000011101000	0000000001100100	2	1
22	1	0000001001000100	0000000111010100	2	2

Table 31. T-Spice Simulation Outputs (hard limited)

To extend the 2-tapline-test case to x-number of taplines, a controller must set the "range-bin valid" control bit for at least x-1 clocks to low between two radar pulses. Due to the

shifted read out, a similar delay as for the FPGA design is achieved. Furthermore, the delay requirement produces additional outputs,

number of DRFM samples per pulse + (x−1) = number of outputs. (9.1)

After the T-Spice simulation results are transformed into a decimal representation, the controller can process the results in Matlab. Figure 89 shows a two-dimensional contour plots for Matlab (upper plot) and T-Spice (lower plot) simulation results in comparison. By visual inspection there is no obvious difference in the preliminary simulation results. Figure 90 shows the results in a 3D view and the graphical representation of the difference between the two simulations. Since the difference is only a plane at level zero, there is no difference. Thus, the simulations produced the same results and the proof is complete. Figure 91 exploits the T-Spice simulation results in a single graph and identifies the specified gain-coefficients for the obtained results.
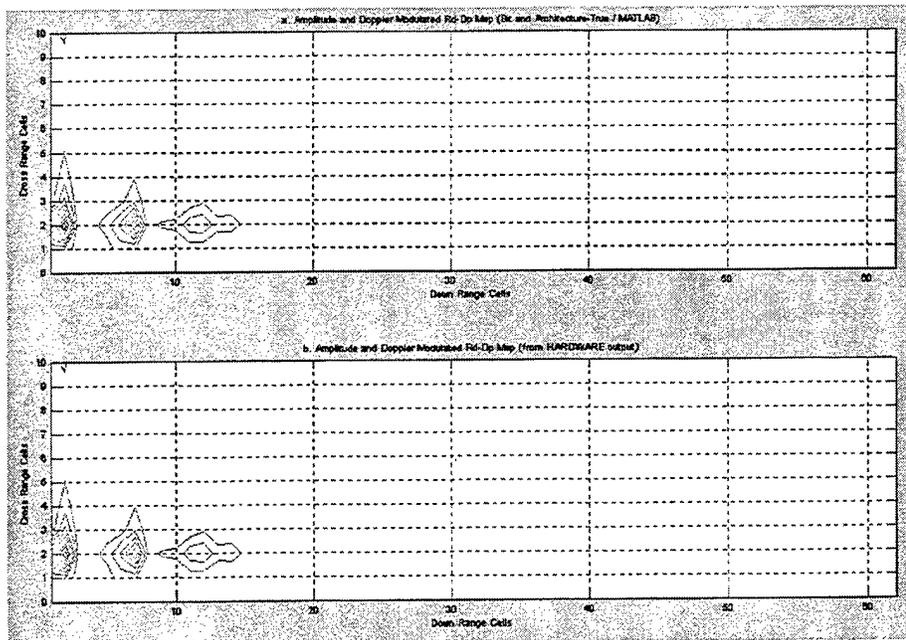
Figure 89. 2D Plot of the Simulation Results
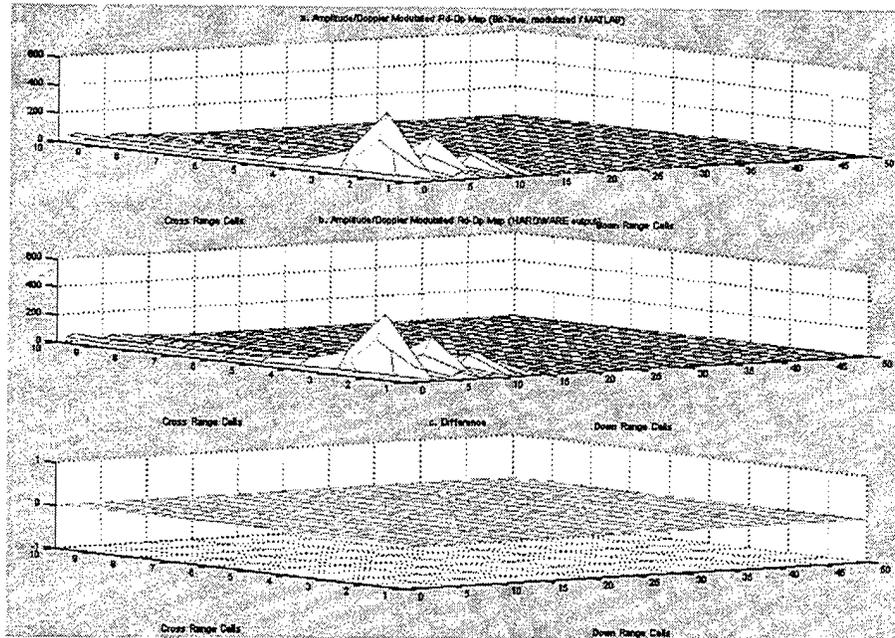
Figure 90. 3D Plot of the Simulation Outputs and Their Comparison
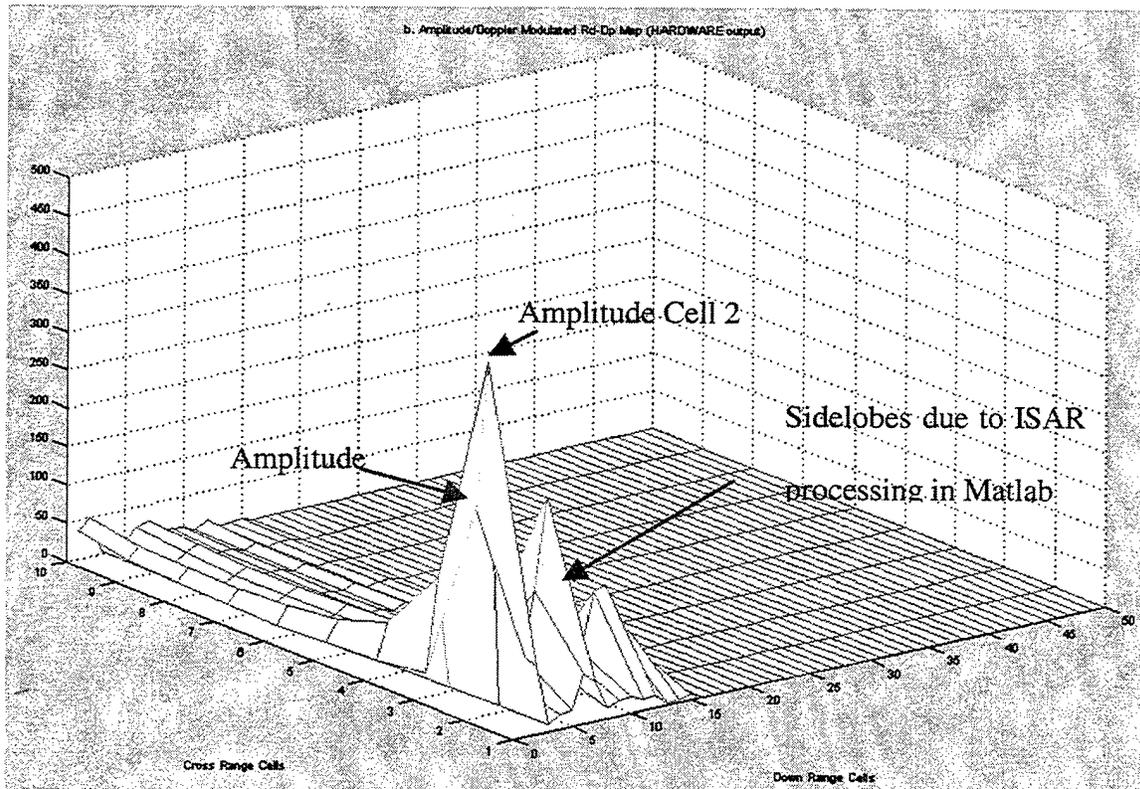
Figure 91. Exploit T-Spice Simulation Results

174

The 2-tapline test case illustrates the testing procedure and can be a guide for future testing. Transistor models, as used in this case, are non linear. Unfortunately T-Spice quickly reaches its limits simulating larger digital circuits. The simulation time increases almost exponential for large circuit simulation and often results in program crashes. The next section offers a method, which indicates how to partly overcome these problems. A simplification of the primary element in a digital circuit, the transistor, will reduce simulation time and increase the simulatable size of the circuit.

C. SIMULATION OF THE 32-TAPLINE CASE

Two approaches have been tested to find an efficient way for testing larger digital circuits. The first method is to replace every single logic gate with a gate definition. The gate definition will replace the gate circuit, e.g., 2NAND, and substitute it with a table of predefined output values. With this substitution the transistor layer could almost be completely eliminated. Unfortunately the code for the replacement is not fully developed and cannot be used for larger circuits. The second method tries to reduce the complexity of the transistor model itself. For this approach the transistors are replaced with simple switches.

1. Switch Model

The replacement of every transistor in the circuit with a simple switch reduces the computational requirements in the simulator tremendously. As shown earlier, the transistor definition is done in S-Edit. The definition for the P-FET and N-FET transistor calls the model in the SPICE-OUTPUT definition. By changing this line, a new model can be called. The new definition is: X${T} %{D} %{G} %{S} %{B} NMOSX. This

175

line defines the ports of a transistor, gate, source, drain and base. NMOSX is the new name for the transistor definition in switch form, e.g., definition for a N-FET switch. Since the definition of the transistors is changed in Level 1, the entire design circuit is affected by this change. Every module calling a transistor will use the switch instead. The schematic layout is still the same, but the mathematical behavior during the SPICE simulations is simplified.

2. Test Setup

The test setup in T-Spice is basically the same as for the two tapline test case. Even so, the circuit is more complex and contains approximately 16 times more nodes. T-Spice can handle circuits up to 300,000 elements. With 32 taplines, the circuit has 290,604 elements. This is very close to the limit and involves a lot of adjustments and initialization to get the simulation to perform.

a. Simulation Commands

S-Edit provides the SPICE translation of the schematic circuit automatically. After the SPICE definition is imported into T-Spice, it has to be modified with simulation commands and initialization commands. The following is an excerpt from a modified SPICE file ready to simulate:

1. Vdd Vdd Gnd DC 5

2. .include "D:\input_table_ship.md"

3. .include "D:\ModelSwitch.md"

4. .options prtdel=400n numnt=150 abstol=500n reltol=0.01

5. .tran 400n 17200n start=390n

Line one defines the voltage range between zero and five volts. Line two includes the test vectors for the simulation. The vectors are defined with the already introduced "bit" command. Line three is the definition for the transistor model, which defines the P-FET and N-FET transistors as switches. The definition is simple, but effective:

```
* Switch-level model definitions for NFETs and PFETs.

.model SWMODN SW VT=2.5 ron=1e12 roff=4000 dv=1

.model SWMODP SW VT=-2.5 ron=4000 roff=1e12 dv=1


.SUBCKT NMOSX D G S B

S2 D S G B SWMODN

.ENDS


.SUBCKT PMOSX D G S B

S1 S D G B SWMODP

.ENDS
```

The first block defines the switch behavior in general. For both switches the threshold voltage in both directions is 2.5 Volts, where the resistance values are inversed between N-definition and P-definition. The resistance determines the switch behavior. Since a P-FET pulls the output high and a N-FET pulls the output low, the resistance values have to be inverted. A value of 1e12 correspond to an open switch, where 4000 is a closed switch with 4000Ω resistance. The sub-circuit definitions are called by the simulator due to the include statement in the third line. The model's sub-circuit defines the order of the transistor nodes so that the switch behaves as expected.

Line four in the SPICE excerpt customizes the simulation. As before *prtdel* defines the readout cycle. *Numnt* defines the maximum number of iterations allowed during the solution of the Kirchhof-Current-Law (KCL) equations during a DC analysis. For a transient analysis, T-Spice first calculates the DC operating point of the circuit. This is a critical calculation for a very large circuit. The default for *numnt* is ten, which is not sufficient. If the circuit does not converge, T-Spice tries to use source stepping to find a DC operating point, which normally fails. A high number of iterations are therefore required. Curiously, T-Spice recommends decreasing *numnt* when the simulation fails. Also part of the .options commands is the definition of tolerances in absolute and relative form. An increasing of the tolerances results in a faster simulation. Nevertheless, definitions that are too loose result in wrong outputs. The values used for absolute tolerance and relative tolerance are very close to the acceptable limits and should not be further increased.

Line five holds the command for the type of simulation. The type is a transient analysis with a maximum step size of 400nsec, a simulation length of 17200nsec and an offset for the first output reading of 390nsec. In conjunction with the *prtdel* setting, the readout is every 400nsec starting at 390nsec so that each sample (defined with a pulse width of 400nsec) is read out only once. The readout is at the end of the sample to catch the solid-state result of the pulse.

T-Spice allows using initialization for certain nodes within a circuit. For the 32-tapline circuit, it is crucial to initialize the D-Flip-Flop outputs and the carry-out bit of the adders. Since the circuit in T-Spice has the same hierarchy as the schematic circuit, the initialization can be done directly in the sub-circuit definition for the D-Flip-

Flop and the adder cell. This will initialize every register cell output and every adder carry-out to the defined values. *Nodeset* sets an initial guess for the iterative DC-operating point calculation. After the first iteration, the specified nodes are allowed to float. Since very large circuits need more than one iteration, the IC command should be used instead. IC sets node voltages for the duration of a DC operating point calculation. The command is inserted in the sub-circuit definition. For a DFFC sub-circuit the line is .ic Q=0 QB=5. This initializes the Q output port to zero volts and its complement to 5 volts. For the adder cell sub-circuit the line is .ic Co=0 and initializes the carry-out to zero volts.

b.   *Input and Output Pads*

The only purpose of this sub-section is to provide an overview of the input and output pads. Multiple bits are collapsed into one single bit, e.g., Phase 0 to Phase 4 (five bits) corresponds to Phase0-4.

Outputs	Function
S_P_Test_Rout	Scan path out for a shift to the right (1 bit).
S_P_Test_Lout	Scan path out for a shift to the left (1 bit).
Tap_outI0-15	Output for processed values in I channel (16 bit).
Tap_outQ0-15	Output for processed values in Q channel (16 bit).
Data_Processed_out	Control bit flags valid output (1 bit).
Overflow_out	Control bit to check for overflow in a 16-bit adder (1 bit).

Table 32. Output Pads for the 32-Tapline Circuit

179

Inputs	Function
S_P_Test Lin	Scan path input to load test values into registers from the left.
S_P_Test Rin	Scan path input to load test values into registers from the right.
Tgt_Extent_in0-4	Input for the truth table to select the number of taplines used for false target generation (5 bits).
LD_Phase_SupTap_A	Select phase-increment registers in Supertap A and reads in from Bus 0 to 31 (1 bit).
LD_Phase_SupTap_B	Select phase-increment registers in Supertap B and reads in from Bus 0 to 31 (1 bit).
LD_Phase_SupTap_C	Select phase-increment registers in Supertap C and reads in from Bus 0 to 31 (1 bit).
LD_Phase_SupTap_D	Select phase-increment registers in Supertap D and reads in from Bus 0 to 31 (1 bit).
LD_Gain_SupTap_AB	Select gain-coefficient registers in Supertap A and B. Supertap A reads in the data from Bus 0 to 15, Supertap B reads in the data from Bus 16 to 31 (1 bit).
LD_Gain_SupTap_CD	Select gain-coefficient registers in Supertap C and D. Supertap C reads in the data from Bus 0 to 15, Supertap D reads in the data from Bus 16 to 31 (1 bit).
Bus0-15	First 16 bits from the 32-input bus (16 bits).
Bus16-31	Second 16 bits from the 32-input bus (16 bits).
Phase0-4	Input for the DRFM-phase samples (5 bits).
use_Phase_inc	Makes the phase-increment and the gain-coefficient stored in the buffer available for data processing (1 bit).
Range_bin_valid	Control input bit that is required to be high when valid DRFM-phase data is present at Phase0-4 (1 bit).
Overflow_in	Control bit that is used for daisy chaining of more Supertaps.
Data_Processed_in	Control bit that is used for daisy chaining of more Supertaps.

Tap_inI0-15	Input for I channel that is used for daisy chaining of more Supertaps (16 bits).
Tap_inQ0-15	Input for Q channel that is used for daisy chaining of more Supertaps (16 bits).
HLD	Chip hold for special operation mode.
LD	Chip load for normal operation mode.
SR	Shift right for scan-path test mode.
SL	Shift left for scan-path test mode.
CLK	Master clock used throughout the chip.

Table 33. Input Pads for the 32-Tapline Circuit

c.   Test Vectors

For simulation every input requires a test vector, even it is zero for the entire simulation time. The test vectors used the bit command to define the input to the pads in binary form: "VinPhase2 Phase2 Gnd bit ({001101101111111101} on=5.0 off=0.0 pw=400n)." VinPhase2 is the name of the voltage source, where the following Phase2 Gnd is the port name measured versus ground. To prevent confusion, the names should be the same. The bit command contains the vectors used during simulation. As defined after the parenthesis one corresponds to five volts and zero to zero volts. The pulse width for each bit is 400nsec. The entire simulation time as specified in the .tran command in T-Spice is 400nsec multiplied by the number of input bits. The length of a line in the editor determines the length of the input vector for one input. Normally 500 input bits can be used without any problems. The pulse width is important to achieve a steady state for each input. The larger the circuit, the higher is the required pulse width. In the manual are no specifications about the maximum pulse width or the maximum length of the input vector. Tests have shown that values higher than 590nsec for the pulse width and a length

181

of more than 500 characters for a line result into a computer crash. Due to these constraints, it was not possible to run a complete simulation of the ship test target mentioned at the end of Chapter 4. Instead, certain radar pulses have been chosen to verify the outputs produced by T-Spice.

A set of Matlab script files was developed to generate accurate test vectors. The Matlab script files convert data used in the Matlab equivalent simulation into a binary two's complement representation and create a simulation-input file using the appropriate syntax. The different files used are shown in Table 34.

Matlab script file	Output (text file)	Remark
convert2binary_rawint.m	converted_rawint.txt	DRFM-phase data
convert2binary_para.m	converted_para.txt	Modulation parameters (phase and gain modulation coefficients)
convert2binary_control.m	converted_control.txt	Control signals

Table 34. Matlab Files to Generate a T-Spice Input File

3. Results

After performing the simulations, the output files have to be examined and checked for correctness. A procedure was developed to examine T-Spice outputs using Matlab. The T-Spice output files are saved as text files and edited, presenting the first set of valid output data in the first row of the text files. Then, the Matlab script file "hard_limiter.m" is used to convert the results into binary two's complement representation. Finally, the script file "compare.m" is used to plot and compare each single output data produced by T-Spice with the results from the equivalent Matlab simulation. An example of test results after simulating one complete radar pulse is shown

in Figure 92 and Figure 93. The simulation refers to the ship test case using 64 radar pulses for the ISAR image integration discussed at the end of Chapter 4. Figure 92 shows a comparison between the single output data for the I-channel generated in Matlab and T-Spice. Figure 93 illustrates the Q-channel results. As for the two-tapline case, there are no differences between the Matlab and the T-Spice simulations, which verify the correctness of the DIS architecture based on the Matlab simulation.
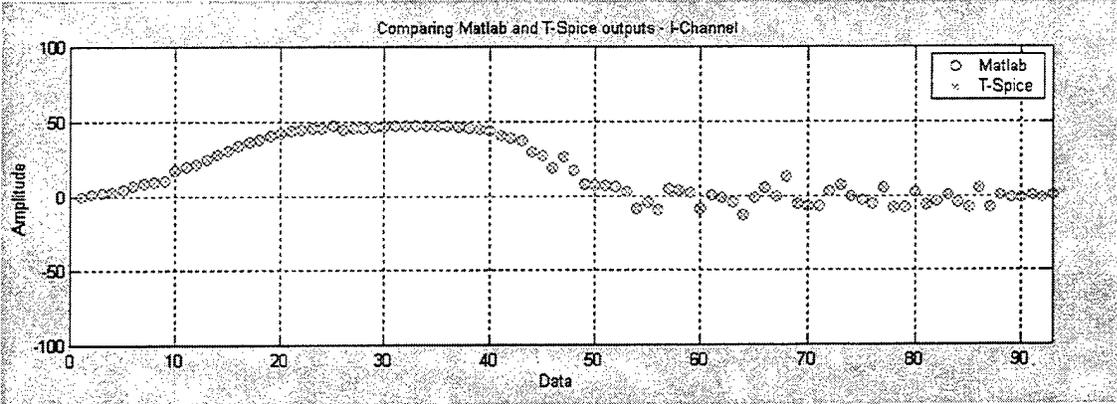
Figure 92. Comparing Matlab and T-Spice Outputs–I-Channel
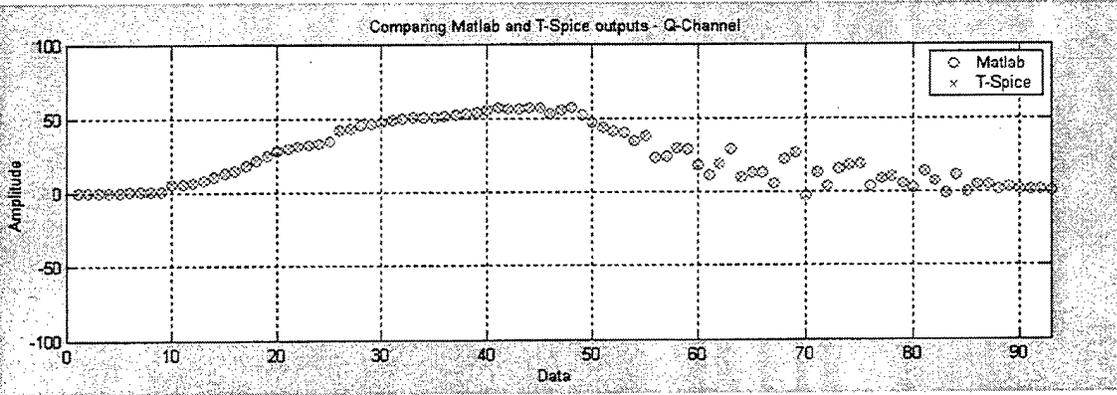
Figure 93. Comparing Matlab and T-Spice Outputs–Q-Channel

183

THIS PAGE INTENTIONALLY LEFT BLANK

X. LAYOUT AND FABRICATION

The cost of the DIS implementation, as with many integrated circuits, is directly proportional to the size of the chip. Moreover, the size of the DIS is directly proportional to the number of taps. In order to keep the implementation costs to a minimum and get a usable demonstration chip, we reduced the size of the design to 8-taplines. This Chapter illustrates the schematic layout of an 8-tapline chip and describes the physical layout in L-Edit. It points out and summarizes only the major differences of previously-discussed designs.

A. 8-TAPLINE SCHEMATICS

The 8-tapline circuit is based on the same hierarchy as the 32-tapline chip, using the double-buffered register tapline. As shown in Figure 94, a Supertap is used to implement the 8-taplines. The Supertap is connected to the 32-bit input bus and loads the phase values and gain-coefficient using a reduced number of control signals. The toplevel 32-to-5-bit decoder is not part of this design. The logic for a decoder representing a truth table to adjust the number of used taplines would increase the size of the design significantly, but gains almost no value for the concept realization. Therefore the user has to ensure a proper setup for normal operation. Since the number of taplines determines the size of the generated false target, only a continuous tapline activation beginning with the first tapline up to the desired one is acceptable. In spite of the missing target-extent decoder, the circuit chosen for fabrication has the same flexibility as the 32-tapline IC. Even a design of 8-taplines guarantees a usable concept demonstrator; however, it

decreases the size of the false target. Nevertheless the fabrication costs are reduced, since the IC area is less than a quarter in size. The first fabrication run will supply several chips. Because the concept provides the capability of daisy chaining, four chips could be interconnected to result in a 32-tapline-chip equivalent without a decoder capability. Thus, the limitations are reduced to a negligible value.
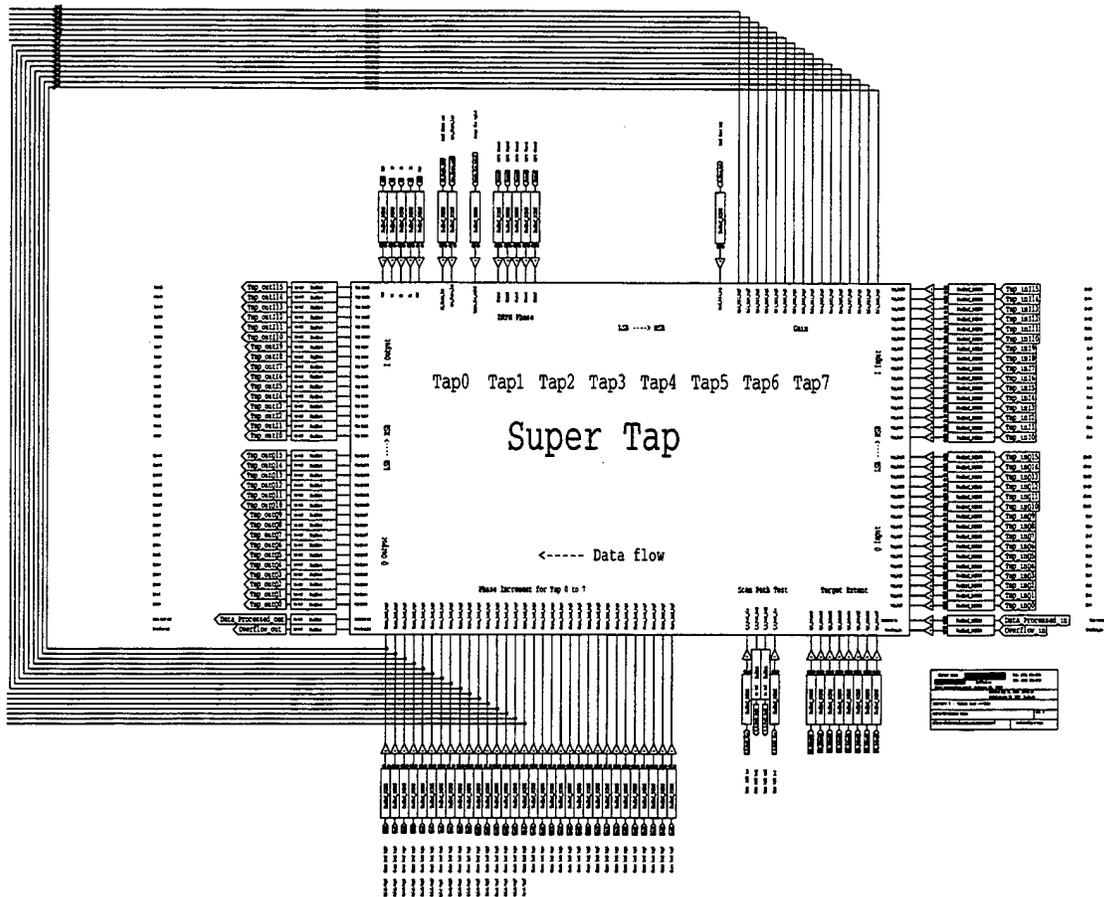
Figure 94. 8-Tapline Chip

186

B. TIMING AND CONTROL

The control requirements for normal chip operation are reduced due to the fact that only 32 phase value and 16 gain-coefficients are loaded. The corresponding output and input pads are listed and explained in Table 35 and Table 36, where a few pad names have changed in comparison to the 32-tapline IC, but these pads still perform the same functions.

Outputs	Function
S_P_Test_Rout	Scan path out for a shift to the right (1 bit).
S_P_Test_Lout	Scan path out for a shift to the left (1 bit).
Tap_outI0-15	Output for processed values in I channel (16 bit).
Tap_outQ0-15	Output for processed values in Q channel (16 bit).
Data_Processed_out	Control bit flags valid output (1 bit).
Overflow_out	Control bit to check for overflow in a 16-bit adder (1 bit).

Table 35. Output Pads for the 8-Tapline Circuit

Inputs	Function
S_P_Test_Lin	Scan path input to load test values into registers from the left.
S_P_Test_Rin	Scan path input to load test values into registers from the right.
Tgt_Extent0-7	Input to select the number of taplines used for false target generation (8 bits), where no decoder logic is implemented.
LD_Phase_inc	Select phase-increment registers in IC and reads in from Bus 0 to 31 (1 bit).
LD_Gain_Reg	Select gain-coefficient registers in IC. Since there are only 16 gain values to load, Bus 0 to 15 are used to read them in (1 bit).
Bus0-15	First 16 bits from the 32-input bus (16 bits).
Bus16-31	Second 16 bits from the 32-input bus (16 bits).
Phase0-4	Input for the DRFM-phase samples (5 bits).
use_Phase_inc	Makes the phase-increment and the gain-coefficient stored in the buffer available for data processing (1 bit).
Range_bin_valid	Control input bit that is required to be high when valid DRFM-phase data is present at Phase0-4 (1 bit).
Overflow_in	Control bit that is used for daisy chaining of more Supertaps.
Data_Processed_in	Control bit that is used for daisy chaining of more Supertaps.
Tap_inI0-15	Input for I channel that is used for daisy chaining of more Supertaps (16 bits).
Tap_inQ0-15	Input for Q channel that is used for daisy chaining of more Supertaps (16 bits).
HLD	Chip hold for special operation mode.
LD	Chip load for normal operation mode.
SR	Shift right for scan-path test mode.
SL	Shift left for scan-path test mode.
CLK	Master clock used throughout the chip.

Table 36. Input Pads for the 8-Tapline Circuit

188

Figure 95 shows the timing diagram for the initial loading phase. In addition to the reduced number of control bits for the gain and phase adjustment, the loading cycle is decreased by four clock cycles. Figure 96 illustrates the readout phase between two radar pulses. Given that the number of taplines is reduced to eight, only seven clock cycles are necessary to read out the processed data remaining in the adder chain after the last DRFM sample is processed (instead of 31 clock cycles as shown before). In general, the timing constraints are very closely related to the 32-tapline design as discussed in detail in previous chapters.
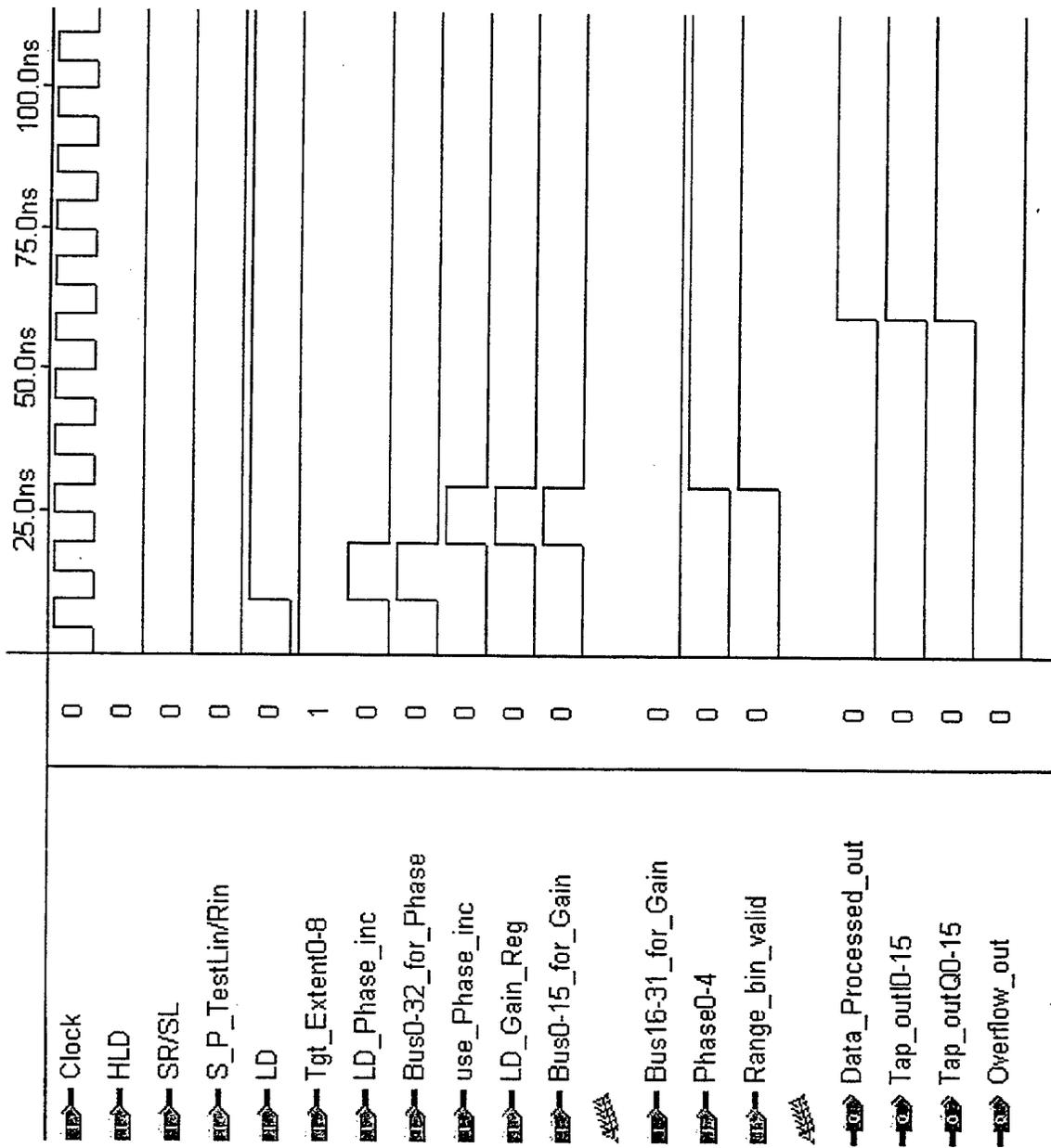
Figure 95. Timing Diagram for the Initial Loading Phase of the 8-Tapline Chip
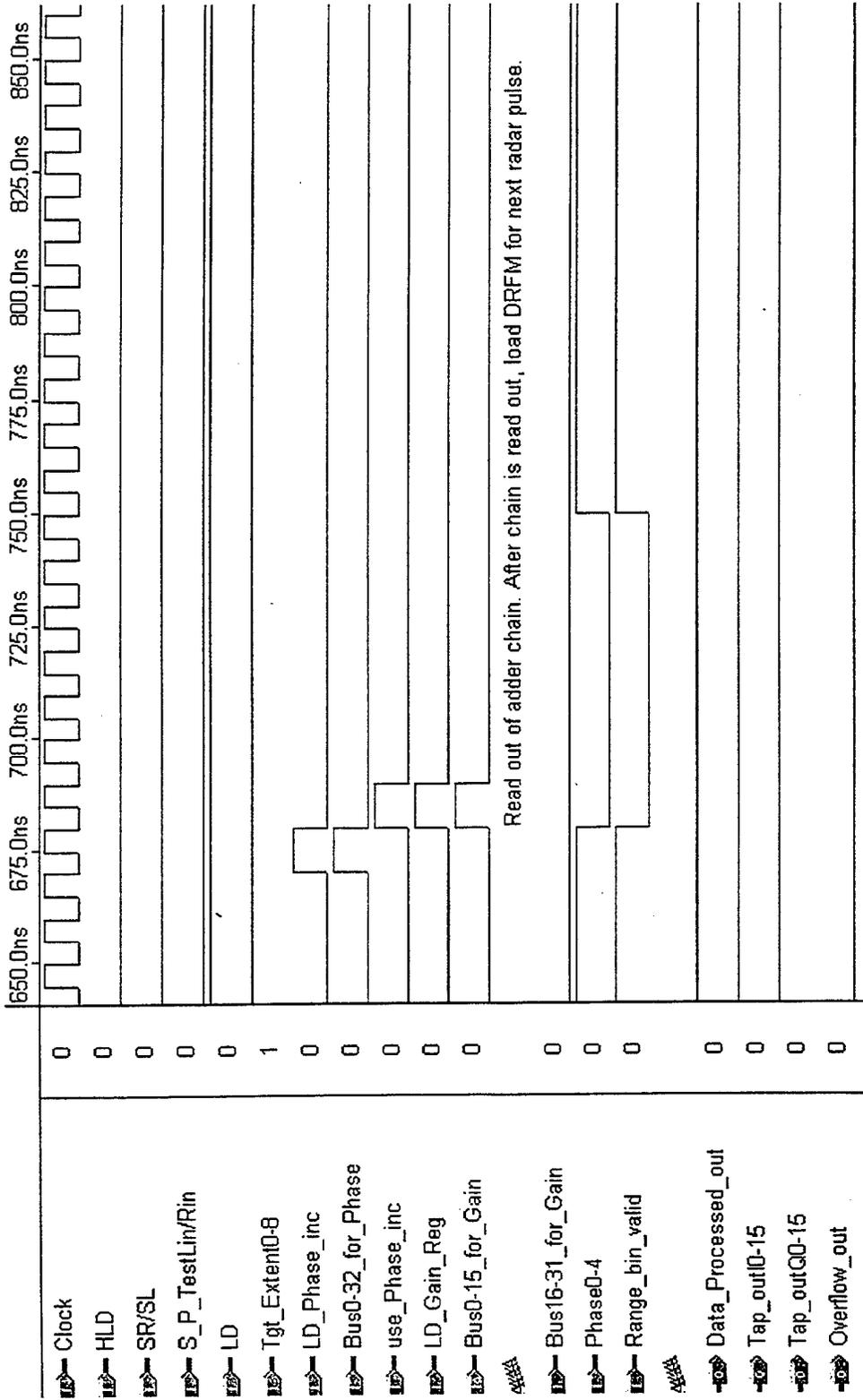
Figure 96. Timing Diagram between Two Radar Pulses for the 8-Tapline Chip

191

C. PHYSICAL LAYOUT GENERATION

The IC layout is automatically generated with the layout editor, L-Edit. A wide range of different optimization parameters affecting both cell placement and network routing are tried with varying degrees of success. The final layout of the chip core with pad frame is shown in Figure 97 and is approximately 5.25 mm by 5.62 mm. With I/O, power, and ground pads and the power and ground distribution buses, the layout is approximately 5.71 mm by 6.07 mm with a total chip area of slightly less than 35 square mm. After layout generation, a design rule check revealed several DRC violations that had to be corrected by hand. Also, additional power and ground pads had to be connected by hand to the power and ground distribution buses. Layout correctness is confirmed using the layout versus schematic comparison tool, LVS. For layout verification the extracted netlist in L-Edit is compared with the netlist of S-Edits circuit representation. At this point of the process minor design incompatibilities are still vacant. As soon as the netlist comparison is passed, the extracted file will be simulated in T-Spice to confirm correct logical functionality. With the T-Spice simulation, the developing process concludes and the resulting files are sent to fabrication. The finished IC will be fabricated through the MOSIS fabrication service at Hewlett Packard on their 0.5 micron CMOS line. The selection of the HP 0.5 micron process was determined to be a reasonable compromise between cost and performance. Although maximum performance will eventually be desired, for this initial, proof-of-concept chip, the moderate performance of the 0.5 micron process is sufficient. It should be noted that the Tanner scalable CMOS library used is compatible with IC fabrication processes down to 0.25 micron without

modification and with only minor modifications it can be used with fabrication processes as small as 0.18 micron. Use of a 0.18 micron fabrication process would allow the DIS design to operate at clock rates in excess of 500 MHz, which is one of the goals for future work.
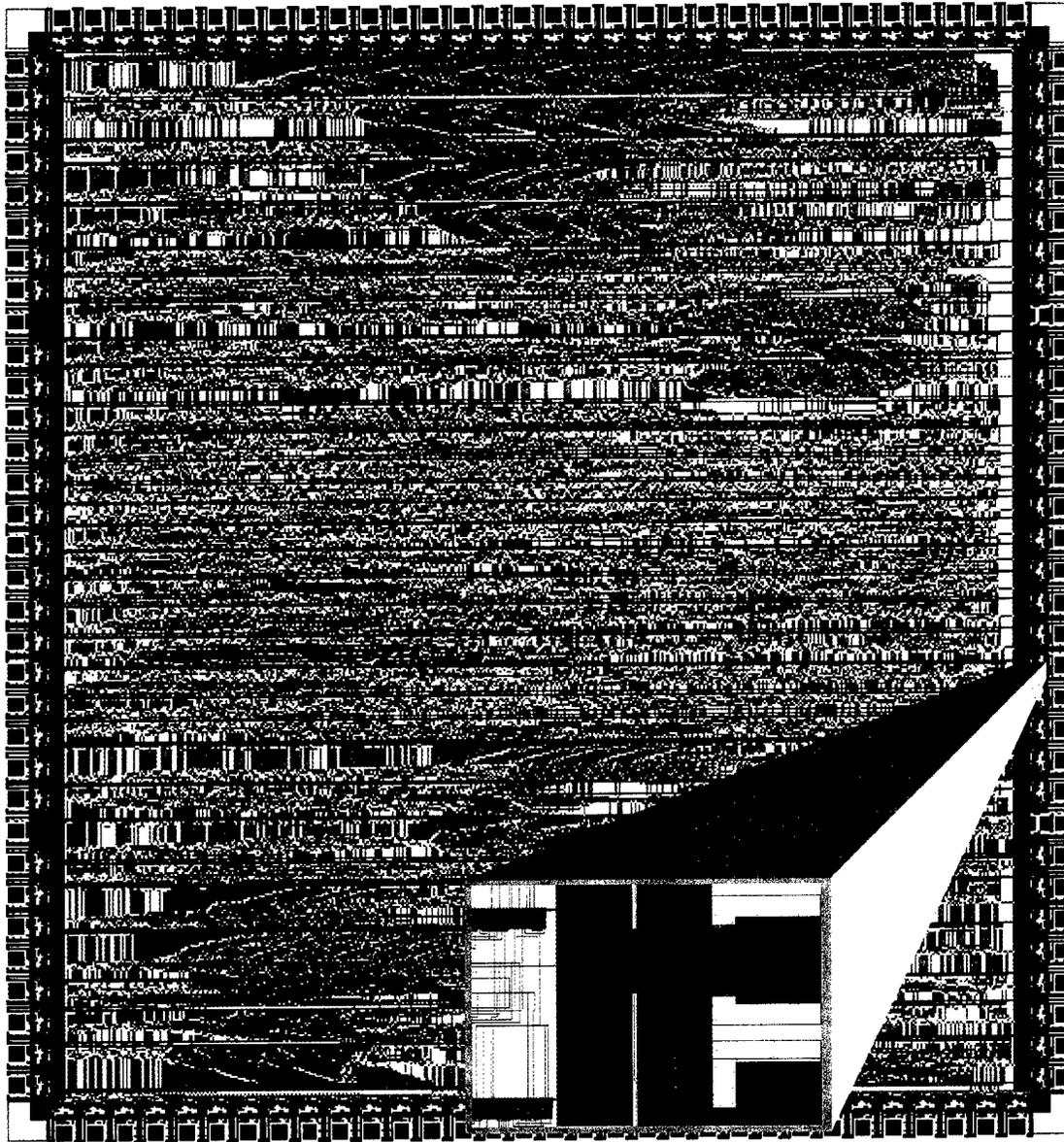
Figure 97. Layout for the 8-tapline Chip Showing Enlarged Pad-Area Region

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. MATLAB CODES

1. DIS SIMULATION FILES–VERSION 4

The DIS simulation files presented in this appendix are the latest version of Matlab codes in this project. The following version of simulating the Digital Image Synthesizer architecture in Matlab has been developed for testing purposes. This modified version (v4) of the original code now represents the new architecture that has been developed during the process of working on the hardware layout in Tanner Tools. The original version (v0) was developed by Sy Yeo, 1998.

A script file (runDISv4.m) has been developed to execute the different files in a more convenient way than is used to run a full simulation. This modified code includes flexibility in choosing the number of taps to be used, proper start-up and shutdown of the taps during processing, "parallel processing" of DRFM-phase data in the taps, and "serial summation" of the results in the taps (partial summation starting from the last taps in use, all the way up to the first tap, which then will be the valid output data). Compared to the v2- and v3-codes, this set of codes is easy to scale-up, can deal with multiple scatterers per range-bin (multiple Doppler that will vary both phase and gain-coefficients between radar pulses). The code can be set in an initial state to run in Version 2 mode (single scatterer per range-bin) if so desired.

Before running the runDISv4.m file, one must extract parameters of the false target one wants to generate if one is working with multiple scatterers per range gate. An appropriate extract_XXX.m file (existing or by modifying an existing file) must be used. After that, the new parameter text file must be called by the simhwchkv4.m file

195

(check/modify line 33 to 40). The graphical user interface (guiv4.m) in this version only serves to get a Doppler offset of the whole false target.

The code also writes data to text files that represent the functionality of the scan-path testing options that are included in the hardware layout, and also to separate I- and Q-data outputs. Minor corrections have also been made to the original code.

a. runDISv4.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% runDISv4.m
% This script file will help you run the Digital Image Syntesizer
% (DIS). This is a modified version that is able to handle different
% target extents (that is, how many taps the user would like to use
% that will represent the radial length of the target, seen from the
% ISAR). The user can also specify some necessary input parameters.
% Created by:
%    LTC Stig Ekestorm, Apr -00
%    Naval Postgraduate School
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% set path
% cd c:\temasek\denise\thesis\final_design\vbfiles

% clear the workspace
clear

% declare global variables, used in outer m-files and functions
global sorm
global dp_pts
global rg_pts
global hda
global printdata

% interactive - use of a dialog box to get inputs parameters from user
title='User Specified Parameters - Matlab DIS Simulation';
prompt={'Single (Version 2) or Multiple (Version 4) Scatterer per range
gate           [1 for Multiple]. If Version 4, then need to run
extract_para_X.m first.',...
         'Number of Doppler cells in the ISAR.',...
         'Number of Range gates in the plots.',...
         'Hardware Data available for comparison [1 for yes].',...
         'Print Intermediate Data to text file (slows down the
simulation) [1 for yes].'};
default={'0','64','200','0','0'};
response=inputdlg(prompt, title, 1, default);
fields={'sorm','dp_pts','rg_pts','hda','printdata'}; % number of
Doppler cells, hardware data available
input=cell2struct(response,fields,1);
% convert cell structure created by dialog box back to numbers
```

```matlab
sorm=str2num(input.sorm);
dp_pts=str2num(input.dp_pts);
rg_pts=str2num(input.rg_pts);
hda=str2num(input.hda);
printdata=str2num(input.printdata);

% run the graphical user interface (GUI) to specify target parameters
disp('Enter the values in the Grapical User Interface')
disp('Press any key to continue')
guiv4
pause

% pre-process signal parameters, simulate ISAR
if sorm == 0
   mathostv4
else
   mathostv4b
end

% simulate the DIS in Matlab
% This simulation does "parallel processing" and then "serial
summation", including:
% - correction at start-up ("initializing outputs from the taps, one
tap after another")
% - correction at the end ("shutting down the taps, one tap after
another")
if sorm == 1,
   if printdata == 1,
      simhwchkv4_write
   else
      simhwchkv4
   end
else
   if printdata == 1,
      simhwchkv2_write
   else
      simhwchkv2
   end
end

% plot results for visual comparison
plothwv4

% end of file
```

b. guiv4.m

```matlab
function [dat] = guiv4(action);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% guiv4.m
% Get inputs from screen
% MAJ Stig Ekestorm, Feb -00
% Modified version of guiv0.m by SY YEO, Jan -98
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
global hf
global h1
global h2
global data
global loc
global patchsize
global txtloc
global count
global ph
global dp_pts

if nargin<1,
    action='start';
end;

if strcmp(action,'start'),

    % initialize the figure

    set(0,'DefaultAxesFontSize',6);

    hf = figure(1); clf
    set(hf,'NextPlot','add');

    set(hf, ...
        'NumberTitle','off', ...
        'Name','Naval PostGraduate School',...
        'backingstore','off',...
        'Units','normalized');

    %rg_pts = 15;
    rg_pts = 62;
    %dp_pts = 64;
    data = [];  loc = [];
    count = 0;
    ph = [];

    h1 = axes('Position',[0 0 1 1],'Visible','off');
    h2 = axes('Position',[0.1 0.1 0.6 0.8]);

    set(hf,'currentaxes',h2);

    xa = 1:rg_pts;
    ya = 0:(dp_pts-1);

    xtick = 0:1:rg_pts;
    set(gca,'XTickMode','manual');
    set(gca,'XLimMode','manual');
    set(gca,'XLim',[1 rg_pts]);
    set(gca,'XTick',xtick);
    set(gca,'XGrid','on');
    set(gca,'GridLineStyle','-');

    set(gca,'YTickMode','manual');
    set(gca,'YLimMode','manual');
    %set(gca,'YLim',[0 dp_pts-1]);
```

```
if dp_pts > 64,
    set(gca,'YLim',[0 64-1]);
else
    set(gca,'YLim',[0 dp_pts-1]);
end
ytick = 0:1:dp_pts;
set(gca,'YTick',ytick);
set(gca,'YGrid','on');
set(gca,'GridLineStyle','-');

xh = xlabel('Range Cell'); set(xh,'FontSize',8); clear xh
yh = ylabel('Doppler'); set(yh,'FontSize',8); clear yh
ht = title('Range-Doppler-Amplitude Map Entry Program');
set(ht,'FontSize',10,'Color',[0 0 1]);

a = uicontrol('Units','normalized', ...
    'BackgroundColor',[.9 .9 .9], ...
    'Position',[0.72 0.80 0.15 0.04], ...
    'Style','text', ...
    'FontSize',6,...
    'String','Range Cell', ...
    'Tag','aText');

b = uicontrol('Units','normalized', ...
    'BackgroundColor',[.9 .9 .9], ...
    'Position',[0.72 0.75 0.15 0.04], ...
    'Style','text', ...
    'FontSize',6,...
    'String','Doppler Cell', ...
    'Tag','bText');

c = uicontrol('Units','normalized', ...
    'BackgroundColor',[.9 .9 .9], ...
    'Position',[0.72 0.65 0.15 0.04], ...
    'Style','text', ...
    'FontSize',6,...
    'String','Amplitude', ...
    'Tag','cText');

c11 = uicontrol('Units','normalized', ...
    'BackgroundColor',[.9 .9 .9], ...
    'Position',[0.72 0.60 0.15 0.04], ...
    'Style','slider','min',0,'max',4,...
    'SliderStep',[0.25 0.5],...
    'Callback','guiv4(''update1'')');

d = uicontrol('Units','normalized', ...
    'BackgroundColor',[.9 .9 .9], ...
    'Position',[0.72 0.50 .15 0.04], ...
    'Style','text', ...
    'FontSize',6,...
    'String','Doppler shift');

d11 = uicontrol('Units','normalized', ...
    'BackgroundColor',[.9 .9 .9], ...
    'Position',[0.72 0.45 0.15 0.04], ...
```

```
      'Style','slider','Min',-10,'Max',10,...
      'SliderStep',[0.05 0.1],...
      'Callback','guiv4(''update1'')');

a1 = uicontrol('Units','normalized', ...
      'BackgroundColor',[1 1 1], ...
      'Position',[0.9 0.80 0.05 0.04], ...
      'Style','text', ...
      'FontSize',6,...
      'String','', ...
      'Tag','a1Text');

set(gcf,'currentaxes',h1);
b1 = uicontrol('Units','normalized', ...
      'BackgroundColor',[1 1 1], ...
      'Position',[0.9 0.75 0.05 0.04], ...
      'Style','text', ...
      'FontSize',6,...
      'String','', ...
      'Tag','a2Text2');

set(gcf,'currentaxes',h1);
c1 = uicontrol('Units','normalized', ...
      'BackgroundColor',[1 1 1], ...
      'Position',[0.9 0.65 0.05 0.04], ...
      'Style','text', ...
      'FontSize',6,...
      'Callback','guiv4(''update'')',...
      'String','');

d1 = uicontrol('Units','normalized', ...
      'BackgroundColor',[1 1 1], ...
      'Position',[0.9 0.50 0.05 0.04], ...
      'Style','text', ...
      'FontSize',6,...
      'Callback','guiv4(''update'')',...
      'String','');

q1 = uicontrol('Units','normalized', ...
      'BackgroundColor','Yellow', ...
      'Position',[0.9 0.10 0.05 0.04], ...
      'Style','pushbutton', ...
      'FontSize',8,...
      'String','SAVE', ...
      'Callback','guiv4(''savequit'')');

q2 = uicontrol('Units','normalized', ...
      'BackgroundColor','Yellow', ...
      'Position',[0.78 0.1 0.1 0.04], ...
      'Style','pushbutton', ...
      'FontSize',8,...
      'String','CLEAR', ...
      'Callback','guiv4(''start'')');

txtloc = [a a1 b b1 c c1 c11 d d1 d11];
% Assign action when mouse button is pressed
```

```
        set(h2,'ButtonDownFcn','guiv4(''down'')');

elseif strcmp(action,'down'),
    % Obtain coordinates of mouse click location in axes units

    set(hf,'currentaxes',h2);
    pt=get(h2,'currentpoint');

    x=pt(1,1); xf = floor(x);
    y=pt(1,2); yf = floor(y);
    [r,c] = size(data);

    set(txtloc(7),'Value',0);
    set(txtloc(9),'Value',0);

    tmp = [x y 1 0];
    loc = [loc tmp];
    tmp = [xf yf 1 0];
    data = [data;tmp];
    [r,c] = size(data);
    ypos = [yf yf+1 yf+1 yf];
    xpos = [xf xf xf+1 xf+1];

    count = count + 1;
    %disp(count);
    txt = ['Tag',num2str(count)];
    ptr = patch(xpos,ypos,[1 1 1]*0.9);
    %disp(ptr);
    set(ptr,'ButtonDownFcn',[ ...
         'guiv4(''update'')']);
    set(ptr,'Tag',txt);
    set(ptr,'UserData',[xf yf 1 0]);
    ph = ptr;
    set(txtloc(2),'String',xf);
    set(txtloc(4),'String',yf);
    set(txtloc(6),'String',1);
    set(txtloc(9),'String',0);

elseif strcmp(action,'update'),
    % Determine the patch that is selected
    ph = gcbo;
    %set(ph,'Selected','on');
    % Retrieve the values for that patch and display it
    % txtloc = [a a1 b b1 c c1 c11 d d1 d11];
    % txtloc 2: Range cell
    % txtloc 4: Doppler cell
    % txtloc 6: Amplitude  txtloc 7: Slider bar
    % txtloc 9: Doppler offset  txtloc 9: Slider bar
    ud = get(ph,'UserData');
    set(txtloc(2),'String',ud(1));
    set(txtloc(4),'String',ud(2));
    set(txtloc(6),'String',ud(3));
    set(txtloc(9),'String',ud(4));
    set(txtloc(7),'Value',ud(3));
    set(txtloc(10),'Value',ud(4));
```

```
elseif strcmp(action,'update1'),
    if (~isempty(ph))
        ph1 = gcbo;
        if ((ph1 == txtloc(7)) | (ph1 == txtloc(10)))
            ud = get(ph,'UserData');
            xf = ud(1);   yf = ud(2);
            ypos = [yf yf+1 yf+1 yf];
            xpos = [xf xf xf+1 xf+1];
            set(ph,'Selected','off');
            % Update the amplitude/Doppler values
            if (ph1 == txtloc(7))
                tmp1 = get(txtloc(7),'Value');
                tmp1 = round(tmp1)
                set(txtloc(6),'String',tmp1);
                set(txtloc(7),'Value',tmp1);
                if (tmp1 < 1),
                    set(txtloc(7),'Value',1);
                    set(txtloc(6),'String','1');
                    tmp1 = 1;
                end
                col = [1 1 1]*(1-tmp1/10);
                set(ph,'FaceColor',col);
                set(ph,'UserData',[ud(1) ud(2) tmp1 ud(4)]);
            end
            if (ph1 == txtloc(10))
                tmp2 = round(get(txtloc(10),'Value'));
                set(txtloc(9),'String',tmp2);
                set(txtloc(10),'Value',tmp2);
                set(ph,'UserData',[ud(1) ud(2) ud(3) tmp2]);
            end
            %disp('HHH');
            %disp(get(ph,'Tag'))
            %disp(get(ph,'UserData'))
        end
    end

elseif strcmp(action,'savequit'),
    dat = [];
    for i = 1:count
        tt = findobj('Tag',['Tag' num2str(i)]);
        tmp = get(tt,'UserData')
        dat = [dat;tmp];
        fprintf('count = %d, Tag = %s  ',count,get(tt,'Tag'));
        disp(tmp);
    end
    save -ascii sigpar1 dat
    close gcbf
end
```

c. mathostv4.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% mathostv4.m
% MAJ Stig Ekestorm, Feb -00
% Modified version of mathostv0.m by SY YEO, Jul-98
%
% Generate pri_dp map and range-Doppler map
% - generates the files for input to hardware
% -- file para.txt contains:
%     line 1: number of range cells
%     line 2: number of pulse in a batch (equals to dp_pts in this
%               program)
%     line 3: extent of target in cells (n: integer); number of taps in
%               delay also equals n (pipeline design)
%     line 4: gain1, gain2, ..., gain n (integer)
%     line 4+n+1: phi0 (pulse 1),
%     line 4+n+2: phi1 (pulse 1),
%     line 4+n+targetExtent: phi-targetExtent (pulse 1),
%     line 4+n+targetExtent+1: phi0 (pulse 2),
%     line 4+n+targetExtent+2: phi1 (pulse 2),
%     line 4+n+2*targetExtent: phi-targetExtent (pulse 2),
%     ...
%     line 4+n+dp_pts*targetExtent: phi-targetExtent (pulse dp_pts)
%
% -- file raw.txt contains the instantaneous phases of simulated DFRM-
%     data (quantized to 45deg step):
%     line 1: pulse 1 (integer)
%     line 2: pulse 2
%     ....
%     line dp_pts: pulse dp_pts
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear

global sorm
global dp_pts
global rg_pts
global doppler_inc
global printdata

set(0,'defaultAxesFontSize',8);

noplot = 0;
Ncontours = 20;

% Parameters
bw = 100e6;
pwc = 1/(1.25*bw);    % compressed pulsewidth
pw =0.5e-6;
prf = 2e3;  pri = 1/prf;
mu = 2*pi*bw/pw;
fs = 1.25*bw;  Ts = 1/fs;
snr = 0;
```

```
% set-up grid
% x-axis(rg), y-axis(dp)
%rg_pts = 200;
%dp_pts = 64;

pri_rg_map = zeros(dp_pts,rg_pts);
pri_rg_mapq = zeros(dp_pts,rg_pts);
pri_rg_map_shift = zeros(dp_pts,rg_pts);
pri_rg_map_shiftq = zeros(dp_pts,rg_pts);
pri_rg_phaseq = zeros(dp_pts,rg_pts);

% insert waveform into grid;
load -ascii sigpar1
sigpar = sigpar1';
doppler_inc = sigpar(4,:);
sigpar([2 4],:) = sigpar([2 4],:)*prf/dp_pts;
[lxs,lys] = size(sigpar);
%t0 = 0:Ts:pw-Ts;
t0 = Ts:Ts:pw;

%for the reduced 2-Tap T-Spice simulation
%TsNew=4*1/fs;
%tnew = 0:TsNew:pw-TsNew;
%tnew = TsNew:TsNew:pw;
%t0 = tnew;

num_chirp_samples = length(t0); if ((num_chirp_samples + lxs) > rg_pts)
disp('Warning : Chirp is clipped - set grid size larger'); end

% open files for writing
f1 = fopen('para.txt','w');
fprintf(f1,'%d\r\n',num_chirp_samples);      % number of range cells
fprintf(f1,'%d\r\n',dp_pts);                 % number of Doppler cells
fprintf(f1,'%d\r\n',lys);                    % target extent

% adjustment to correct multiplication factors for the amplitude (gain)
value
for i = 1:lys
   switch sigpar(3,i)
   case {1}
      sigpar(3,i)=1; % no shift, multiplication by 1, hardware bit "00"
   case {2}
      sigpar(3,i)=2; % shift by 1, multiplication by 2, hardware bit
"01"
   case {3}
      sigpar(3,i)=4; % shift by 2, multiplication by 4, hardware bit
"10"
   case {4}
      sigpar(3,i)=8; % shift by 3, multiplication by 8, hardware bit
"11"
   end
   fprintf(f1,'%d\r\n',sigpar(3,i));   % gain1, gain2, ..., gainN
end

nbitsph = 3;
nbitsdop = 5;
```

```
nbitsamp = 8;
b = 2*pi/(2^nbitsph);
a = 2*pi/(2^nbitsamp);
p = 2*pi/(2^nbitsdop);
%for the reduced 2-Tap T-Spice simulation, Nov -99
%p = 2*pi/(dp_pts);


for idx1 = 1:dp_pts        % Repeat for total number of pulses within
batch
    t1 = t0 + (idx1)*pri;
    %t1 = t0 + (idx1-1)*pri;
    for idx = 1:lys
        %**** approximation used here, assume phase change due to Doppler
within a chirp is constant
        %**** since the Doppler is tens of hertz compared to the MHz
chirp bandwidth
        oldphase = mu*t1.*t1/2 + 2*pi*sigpar(2,idx)*t1;
        %oldphase = mu*t1.*t1/2 + 2*pi*sigpar(2,idx)*(idx1-1)*pri;
        oldphase = mod(oldphase,2*pi);

        % quantize the oldphase to 1 of 8 phases
        int_oldphase = round(oldphase/b);
        oldphaseq = b*int_oldphase;          % quantize the phase
        xc = exp(sqrt(-1)*oldphase);
        lx = (sigpar(1,idx)):(sigpar(1,idx))+length(xc)-1;
        pri_rg_map(idx1,lx) = xc+pri_rg_map(idx1,lx);
        pri_rg_phaseq(idx1,lx) = int_oldphase;


        xcq = exp(sqrt(-1)*oldphaseq);
        xcq = p*round(xcq/p);          % quantize the phase
        pri_rg_mapq(idx1,lx) = xcq+pri_rg_mapq(idx1,lx);
        % phase focusing
        dopphase = 2*pi*sigpar(4,idx)*(idx1)*pri; % approximation used
here
        %dopphase = 2*pi*sigpar(4,idx)*(idx1-1)*pri; % approximation used
here
        newphase = oldphase + dopphase*ones(size(oldphase));
        xI = cos(newphase);
        xQ = sin(newphase);
        x1 = sigpar(3,idx)*(xI+sqrt(-1)*xQ);
        pri_rg_map_shift(idx1,lx) = pri_rg_map_shift(idx1,lx) + x1;


        int_dopphaseq = round(dopphase/p);
        dopphaseq = int_dopphaseq*p;
        newphaseq = oldphaseq + dopphaseq;
        xI = cos(newphaseq);
        xQ = sin(newphaseq);
        xI = round(xI/a)*a;
        xQ = round(xQ/a)*a;
        x1 = sigpar(3,idx)*(xI+sqrt(-1)*xQ);
        pri_rg_map_shiftq(idx1,lx) = pri_rg_map_shiftq(idx1,lx) + x1;


        % store the dopphase value (ignore intrapulse phase change since
it is small)
        %fprintf(f1,'%d\r\n',int_dopphaseq);               %originals
code, incrementation of phase modulation coefficients
```

```
        %fprintf(f1,'%d\r\n',mod(int_dopphaseq,32));           %to get
true phase modulation coefficients each PRI
        fprintf(f1,'%d\r\n',mod(2*fix(int_dopphaseq/2),32));   %to
represent phase modulation coefficients using 4-bits words


    end
end
fclose(f1);


noise = randn(size(pri_rg_map))*c_snr(snr); noise = 0;
pri_rg_map = pri_rg_map + noise;
pri_rg_map_shift = pri_rg_map_shift + noise;


%------
save pulse1 pri_rg_map_shiftq


%---------------
% Perform pulse compression
% (a) for the non-quantized phase case
disp('Creating reference waveform');
ph = (mu*t1.*t1/2);
crefc = exp(sqrt(-1)*ph);
cref = conj(fft(crefc,2*rg_pts-1));
save pc_ref cref t1
pc_ref_map = fft(pri_rg_map.',2*rg_pts-1).';
pc_ref_map_shift = fft(pri_rg_map_shift.',2*rg_pts-1).';


disp('Performing pulse compression');
pri_rg_map1 = zeros(size(pri_rg_map));
pri_rg_map2 = zeros(size(pri_rg_map));


%--- Compress the original signals
for idx = 1:dp_pts
    tmp = cref.*pc_ref_map(idx,:);
    tmp1 = fftshift(ifft(tmp));
    pri_rg_map1(idx,:) =  tmp1(rg_pts:end);
end


%--- Compress the Doppler shifted signals

for idx = 1:dp_pts
    tmp = cref.*pc_ref_map_shift(idx,:);
    tmp1 = fftshift(ifft(tmp));
    pri_rg_map2(idx,:) =  tmp1(rg_pts:end);
end

% Compute the rg-dop map
disp('Plotting ... r-d map');


dp_rg_map = fft(pri_rg_map1,dp_pts);
dp_rg_map_shift = fft(pri_rg_map2,dp_pts);


[lx,ly] = size(dp_rg_map);
rax = 1:(length(ly));
dax = 0:(length(lx))-1;
```

```
dpy = abs(dp_rg_map);
dpy_shift = abs(dp_rg_map_shift);

if (noplot == 0)
    figure(1);

    subplot(2,1,1);
    h = contour(dpy,Ncontours); grid
    title('a. Original Rd-Dp Map');
    axis([1 62 0 dp_pts])
    xlabel('Down Range Cells');   ylabel('Cross Range Cells');
    subplot(2,1,2);
    h = contour(dpy_shift,Ncontours); grid
    axis([1 62 0 dp_pts])
    title('b. Amplitude and Doppler Modulated Rd-Dp Map');
    xlabel('Down Range Cells');   ylabel('Cross Range Cells');

    % Perform pulse compression
    % (b) for the quantized phase case
    disp('Performing pulse compression for quantized phase case');
    pc_ref_mapq = fft(pri_rg_mapq.',2*rg_pts-1).';
    pc_ref_map_shiftq = fft(pri_rg_map_shiftq.',2*rg_pts-1).';
    pri_rg_map3 = zeros(size(pri_rg_mapq));
    pri_rg_map4 = zeros(size(pri_rg_mapq));

    %--- Compress the original signals

    for idx = 1:dp_pts
        tmp = cref.*pc_ref_mapq(idx,:);
        tmp1 = fftshift(ifft(tmp));
        pri_rg_map3(idx,:) =  tmp1(rg_pts:end);
    end

    %--- Compress the Doppler shifted signals

    for idx = 1:dp_pts
        tmp = cref.*pc_ref_map_shiftq(idx,:);
        tmp1 = fftshift(ifft(tmp));
        pri_rg_map4(idx,:) =  tmp1(rg_pts:end);
    end

    % Compute the rg-dop map
    disp('Plotting ... r-d map');

    dp_rg_mapq = fft(pri_rg_map3);
    dp_rg_map_shiftq = fft(pri_rg_map4);

    [lx,ly] = size(dp_rg_mapq);
    rax = 1:(length(ly));
    dax = 0:(length(lx))-1;

    dpyq = abs(dp_rg_mapq);
    dpy_shiftq = abs(dp_rg_map_shiftq);

    % -- Simulation of phase quantizing DRFM
    % Now convert amplitude to phase.
```

```matlab
    % Convert phase to positive numbers between 0-360deg, so do not need
to handle
    % negative numbers in Altera.
    pri_rg_mapq_angle = mod(pri_rg_phaseq,2*pi);
    pri_rg_mapq_shift_angle = angle(pri_rg_map_shiftq);

    f2 = fopen('rawint.txt','w');
    [lx,ly] = size(pri_rg_mapq_angle);
    deltaDegrees = 2*pi/(2^nbitsdop);
    for i = 1:lx
        int_raw = round(pri_rg_mapq_angle(i,1:num_chirp_samples-
1)/deltaDegrees); % need to store in Visual basic text file format
        fprintf(f2,'%d,',int_raw);
        int_raw =
round(pri_rg_mapq_angle(i,num_chirp_samples)/deltaDegrees);
        fprintf(f2,'%d\r\n',int_raw);
    end;
    fclose(f2);
    q = 2*pi/(2^nbitsph);
    pri_rg_mapq_drfm = exp(sqrt(-1)*(round(pri_rg_mapq_angle/q))*q);
    pri_rg_mapq_shift_drfm = exp(sqrt(-
1)*(round(pri_rg_mapq_shift_angle/q))*q);

    % Perform pulse compression
    % (c) for the quantized phase case with phase DFRM model
    disp('Performing pulse compression for quantized phase case
(simulates phase DFRM effects)');
    pc_ref_mapq_drfm = fft(pri_rg_mapq_drfm.',2*rg_pts-1).';
    pc_ref_mapq_shift_drfm = fft(pri_rg_mapq_shift_drfm.',2*rg_pts-1).';
    pri_rg_map5 = zeros(size(pri_rg_mapq_drfm));
    pri_rg_map6 = zeros(size(pri_rg_mapq_shift_drfm));

    %--- Compress the original signals

    for idx = 1:dp_pts
        tmp = cref.*pc_ref_mapq_drfm(idx,:);
        tmp1 = fftshift(ifft(tmp));
        pri_rg_map5(idx,:) =  tmp1(rg_pts:end);
    end

    %--- Compress the Doppler shifted signals

    for idx = 1:dp_pts
        tmp = cref.*pc_ref_mapq_shift_drfm(idx,:);
        tmp1 = fftshift(ifft(tmp));
        pri_rg_map6(idx,:) =  tmp1(rg_pts:end);
    end

    % Compute the rg-dop map
    disp('Plotting ... r-d map');

    dp_rg_mapq_drfm = fft(pri_rg_map5);
    dp_rg_map_shiftq_drfm = fft(pri_rg_map6);

    [lx,ly] = size(dp_rg_mapq_drfm);
    rax = 1:(length(ly));
```

```
    dax = 0:(length(lx));
    %dax = 0:(length(lx))-1;

    dpyq_drfm = abs(dp_rg_mapq_drfm);
    dpyq_shift_drfm = abs(dp_rg_map_shiftq);

    save plot dpyq dpyq_shift_drfm

end

figure(1); print -dtiff simhost1
```

d. mathostv4b.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% mathostv4b.m
% MAJ Stig Ekestorm, Feb -00
% Modified version of mathostv0.m by SY YEO, Jul -98
%
% Generate pri_dp map and range-Doppler map
% - generates the files for input to hardware
% -- file para.txt contains:
%     line 1: number of range cells
%     line 2: number of pulse in a batch (equals to dp_pts in this
%                 program)
%     line 3: extent of target in cells (n: integer); number of taps in
%                 delay also equals n (pipeline design)
%     line 4: gain1, gain2, ..., gain n (integer)
%     line 4+n+1: phi0 (pulse 1),
%     line 4+n+2: phi1 (pulse 1),
%     line 4+n+targetExtent: phi-targetExtent (pulse 1),
%     line 4+n+targetExtent+1: phi0 (pulse 2),
%     line 4+n+targetExtent+2: phi1 (pulse 2),
%     line 4+n+2*targetExtent: phi-targetExtent (pulse 2),
%
%     ...
%     line 4+n+dp_pts*targetExtent: phi-targetExtent (pulse dp_pts)
%
% -- file raw.txt contains the instantaneous phases of simulated DFRM-
%     data (quantized to 45deg step):
%     line 1: pulse 1 (integer)
%     line 2: pulse 2
%
%     ....
%     line dp_pts: pulse dp_pts
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear

global sorm
global dp_pts
global rg_pts
global doppler_inc
global printdata

set(0,'defaultAxesFontSize',8);
```

```
noplot = 0;
Ncontours = 20;

% Parameters
%bw = 100e6;
%pwc = 1/(1.25*bw);   % compressed pulsewidth
%pw =0.5e-6;
%prf = 2e3; pri = 1/prf;
%mu = 2*pi*bw/pw;
%fs = 1.25*bw; Ts = 1/fs;
%snr = 0;

bw = 100e6;
bw2 = 1.25*bw;        % bandwidth of the chirp signal, delta
pwc = 1/(1.25*bw);    % compressed pulsewidth
pw =0.5e-6;           % uncompressed pulsewidth
prf = 2e3;            % PRF
pri = 1/prf;          % PRI
k = bw2/pw;           % pulse compression rate, delta / uncompressed
pulsewidth
%mu = 2*pi*bw/pw;
fs = 1.25*bw;         % sampling frequency
Ts = 1/fs;            % sampling time step
snr = 0;              % no extra noise added

% set-up grid
% x-axis(rg), y-axis(dp)
%rg_pts = 200;
%dp_pts = 64;

pri_rg_map = zeros(dp_pts,rg_pts);
pri_rg_mapq = zeros(dp_pts,rg_pts);
pri_rg_map_shift = zeros(dp_pts,rg_pts);
pri_rg_map_shiftq = zeros(dp_pts,rg_pts);
pri_rg_phaseq = zeros(dp_pts,rg_pts);

% insert waveform into grid;
load -ascii sigpar1
sigpar = sigpar1';
doppler_inc = sigpar(4,:);
%sigpar([2 4],:) = sigpar([2 4],:)*prf/dp_pts;
sigpar(2,:) = sigpar(2,:) * 0 + 1000-9*31.25;    %to create an
artificial Doppler offset for the Ship Case, 32 Taps
%sigpar(2,:) = sigpar(2,:) * 0 + 1000-5*31.25;   %to create an
artificial Doppler offset for the Ship Case, 16 Taps
[lxs,lys] = size(sigpar);
%t0 = 0:Ts:pw-Ts;
t0 = Ts:Ts:pw;

%for the reduced 2-Tap T-Spice simulation
%TsNew=4*1/fs;
%tnew = 0:TsNew:pw-TsNew;
%tnew = TsNew:TsNew:pw;
%t0 = tnew;
```

```
num_chirp_samples = length(t0);
if ((num_chirp_samples + lxs) > rg_pts)
    disp('Warning : Chirp is clipped - set grid size larger');
end


% open files for writing
f1 = fopen('para.txt','w');
fprintf(f1,'%d\r\n',num_chirp_samples);     % number of range cells
fprintf(f1,'%d\r\n',dp_pts);                % number of Doppler cells
fprintf(f1,'%d\r\n',lys);                   % target extent

% adjustment to correct multiplication factors for the amplitude (gain)
value
for i = 1:lys
    switch sigpar(3,i)
    case {1}
        sigpar(3,i)=1; % no shift, multiplication by 1, hardware bit "00"
    case {2}
        sigpar(3,i)=2; % shift by 1, multiplication by 2, hardware bit
"01"
    case {3}
        sigpar(3,i)=4; % shift by 2, multiplication by 4, hardware bit
"10"
    case {4}
        sigpar(3,i)=8; % shift by 3, multiplication by 8, hardware bit
"11"
    end
    fprintf(f1,'%d\r\n',sigpar(3,i));    % gain1, gain2, ..., gainN
end

nbitsph = 3;
nbitsdop = 5;
nbitsamp = 8;
b = 2*pi/(2^nbitsph);
a = 2*pi/(2^nbitsamp);
p = 2*pi/(2^nbitsdop);
%for the reduced 2-Tap T-Spice simulation, Nov -99
%p = 2*pi/(dp_pts);

for idx1 = 1:dp_pts        % Repeat for total number of pulses within
batch
    t1 = t0 + (idx1)*pri;
    %t1 = t0 + (idx1-1)*pri;
    for idx = 1:lys
        %**** approximation used here, assume phase change due to Doppler
within a chirp is constant
        %**** since the Doppler is tens of hertz compared to the MHz
chirp bandwidth

        oldphase = 2*pi*((k*t1.*t1)/2 + sigpar(2,idx)*t1);
        %oldphase = mu*t1.*t1/2 + 2*pi*sigpar(2,idx)*t1;
        %oldphase = mu*t1.*t1/2 + 2*pi*sigpar(2,idx)*(idx1-1)*pri;


        oldphase = mod(oldphase,2*pi);
```

```matlab
      % quantize the oldphase to 1 of 8 phases
      int_oldphase = round(oldphase/b);
      oldphaseq = b*int_oldphase;        % quantize the phase
      xc = exp(sqrt(-1)*oldphase);
      lx = (sigpar(1,idx)):(sigpar(1,idx))+length(xc)-1;
      pri_rg_map(idx1,lx) = xc+pri_rg_map(idx1,lx);
      pri_rg_phaseq(idx1,lx) = int_oldphase;

      xcq = exp(sqrt(-1)*oldphaseq);
      xcq = p*round(xcq/p);          % quantize the phase
      pri_rg_mapq(idx1,lx) = xcq+pri_rg_mapq(idx1,lx);
      % phase focusing
      dopphase = 2*pi*sigpar(4,idx)*(idx1)*pri; % approximation used
here
      %dopphase = 2*pi*sigpar(4,idx)*(idx1-1)*pri; % approximation used
here
      newphase = oldphase + dopphase*ones(size(oldphase));
      xI = cos(newphase);
      xQ = sin(newphase);
      x1 = sigpar(3,idx)*(xI+sqrt(-1)*xQ);
      pri_rg_map_shift(idx1,lx) = pri_rg_map_shift(idx1,lx) + x1;

      int_dopphaseq = round(dopphase/p);
      dopphaseq = int_dopphaseq*p;
      newphaseq = oldphaseq + dopphaseq;
      xI = cos(newphaseq);
      xQ = sin(newphaseq);
      xI = round(xI/a)*a;
      xQ = round(xQ/a)*a;
      x1 = sigpar(3,idx)*(xI+sqrt(-1)*xQ);
      pri_rg_map_shiftq(idx1,lx) = pri_rg_map_shiftq(idx1,lx) + x1;

      % store the dopphase value (ignore intrapulse phase change since
it is small)
      %fprintf(f1,'%d\r\n',int_dopphaseq);            %originals
code, incrementation of phase modulation coefficients
      %fprintf(f1,'%d\r\n',mod(int_dopphaseq,32));       %to get
true phase modulation coefficients each PRI
      fprintf(f1,'%d\r\n',mod(2*fix(int_dopphaseq/2),32));  %to
represent phase modulation coefficients using 4-bits words

    end
end
fclose(f1);

noise = randn(size(pri_rg_map))*c_snr(snr); noise = 0;
pri_rg_map = pri_rg_map + noise;
pri_rg_map_shift = pri_rg_map_shift + noise;

%------
save pulse1 pri_rg_map_shiftq

%---------------
% Perform pulse compression
% (a) for the non-quantized phase case
disp('Creating reference waveform');
```

```matlab
ph = 2*pi*((k*t1.*t1)/2 + sigpar(2,1)*t1);
crefc = sqrt(j*k)*exp(j*ph);
cref = conj(fft(crefc,2*rg_pts-1));
save pc_ref cref t1

%ph = (mu*t1.*t1/2);
%crefc = exp(sqrt(-1)*ph);
%cref = conj(fft(crefc,2*rg_pts-1));
%save pc_ref cref t1
pc_ref_map = fft(pri_rg_map.',2*rg_pts-1).';
pc_ref_map_shift = fft(pri_rg_map_shift.',2*rg_pts-1).';

disp('Performing pulse compression');
pri_rg_map1 = zeros(size(pri_rg_map));
pri_rg_map2 = zeros(size(pri_rg_map));

%--- Compress the original signals
for idx = 1:dp_pts
    tmp = cref.*pc_ref_map(idx,:);
    tmp1 = fftshift(ifft(tmp));
    pri_rg_map1(idx,:) =  tmp1(rg_pts:end);
end

%--- Compress the Doppler shifted signals

for idx = 1:dp_pts
    tmp = cref.*pc_ref_map_shift(idx,:);
    tmp1 = fftshift(ifft(tmp));
    pri_rg_map2(idx,:) =  tmp1(rg_pts:end);
end

% Compute the rg-dop map
disp('Plotting ... r-d map');

dp_rg_map = fft(pri_rg_map1,dp_pts);
dp_rg_map_shift = fft(pri_rg_map2,dp_pts);

[lx,ly] = size(dp_rg_map);
rax = 1:(length(ly));
dax = 0:(length(lx))-1;

dpy = abs(dp_rg_map);
dpy_shift = abs(dp_rg_map_shift);

if (noplot == 0)
   figure(1);

   subplot(2,1,1);
   h = contour(dpy,Ncontours); grid
   title('a. Original Rd-Dp Map');
   axis([1 62 0 dp_pts])
   xlabel('Down Range Cells');   ylabel('Cross Range Cells');
   subplot(2,1,2);
   h = contour(dpy_shift,Ncontours); grid
   axis([1 62 0 dp_pts])
```

```matlab
title('b. Amplitude and Doppler Modulated Rd-Dp Map');
xlabel('Down Range Cells');   ylabel('Cross Range Cells');

% Perform pulse compression
% (b) for the quantized phase case
disp('Performing pulse compression for quantized phase case');
pc_ref_mapq = fft(pri_rg_mapq.',2*rg_pts-1).';
pc_ref_map_shiftq = fft(pri_rg_map_shiftq.',2*rg_pts-1).';
pri_rg_map3 = zeros(size(pri_rg_mapq));
pri_rg_map4 = zeros(size(pri_rg_mapq));

%--- Compress the original signals

for idx = 1:dp_pts
    tmp = cref.*pc_ref_mapq(idx,:);
    tmp1 = fftshift(ifft(tmp));
    pri_rg_map3(idx,:) =  tmp1(rg_pts:end);
end

%--- Compress the Doppler shifted signals

for idx = 1:dp_pts
    tmp = cref.*pc_ref_map_shiftq(idx,:);
    tmp1 = fftshift(ifft(tmp));
    pri_rg_map4(idx,:) =  tmp1(rg_pts:end);
end

% Compute the rg-dop map
disp('Plotting ... r-d map');

dp_rg_mapq = fft(pri_rg_map3);
dp_rg_map_shiftq = fft(pri_rg_map4);

[lx,ly] = size(dp_rg_mapq);
rax = 1:(length(ly));
dax = 0:(length(lx))-1;

dpyq = abs(dp_rg_mapq);
dpy_shiftq = abs(dp_rg_map_shiftq);

% -- Simulation of phase quantizing DRFM
% Now convert amplitude to phase.
% Convert phase to positive numbers between 0-360deg, so do not need
to handle
% negative numbers in Altera.
pri_rg_mapq_angle = mod(pri_rg_phaseq,2*pi);
pri_rg_mapq_shift_angle = angle(pri_rg_map_shiftq);

f2 = fopen('rawint.txt','w');
[lx,ly] = size(pri_rg_mapq_angle);
deltaDegrees = 2*pi/(2^nbitsdop);
for i = 1:lx
    int_raw = round(pri_rg_mapq_angle(i,1:num_chirp_samples-
1)/deltaDegrees); % need to store in Visual basic text file format
    fprintf(f2,'%d,',int_raw);
```

```matlab
        int_raw =
round(pri_rg_mapq_angle(i,num_chirp_samples)/deltaDegrees);
        fprintf(f2,'%d\r\n',int_raw);
    end;
    fclose(f2);
    q = 2*pi/(2^nbitsph);
    pri_rg_mapq_drfm = exp(sqrt(-1)*(round(pri_rg_mapq_angle/q))*q);
    pri_rg_mapq_shift_drfm = exp(sqrt(-
1)*(round(pri_rg_mapq_shift_angle/q))*q);


    % Perform pulse compression
    % (c) for the quantized phase case with phase DFRM model
    disp('Performing pulse compression for quantized phase case
(simulates phase DFRM effects)');
    pc_ref_mapq_drfm = fft(pri_rg_mapq_drfm.',2*rg_pts-1).';
    pc_ref_mapq_shift_drfm = fft(pri_rg_mapq_shift_drfm.',2*rg_pts-1).';
    pri_rg_map5 = zeros(size(pri_rg_mapq_drfm));
    pri_rg_map6 = zeros(size(pri_rg_mapq_shift_drfm));


    %--- Compress the original signals

    for idx = 1:dp_pts
        tmp = cref.*pc_ref_mapq_drfm(idx,:);
        tmp1 = fftshift(ifft(tmp));
        pri_rg_map5(idx,:) =  tmp1(rg_pts:end);
    end

    %--- Compress the Doppler shifted signals

    for idx = 1:dp_pts
        tmp = cref.*pc_ref_mapq_shift_drfm(idx,:);
        tmp1 = fftshift(ifft(tmp));
        pri_rg_map6(idx,:) =  tmp1(rg_pts:end);
    end

    % Compute the rg-dop map
    disp('Plotting ... r-d map');

    dp_rg_mapq_drfm = fft(pri_rg_map5);
    dp_rg_map_shiftq_drfm = fft(pri_rg_map6);

    [lx,ly] = size(dp_rg_mapq_drfm);
    rax = 1:(length(ly));
    dax = 0:(length(lx));
    %dax = 0:(length(lx))-1;

    dpyq_drfm = abs(dp_rg_mapq_drfm);
    dpyq_shift_drfm = abs(dp_rg_map_shiftq);

    save plot dpyq dpyq_shift_drfm

end

figure(1); print -dtiff simhost1
```

e. simhwchkv4.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% simhwchkv4.m
% MAJ Stig Ekestorm, Feb -00
% Modified version of simhwchkv0.m
% Purpose: This program performs a architectural true simulation of the
% Digital Image Synthesizer hardware
% Modifications will perform "parallel processing" and then "serial
% summation" including:
% - correction at start-up ("initialize outputs from the taps, one tap
%   after another")
% - correction at the end ("shutting down the taps, one tap after
%   another")
% Original file: simhwchk.m by SY YEO, Jul -98
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear

global dp_pts
global rg_pts
global doppler_inc

set(0,'defaultAxesFontSize',8);

noPlot = 0;
Ncontours = 20;

depthLUT = 32;
widthLUTFile = 2; % in units of number of hex digits
widthLUT = 8;  % n bits
ndopbits = 5;

%****************************
% Read from data files
%****************************
% Read from para.dat
%fid = fopen('para.txt','r'); %opens para.txt to be read
%fid = fopen('paraMULTI.txt','r'); %opens paraMULTI.txt to be read
%fid = fopen('paraMULTIq5.txt','r'); %opens paraMULTIq5.txt to be read
%fid = fopen('paraMULTIq4.txt','r'); %opens paraMULTIq4.txt to be read
%fid = fopen('paraMULTIq4NEW.txt','r'); %opens paraMULTIq4NEW.txt to be
read
%fid = fopen('paraMULTIq4Vcase1.txt','r'); %opens paraMULTIq4Vcase.txt
to be read
fid = fopen('paraMULTIq4Vcase2.txt','r'); %opens paraMULTIq4Ship1.txt
to be read

tmp = fscanf(fid,'%f'); %reads in the values, for non-quantized test
case
%tmp = fscanf(fid,'%d'); %reads in the values

nRangeCell = tmp(1); %1st value: 62, represents the number of range
cells
```

```
nDopplerCell = tmp(2); %2nd value: 64, represents the number of radar
pulses
targetExtent = tmp(3); %3rd value: 3, represents the radial length of
the target expressed in number of range cells

gain = tmp(4:4+targetExtent*nDopplerCell-1);
gainRev = reshape(gain,targetExtent,nDopplerCell);
tmp1 = tmp(4+targetExtent*nDopplerCell:end);
phi = reshape(tmp1,targetExtent,nDopplerCell);

%gain = tmp(4:4+targetExtent-1); %4th to 6th values: 1,2,4 - the gain
value for each tap
%gainRev = fliplr(gain);
%tmp1 = tmp(4+targetExtent:end); % 7th to last value: the phase-
increment values for each tap
%phi = reshape(tmp1,targetExtent,nDopplerCell); %3x64 matrix with zeros
in the 1st column
fclose(fid);

% Read from rawint.dat
raw = zeros(nDopplerCell,nRangeCell); %create a 64x62 matrix,
initialized to zeros
fid = fopen('rawint.txt','r'); %open rawint.txt to be read
for j = 1:nDopplerCell
    for k = 1:nRangeCell-1
        raw(j,k) = fscanf(fid,'%d',1);
        comma = fscanf(fid,'%c',1);
    end
    raw(j,nRangeCell) = fscanf(fid,'%d',1);
end
fclose(fid);
[row,col] = size(raw);
raw = [raw,zeros(row,targetExtent-1)]; %raw: 64x64 matrix, last 2
columns with zeros
%raw = [raw,zeros(row,targetExtent)]; %raw: 64x65 matrix, last 3
columns with zeros

% Read from the LUT files.
load -ascii cosine.txt  % variable is cosine
load -ascii sine.txt    % variable is sine

% initialize some intermediate variables and vectors
Tgt_Extent=targetExtent;
DRFM_Phase=raw;
GainRev = gain';
gainRev = gainRev';
Phase_inc=phi;
phiRev = zeros(Tgt_Extent,1);
depthLUT = 32;
phaseAdderOut = zeros(Tgt_Extent,1);
lutOut = zeros(Tgt_Extent,1);
tapOut = zeros(nRangeCell + (Tgt_Extent-1),Tgt_Extent);

% open files to write results to
%f1 = fopen('checkv2.txt','w');  % "scan-path test"
f2 = fopen('Iout.txt','w');      % I-values, final output
```

```matlab
f3 = fopen('Qout.txt','w');      % Q-values, final output
f4 = fopen('Iout_bin.txt','w');  % I-values in 2-complement binary,
final output
f5 = fopen('Qout_bin.txt','w');  % Q-values in 2-complement binary,
final output

% signal processing
for batchCnt = 1:nDopplerCell,

    disp(['Processing Pulse 'num2str(batchCnt)]);

    for intraPulseCnt = 1:(nRangeCell + (Tgt_Extent-1)), % clock cycle

        % --- This part simulates the intra pulse processing in hardware

        %This part does "parallel processing" and then "serial summation"

        % "parallel processing"    .

        % initialize some intermediate variables and vectors
        tap=zeros(1,Tgt_Extent);

        % extraxt DRFM-phase data
        DRFM_data=DRFM_Phase(batchCnt,intraPulseCnt);
        for idx=1:Tgt_Extent,
            tap(idx)=DRFM_data;
        end

        % phase addition (add phase-increment (Doppler offset) to DFRM
phase data)
        phaseAdderOut=tap(1:Tgt_Extent)' + Phase_inc(:,batchCnt);

        % phase-amplitude look-up (to obtain complex time signal)
        %tmp=mod(phaseAdderOut,32)/32*2*pi;                %test case
with non-quantized phase and LUT
        %lutOut = cos(tmp) + sqrt(-1)*sin(tmp);            %test case
with non-quantized phase and LUT
        tmp = mod(phaseAdderOut,depthLUT) + 1;             %original DIS
code
        lutOut = cosine(tmp) + sqrt(-1)*sine(tmp);         %original DIS
code

        % correction at the end ("shutting down the taps one tap after
another")
        if intraPulseCnt>nRangeCell,
            for idx2=1:(intraPulseCnt-nRangeCell),
                lutOut((idx2),:)=0;
            end
        end

        % gain modulation, and storing values in an intermediate matrix
        if intraPulseCnt<=nRangeCell,
            GainOut = gainRev(batchCnt,:)'.*lutOut;
            for idx3=0:Tgt_Extent-1,
                tapOut(intraPulseCnt+idx3,idx3+1)=GainOut(idx3+1);
            end
```

218

```matlab
    end

    % final accumulation - "serial summation"
    % - 1st: extract partial sums (I and Q)
    % - 2nd: extract final sums (I and Q)
    tapNew=tapOut;
    add=0;
    tt=Tgt_Extent;

    if tt>=2,
        while tt>=2,
            add=add+1;
            tapNew(intraPulseCnt,tt-
1)=tapNew(intraPulseCnt,tt)+tapNew(intraPulseCnt,tt-1);
            partial_tapsum(intraPulseCnt,1)=tapNew(intraPulseCnt,tt-1);
            tt=tt-1;
        end
        tt=tt-1;
    end

    if tt==1,
        partial_tapsum(intraPulseCnt,1)=tapOut(intraPulseCnt,1);
    end

    Iout=real(partial_tapsum(intraPulseCnt,1));
    Qout=imag(partial_tapsum(intraPulseCnt,1));

  % write final results (I and Q) to separate files
    format long
    fprintf(f2,'%5.7f\n',Iout);
    fprintf(f3,'%5.7f\n',Qout);
    fprintf(f4,'%d',dec2two(Iout,8,7));
    fprintf(f4,'\r\n');
    fprintf(f5,'%d',dec2two(Qout,8,7));
    fprintf(f5,'\r\n');

  end %intraPulseCnt

  finalAdderOut(batchCnt,:)=conj(partial_tapsum');

end %batchCnt

% close files
fclose(f2);
fclose(f3);
fclose(f4);
fclose(f5);

%****************************
% Pulse Compression
%****************************
%--- Compress the Doppler shifted signals
load pc_ref
priRgMapShift = zeros(nDopplerCell,rg_pts);
tic
pcRefMapShift = fft(finalAdderOut.',2*rg_pts-1).';
```

```
for idx = 1:nDopplerCell
    tmp = cref.*pcRefMapShift(idx,:);
    tmp1 = fftshift(ifft(tmp));
    priRgMapShift(idx,1:end-targetExtent+1) =  tmp1(rg_pts+targetExtent-
1:end);
end
dpRgMapShiftMOD = abs(fft(priRgMapShift));
%dpRgMapShift = abs(fft(priRgMapShift));
toc

dpRgMapShiftMOD4Vcase2=dpRgMapShiftMOD;
finalAdderOutVcase2=finalAdderOut;
save plotMOD4Vcase2 dpRgMapShiftMOD4Vcase2
%dpRgMapShiftMOD4Ship2=dpRgMapShiftMOD;
%save plotMOD4Ship2 dpRgMapShiftMOD4Ship2
%%dpRgMapShiftMOD4NEW=dpRgMapShiftMOD;
%save plotMOD4NEW dpRgMapShiftMOD4NEW
%dpRgMapShiftMOD4=dpRgMapShiftMOD;
%save plotMOD4 dpRgMapShiftMOD4
%dpRgMapShiftMOD5=dpRgMapShiftMOD;
%save plotMOD5 dpRgMapShiftMOD5
%dpRgMapShiftMODnot=dpRgMapShiftMOD;
%save plotMODnot dpRgMapShiftMODnot
save plotMOD dpRgMapShiftMOD

save fAddOut finalAdderOut

%****************************
% Display
%****************************
if (noPlot == 0)
    figure(2);
    load plot.mat
    subplot(2,1,1);
    h = contour(dpyq,Ncontours); grid; axis([1 64 0 dp_pts])
    %h = contour(dpyq_shift_drfm,Ncontours); grid; axis([0 20 0 32])
    title('a. Amplitude and Doppler Modulated Rd-Dp Map (unmodulated /
MATLAB) ');
    xlabel('Down Range Cells');   ylabel('Cross Range Cells');
    axis([1 62 0 dp_pts])
    subplot(2,1,2);
    h = contour(dpRgMapShiftMOD,Ncontours); grid; axis([1 64 0 dp_pts])
    %h = contour(dpRgMapShift,Ncontours); grid; axis([1 20 0 32])
    title('b. Amplitude and Doppler Modulated Rd-Dp Map (Bit-True,
modulated / MATLAB) ');
    xlabel('Down Range Cells');   ylabel('Cross Range Cells');
    axis([1 62 0 dp_pts])
end
```

f. simhwchkv4_write.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% simhwchkv4_write.m
% MAJ Stig Ekestorm, Feb -00
% Modified version of simhwchkv0.m
% Purpose: This program performs a architectural true simulation of the
% Digital Image Synthesizer hardware
% Modifications will perform "parallel processing" and then "serial
% summation" including:
% - correction at start-up ("initialize outputs from the taps, one tap
%   after another")
% - correction at the end ("shutting down the taps, one tap after
%   another")
% Original file: simhwchk.m by SY YEO, Jul -98
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear

global dp_pts
global rg_pts
global doppler_inc

set(0,'defaultAxesFontSize',8);

noPlot = 0;
Ncontours = 20;

depthLUT = 32;
widthLUTFile = 2; % in units of number of hex digits
widthLUT = 8;   % n bits
ndopbits = 5;

%****************************
% Read from data files
%****************************
% Read from para.dat
%fid = fopen('para.txt','r'); %opens para.txt to be read
%fid = fopen('paraMULTI.txt','r'); %opens paraMULTI.txt to be read
%fid = fopen('paraMULTIq5.txt','r'); %opens paraMULTIq5.txt to be read
%fid = fopen('paraMULTIq4.txt','r'); %opens paraMULTIq4.txt to be read
%fid = fopen('paraMULTIq4NEW.txt','r'); %opens paraMULTIq4NEW.txt to be
read
%fid = fopen('paraMULTIq4Ship64a.txt','r'); %opens paraMULTIq4Ship1.txt
to be read
fid = fopen('paraMULTIq4Vcase2.txt','r'); %opens paraMULTIq4Ship1.txt
to be read

%tmp = fscanf(fid,'%f'); %reads in the values, for non-quantized test
case
tmp = fscanf(fid,'%d'); %reads in the values

nRangeCell = tmp(1); %1st value: 62, represents the number of range
cells
```

```matlab
nDopplerCell = tmp(2); %2nd value: 64, represents the number of radar
pulses
targetExtent = tmp(3); %3rd value: 3, represents the radial length of
the target expressed in number of range cells

gain = tmp(4:4+targetExtent*nDopplerCell-1);
gainRev = reshape(gain,targetExtent,nDopplerCell);
tmp1 = tmp(4+targetExtent*nDopplerCell:end);
phi = reshape(tmp1,targetExtent,nDopplerCell);

%gain = tmp(4:4+targetExtent-1); %4th to 6th values: 1,2,4 - the gain
value for each tap
%gainRev = fliplr(gain);
%tmp1 = tmp(4+targetExtent:end); % 7th to last value: the phase-
increment values for each tap
%phi = reshape(tmp1,targetExtent,nDopplerCell); %3x64 matrix with zeros
in the 1st column
fclose(fid);

% Read from rawint.dat
raw = zeros(nDopplerCell,nRangeCell); %create a 64x62 matrix,
initialized to zeros
fid = fopen('rawint.txt','r'); %open rawint.txt to be read
for j = 1:nDopplerCell
    for k = 1:nRangeCell-1
        raw(j,k) = fscanf(fid,'%d',1);
        comma = fscanf(fid,'%c',1);
    end
    raw(j,nRangeCell) = fscanf(fid,'%d',1);
end
fclose(fid);
[row,col] = size(raw);
raw = [raw,zeros(row,targetExtent-1)]; %raw: 64x64 matrix, last 2
columns with zeros
%raw = [raw,zeros(row,targetExtent)]; %raw: 64x65 matrix, last 3
columns with zeros

% Read from the LUT files.
load -ascii cosine.txt  % variable is cosine
load -ascii sine.txt    % variable is sine

% initialize some intermediate variables and vectors
Tgt_Extent=targetExtent;
DRFM_Phase=raw;
GainRev = gain';
gainRev = gainRev';
Phase_inc=phi;
phiRev = zeros(Tgt_Extent,1);
depthLUT = 32;
phaseAdderOut = zeros(Tgt_Extent,1);
lutOut = zeros(Tgt_Extent,1);
tapOut = zeros(nRangeCell + (Tgt_Extent-1),Tgt_Extent);

% open files to write results to
f1 = fopen('checkv4.txt','w');   % "scan-path test"
f2 = fopen('Iout.txt','w');      % I-values, final output
```

222

```matlab
f3 = fopen('Qout.txt','w');       % Q-values, final output
f4 = fopen('Iout_bin.txt','w');   % I-values in 2-complement binary,
final output
f5 = fopen('Qout_bin.txt','w');   % Q-values in 2-complement binary,
final output

% signal processing
for batchCnt = 1:nDopplerCell,

    fprintf(f1,'%s','**********************************************');
    fprintf(f1,'\r\n');
    fprintf(f1,'\r\n');
    fprintf(f1,'%s:%d\r\n','batchCnt (Radar Pulse)',batchCnt);
    fprintf(f1,'\r\n');
    disp(['Processing Pulse 'num2str(batchCnt)]);

    for intraPulseCnt = 1:(nRangeCell + (Tgt_Extent-1)), % clock cycle

        fprintf(f1,'%s:%d\r\n','intraPulseCnt',intraPulseCnt);
        fprintf(f1,'\r\n');
        fprintf(f1,'\t\t\t\t\t\t\t\t\t\t%s:%d\r\n','Clock
Pulse',intraPulseCnt);

        % --- This part simulates the intra pulse processing in hardware

        %This part does "parallel processing" and then "serial summation"

        % "parallel processing"

        % initialize some intermediate variables and vectors
        tap=zeros(1,Tgt_Extent);

        % extraxt DRFM-phase data
        DRFM_data=DRFM_Phase(batchCnt,intraPulseCnt);
        fprintf(f1,'%s\r\n','DRFM-phase Data processed:');
        for idx=1:Tgt_Extent,
            tap(idx)=DRFM_data;
        end
        for idx=1:Tgt_Extent,
            fprintf(f1,'%s%d','   tap',idx-1);
            DRFM_bin=dec2two(DRFM_data,5,0);
            fprintf(f1,':%d\r\n',DRFM_data);
            fprintf(f1,' %d',DRFM_bin(1,2:6));
            fprintf(f1,'\r\n');
            fprintf(f1,'\r\n');
        end

        % phase addition (add phase-increment (Doppler offset) to DFRM
phase data)
        phaseAdderOut=tap(1:Tgt_Extent)' + Phase_inc(:,batchCnt);
        fprintf(f1,'\r\n');
        fprintf(f1,'%s\r\n','Phase-increment:');
        for idx=1:Tgt_Extent,
            fprintf(f1,'%s%d','   tap',idx-1);
            %fprintf(f1,':%d\n',doppler_inc(idx));
            %fprintf(f1,' %d',dec2two(doppler_inc(idx),3,0));
```

223

```
        fprintf(f1,':%d\n',Phase_inc(idx,batchCnt));
        phase_inc=dec2two(Phase_inc(idx,batchCnt),5,0);
        fprintf(f1,' %d',phase_inc(1,2:5));
        fprintf(f1,'\r\n');
        fprintf(f1,'\r\n');
    end
    fprintf(f1,'\r\n');
    fprintf(f1,'%s\r\n','Phase-rotation Register:');
    for idx=1:Tgt_Extent,
        fprintf(f1,'%s%d','    tap',idx-1);
        fprintf(f1,':%d\n',mod(Phase_inc(idx,batchCnt),32));
        Pi=dec2two(mod(Phase_inc(idx,batchCnt),32),5,0);
        fprintf(f1,' %d',Pi(1,2:6));
        fprintf(f1,'\r\n');
        fprintf(f1,'\r\n');
    end
    fprintf(f1,'\r\n');
    fprintf(f1,'\t\t\t\t\t\t\t\t\t%s:%d\r\n','Clock
Pulse',intraPulseCnt+1);
    fprintf(f1,'%s\r\n','After Phase Adder:');
    for idx=1:Tgt_Extent,
        fprintf(f1,'%s%d','    tap',idx-1);
        fprintf(f1,':%d\n',mod(phaseAdderOut(idx),32));
        pAO=dec2two(mod(phaseAdderOut(idx),32),5,0);
        fprintf(f1,' %d',pAO(1,2:6));
        fprintf(f1,'\r\n');
        fprintf(f1,'\r\n');
    end
    fprintf(f1,'\r\n');
    fprintf(f1,'\t\t\t\t\t\t\t\t\t%s:%d\r\n','Clock
Pulse',intraPulseCnt+2);

    % phase-amplitude look-up (to obtain complex time signal)

    %    tmp=mod(phaseAdderOut,32)/32*2*pi;                %test case
with non-quantized phase and LUT
    %    lutOut = cos(tmp) + sqrt(-1)*sin(tmp);            %test case
with non-quantized phase and LUT
    tmp = mod(phaseAdderOut,depthLUT) + 1;                %original DIS
code
    lutOut = cosine(tmp) + sqrt(-1)*sine(tmp);            %original DIS
code

    fprintf(f1,'%s\r\n','After LUT (I-values):');
    for idx=1:Tgt_Extent,
        fprintf(f1,'%s%d','    tap',idx-1);
        fprintf(f1,':%5.7f\n',real(lutOut(idx)));
        fprintf(f1,' %d',dec2two(real(lutOut(idx)),0,7));
        fprintf(f1,'\r\n');
        fprintf(f1,'\r\n');
    end
    fprintf(f1,'%s\r\n','After LUT (Q-values):');
    for idx=1:Tgt_Extent,
        fprintf(f1,'%s%d','    tap',idx-1);
        fprintf(f1,':%5.7f\n',imag(lutOut(idx)));
        fprintf(f1,' %d',dec2two(imag(lutOut(idx)),0,7));
```

```matlab
            fprintf(f1,'\r\n');
            fprintf(f1,'\r\n');
        end

        % correction at the end ("shutting down the taps one tap after
another")
        if intraPulseCnt>nRangeCell,
            for idx2=1:(intraPulseCnt-nRangeCell),
                lutOut((idx2),:)=0;
            end
        end

        fprintf(f1,'\r\n');
        fprintf(f1,'\t\t\t\t\t\t\t\t\t\t%s:%d\r\n','Clock
Pulse',intraPulseCnt+3);

        % gain modulation, and storing values in an intermediate matrix
        fprintf(f1,'%s\r\n','Gain Value:');
        for idx=1:Tgt_Extent,
        %    if GainRev(1,idx)==1,
        %       GainRev2=0;
        %    elseif GainRev(1,idx)==2,
        %       GainRev2=1;
        %    elseif GainRev(1,idx)==4,
        %       GainRev2=2;
        %    elseif GainRev(1,idx)==8,
        %       GainRev2=3;
        %    end
            fprintf(f1,'%s%d','    tap',idx-1);
            fprintf(f1,':%5.7f\n',gainRev(batchCnt,idx));
            GRbin=dec2two(gainRev(batchCnt,idx),2,0);
            fprintf(f1,'  %d',GRbin(1,2:3));
            fprintf(f1,'\r\n');
            fprintf(f1,'\r\n');
        end
        if intraPulseCnt<=nRangeCell,
            GainOut = gainRev(batchCnt,:)'.*lutOut;
            for idx3=0:Tgt_Extent-1,
                tapOut(intraPulseCnt+idx3,idx3+1)=GainOut(idx3+1);
            end
        end
        fprintf(f1,'%s\r\n','After Gain Block (I-values):');
        for idx=1:Tgt_Extent,
            fprintf(f1,'%s%d','    tap',idx-1);
            fprintf(f1,':%5.7f\n',real(GainOut(idx)));
            fprintf(f1,'  %d',dec2two(real(GainOut(idx)),3,7));
            fprintf(f1,'\r\n');
            fprintf(f1,'\r\n');
        end
        fprintf(f1,'%s\r\n','After Gain Block (Q-values):');
        for idx=1:Tgt_Extent,
            fprintf(f1,'%s%d','    tap',idx-1);
            fprintf(f1,':%5.7f\n',imag(GainOut(idx)));
            fprintf(f1,'  %d',dec2two(imag(GainOut(idx)),3,7));
            fprintf(f1,'\r\n');
            fprintf(f1,'\r\n');
```

```
        end

        fprintf(f1,'\r\n');
        fprintf(f1,'\t\t\t\t\t\t\t\t\t\t%s:%d\r\n','Clock
Pulse',intraPulseCnt+4);

        % final accumulation - "serial summation"
        % - 1st: extract partial sums (I and Q)
        % - 2nd: extract final sums (I and Q)
        tapNew=tapOut;
        add=0;
        tt=Tgt_Extent;
        fprintf(f1,'%s\r\n','Final values in each tap (I- and Q-
values):');
        fprintf(f1,'%s%d','    Final I-value in tap',tt-1);
        fprintf(f1,':%5.7f\n',real(tapNew(intraPulseCnt,tt)));
        fprintf(f1,' %d',dec2two(real(tapNew(intraPulseCnt,tt)),8,7));
        fprintf(f1,'\r\n');
        fprintf(f1,'\r\n');
        fprintf(f1,'%s%d','    Final Q-value in tap',tt-1);
        fprintf(f1,':%5.7f\n',imag(tapNew(intraPulseCnt,tt)));
        fprintf(f1,' %d',dec2two(imag(tapNew(intraPulseCnt,tt)),8,7));
        fprintf(f1,'\r\n');
        fprintf(f1,'\r\n');

        if tt>=2,
            while tt>=2,
                add=add+1;
                tapNew(intraPulseCnt,tt-
1)=tapNew(intraPulseCnt,tt)+tapNew(intraPulseCnt,tt-1);
                fprintf(f1,'%s%d','    Final I-value in tap',tt-2);
                fprintf(f1,':%5.7f\n',real(tapNew(intraPulseCnt,tt-1)));
                fprintf(f1,' %d',dec2two(real(tapNew(intraPulseCnt,tt-
1)),8,7));
                fprintf(f1,'\r\n');
                fprintf(f1,'\r\n');
                fprintf(f1,'%s%d','    Final Q-value in tap',tt-2);
                fprintf(f1,':%5.7f\n',imag(tapNew(intraPulseCnt,tt-1)));
                fprintf(f1,' %d',dec2two(imag(tapNew(intraPulseCnt,tt-
1)),8,7));
                fprintf(f1,'\r\n');
                fprintf(f1,'\r\n');
                partial_tapsum(intraPulseCnt,1)=tapNew(intraPulseCnt,tt-1);
                tt=tt-1;
            end
            tt=tt-1;
        end

        if tt==1,
            partial_tapsum(intraPulseCnt,1)=tapOut(intraPulseCnt,1);
        end

        fprintf(f1,'\r\n');
        Iout=real(partial_tapsum(intraPulseCnt,1));
        Qout=imag(partial_tapsum(intraPulseCnt,1));
        fprintf(f1,'%s\r\n','Final Output values (I- and Q-values):');
```

```
      fprintf(f1,'%s%d','  Iout - Final I-value for
intraPulseCnt',intraPulseCnt);
      fprintf(f1,':%5.7f\n',Iout);
      fprintf(f1,' %d',dec2two(Iout,8,7));
      fprintf(f1,'\r\n');
      fprintf(f1,'\r\n');
      fprintf(f1,'%s%d','  Qout - Final Q-value for
intraPulseCnt',intraPulseCnt);
      fprintf(f1,':%5.7f\n',Qout);
      fprintf(f1,' %d',dec2two(Qout,8,7));
      fprintf(f1,'\r\n');
      fprintf(f1,'\r\n');
      fprintf(f1,'%s','----------------------------------------------
');
      fprintf(f1,'\r\n');

  % write final results (I and Q) to separate files
      format long
      fprintf(f2,'%5.7f\n',Iout);
      fprintf(f3,'%5.7f\n',Qout);
      fprintf(f4,'%d',dec2two(Iout,8,7));
      fprintf(f4,'\r\n');
      fprintf(f5,'%d',dec2two(Qout,8,7));
      fprintf(f5,'\r\n');

  end %intraPulseCnt

  finalAdderOut(batchCnt,:)=conj(partial_tapsum');

end %batchCnt

% close files
fclose(f1);
fclose(f2);
fclose(f3);
fclose(f4);
fclose(f5);

%****************************
% Pulse Compression
%****************************
%--- Compress the Doppler shifted signals
load pc_ref
priRgMapShift = zeros(nDopplerCell,rg_pts);
tic
pcRefMapShift = fft(finalAdderOut.',2*rg_pts-1).';
for idx = 1:nDopplerCell
    tmp = cref.*pcRefMapShift(idx,:);
    tmp1 = fftshift(ifft(tmp));
    priRgMapShift(idx,1:end-targetExtent+1) =  tmp1(rg_pts+targetExtent-
1:end);
end
dpRgMapShiftMOD = abs(fft(priRgMapShift));
%dpRgMapShift = abs(fft(priRgMapShift));
toc
```

227

```
dpRgMapShiftMOD4Ship64a=dpRgMapShiftMOD;
finalAdderOut64a = finalAdderOut;
save plotMOD4Ship64a dpRgMapShiftMOD4Ship64a finalAdderOut64a
%dpRgMapShiftMOD4Ship2=dpRgMapShiftMOD;
%save plotMOD4Ship2 dpRgMapShiftMOD4Ship2
%%dpRgMapShiftMOD4NEW=dpRgMapShiftMOD;
%save plotMOD4NEW dpRgMapShiftMOD4NEW
%dpRgMapShiftMOD4=dpRgMapShiftMOD;
%save plotMOD4 dpRgMapShiftMOD4
%dpRgMapShiftMOD5=dpRgMapShiftMOD;
%save plotMOD5 dpRgMapShiftMOD5
%dpRgMapShiftMODnot=dpRgMapShiftMOD;
%save plotMODnot dpRgMapShiftMODnot
save plotMOD dpRgMapShiftMOD
save fAddOut finalAdderOut

%****************************
% Display
%****************************
if (noPlot == 0)
   figure(2);
   load plot.mat
   subplot(2,1,1);
   h = contour(dpyq,Ncontours); grid; axis([1 64 0 dp_pts])
   %h = contour(dpyq_shift_drfm,Ncontours); grid; axis([0 20 0 32])
   title('a. Amplitude and Doppler Modulated Rd-Dp Map (unmodulated /
MATLAB) ');
   xlabel('Down Range Cells');   ylabel('Cross Range Cells');
   axis([1 62 0 dp_pts])
   subplot(2,1,2);
   h = contour(dpRgMapShiftMOD,Ncontours); grid; axis([1 64 0 dp_pts])
   %h = contour(dpRgMapShift,Ncontours); grid; axis([1 20 0 32])
   title('b. Amplitude and Doppler Modulated Rd-Dp Map (Bit-True,
modulated / MATLAB) ');
   xlabel('Down Range Cells');   ylabel('Cross Range Cells');
   axis([1 62 0 dp_pts])
end
```

g. simhwchkv2.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% simhwchkv2.m
% MAJ Stig Ekestorm, Sep -99
% Modified version of simhwchkv0.m
% Purpose: This program performs a architectural true simulation of the
% Digital Image Synthesizer hardware
% Modifications will perform "parallel processing" and then "serial
% summation" including:
% - correction at start-up ("initialize outputs from the taps, one tap
%   after another")
% - correction at the end ("shutting down the taps, one tap after
%   another")
% Original file: simhwchk.m by SY YEO, Jul -98
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

228

```
clear

global dp_pts
global rg_pts
global doppler_inc

set(0,'defaultAxesFontSize',8);

noPlot = 0;
Ncontours = 20;

depthLUT = 32;
widthLUTFile = 2; % in units of number of hex digits
widthLUT = 8;   % n bits
ndopbits = 5;

%****************************
% Read from data files
%****************************
% Read from para.dat
fid = fopen('para.txt','r'); %opens para.txt to be read
%fid = fopen('paraMULTI.txt','r'); %opens paraMULTI.txt to be read
%fid = fopen('paraMULTIq5.txt','r'); %opens paraMULTIq5.txt to be read
%fid = fopen('paraMULTIq4.txt','r'); %opens paraMULTIq4.txt to be read
%fid = fopen('paraMULTIq4NEW.txt','r'); %opens paraMULTIq4NEW.txt to be
read
%fid = fopen('paraMULTIq4Ship1.txt','r'); %opens paraMULTIq4Ship1.txt
to be read

%tmp = fscanf(fid,'%f'); %reads in the values, for non-quantized test
case
tmp = fscanf(fid,'%d'); %reads in the values

nRangeCell = tmp(1); %1st value: 62, represents the number of range
cells
nDopplerCell = tmp(2); %2nd value: 64, represents the number of radar
pulses
targetExtent = tmp(3); %3rd value: 3, represents the radial length of
the target expressed in number of range cells
gain = tmp(4:4+targetExtent-1); %4th to 6th values: 1,2,4 - the gain
value for each tap
gainRev = fliplr(gain);
tmp1 = tmp(4+targetExtent:end); % 7th to last value: the phase-
increment values for each tap
phi = reshape(tmp1,targetExtent,nDopplerCell); %3x64 matrix with zeros
in the 1st column
fclose(fid);

% Read from rawint.dat
raw = zeros(nDopplerCell,nRangeCell); %create a 64x62 matrix,
initialized to zeros
fid = fopen('rawint.txt','r'); %open rawint.txt to be read
for j = 1:nDopplerCell
    for k = 1:nRangeCell-1
        raw(j,k) = fscanf(fid,'%d',1);
```

```matlab
        comma = fscanf(fid,'%c',1);
    end
    raw(j,nRangeCell) = fscanf(fid,'%d',1);
end
fclose(fid);
[row,col] = size(raw);
raw = [raw,zeros(row,targetExtent-1)]; %raw: 64x64 matrix, last 2
columns with zeros
%raw = [raw,zeros(row,targetExtent)]; %raw: 64x65 matrix, last 3
columns with zeros

% Read from the LUT files.
load -ascii cosine.txt  % variable is cosine
load -ascii sine.txt    % variable is sine

% initialize some intermediate variables and vectors
Tgt_Extent=targetExtent;
DRFM_Phase=raw;
GainRev = gain';
Phase_inc=phi;
phiRev = zeros(Tgt_Extent,1);
depthLUT = 32;
phaseAdderOut = zeros(Tgt_Extent,1);
lutOut = zeros(Tgt_Extent,1);
tapOut = zeros(nRangeCell + (Tgt_Extent-1),Tgt_Extent);

% open files to write results to
%f1 = fopen('checkv2.txt','w');  % "scan-path test"
f2 = fopen('Iout.txt','w');       % I-values, final output
f3 = fopen('Qout.txt','w');       % Q-values, final output
f4 = fopen('Iout_bin.txt','w');   % I-values in 2-complement binary,
final output
f5 = fopen('Qout_bin.txt','w');   % Q-values in 2-complement binary,
final output

% signal processing
for batchCnt = 1:nDopplerCell,

    disp(['Processing Pulse 'num2str(batchCnt)]);

    for intraPulseCnt = 1:(nRangeCell + (Tgt_Extent-1)), % clock cycle

        % --- This part simulates the intra pulse processing in hardware

        %This part does "parallel processing" and then "serial summation"

        % "parallel processing"

        % initialize some intermediate variables and vectors
        tap=zeros(1,Tgt_Extent);

        % extraxt DRFM-phase data
        DRFM_data=DRFM_Phase(batchCnt,intraPulseCnt);
        for idx=1:Tgt_Extent,
            tap(idx)=DRFM_data;
        end
```

```
      % phase addition (add phase-increment (Doppler offset) to DFRM
phase data)
      phaseAdderOut=tap(1:Tgt_Extent)' + Phase_inc(:,batchCnt);

      % phase-amplitude look-up (to obtain complex time signal)

      %tmp=mod(phaseAdderOut,32)/32*2*pi;              %test case
with non-quantized phase and LUT
      %lutOut = cos(tmp) + sqrt(-1)*sin(tmp);          %test case
with non-quantized phase and LUT
      tmp = mod(phaseAdderOut,depthLUT) + 1;           %original DIS
code
      lutOut = cosine(tmp) + sqrt(-1)*sine(tmp);       %original DIS
code

      % correction at the end ("shutting down the taps one tap after
another")
      if intraPulseCnt>nRangeCell,
         for idx2=1:(intraPulseCnt-nRangeCell),
            lutOut((idx2),:)=0;
         end
      end

      % gain modulation, and storing values in an intermediate matrix
      for idx=1:Tgt_Extent,
         if GainRev(1,idx)==1,
            GainRev2=0;
         elseif GainRev(1,idx)==2,
            GainRev2=1;
         elseif GainRev(1,idx)==4,
            GainRev2=2;
         elseif GainRev(1,idx)==8,
            GainRev2=3;
         end
      end
      if intraPulseCnt<=nRangeCell,
         GainOut = GainRev.*lutOut';
         for idx3=0:Tgt_Extent-1,
            tapOut(intraPulseCnt+idx3,idx3+1)=GainOut(idx3+1);
         end
      end

      % final accumulation - "serial summation"
      % - 1st: extract partial sums (I and Q)
      % - 2nd: extract final sums (I and Q)
      tapNew=tapOut;
      add=0;
      tt=Tgt_Extent;

      if tt>=2,
         while tt>=2,
            add=add+1;
            tapNew(intraPulseCnt,tt-
1)=tapNew(intraPulseCnt,tt)+tapNew(intraPulseCnt,tt-1);
            partial_tapsum(intraPulseCnt,1)=tapNew(intraPulseCnt,tt-1);
```

```
            tt=tt-1;
        end
        tt=tt-1;
    end

    if tt==1,
        partial_tapsum(intraPulseCnt,1)=tapOut(intraPulseCnt,1);
    end

    Iout=real(partial_tapsum(intraPulseCnt,1));
    Qout=imag(partial_tapsum(intraPulseCnt,1));

% write final results (I and Q) to separate files
    format long
    fprintf(f2,'%5.7f\n',Iout);
    fprintf(f3,'%5.7f\n',Qout);
    fprintf(f4,'%d',dec2two(Iout,8,7));
    fprintf(f4,'\r\n');
    fprintf(f5,'%d',dec2two(Qout,8,7));
    fprintf(f5,'\r\n');

    end %intraPulseCnt

    finalAdderOut(batchCnt,:)=partial_tapsum';

end %batchCnt

% close files
%fclose(f1);
fclose(f2);
fclose(f3);
fclose(f4);
fclose(f5);

%****************************
% Pulse Compression
%****************************
%--- Compress the Doppler shifted signals
load pc_ref
priRgMapShift = zeros(nDopplerCell,rg_pts);
tic
pcRefMapShift = fft(finalAdderOut.',2*rg_pts-1).';
for idx = 1:nDopplerCell
    tmp = cref.*pcRefMapShift(idx,:);
    tmp1 = fftshift(ifft(tmp));
    priRgMapShift(idx,1:end-targetExtent+1) =  tmp1(rg_pts+targetExtent-
1:end);
end
dpRgMapShiftMOD = abs(fft(priRgMapShift));
%dpRgMapShift = abs(fft(priRgMapShift));
toc

dpRgMapShiftMOD4Ship1b=dpRgMapShiftMOD;
save plotMOD4Ship1b dpRgMapShiftMOD4Ship1b
%dpRgMapShiftMOD4Ship2=dpRgMapShiftMOD;
%save plotMOD4Ship2 dpRgMapShiftMOD4Ship2
```

```
%%dpRgMapShiftMOD4NEW=dpRgMapShiftMOD;
%save plotMOD4NEW dpRgMapShiftMOD4NEW
%dpRgMapShiftMOD4=dpRgMapShiftMOD;
%save plotMOD4 dpRgMapShiftMOD4
%dpRgMapShiftMOD5=dpRgMapShiftMOD;
%save plotMOD5 dpRgMapShiftMOD5
%dpRgMapShiftMODnot=dpRgMapShiftMOD;
%save plotMODnot dpRgMapShiftMODnot
save plotMOD dpRgMapShiftMOD

save fAddOut finalAdderOut

%****************************
% Display
%****************************
if (noPlot == 0)
   figure(2);
   load plot.mat
   subplot(2,1,1);
   h = contour(dpyq,Ncontours); grid; axis([1 64 0 dp_pts])
   %h = contour(dpyq_shift_drfm,Ncontours); grid; axis([0 20 0 32])
   title('a. Amplitude and Doppler Modulated Rd-Dp Map (unmodulated /
MATLAB) ');
   xlabel('Down Range Cells');   ylabel('Cross Range Cells');
   axis([1 62 0 dp_pts])
   subplot(2,1,2);
   h = contour(dpRgMapShiftMOD,Ncontours); grid; axis([1 64 0 dp_pts])
   %h = contour(dpRgMapShift,Ncontours); grid; axis([1 20 0 32])
   title('b. Amplitude and Doppler Modulated Rd-Dp Map (Bit-True,
modulated / MATLAB) ');
   xlabel('Down Range Cells');   ylabel('Cross Range Cells');
   axis([1 62 0 dp_pts])
end
```

h. simhwchkv2_write.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% simhwchkv2_write.m
% MAJ Stig Ekestorm, Sep -99
% Modified version of simhwchkv0.m
% Purpose: This program performs a architectural true simulation of the
% Digital Image Synthesizer hardware
% Modifications will perform "parallel processing" and then "serial
% summation" including:
% - correction at start-up ("initialize outputs from the taps, one tap
%   after another")
% - correction at the end ("shutting down the taps, one tap after
%   another")
% Original file: simhwchk.m by SY YEO, Jul -98
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear

global dp_pts
```

```matlab
global rg_pts
global doppler_inc

set(0,'defaultAxesFontSize',8);

noPlot = 0;
Ncontours = 20;

depthLUT = 32;
widthLUTFile = 2; % in units of number of hex digits
widthLUT = 8;  % n bits
ndopbits = 5;

%****************************
% Read from data files
%****************************
% Read from para.dat
fid = fopen('para.txt','r'); %opens para.txt to be read
%fid = fopen('paraMULTI.txt','r'); %opens paraMULTI.txt to be read
%fid = fopen('paraMULTIq5.txt','r'); %opens paraMULTIq5.txt to be read
%fid = fopen('paraMULTIq4.txt','r'); %opens paraMULTIq4.txt to be read
%fid = fopen('paraMULTIq4NEW.txt','r'); %opens paraMULTIq4NEW.txt to be
read
%fid = fopen('paraMULTIq4Ship64a.txt','r'); %opens paraMULTIq4Ship1.txt
to be read

%tmp = fscanf(fid,'%f'); %reads in the values, for non-quantized test
case
tmp = fscanf(fid,'%d'); %reads in the values

nRangeCell = tmp(1); %1st value: 62, represents the number of range
cells
nDopplerCell = tmp(2); %2nd value: 64, represents the number of radar
pulses
targetExtent = tmp(3); %3rd value: 3, represents the radial length of
the target expressed in number of range cells
gain = tmp(4:4+targetExtent-1); %4th to 6th values: 1,2,4 - the gain
value for each tap
gainRev = fliplr(gain);
tmp1 = tmp(4+targetExtent:end); % 7th to last value: the phase-
increment values for each tap
phi = reshape(tmp1,targetExtent,nDopplerCell); %3x64 matrix with zeros
in the 1st column
fclose(fid);

% Read from rawint.dat
raw = zeros(nDopplerCell,nRangeCell); %create a 64x62 matrix,
initialized to zeros
fid = fopen('rawint.txt','r'); %open rawint.txt to be read
for j = 1:nDopplerCell
    for k = 1:nRangeCell-1
        raw(j,k) = fscanf(fid,'%d',1);
        comma = fscanf(fid,'%c',1);
    end
    raw(j,nRangeCell) = fscanf(fid,'%d',1);
end
```

```
fclose(fid);
[row,col] = size(raw);
raw = [raw,zeros(row,targetExtent-1)]; %raw: 64x64 matrix, last 2
columns with zeros
%raw = [raw,zeros(row,targetExtent)]; %raw: 64x65 matrix, last 3
columns with zeros

% Read from the LUT files.
load -ascii cosine.txt   % variable is cosine
load -ascii sine.txt     % variable is sine

% initialize some intermediate variables and vectors
Tgt_Extent=targetExtent;
DRFM_Phase=raw;
GainRev = gain';
Phase_inc=phi;
phiRev = zeros(Tgt_Extent,1);
depthLUT = 32;
phaseAdderOut = zeros(Tgt_Extent,1);
lutOut = zeros(Tgt_Extent,1);
tapOut = zeros(nRangeCell + (Tgt_Extent-1),Tgt_Extent);

% open files to write results to
f1 = fopen('checkv2.txt','w');    % "scan-path test"
f2 = fopen('Iout.txt','w');       % I-values, final output
f3 = fopen('Qout.txt','w');       % Q-values, final output
f4 = fopen('Iout_bin.txt','w');   % I-values in 2-complement binary,
final output
f5 = fopen('Qout_bin.txt','w');   % Q-values in 2-complement binary,
final output

% signal processing
for batchCnt = 1:nDopplerCell,

    fprintf(f1,'%s','***********************************************');
    fprintf(f1,'\r\n');
    fprintf(f1,'\r\n');
    fprintf(f1,'%s:%d\r\n','batchCnt (Radar Pulse)',batchCnt);
    fprintf(f1,'\r\n');
    disp(['Processing Pulse 'num2str(batchCnt)]);

    for intraPulseCnt = 1:(nRangeCell + (Tgt_Extent-1)), % clock cycle

        fprintf(f1,'%s:%d\r\n','intraPulseCnt',intraPulseCnt);
        fprintf(f1,'\r\n');
        fprintf(f1,'\t\t\t\t\t\t\t\t\t\t%s:%d\r\n','Clock
Pulse',intraPulseCnt);

        % --- This part simulates the intra pulse processing in hardware

        %This part does "parallel processing" and then "serial summation"

        % "parallel processing"

        % initialize some intermediate variables and vectors
        tap=zeros(1,Tgt_Extent);
```

235

```matlab
% extraxt DRFM-phase data
DRFM_data=DRFM_Phase(batchCnt,intraPulseCnt);
fprintf(f1,'%s\r\n','DRFM-phase Data processed:');
for idx=1:Tgt_Extent,
    tap(idx)=DRFM_data;
end
for idx=1:Tgt_Extent,
    fprintf(f1,'%s%d','    tap',idx-1);
    DRFM_bin=dec2two(DRFM_data,5,0);
    fprintf(f1,':%d\r\n',DRFM_data);
    fprintf(f1,' %d',DRFM_bin(1,2:6));
    fprintf(f1,'\r\n');
    fprintf(f1,'\r\n');
end


% phase addition (add phase-increment (Doppler offset) to DFRM
phase data)
phaseAdderOut=tap(1:Tgt_Extent)' + Phase_inc(:,batchCnt);
fprintf(f1,'\r\n');
fprintf(f1,'%s\r\n','Phase-increment:');
for idx=1:Tgt_Extent,
    fprintf(f1,'%s%d','    tap',idx-1);
    %fprintf(f1,':%d\n',doppler_inc(idx));
    %fprintf(f1,' %d',dec2two(doppler_inc(idx),3,0));
    fprintf(f1,':%d\n',Phase_inc(idx,batchCnt));
    phase_inc=dec2two(Phase_inc(idx,batchCnt),5,0);
    fprintf(f1,' %d',phase_inc(1,2:5));
    fprintf(f1,'\r\n');
    fprintf(f1,'\r\n');
end
fprintf(f1,'\r\n');
fprintf(f1,'%s\r\n','Phase-rotation Register:');
for idx=1:Tgt_Extent,
    fprintf(f1,'%s%d','    tap',idx-1);
    fprintf(f1,':%d\n',mod(Phase_inc(idx,batchCnt),32));
    Pi=dec2two(mod(Phase_inc(idx,batchCnt),32),5,0);
    fprintf(f1,' %d',Pi(1,2:6));
    fprintf(f1,'\r\n');
    fprintf(f1,'\r\n');
end
fprintf(f1,'\r\n');
fprintf(f1,'\t\t\t\t\t\t\t\t\t\t%s:%d\r\n','Clock
Pulse',intraPulseCnt+1);
fprintf(f1,'%s\r\n','After Phase Adder:');
for idx=1:Tgt_Extent,
    fprintf(f1,'%s%d','    tap',idx-1);
    fprintf(f1,':%d\n',mod(phaseAdderOut(idx),32));
    pAO=dec2two(mod(phaseAdderOut(idx),32),5,0);
    fprintf(f1,' %d',pAO(1,2:6));
    fprintf(f1,'\r\n');
    fprintf(f1,'\r\n');
end
fprintf(f1,'\r\n');
fprintf(f1,'\t\t\t\t\t\t\t\t\t\t%s:%d\r\n','Clock
Pulse',intraPulseCnt+2);
```

236

```
% phase-amplitude look-up (to obtain complex time signal)

%    tmp=mod(phaseAdderOut,32)/32*2*pi;                    %test case
with non-quantized phase and LUT
%    lutOut = cos(tmp) + sqrt(-1)*sin(tmp);                %test case
with non-quantized phase and LUT
     tmp = mod(phaseAdderOut,depthLUT) + 1;                %original DIS
code
     lutOut = cosine(tmp) + sqrt(-1)*sine(tmp);            %original DIS
code


     fprintf(f1,'%s\r\n','After LUT (I-values):');
     for idx=1:Tgt_Extent,
         fprintf(f1,'%s%d','   tap',idx-1);
         fprintf(f1,':%5.7f\n',real(lutOut(idx)));
         fprintf(f1,' %d',dec2two(real(lutOut(idx)),0,7));
         fprintf(f1,'\r\n');
         fprintf(f1,'\r\n');
     end
     fprintf(f1,'%s\r\n','After LUT (Q-values):');
     for idx=1:Tgt_Extent,
         fprintf(f1,'%s%d','   tap',idx-1);
         fprintf(f1,':%5.7f\n',imag(lutOut(idx)));
         fprintf(f1,' %d',dec2two(imag(lutOut(idx)),0,7));
         fprintf(f1,'\r\n');
         fprintf(f1,'\r\n');
     end


     % correction at the end ("shutting down the taps one tap after
another")
     if intraPulseCnt>nRangeCell,
         for idx2=1:(intraPulseCnt-nRangeCell),
             lutOut((idx2),:)=0;
         end
     end


     fprintf(f1,'\r\n');
     fprintf(f1,'\t\t\t\t\t\t\t\t\t\t%s:%d\r\n','Clock
Pulse',intraPulseCnt+3);


     % gain modulation, and storing values in an intermediate matrix
     fprintf(f1,'%s\r\n','Gain Value:');
     for idx=1:Tgt_Extent,
         if GainRev(1,idx)==1,
             GainRev2=0;
         elseif GainRev(1,idx)==2,
             GainRev2=1;
         elseif GainRev(1,idx)==4,
             GainRev2=2;
         elseif GainRev(1,idx)==8,
             GainRev2=3;
         end
         fprintf(f1,'%s%d','   tap',idx-1);
         fprintf(f1,':%5.7f\n',GainRev(1,idx));
         GRbin=dec2two(GainRev2,2,0);
```

237

```
        fprintf(f1,' %d',GRbin(1,2:3));
        fprintf(f1,'\r\n');
        fprintf(f1,'\r\n');
    end
    if intraPulseCnt<=nRangeCell,
        GainOut = GainRev'.*lutOut;
        for idx3=0:Tgt_Extent-1,
            tapOut(intraPulseCnt+idx3,idx3+1)=GainOut(idx3+1);
        end
    end
    fprintf(f1,'%s\r\n','After Gain Block (I-values):');
    for idx=1:Tgt_Extent,
        fprintf(f1,'%s%d','    tap',idx-1);
        fprintf(f1,':%5.7f\n',real(GainOut(idx)));
        fprintf(f1,' %d',dec2two(real(GainOut(idx)),3,7));
        fprintf(f1,'\r\n');
        fprintf(f1,'\r\n');
    end
    fprintf(f1,'%s\r\n','After Gain Block (Q-values):');
    for idx=1:Tgt_Extent,
        fprintf(f1,'%s%d','    tap',idx-1);
        fprintf(f1,':%5.7f\n',imag(GainOut(idx)));
        fprintf(f1,' %d',dec2two(imag(GainOut(idx)),3,7));
        fprintf(f1,'\r\n');
        fprintf(f1,'\r\n');
    end

    fprintf(f1,'\r\n');
    fprintf(f1,'\t\t\t\t\t\t\t\t\t\t%s:%d\r\n','Clock
Pulse',intraPulseCnt+4);

    % final accumulation - "serial summation"
    % - 1st: extract partial sums (I and Q)
    % - 2nd: extract final sums (I and Q)
    tapNew=tapOut;
    add=0;
    tt=Tgt_Extent;
    fprintf(f1,'%s\r\n','Final values in each tap (I- and Q-
values):');
    fprintf(f1,'%s%d','    Final I-value in tap',tt-1);
    fprintf(f1,':%5.7f\n',real(tapNew(intraPulseCnt,tt)));
    fprintf(f1,' %d',dec2two(real(tapNew(intraPulseCnt,tt)),8,7));
    fprintf(f1,'\r\n');
    fprintf(f1,'\r\n');
    fprintf(f1,'%s%d','    Final Q-value in tap',tt-1);
    fprintf(f1,':%5.7f\n',imag(tapNew(intraPulseCnt,tt)));
    fprintf(f1,' %d',dec2two(imag(tapNew(intraPulseCnt,tt)),8,7));
    fprintf(f1,'\r\n');
    fprintf(f1,'\r\n');

    if tt>=2,
        while tt>=2,
            add=add+1;
            tapNew(intraPulseCnt,tt-
1)=tapNew(intraPulseCnt,tt)+tapNew(intraPulseCnt,tt-1);
            fprintf(f1,'%s%d','    Final I-value in tap',tt-2);
```

```
            fprintf(f1,':%5.7f\n',real(tapNew(intraPulseCnt,tt-1)));
            fprintf(f1,' %d',dec2two(real(tapNew(intraPulseCnt,tt-
1)),8,7));
            fprintf(f1,'\r\n');
            fprintf(f1,'\r\n');
            fprintf(f1,'%s%d','   Final Q-value in tap',tt-2);
            fprintf(f1,':%5.7f\n',imag(tapNew(intraPulseCnt,tt-1)));
            fprintf(f1,' %d',dec2two(imag(tapNew(intraPulseCnt,tt-
1)),8,7));
            fprintf(f1,'\r\n');
            fprintf(f1,'\r\n');
            partial_tapsum(intraPulseCnt,1)=tapNew(intraPulseCnt,tt-1);
            tt=tt-1;
        end
        tt=tt-1;
    end

    if tt==1,
        partial_tapsum(intraPulseCnt,1)=tapOut(intraPulseCnt,1);
    end

    fprintf(f1,'\r\n');
    Iout=real(partial_tapsum(intraPulseCnt,1));
    Qout=imag(partial_tapsum(intraPulseCnt,1));
    fprintf(f1,'%s\r\n','Final Output values (I- and Q-values):');
    fprintf(f1,'%s%d','   Iout - Final I-value for
intraPulseCnt',intraPulseCnt);
    fprintf(f1,':%5.7f\n',Iout);
    fprintf(f1,' %d',dec2two(Iout,8,7));
    fprintf(f1,'\r\n');
    fprintf(f1,'\r\n');
    fprintf(f1,'%s%d','   Qout - Final Q-value for
intraPulseCnt',intraPulseCnt);
    fprintf(f1,':%5.7f\n',Qout);
    fprintf(f1,' %d',dec2two(Qout,8,7));
    fprintf(f1,'\r\n');
    fprintf(f1,'\r\n');
    fprintf(f1,'%s','----------------------------------------------
');
    fprintf(f1,'\r\n');

  % write final results (I and Q) to separate files
    format long
    fprintf(f2,'%5.7f\n',Iout);
    fprintf(f3,'%5.7f\n',Qout);
    fprintf(f4,'%d',dec2two(Iout,8,7));
    fprintf(f4,'\r\n');
    fprintf(f5,'%d',dec2two(Qout,8,7));
    fprintf(f5,'\r\n');

  end %intraPulseCnt

  finalAdderOut(batchCnt,:)=conj(partial_tapsum');

end %batchCnt
```

```
% close files
fclose(f1);
fclose(f2);
fclose(f3);
fclose(f4);
fclose(f5);

%*****************************
% Pulse Compression
%*****************************
%--- Compress the Doppler shifted signals
load pc_ref
priRgMapShift = zeros(nDopplerCell,rg_pts);
tic
pcRefMapShift = fft(finalAdderOut.',2*rg_pts-1).';
for idx = 1:nDopplerCell
    tmp = cref.*pcRefMapShift(idx,:);
    tmp1 = fftshift(ifft(tmp));
    priRgMapShift(idx,1:end-targetExtent+1) =  tmp1(rg_pts+targetExtent-
1:end);
end
dpRgMapShiftMOD = abs(fft(priRgMapShift));
%dpRgMapShift = abs(fft(priRgMapShift));
toc

dpRgMapShiftMOD4Ship64a=dpRgMapShiftMOD;
finalAdderOut64a = finalAdderOut;
save plotMOD4Ship64a dpRgMapShiftMOD4Ship64a finalAdderOut64a
%dpRgMapShiftMOD4Ship2=dpRgMapShiftMOD;
%save plotMOD4Ship2 dpRgMapShiftMOD4Ship2
%%dpRgMapShiftMOD4NEW=dpRgMapShiftMOD;
%save plotMOD4NEW dpRgMapShiftMOD4NEW
%dpRgMapShiftMOD4=dpRgMapShiftMOD;
%save plotMOD4 dpRgMapShiftMOD4
%dpRgMapShiftMOD5=dpRgMapShiftMOD;
%save plotMOD5 dpRgMapShiftMOD5
%dpRgMapShiftMODnot=dpRgMapShiftMOD;
%save plotMODnot dpRgMapShiftMODnot
save plotMOD dpRgMapShiftMOD

save fAddOut finalAdderOut

%*****************************
% Display
%*****************************
if (noPlot == 0)
    figure(2);
    load plot.mat
    subplot(2,1,1);
    h = contour(dpyq,Ncontours); grid; axis([1 64 0 dp_pts])
    %h = contour(dpyq_shift_drfm,Ncontours); grid; axis([0 20 0 32])
    title('a. Amplitude and Doppler Modulated Rd-Dp Map (unmodulated /
MATLAB) ');
    xlabel('Down Range Cells');   ylabel('Cross Range Cells');
    axis([1 62 0 dp_pts])
    subplot(2,1,2);
```

```
    h = contour(dpRgMapShiftMOD,Ncontours); grid; axis([1 64 0 dp_pts])
    %h = contour(dpRgMapShift,Ncontours); grid; axis([1 20 0 32])
    title('b. Amplitude and Doppler Modulated Rd-Dp Map (Bit-True,
modulated / MATLAB) ');
    xlabel('Down Range Cells');   ylabel('Cross Range Cells');
    axis([1 62 0 dp_pts])
end
```

i. plothwv4.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% plothwv4.m
% MAJ Stig Ekestorm, Feb -00
% Modified version of plothwv1.m by Stig Ekestorm, Aug -99
% Original file: plothwv0.m by SY YEO, Aug -98
% This version processes the output from the LUT
% Works in concert with mathostv4.m and simhwchkv4.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear

global hda
global dp_pts
global rg_pts

set(0,'defaultAxesFontSize',7)

noplot = 0;

%to load data from hardware output files
if hda==1,
    load -ascii imagei.txt
    load -ascii imageq.txt
end

fid = fopen('para.txt','r');
tmp = fscanf(fid,'%d');
nRangeCell = tmp(1);
nDopplerCell = tmp(2);
targetExtent = tmp(3);
fclose(fid);

%for getting the data form hardware in the right format
if hda==1,
    image = reshape(image,nRangeCell+(targetExtent-1),nDopplerCell);
    image = imagei - j*imageq;
    image = reshape(image,nRangeCell+(targetExtent-1),nDopplerCell);
%for ASIC simulation
    %image = reshape(image,nRangeCell+targetExtent,nDopplerCell); %for
FPGA 3-tap simulation
    image = image';
end

load fAddOut
```

241

```matlab
if (noplot == 0)
   %****************************
   % Pulse Compression
   %****************************
   %--- Compress the Doppler shifted signals
   figure(3);
   orient tall

   load plot.mat
   load plotMOD.mat

   Ncontours = 9;
   subplot(2,1,1);
   %h = contour(dpyq_shift_drfm,Ncontours); grid, axis([1 62 0 64])
   h = contour(dpRgMapShiftMOD,Ncontours); grid; axis([1 62 0 dp_pts])
   title('a. Amplitude and Doppler Modulated Rd-Dp Map (Bit and
Architecture-True / MATLAB)');
   xlabel('Down Range Cells');   ylabel('Cross Range Cells');

   %to post-process data from hardware
   if hda==1,
      load pc_ref
      priRgMapShift = zeros(nDopplerCell.',rg_pts);
      tic
      pcRefMapShift = fft(image.',2*rg_pts-1).';
      for idx = 1:nDopplerCell
         tmp = cref.*pcRefMapShift(idx,:);
         tmp1 = fftshift(ifft(tmp));
         priRgMapShift(idx,1:end-targetExtent+1) =
tmp1(rg_pts+targetExtent-1:end);
      end
      dpRgMapShift = abs(fft(priRgMapShift));
      toc
   end

   subplot(2,1,2);
   if hda==1,
      h = contour(dpRgMapShift,Ncontours);
      grid, axis([1 62 0 dp_pts])
   end
   title('b. Amplitude and Doppler Modulated Rd-Dp Map (from HARDWARE
output)');
   xlabel('Down Range Cells');   ylabel('Cross Range Cells');
end

figure(3)
print -dtiff hwres

figure(4)
subplot(3,1,1);
h = mesh(dpRgMapShiftMOD); grid;
%h = mesh(dpyq_shift_drfm); grid;
title('a. Amplitude/Doppler Modulated Rd-Dp Map (Bit-True, modulated /
MATLAB)');
xlabel('Down Range Cells');   ylabel('Cross Range Cells'); grid
```

242

```
subplot(3,1,2);
%to plot hardware output
if hda==1,
    h = mesh(dpRgMapShift); grid;
end
title('b. Amplitude/Doppler Modulated Rd-Dp Map (HARDWARE output)');
xlabel('Down Range Cells');   ylabel('Cross Range Cells'); grid
subplot(3,1,3);
%to plot difference between Matlab simulation and hardware output
if hda==1,
    %h = mesh(dpyq_shift_drfm/max(max(dpyq_shift_drfm))-
dpRgMapShift/max(max(dpRgMapShift))); grid;
    h = mesh(dpRgMapShiftMOD-dpRgMapShift); grid; %plot the real
difference, Stig Aug-99
    %h = mesh(dpyq_shift_drfm-dpRgMapShift); grid; %plot the real
difference, Stig Aug-99
end
title('c. Difference');
xlabel('Down Range Cells');   ylabel('Cross Range Cells'); grid
print -dtiff diffplot


%for comparison 5 Oct -99, Stig Ekestorm
%to plot MATLAB simulation output separately
figure(5)
h = mesh((dpRgMapShiftMOD/max(max(dpRgMapShiftMOD)))); grid;
%normalized
%h = mesh(dpRgMapShiftMOD); grid;
title('a. Amplitude/Doppler Modulated Rd-Dp Map (Bit-True, modulated /
MATLAB)');
xlabel('Down Range Cells');   ylabel('Cross Range Cells'); grid


figure(6);
%to plot hardware output separately
if hda==1,
    h = mesh(dpRgMapShift); grid;
end
title('b. Amplitude/Doppler Modulated Rd-Dp Map (HARDWARE output)');
xlabel('Down Range Cells');   ylabel('Cross Range Cells'); grid
```

2.   COMMON FILES IN ALL VERSIONS (VERSION 1 TO 4)

These files are used in all versions. The two first files (cosine.txt and sine.txt) represent the look-up tables. The next three files (genLUT.m, genfixptv0.m and genfloat.m) are used to create the look-up tables [Ref. 6]. Two Matlab functions (dec2two.m and two2dec.m) have been developed to translate from decimal

representation to binary two's complement representation and vice versa. Two extra plot-files are also presented. The first plot-file (plot_like_NRL_image.m) has been used to plot simulation results in a comparable way to a real ISAR image. The second plot-file (plot_in_dB.m) can be used to examine the results in the range-Doppler map as normalized amplitude values in dB.

a. cosine.txt

```
 9.9218750e-001
 9.6875000e-001
 9.0625000e-001
 8.1250000e-001
 6.7968750e-001
 5.2343750e-001
 3.4375000e-001
 1.4843750e-001
-4.6875000e-002
-2.4218750e-001
-4.2968750e-001
-6.0156250e-001
-7.5000000e-001
-8.6718750e-001
-9.4531250e-001
-9.8437500e-001
-9.8437500e-001
-9.4531250e-001
-8.6718750e-001
-7.5000000e-001
-6.0156250e-001
-4.2968750e-001
-2.4218750e-001
-4.6875000e-002
 1.4843750e-001
 3.4375000e-001
 5.2343750e-001
 6.7968750e-001
 8.1250000e-001
 9.0625000e-001
 9.6875000e-001
 9.9218750e-001
```

b. sine.txt

```
 0.0000000e+000
 1.9531250e-001
 3.9062500e-001
 5.6250000e-001
 7.1875000e-001
 8.3593750e-001
 9.2968750e-001
 9.7656250e-001
 9.8437500e-001
 9.5312500e-001
 8.9062500e-001
 7.8125000e-001
 6.4062500e-001
 4.7656250e-001
 2.9687500e-001
 9.3750000e-002
-9.3750000e-002
-2.9687500e-001
-4.7656250e-001
-6.4062500e-001
-7.8125000e-001
-8.9062500e-001
-9.5312500e-001
-9.8437500e-001
-9.7656250e-001
-9.2968750e-001
-8.3593750e-001
-7.1875000e-001
-5.6250000e-001
-3.9062500e-001
-1.9531250e-001
 0.0000000e+000
```

c. genLUT.m

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% genLUT.m
% mfile to generate memory initialization file (cos and sin look-up)
% for altera memory initialization
% Created by: SY YEO, Jul -98
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear

% parameters
depth = 32;
nbits = 8;

del = 1/(2^(nbits-1));
amp = (1-del);

cosine = zeros(depth,1);
```

245

```
for i = 0:depth-1
    cosine(i+1) = amp*cos(2*pi*i/(depth-1));
end
[i,j] = find(abs(cosine) < 4*eps);
p = isempty(i) + isempty(j);
if (p == 0)
    cosine(i,j) = 0;
end
cosine_fixpt = genfixptv0(cosine,nbits);
inv_cosine_fixpt = genfloat(cosine_fixpt,nbits);

fid = fopen('cos.mif','w');
%fid1 = fopen('cos.txt','w');

fprintf(fid,'-- MAX+plus II : memory initialization file');
fprintf(fid,'\n');

txt = (['WIDTH = ' num2str(nbits) '\n']);
%fprintf(fid,'WIDTH = 8\n');
fprintf(fid,txt);
txt = (['DEPTH = ' num2str(depth) '\n']);
%fprintf(fid,'DEPTH = 32\n');
fprintf(fid,txt);

fprintf(fid,'ADDRESS_RADIX = HEX\n');
fprintf(fid,'DATA_RADIX = HEX\n');
fprintf(fid,'\n');

fprintf(fid,'CONTENT BEGIN\n');
tmp1 = dec2hex(bin2dec(cosine_fixpt),2);
t_cosine_fixpt = cosine_fixpt';
for i = 1:depth
    tmp = dec2hex(i-1,2);
    fprintf(fid,'%s\t:\t%s;\n',tmp,tmp1(i,:)');
    %fprintf(fid1,'%s\n',t_cosine_fixpt(:,i));
    %fprintf(fid1,'%s\n',tmp1(i,:)');
end
fprintf(fid,'END;\n');
fclose(fid);
%fclose(fid1);

disp('--Check--')
[cosine inv_cosine_fixpt]
ddcos = std(cosine-inv_cosine_fixpt);

%%%% Repeat for the sin LUT
sine = zeros(depth,1);
for i = 0:depth-1
    sine(i+1) = amp*sin(2*pi*i/(depth-1));
end
[i,j] = find(abs(sine) < 4*eps);
p = isempty(i) + isempty(j);
if (p == 0)
    sine(i,j) = 0;
end
sine_fixpt = genfixptv0(sine,nbits);
```

```
inv_sine_fixpt = genfloat(sine_fixpt,nbits);

fid = fopen('sin.mif','w');
%fid1 = fopen('sin.txt','w');

fprintf(fid,'-- MAX+plus II : memory initialization file');
fprintf(fid,'\n');

txt = (['WIDTH = ' num2str(nbits) '\n']);
fprintf(fid,txt);
txt = (['DEPTH = ' num2str(depth) '\n']);
fprintf(fid,txt);

fprintf(fid,'ADDRESS_RADIX = HEX\n');
fprintf(fid,'DATA_RADIX = HEX\n');
fprintf(fid,'\n');

fprintf(fid,'CONTENT BEGIN\n');
tmp1 = dec2hex(bin2dec(sine_fixpt),2);
t_sine_fixpt = sine_fixpt';
for i = 1:depth
    tmp = dec2hex(i-1,2);
    fprintf(fid,'%s\t:\t%s;\n',tmp,tmp1(i,:)');
    %fprintf(fid1,'%s\n',t_sine_fixpt(:,i));
%    fprintf(fid1,'%s\n',tmp1(i,:)');
end
fprintf(fid,'END;\n');
fclose(fid);
%fclose(fid1);

[sine inv_sine_fixpt]
disp('Std in Reconstruction Errors (cos)')
ddcos
disp('Std in Reconstruction Errors (sin)')
ddsin = std(sine-inv_sine_fixpt)

cosinefp = inv_cosine_fixpt
sinefp = inv_sine_fixpt

save -ascii cosine.txt cosinefp
save -ascii sine.txt sinefp

figure(1);
ll = 1:length(cosinefp);
plot(ll,cosinefp,ll,cosine); grid;

%end of file
```

d. genfixptv0.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% genfixptv0.m
% Fixed point and floating conversion
% PROGRAM in Matlab converts from FLOATING POINT to FIX POINT
% given a number of BITS (nbits) for fix point representation
% Note: that the decimal numbers must be scaled to +- 1.0
% function [out] = genfixpt(in,nbits);
% Created by: SY YEO, Jul -98
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [out] = genfixpt(in,nbits);

del = 1/(2^(nbits-1));

num = in(:);

% Convert to binary
len = length(num);
num1 = []; numb = [];
for i = 1:len
    if (num(i) >= 0.0)
        if num(i) == 1
            num(i) = 1 - del;
        end
        num1 = [num1; fix(num(i)/del)];
        numb = [numb; dec2bin(num(i)/del,nbits)];
    else
        tmp = abs(num(i));
        tmp = dec2bin(tmp/del,nbits);
        if (bin2dec(tmp) ~= 0)
            for k = 1:length(tmp)
                if (tmp(k) == '0')
                    tmp(k) = '1';
                else
                    tmp(k) = '0';
                end
            end
            tmp = bin2dec(tmp)*del + del;
            tmp = dec2bin(tmp/del,nbits);
        end
        numb = [numb; tmp];
    end
end
out = numb;

%end of file
```

248

e. genfloat.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% genfloat.m
% Floating and fix point conversion
% PROGRAM in Matlab converts from FIX POINT to FLOATING POINT
% given a number of BITS (nbits) for fix point representation
% Note: that the decimal numbers must be scaled to +- 1.0
% function [out] = genfloat(in,nbits);
% Created by: SY YEO, Jul -98
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [out] = genfixpt(in,nbits);

numb = in;
del = 1/(2^(nbits-1));

[len,c] = size(numb);
num2 = [];
for i = 1:len
    if (numb(i,1) == '1')
        tmp = numb(i,:);
        for i = 1:length(tmp)    % invert all bits
            if (tmp(i) == '0')
                tmp(i) = '1';
            else
                tmp(i) = '0';
            end
        end
        tmp = -1*(bin2dec(tmp)*del + del); % add a BINARY one!
        num2 = [num2; tmp];
    else
        num2 = [num2; bin2dec(numb(i,:))*del];
    end
end
out = num2;

%end of file
```

f. dec2two.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% dec2two.m
% This MATLAB function converts a number in decimal representation
% (positive or negative) to a vector in binary 2-complement
% representation. With a slight modification the output can be
% presented in as a string with a "." character separating integer and
% fractional parts. The user has to specify the number to be converted
% and the format for the %binary presentation (number of bits used for
% the integer part and the %fractional part). A sign bit will
% automatically be included in the output vector (string).
%
% Function call:
%    dec2two(dec,integerbits,fractionbits)
```

```matlab
%
% User inputs:
%    dec - the number in decimal representation to be converted
%    integerbits - # of bits to represent the integer part
%    fractionbits - # of bits to represent the fractional part
%
% Example (1):
%    type in the MATLAB Command Window: dec2two(2.75,8,4)
%    returned answer: 0  0  0  0  0  0  0  1  0  1  1  0  0
%    (returned answer: 0  0  0  0  0  0  0  1  0 . 1  1  0  0)
%
% Example (2):
%    type in the MATLAB Command Window: dec2two(-2.75,8,4)
%    returned answer: 1  1  1  1  1  1  1  0  1  0  1  0  0
%    (returned answer: 1  1  1  1  1  1  1  0  1 . 0  1  0  0)
%
% Created by:
%    MAJ Stig Ekestorm, Oct -99
%    Naval Postgraduate School
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [out] = dec2two(dec,integerbits,fractionbits);

%binary format to be displayed
signbit=1;                              %# of bits to represent
the sign
%integerbits                            %# of bits to represent
the integer part
%fractionbits                           %# of bits to represent
the fractional part

%initialize output vector
aa=signbit+integerbits+fractionbits;    %length of output
vector
bb=zeros(1,aa);                         %initialize output
vector to zero

%check if the number is negative
if dec<0,                               %if negative number
    dec=dec*(-1);                       %turn number into
positive
    bb(1,1)=1;                          %set sign bit to "1"
end                                     %end if statement

%integer part
mm=floor(dec);                          %extract the integer
part
for idx1=2:integerbits+1,               %integer bits (sign bit
not included)
    cc=2^(integerbits+1-idx1);          %binary bit
representation
    bb(1,idx1)=floor(mm/cc);            %set each bit after
integer division
    mm=rem(mm,cc);                      %extract reminder after
division
end                                     %end for loop
```

250

```
%fractional part
ff=dec-floor(dec);                            %extract fractional
part
for idx2=signbit+integerbits+1:aa,            %fraction bits
    dd=1/(2^(idx2-(signbit+integerbits)));    %binary bit
representation
    bb(1,idx2)=floor(ff/dd);                  %set each bit after
integer division
    ff=rem(ff,dd);                            %extract reminder after
division
end                                           %end for loop

%adjust negative value to 2-complement representation
if bb(1,1)==1,                                %if negative value
    for idx3=1:aa,                            %index for switching
bit values
        if bb(1,idx3)==2,
            bb(1,idx3)=0;
        end
        if bb(1,idx3)==0,                     %switch all "0"
            bb(1,idx3)=1;                     %to "1"
        else
            bb(1,idx3)=0;                     %and vice versa
        end                                   %end if/else statement
    end                                       %end for loop
    bb(1,aa)=bb(1,aa)+1;                      %add "1" to the LSB
    idx4=aa;                                  %index for binary
addition
    while bb(1,idx4)==2,                      %if carry (bit-to-bit)
        bb(1,idx4-1)=bb(1,idx4-1)+1;          %add "1" to next higher
bit
        bb(1,idx4)=0;                         %set current bit to "0"
        idx4=idx4-1;                          %decrement index
(higher bit)
    end                                       %end while loop
    bb(1,1)=1;
end                                           %end if/else statement

%return output in string format (integer and fractional parts separated
by ".")
%out=([num2str(bb(1,1:signbit+integerbits)),' . '...
%        ,num2str(bb(1,signbit+integerbits+1:aa))]);

%return output in vector format (integer and fractional parts without
separation)
out=bb;

%end of file
```

g. two2dec.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% two2dec.m
% This MATLAB function convert vectors in binary 2-complement
% representation to numbers in decimal representation (positive and
% negative). The user has to specify a vector or a set of vectors in a
% matrix to be converted, and the format for the binary presentation
% (number of bits used for the fractional part). The first bit is
% assumed to be a sign bit for the binary vector.
%
% Function call:
%    two2dec(two,fractionbits)
%
% User inputs:
%    two           - the vector/matrix of vectors in binary 2-complement
%                    representation to be converted to decimal number
%    fractionbits - # of bits that represent the fractional part
%
% Output:
%    number/set of numbers in decimal representation
%
% Created by:
%    MAJ Stig Ekestorm, Nov -99
%    Naval Postgraduate School
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%function call
function [out] = two2dec(two,fractionbits);

%determine the size of the matrix of vectors to be converted
[row col]=size(two);

%initialize vectors/variables used
integer=two(:,1:col-fractionbits);   %integer part
fraction=two(:,col-fractionbits+1:col);    %fractional part
total=[integer fraction];
[rowi coli]=size(integer); %size of integer part
[rowf colf]=size(fraction);    %size of fractional part
[rowt colt]=size(total);    %size of fractional part
out=zeros(row,col/col); %output vector

%convert bit pattern representing binary 2-complement number to decimal
number
for idx1=1:row,    %vector-by-vector

    %if positive number - convert
    if integer(idx1,1)==0,   %if sign bit is "0" (positive)
       testi=fliplr(integer(idx1,1:coli)); %flip integer part of vector
       testf=fraction(idx1,1:colf); %fractional part of vector
       for idx2=1:coli-1,    %integer part, bit-by-bit
          if testi(1,idx2)==1, %check for ones
             out(idx1,1)=out(idx1,1)+2^(idx2-1); %add decimal value
          end    %end if
       end    %end for
```

```
    for idx2=1:colf, %fractional part, bit-by-bit
        if testf(1,idx2)==1, %check for ones
            out(idx1,1)=out(idx1,1)+2^(-(idx2));   %add decimal value
        end   %end if
    end   %end for

%if negative number - adjust and convert
else  %if sign bit is "1" (negative)
    for idx4=1:colt,  %index for switching all bit values
        if total(idx1,idx4)==0, %switch all "0"
            total(idx1,idx4)=1;  %to "1"
        else
            total(idx1,idx4)=0;  %and vice versa
        end   %end if/else statement
    end   %end for loop
    if colt>1,  %must be...
        total(idx1,colt)=total(idx1,colt)+1;   %add "1" to the LSB
        idx5=colt;  %index for binary addition
        while total(idx1,idx5)==2, %if carry (bit-to-bit)
            total(idx1,idx5-1)=total(idx1,idx5-1)+1;  %add "1" to next
higher bit
            total(idx1,idx5)=0;  %set current bit to "0"
            idx5=idx5-1;   %decrement index (higher bit)
            if idx5==1, %if this is the last bit
                total(idx1,idx5)=1;  %reset sign bit to "1"
            end   %end if
        end   %end while
    end   %end if
    testi=fliplr(total(idx1,1:coli));   %flip integer part of vector
    testf=total(idx1,coli+1:coli+colf); %fractional part of vector
    for idx2=1:coli-1,   %integer part, bit-by-bit
        if testi(1,idx2)==1, %check for ones
            out(idx1,1)=out(idx1,1)+2^(idx2-1); %add decimal value
        end   %end if
    end   %end for
    for idx2=1:colf, %fractional part, bit-by-bit
        if testf(1,idx2)==1, %check for ones
            out(idx1,1)=out(idx1,1)+2^(-(idx2));   %add decimal value
        end   %end if
    end   %end for
    out(idx1,1)=out(idx1,1)*(-1); %assign a negative value

    end   %end if

end   %end for

%end of file
```

h. plot_like_NRL_image.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% plot_like_NRL_image.m
% This MATLAB script file can be used to plot image in colors similar
% to the ship case refered to at NRL homepage
% (http://radar-www.nrl.navy.mil/Areas/ISAR)
% Created by:
%    MAJ Stig Ekestorm, Feb -00
%    Naval Postgraduate School
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%load plotMOD4Ship1_64
%load plotMOD4ShipK
%load plotMOD4VcaseKb
load plotMOD4Ship8_64
%load plotMOD

%Ship1 =
dpRgMapShiftMOD4Ship1_64(1:64,1:62)/max(max(dpRgMapShiftMOD4Ship1_64(1:
64,1:62)));
%Ship2 =
dpRgMapShiftMOD4DIS2000(1:256,1:62)/max(max(dpRgMapShiftMOD4DIS2000(1:2
56,1:62)));
%Ship2 =
dpRgMapShiftMOD4ShipK(1:64,1:62)/max(max(dpRgMapShiftMOD4ShipK(1:64,1:6
2)));
%Ship2 =
dpRgMapShiftMOD4VcaseKb(1:64,1:62)/max(max(dpRgMapShiftMOD4VcaseKb(1:64
,1:62)));
Ship2 = dpRgMapShiftMOD4Ship8_64(1:64-
1,1:62)/max(max(dpRgMapShiftMOD4Ship8_64(1:64-1,1:62)));

colordef white
figure(64)
colordef black
colormap(hot(100))
%contour(Ship2(:,:),20)
contour(Ship2(:,:),100)
title('8-Tapline Ship Target - 64 Radar Pulses')
xlabel('Range')
ylabel('Doppler')
%axis([0 10 24 38])
axis square

%end of file
```

i. plot_in_dB.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% plot_in_dB.m
% This script file will help you to plot results in dB.
% 1st run plot_like_NRL_image.m, then run this file with the
% appropriate input matrix specified in the mesh-command line.
% Created by:
%    LTC Stig Ekestorm, Apr -00
%    Naval Postgraduate School
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

colordef white
colormap('default')

figure(1)
subplot(2,1,1)
mesh(20*log10(Ship2))        %convert to normalized voltage to dB
axis([0 62 0 64 -100 0])
title('8-Tapline Ship Target - 64 Radar Pulses')
xlabel('Range')
ylabel('Doppler')
zlabel('Normalized Amplitude [dB]')
view(0,0)                     %view along the range axis

subplot(2,1,2)
mesh(20*log10(Ship2))        %convert to normalized voltage to dB
axis([0 62 0 64 -100 0])
%title('8-Tapline Ship Target - 64 Radar Pulses')
xlabel('Range')
ylabel('Doppler')
zlabel('Normalized Amplitude [dB]')
view(90,0)                    %view along the Doppler axis

%end of file
```

3. GENERATING PARAMETERS FOR MULTIPLE SCATTERERS PER RANGE-GATE

Two examples of extract files are presented. The first one for the "V"-case and the other for the ship simulation case. Both are discussed in the end of Chapter IV.

a. extract_para_v4_Vcase.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% extract_phase_v4_Vcase.m
% To extract Phase and Magnitude information to the DIS chip
% Print modified para.txt file
% Created by:
%   MAJ Stig Ekestorm, Feb -00
%   Naval Postgraduate School
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear

%parameters from mathostv2.m
rg_pts=62;                  %# of range-bins

dp_pts=64;                  %# of Doppler bins (same as number of radar
pulses)
%dp_pts=128;                %# of Doppler bins (same as number of radar
pulses)
%dp_pts=256;                %# of Doppler bins (same as number of radar
pulses)
%dp_pts=512;                %# of Doppler bins (same as number of radar
pulses)
%dp_pts=1024;               %# of Doppler bins (same as number of radar
pulses)
%dp_pts=4096;               %# of Doppler bins (same as number of radar
pulses)

bw = 100e6;
pwc = 1/(1.25*bw);          %compressed pulsewidth
pw =0.5e-6;
prf = 2e3;                  %PRF = 2 kHz
pri = 1/prf;                %PRI = 0.5 msec
mu = 2*pi*bw/pw;
fs = 1.25*bw;
Ts = 1/fs;
snr = 0;
nbitsdop = 5;               %# of bits used for precision of the Doppler
phase
p = 2*pi/(2^nbitsdop);      %quant factor
```

```
%additional parameters

dopplerbin=prf/dp_pts;
N=dp_pts;
NP=2*N-1;                   %number of points for the fft
NP=N;
tstop=pri*dp_pts;           %32ms for 64 radar pulses (0.5ms * 64 radar
pulses)
ti=linspace(0,tstop,N);

numtaps=16;
numfreq=[1 1 1 1 1 1 1 1 1 1 1 2 2 2 1 1];   %# of Doppler frequencies
in each range-bin
freq=zeros(numtaps,dp_pts);
freq(1,1:1)=[32.25];
freq(2,1:2)=[5 63.5];
freq(3,1:2)=[-30.25 94.75];
freq(4,1:2)=[-61.5 126];
freq(5,1:1)=[-92.75];
freq(6,1:1)=[-124];
zeropad=zeros(10,dp_pts);
freq=[zeropad;freq];

for idx1 = 1:numtaps
    for idx2 = 1:dp_pts
        if freq(idx1,idx2) == 0
            %do nothing
        else
            freq(idx1,idx2) = freq(idx1,idx2) + 1000;
        end
    end
end

for idx1 = 1:10
    freq(idx1,1) = idx1*10-10;
end

AA=zeros(numtaps,dp_pts);
%AA(1,1:1)=[1e6];
%AA(2,1:2)=[1e6 1e6];
%AA(3,1:2)=[1e6 1e6];
%AA(4,1:2)=[1e6 1e6];
%AA(5,1:1)=[1e6];
%AA(6,1:1)=[1e6];
AA(1,1:1)=[1];
AA(2,1:2)=[2 2];
AA(3,1:2)=[4 4];
AA(4,1:2)=[2 8];
AA(5,1:1)=[4];
AA(6,1:1)=[8];
zeropad=zeros(10,dp_pts);
AA=[zeropad;AA];

targetSum = zeros(numtaps,dp_pts);

for idx1 = 1:numtaps
```

```matlab
    target = zeros(dp_pts,length(ti));
    for idx2 = 1:numfreq(idx1)              %Doppler frequencies
        for idx3 = 1:length(ti)             %create each signal seperately
            fi = rand(1)*0.1;               %random initial phase shift
            %target(idx2,idx3) = 1*exp(-j*2*pi*freq(idx1,idx2)*ti(1,idx3)
+ fi);
            target(idx2,idx3) = AA(idx1,idx2)*exp(-
j*2*pi*freq(idx1,idx2)*ti(1,idx3) + fi);
        end
    end
    for idx4 = 1:numfreq(idx1)                  %create the combined
signal for that range gate
        targetSum(idx1,:) = targetSum(idx1,:) + target(idx4,:);
    end
end

%amplitude values                    %use manual gain values since we can
only do 1, 2, 4, or 8

%gain = [1 2 2 2 1 1];
%gain = input('Enter gain value for each range-bin (enter values as a
row vector): ');
%gain=[1 1·1 2 4 2 1 1 8 1 2 4 4 4 4 8 8 8 4 4 2 1 1 2 8 1 4 1 2 2 1
1];
%gain=[4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4];
%gain=[2 2 2 4 4 2 2 2 8 2 2 4 4 4 4 4 4 4 4 2 2 2 2 8 2 4 2 2 4 4 2
2];
%gain=[4 2 2 2 4 2 2 2 8 4 2 4 2 2 4 4 4 4 4 4 4 4 2 2 8 4 4 2 4 4 4
2];

amp = abs(targetSum);
ampmax = max(amp);
ampmaxmax = max(ampmax);
for i1 = 1:numtaps
    for i2 = 1:dp_pts
        if amp(i1,i2)/ampmaxmax > 0.8
            ampq(i1,i2) = 8;
        end
        if amp(i1,i2)/ampmaxmax < 0.8
            ampq(i1,i2) = 4;
        end
        if amp(i1,i2)/ampmaxmax < 0.4
            ampq(i1,i2) = 2;
        end
        if amp(i1,i2)/ampmaxmax < 0.2
            ampq(i1,i2) = 1;
        end
    end
end

gain = ampq;

%phase-increment values

vphase1=zeros(numtaps,dp_pts);
```

```
vphase2=zeros(numtaps,dp_pts);
vphase3=zeros(numtaps,dp_pts);
for idx5 = 1:numtaps
    vphase1(idx5,:)=angle(targetSum(idx5,:));
    vphase2=vphase1*2^5/(2*pi);
    vphase3=round(mod(vphase2,2^5));
end

phaseinc=zeros(numtaps,dp_pts);
for idx6 = 1:numtaps
    for idx7 = 1:dp_pts
        if idx7==1,
            phaseinc(idx6,idx7)=phaseinc(idx6,idx7)+vphase2(idx6,idx7);
        else
            phaseinc(idx6,idx7)=phaseinc(idx6,idx7-1)+vphase2(idx6,idx7-
1)-vphase2(idx6,idx7);
        end
    end
end

for idx8 = 1:dp_pts
    for idx9 = 1:numtaps
        phasecoeff(idx9,idx8)=2*fix(mod(phaseinc(idx9,idx8),32)/2);  %4-
bit phase modulation coefficient
    end
end

%write modulation parameters to text file

f4 = fopen('paraMULTIq4Vcase3.txt','w');
%f4 = fopen('paraMULTIq4Ship2_1024.txt','w');
%f4 = fopen('paraMULTIq4Ship1.txt','w');

fprintf(f4,'%d\r\n',rg_pts);           %# of range-bins
fprintf(f4,'%d\r\n',dp_pts);           %# of Doppler bins
fprintf(f4,'%d\r\n',numtaps);          %# of tap lines (target
extent)

for aa=1:dp_pts
    for bb=1:numtaps
        fprintf(f4,'%d\r\n',gain(bb,aa));
    end
end

% adjustment to correct multiplication factors for the amplitude (gain)
value
%for i = 1:length(gain)
    %switch gain(1,i)
    %case {1}
    %    gain(1,i)=1;  %no shift, multiplication by 1, hardware bit "00"
    %case {2}
    %    gain(1,i)=2;  %shift by 1, multiplication by 2, hardware bit
"01"
    %case {3}
    %    gain(1,i)=4;  %shift by 2, multiplication by 4, hardware bit
"10"
```

259

```
   %case {4}
   %   gain(1,i)=8;  %shift by 3, multiplication by 8, hardware bit
"11"
   %end
   %fprintf(f1,'%d\r\n',gain(1,i));    %gain1, gain2, ..., gainN
   %fprintf(f2,'%d\r\n',gain(1,i));    %gain1, gain2, ..., gainN
   %fprintf(f3,'%d\r\n',gain(1,i));    %gain1, gain2, ..., gainN
   %fprintf(f4,'%d\r\n',gain(1,i));    %gain1, gain2, ..., gainN
%end

for aa=1:dp_pts
   for bb=1:numtaps
      fprintf(f4,'%d\r\n',phasecoeff(bb,aa));
   end
end

fclose(f4);

%end of file
```

b. extract_para_v4_Ship64.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% extract_para_v4_Ship64.m
% To extract Phase and Magnitude information of the Ship test case
% to be used for Matlab and T-Spice simulation of the DIS chip
% Prints modified para.txt file
% Created by:
%   MAJ Stig Ekestorm, Feb -00
%   Naval Postgraduate School
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%parameters from mathostv2.m

rg_pts=62;              %# of range-bins

dp_pts=64;              %# of Doppler bins (same as number of radar
pulses)
%dp_pts=128;            %# of Doppler bins (same as number of radar
pulses)
%dp_pts=256;            %# of Doppler bins (same as number of radar
pulses)
%dp_pts=512;            %# of Doppler bins (same as number of radar
pulses)
%dp_pts=1024;           %# of Doppler bins (same as number of radar
pulses)
%dp_pts=4096;           %# of Doppler bins (same as number of radar
pulses)

bw = 100e6;
pwc = 1/(1.25*bw);      %compressed pulsewidth
pw =0.5e-6;
prf = 2e3;              %PRF = 2 kHz
pri = 1/prf;            %PRI = 0.5 msec
mu = 2*pi*bw/pw;
```

```
fs = 1.25*bw;
Ts = 1/fs;
snr = 0;
nbitsdop = 5;              %# of bits used for precision of the Doppler
phase
p = 2*pi/(2^nbitsdop);    %quant factor

%Additional parameters

dopplerbin=prf/dp_pts;
N=dp_pts;                  %# of Doppler bins
NP=2*N-1;                  %number of points for the fft
NP=N;                      %change variable
tstop=pri*dp_pts;          %32ms for 64 radar pulses (0.5ms * 64 radar
pulses)
ti=linspace(0,tstop,N);

%false target parameters

numtaps=32;                %# of taps used

%Doppler frequencies per range-bin

freq=zeros(32,dp_pts);
freq(1,1:2)=    [15 45];
freq(2,1:3)=    [17 47 72];
freq(3,1:3)=    [   49 74 100];
freq(4,1:4)=    [   51 77 103 130];
freq(5,1:6)=    [      79 105 132 160 195 231];
freq(6,1:4)=    [      80 108 134 162];
freq(7,1:2)=   .[         111 137];
freq(8,1:3)=    [      113 139 166];
freq(9,1:16)=   [               141 168 199 230 260 291 321 351 379 409
440 471 505 536 571 601];
freq(10,1:4)=   [               142 169 201 232];
freq(11,1:4)=   [                   171 204 234 262];
freq(12,1:11)=  [                   173 206 236 264 294 323 353 381 411
442 475];
freq(13,1:8)=   [                        208 237 265 295 324 354 382 412];
freq(14,1:8)=   [                        212 238 266 296 325 355 383 413];
freq(15,1:8)=   [                            241 267 297 326 356 384 414
445];
freq(16,1:10)=  [                        242 268 298 327 357 385 415
446 481 515];
freq(17,1:11)=  [                            269 299 328 358 386 416
447 482 516 546 582];
freq(18,1:9)=   [                            270 300 329 359 387 417
448 483 518];
freq(19,1:7)=   [                                301 329 360 389 418
449 484];
freq(20,1:6)=   [                                302 330 361 390 419
450];
freq(21,1:4)=   [                                    331 362 391 420];
freq(22,1:3)=   [                                    332 363 392];
freq(23,1:3)=   [                                        364 393 422];
```

```
freq(24,1:4)=   [                                                    365 394 423
454];
freq(25,1:16)= [                                                         395 425
455 491 526 556 590 620 651 681 714 741 776 808 840 869];
freq(26,1:3)=   [                                                        396 426
456];
freq(27,1:6)=   [                                                            427
457 493 528 558 592];
freq(28,1:3)=   [                                                            428
458 494];
freq(29,1:3)=   [
459 495 530];
freq(30,1:4)=   [
460 496 531 560];
freq(31,1:2)=   [
497 532];
freq(32,1:2)=   [
498 533];


numfreq=[2 3 3 4 6 4 2 3 16 4 4 11 8 8 8 10 11 9 7 6 4 3 3 4 16 3 6 3 3
4 2 2];   %# of Doppler frequencies in each range-bin

%amplitude of each scatterer

A = ones(numtaps,dp_pts);                    %all amplitudes set to "1" for
simplification, representing equal strong scatterers

%create a combined signal per range gate

targetSum = zeros(numtaps,dp_pts);

for idx1 = 1:numtaps
   target = zeros(dp_pts,length(ti));
   for idx2 = 1:numfreq(idx1)              %Doppler frequencies
      for idx3 = 1:length(ti)             %create each signal seperately
         fi = rand(1)*0.1;                %random initial phase shift
         target(idx2,idx3) = A(idx1,idx2)*exp(-
j*2*pi*freq(idx1,idx2)*ti(1,idx3) + fi);
      end
   end
   for idx4 = 1:numfreq(idx1)             %create the combined signal for
that range gate
      targetSum(idx1,:) = targetSum(idx1,:) + target(idx4,:);
   end
end

%extract gain (amplitude) coefficients

amp = abs(targetSum);                        %extract magnetude
ampmax = max(amp);                           %find max
ampmaxmax = max(ampmax);                     %find max
for i1 = 1:numtaps                           %hard limit magnetude using 4
levels
   for i2 = 1:dp_pts
      if amp(i1,i2)/ampmaxmax > 0.8
         ampq(i1,i2) = 8;
```

```
        end
        if amp(i1,i2)/ampmaxmax < 0.8
            ampq(i1,i2) = 4;
        end
        if amp(i1,i2)/ampmaxmax < 0.4
            ampq(i1,i2) = 2;
        end
        if amp(i1,i2)/ampmaxmax < 0.2
            ampq(i1,i2) = 1;
        end
    end
end

gain = ampq;                        %assign quantized magnetude values
to gain-coefficient matrix

%extract phase angle

vphase1=zeros(numtaps,dp_pts);      %initialize
vphase2=zeros(numtaps,dp_pts);      %initialize
vphase3=zeros(numtaps,dp_pts);      %initialize
for idx5 = 1:numtaps                            %for each range gate,
    vphase1(idx5,:)=angle(targetSum(idx5,:)); %extract phase angle
    vphase2=vphase1*2^5/(2*pi);                 %adjust value to be
between 0 and 32 (dec)
    vphase3=round(mod(vphase2,2^5));            %round and mod32 to get a
5-bit binary representation
end

%extract phase coefficients

phaseinc=zeros(numtaps,dp_pts);     %initialize
for idx6 = 1:numtaps                %turn values into phase-increments
    for idx7 = 1:dp_pts
        if idx7==1,
            phaseinc(idx6,idx7)=phaseinc(idx6,idx7)+vphase2(idx6,idx7);
        else
            phaseinc(idx6,idx7)=phaseinc(idx6,idx7-1)+vphase2(idx6,idx7-
1)-vphase2(idx6,idx7);
        end
    end
end

for idx8 = 1:dp_pts                 %adjust values to represent the
number of bits used in hardware
    for idx9 = 1:numtaps
        phasecoeff(idx9,idx8)=2*fix(mod(phaseinc(idx9,idx8),32)/2); %4-
bit phase modulation coefficient
    end
end

%write modulation parameters of the false target to text file

%f4 = fopen('paraMULTIq4Vcase1.txt','w');
f4 = fopen('paraMULTIq4Ship3_64.txt','w');
%f4 = fopen('paraMULTIq4Ship1.txt','w');
```

263

```
fprintf(f4,'%d\r\n',rg_pts);          %# of range-bins
fprintf(f4,'%d\r\n',dp_pts);          %# of Doppler bins
fprintf(f4,'%d\r\n',numtaps);         %# of taplines (target
extent)

for aa=1:dp_pts                       %gain modulation coefficients
    for bb=1:numtaps
        fprintf(f4,'%d\r\n',gain(bb,aa));
    end
end

for aa=1:dp_pts                       %phase modulation
coefficients
    for bb=1:numtaps
        fprintf(f4,'%d\r\n',phasecoeff(bb,aa));
    end
end

fclose(f4);

%end of file
```

4.   CREATING TEST VECTORS IN T-SPICE

Creating a long test vector in binary format can be tedious if it must be done manually and the probability of making mistakes cannot be ignored. These three files presented here have been used for transforming data, parameters, and control signals, used for Matlab simulations, into T-Spice format including appropriate T-Spice commands in a computer process. The outputs from these files are in text-file format, which can easily be added together for a complete T-Spice input file.

a. convert2binary_rawint.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% convert2binary_rawint.m
% To convert input values to T-Spice input vector format
% to be used for T-Spice simulation of the DIS chip.
% Prints modified DRFM-phase data as binary test vector.
% Created by:
%    MAJ Stig Ekestorm, Feb -00
%    Naval Postgraduate School
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```matlab
set = 1;                              %what set of radar pulses
norp = 1;                             %number of radar pulses in
one set
norp1 = 1;                            %number of radar pulses in
one set

%nDopplerCell=8;
%nRangeCell=10;
nDopplerCell=64;
nRangeCell=62;
pw=400;
numzero=31;

drfm=ones(nDopplerCell,nRangeCell);

fid = fopen('rawintTspice.txt','r'); %open rawint.txt to be read
for j = 1:nDopplerCell
    for k = 1:nRangeCell-1
        drfm(j,k) = fscanf(fid,'%d',1);
        comma = fscanf(fid,'%c',1);
    end
    drfm(j,nRangeCell) = fscanf(fid,'%d',1);
end
fclose(fid);

f1 = fopen('drfm_bin1.txt','w'); %DRFM-phase data in bimary,
intermediate file
for j = 1:nDopplerCell
    %fprintf(f1,'\r\n');
    %fprintf(f1,'%s%d','Radar Pulse: ',j);
    %fprintf(f1,'\r\n');
    for k = 1:nRangeCell
        drfm_bin = dec2two(drfm(j,k),6,0);
        fprintf(f1,' %d',drfm_bin(1,3:7));
        fprintf(f1,'\r\n');
    end
end
fclose(f1);

drfm2=zeros(nDopplerCell*nRangeCell,5);
fid = fopen('drfm_bin1.txt','r'); %
for j = 1:nDopplerCell*nRangeCell
    for k = 1:4
        drfm2(j,k) = fscanf(fid,'%d',1);
    end
    drfm2(j,5) = fscanf(fid,'%d',1);
end
fclose(fid);

drfm3=drfm2';

f2 = fopen('drfm_bin2.txt','w'); %DRFM-phase data in binary
f3 = fopen('drfm_bin3.txt','w'); %DRFM-phase data in binary, T-Spice
format
for j = 1:5
    for k = 1:nDopplerCell*nRangeCell
```

```
        if j==1 & k==1,
            fprintf(f3,'%s','VinPhase4 Phase4 Gnd bit        ({');
        end
        if j==2 & k==1,
            fprintf(f3,'%s','VinPhase3 Phase3 Gnd bit        ({');
        end
        if j==3 & k==1,
            fprintf(f3,'%s','VinPhase2 Phase2 Gnd bit        ({');
        end
        if j==4 & k==1,
            fprintf(f3,'%s','VinPhase1 Phase1 Gnd bit        ({');
        end
        if j==5 & k==1,
            fprintf(f3,'%s','VinPhase0 Phase0 Gnd bit        ({');
        end
        drfm_bin = drfm3(j,k);
        fprintf(f2,'%d',drfm_bin);
        fprintf(f3,'%d',drfm_bin);
    end
    fprintf(f3,'%s','} on=5.0 off=0.0 pw=',num2str(pw),'n)');
    fprintf(f2,'\r\n');
    fprintf(f3,'\r\n');
end
fclose(f2);
fclose(f3);

drfm4 = [];
zeropadstart = zeros(5,7);      %1 for sync clear, 6 for loading gain and
phase mudulation coefficients
zeropad = zeros(5,numzero);     %31 for reading out results between radar
pulses
for k = 1:nDopplerCell
    if k==1,
        drfm4 = [drfm4,zeropadstart,drfm3(1:5,1:nRangeCell)];
    else
        drfm4 = [drfm4,zeropad,drfm3(1:5,(k-
1)*nRangeCell+1:k*nRangeCell)];
    end
end
drfm4 = [drfm4,zeropad];

f4 = fopen('converted_rawint_1.txt','w'); %DRFM-phase data in binary,
padded with 31 col's of zeros after every radar pulse, T-Spice format

start = (1+6)+(set-1)*norp1*(62+31) + 1;  %start bit of the set
if set == 1                               %1st set (special case)
    start = 1;                            %start bit is then the 1st
bit
end
stop = (start-1) + norp*(62+31);          %stop bit of the set
if start == 1                             %1st set (special case)
    stop = (1+6) + norp*(62+31);          %7 bits for sync clear and
load, then 3 radar pulses (3*(62+31))
end

for j = 1:5
```

```matlab
    for k = start:stop                          %from start bit to stop bit
    %for k = 1:length(drfm4)
    %length(zeropadstart)+nDopplerCell*(nRangeCell+length(zeropad))
        if j==1 & k==start,
            fprintf(f4,'%s','VinPhase4 Phase4 Gnd bit    ({');
            if k>1
                fprintf(f4,'%s','0000000');
            end
        end
        if j==2 & k==start,
            fprintf(f4,'%s','VinPhase3 Phase3 Gnd bit       ({');
            if k>1
                fprintf(f4,'%s','0000000');
            end
        end
        if j==3 & k==start,
            fprintf(f4,'%s','VinPhase2 Phase2 Gnd bit       ({');
            if k>1
                fprintf(f4,'%s','0000000');
            end
        end
        if j==4 & k==start,
            fprintf(f4,'%s','VinPhase1 Phase1 Gnd bit       ({');
            if k>1
                fprintf(f4,'%s','0000000');
            end
        end
        if j==5 & k==start,
            fprintf(f4,'%s','VinPhase0 Phase0 Gnd bit       ({');
            if k>1
                fprintf(f4,'%s','0000000');
            end
        end
        drfm_bin = drfm4(j,k);
        fprintf(f4,'%d',drfm_bin);
    end
    fprintf(f4,'%s','} on=5.0 off=0.0 pw=',num2str(pw),'n)');
    fprintf(f4,'\r\n');
end
fprintf(f4,'%s','*');
fprintf(f4,'\r\n');
fclose(f4);

save converted_rawint drfm4

%end of file
```

b. convert2binary_para.m

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% convert2binary_para.m
% To convert input values to T-Spice input vector format
% to be used for T-Spice simulation of the DIS chip.
% Prints modified gain and phase modulation parameter data as binary
```

```matlab
% test vector.
% Created by:
%    MAJ Stig Ekestorm, Feb -00
%    Naval Postgraduate School
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

set = 1;                                    %what set of radar pulses
norp = 1;                                   %number of radar pulses used
in the set
norp1 = 1;                                  %number of radar pulses used
in the set

%simulation parameters
pw=400;                                     %pulsewidth of simulation
pulse in T-Spice
pbit=4;                                     %# of bits used to represent
Phase Modulation Coefficient

%extract values - select input file
%fid = fopen('para.txt','r');              %open para.txt to be read
%fid = fopen('paraMULTI.txt','r');         %open paraMULTI.txt to be
read
%fid = fopen('paraMULTIq5.txt','r');       %open paraMULTIq5.txt to be
read
%fid = fopen('paraMULTIq4.txt','r');       %open paraMULTIq4.txt to be
read
fid = fopen('paraMULTIq4Ship3_64n.txt','r');    %open
paraMULTIq4ShipXXX.txt to be read

%extract scaling parameters
for j = 1:3
    if j==1,
        nRangeCell = fscanf(fid,'%d',1);    %# of DRFM-phase samples per
radar pulse, range-bins
    end
    if j==2,
        nDopplerCell = fscanf(fid,'%d',1);  %# of radar pulses, Doppler
bins
    end
    if j==3,
        targetExtent = fscanf(fid,'%d',1);  %# of Tapline used, target
extent
    end
end

%initialize matrices
gain=zeros(targetExtent,1);
phase=zeros(nDopplerCell,targetExtent);

%extract gain modulation values
for j = 1:nDopplerCell
    for k = 1:targetExtent
        gain(j,k) = fscanf(fid,'%d',1);
    end
end
```

268

```
%extraxt phase modulation values
for j = 1:nDopplerCell
    for k = 1:targetExtent
        phase(j,k) = fscanf(fid,'%d',1);
    end
end
fclose(fid);

%convert gain modulation coefficients
%adjustment to correct multiplication factors for the amplitude (gain)
value
for j = 1:nDopplerCell
    for k = 1:targetExtent
        switch gain(j,k)
        case {1}
            gain(j,k)=0;   %no shift, multiplication by 1, hardware bit
"00"
        case {2}
            gain(j,k)=1;   %shift by 1, multiplication by 2, hardware bit
"01"
        case {4}
            gain(j,k)=2;   %shift by 2, multiplication by 4, hardware bit
"10"
        case {8}
            gain(j,k)=3;   %shift by 3, multiplication by 8, hardware bit
"11"
        end
    end
end

f1 = fopen('para_gain_bin1.txt','w');  %
for j = 1:nDopplerCell
    for k = 1:targetExtent
        gain_bin = dec2two(gain(j,k),2,0);
        fprintf(f1,' %d',gain_bin(1,2:3));
        fprintf(f1,'\r\n');
    end
end
fclose(f1);

gain2=zeros(nDopplerCell*targetExtent,2);
fid = fopen('para_gain_bin1.txt','r'); %
for j = 1:nDopplerCell*targetExtent
    for k = 1:2
        gain2(j,k) = fscanf(fid,'%d',1);
    end
end
fclose(fid);

gain2MOD=fliplr(gain2);
gain3=gain2MOD';

f2 = fopen('para_gain_bin2.txt','w');  %
for j = 1:2
    for k = 1:nDopplerCell*targetExtent
        gain_bin = gain3(j,k);
```

```
        fprintf(f2,'%d',gain_bin);
    end
    fprintf(f2,'\r\n');
end
fclose(f2);


f3 = fopen('para_gain_bin3.txt','w');   %in format for Toplevel input
file
gain4 = zeros(2,nDopplerCell*32);
[row col]=size(gain3(1:2,1:nDopplerCell*targetExtent));

for m = 1:nDopplerCell
    for j = 1:targetExtent
        for k = 1:2
            if m==1,
                gain4(k,j) = gain3(k,j);
            else
                gain4(k,j+(m-1)*32) = gain3(k,j+(m-1)*targetExtent);
            end
        end
    end
end

gain5=reshape(gain4,8*pbit,2*nDopplerCell);

for j = 1:8*pbit
    for k = 1:2*nDopplerCell
        fprintf(f3,'%d',gain5(j,k));
    end
    fprintf(f3,'\r\n');
end
fclose(f3);

%convert phase modulation coefficients
f4 = fopen('para_phase_bin1.txt','w'); %
for j = 1:nDopplerCell
    for k = 1:targetExtent
        phase_bin = dec2two(phase(j,k),5,0);
        fprintf(f4,' %d',phase_bin(1,2:pbit+1));
        fprintf(f4,'\r\n');
    end
end
fclose(f4);

phase2=zeros(nDopplerCell*targetExtent,pbit);
fid = fopen('para_phase_bin1.txt','r'); %
for j = 1:nDopplerCell*targetExtent
    for k = 1:pbit
        phase2(j,k) = fscanf(fid,'%d',1);
    end
end
fclose(fid);

phase2MOD=fliplr(phase2);
phase3=phase2MOD';
```

```
f5 = fopen('para_phase_bin2.txt','w'); %
for j = 1:pbit
    for k = 1:nDopplerCell*targetExtent
        phase_bin = phase3(j,k);
        fprintf(f5,'%d',phase_bin);
    end
    fprintf(f5,'\r\n');
end
fclose(f5);

f6 = fopen('para_phase_bin3.txt','w'); %in format for Toplevel input
file
phase4 = zeros(pbit,nDopplerCell*32);
[row col]=size(phase3(1:pbit,1:nDopplerCell*targetExtent));

for m = 1:nDopplerCell
    for j = 1:targetExtent
        for k = 1:pbit
            if m==1,
                phase4(k,j) = phase3(k,j);
            else
                phase4(k,j+(m-1)*32) = phase3(k,j+(m-1)*targetExtent);
            end
        end
    end
end

phase5=reshape(phase4,8*pbit,4*nDopplerCell);

for j = 1:8*pbit
    for k = 1:4*nDopplerCell
        fprintf(f6,'%d',phase5(j,k));
    end
    fprintf(f6,'\r\n');
end
fclose(f6);

%make one matrix of Gain and Phase Modulation Coefficients
phasegain = [];
zeropadstart = zeros(32,1);              %1 for sync clear
zeropad = zeros(32,nRangeCell+31-6);     %62 for processing DRFM-phase
data, (31-6) for reading out values and loading new gain and phase
modulation coefficients
zeropadend = zeros(32,nRangeCell+31);    %62+31 for the last radar pulse
and final readout


start = set*(norp1-1);                    %start radar pulse of the set
%start = set*norp + 1;                    %start radar pulse of the set
%start = (1+6)+(set-1)*norp*(62+31) + 1;  %start radar pulse of the set
if set == 1                               %1st set (special case)
    start = 1;                            %start radar pulse is then
the 1st radar pulse
end
stop = set*(norp1-1) + (norp-1);          %stop radar pulse of the set
%stop = set*norp + norp;                  %stop radar pulse of the set
```

```
if start == 1                                %1st set (special case)
    stop = norp;                             %just the number of radar
pulses
end
%stop = (start-1) + norp*(62+31);            %stop radar pulse of the set
%if start == 1                               %1st set (special case)
%    stop = (1+6) + norp*(62+31);            %
%end


for k = start:stop
%for k = 1:nDopplerCell
    if k==start,
    %if k==1,
        phasegain =
[phasegain,zeropadstart,phase5(1:8*pbit,1:4),gain5(1:8*pbit,1:2)];
    else
        phasegain = [phasegain,zeropad,phase5(1:8*pbit,(k-
1)*4+1:k*4),gain5(1:8*pbit,(k-1)*2+1:k*2)];
    end
end
phasegain = [phasegain,zeropadend];


f7 = fopen('converted_para_1.txt','w');    %gain and phase modulation
coefficients in binary, padded zeros during every radar pulse and time
for readout, T-Spice format
for j = 1:8*pbit
    for k = 1:length(phasegain)
        if j==1 & k==1,
            fprintf(f7,'%s','VinBus0 Bus0 Gnd bit          ({');
        end
        if j==2 & k==1,
            fprintf(f7,'%s','VinBus1 Bus1 Gnd bit          ({');
        end
        if j==3 & k==1,
            fprintf(f7,'%s','VinBus2 Bus2 Gnd bit          ({');
        end
        if j==4 & k==1,
            fprintf(f7,'%s','VinBus3 Bus3 Gnd bit          ({');
        end
        if j==5 & k==1,
            fprintf(f7,'%s','VinBus4 Bus4 Gnd bit          ({');
        end
        if j==6 & k==1,
            fprintf(f7,'%s','VinBus5 Bus5 Gnd bit          ({');
        end
        if j==7 & k==1,
            fprintf(f7,'%s','VinBus6 Bus6 Gnd bit          ({');
        end
        if j==8 & k==1,
            fprintf(f7,'%s','VinBus7 Bus7 Gnd bit          ({');
        end
        if j==9 & k==1,
            fprintf(f7,'%s','VinBus8 Bus8 Gnd bit          ({');
        end
        if j==10 & k==1,
            fprintf(f7,'%s','VinBus9 Bus9 Gnd bit          ({');
```

```
end
if j==11 & k==1,
    fprintf(f7,'%s','VinBus10 Bus10 Gnd bit       ({');
end
if j==12 & k==1,
    fprintf(f7,'%s','VinBus11 Bus11 Gnd bit       ({');
end
if j==13 & k==1,
    fprintf(f7,'%s','VinBus12 Bus12 Gnd bit       ({');
end
if j==14 & k==1,
    fprintf(f7,'%s','VinBus13 Bus13 Gnd bit       ({');
end
if j==15 & k==1,
    fprintf(f7,'%s','VinBus14 Bus14 Gnd bit       ({');
end
if j==16 & k==1,
    fprintf(f7,'%s','VinBus15 Bus15 Gnd bit       ({');
end
if j==17 & k==1,
    fprintf(f7,'%s','VinBus16 Bus16 Gnd bit       ({');
end
if j==18 & k==1,
    fprintf(f7,'%s','VinBus17 Bus17 Gnd bit       ({');
end
if j==19 & k==1,
    fprintf(f7,'%s','VinBus18 Bus18 Gnd bit       ({');
end
if j==20 & k==1,
    fprintf(f7,'%s','VinBus19 Bus19 Gnd bit       ({');
end
if j==21 & k==1,
    fprintf(f7,'%s','VinBus20 Bus20 Gnd bit       ({');
end
if j==22 & k==1,
    fprintf(f7,'%s','VinBus21 Bus21 Gnd bit       ({');
end
if j==23 & k==1,
    fprintf(f7,'%s','VinBus22 Bus22 Gnd bit       ({');
end
if j==24 & k==1,
    fprintf(f7,'%s','VinBus23 Bus23 Gnd bit       ({');
end
if j==25 & k==1,
    fprintf(f7,'%s','VinBus24 Bus24 Gnd bit       ({');
end
if j==26 & k==1,
    fprintf(f7,'%s','VinBus25 Bus25 Gnd bit       ({');
end
if j==27 & k==1,
    fprintf(f7,'%s','VinBus26 Bus26 Gnd bit       ({');
end
if j==28 & k==1,
    fprintf(f7,'%s','VinBus27 Bus27 Gnd bit       ({');
end
if j==29 & k==1,
```

```
        fprintf(f7,'%s','VinBus28 Bus28 Gnd bit        ({');
    end
    if j==30 & k==1,
        fprintf(f7,'%s','VinBus29 Bus29 Gnd bit        ({');
    end
    if j==31 & k==1,
        fprintf(f7,'%s','VinBus30 Bus30 Gnd bit        ({');
    end
    if j==32 & k==1,
        fprintf(f7,'%s','VinBus31 Bus31 Gnd bit        ({');
    end
    phasegain_bin = phasegain(j,k);
    fprintf(f7,'%d',phasegain_bin);
    end
    fprintf(f7,'%s','} on=5.0 off=0.0 pw=',num2str(pw),'n)');
    fprintf(f7,'\r\n');
end
fprintf(f7,'%s','*');
fprintf(f7,'\r\n');
fclose(f7);

%end of file
```

c. convert2binary_control.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% convert2binary_control.m
% To convert input values to T-Spice input vector format
% to be used for T-Spice simulation of the DIS chip.
% Prints chip control signals as binary test vector.
% Created by:
%    MAJ Stig Ekestorm, Feb -00
%    Naval Postgraduate School
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

nurp = 1;                               %number of radar pulses used
in the set
nurp1 = 1;                              %number of radar pulses used
in the set
nDopplerCell=nurp;
%nDopplerCell=64;
nRangeCell=62;
numzero=31;
pwc=200;                                %pulsewidth of clock pulse in
T-Spice
pw=2*pwc;                               %pulsewidth of signal pulse
in T-Spice
load converted_rawint                   %variable is drfm4

f1 = fopen('converted_control_1.txt','w');   %control signals in
binary, T-Spice format
%
fprintf(f1,'%s','*');
fprintf(f1,'\r\n');
```

```
%clock
fprintf(f1,'%s','VinCLK CLK Gnd bit                          ({01');
fprintf(f1,'%s','} on=5.0 off=0.0 pw=',num2str(pwc),'n)');
fprintf(f1,'\r\n');
fprintf(f1,'%s','*');
fprintf(f1,'\r\n');
%hold
fprintf(f1,'%s','VinHLD HLD Gnd bit                          ({0');
fprintf(f1,'%s','} on=5.0 off=0.0 pw=',num2str(pw),'n)');
fprintf(f1,'\r\n');
%load
one1=ones(1,1+6+nurp*(62+31)-1);
%one1=ones(1,length(drfm4)-1);
for k = 1:1+6+nurp*(62+31)
%for k = 1:length(drfm4)
    if k==1'
        fprintf(f1,'%s','VinLD LD Gnd bit                     ({0');
    else
        fprintf(f1,'%d',one1(1,k-1));
    end
end
fprintf(f1,'%s','} on=5.0 off=0.0 pw=',num2str(pw),'n)');
fprintf(f1,'\r\n');
%scan right
fprintf(f1,'%s','VinSR SR Gnd bit                            ({0');
fprintf(f1,'%s','} on=5.0 off=0.0 pw=',num2str(pw),'n)');
fprintf(f1,'\r\n');
%scan left
fprintf(f1,'%s','VinSL SL Gnd bit                            ({0');
fprintf(f1,'%s','} on=5.0 off=0.0 pw=',num2str(pw),'n)');
fprintf(f1,'\r\n');
fprintf(f1,'%s','*');
fprintf(f1,'\r\n');
%scan right in
fprintf(f1,'%s','VinS_P_Test_Rin S_P_Test_Rin Gnd bit     ({0');
fprintf(f1,'%s','} on=5.0 off=0.0 pw=',num2str(pw),'n)');
fprintf(f1,'\r\n');
%scan left in
fprintf(f1,'%s','VinS_P_Test_Lin S_P_Test_Lin Gnd bit     ({0');
fprintf(f1,'%s','} on=5.0 off=0.0 pw=',num2str(pw),'n)');
fprintf(f1,'\r\n');
fprintf(f1,'%s','*');
fprintf(f1,'\r\n');
%range-bin valid
one2 = ones(1,nRangeCell);
zero2 = zeros(1,numzero);
block = [];
for k = 1:nDopplerCell
    block = [block,one2,zero2];
end
for k = 1:length(block)+1
    if k==1'
        fprintf(f1,'%s','VinRange_bin_valid Range_bin_valid Gnd bit
        ({0000000');
    else
        fprintf(f1,'%d',block(1,k-1));
```

275

```
        end
end
fprintf(f1,'%s','} on=5.0 off=0.0 pw=',num2str(pw),'n)');
fprintf(f1,'\r\n');
fprintf(f1,'%s','*');
fprintf(f1,'\r\n');
%load phase A
zero3 = zeros(1,nRangeCell+numzero-6);
PhaseA = [1,0,0,0,0,0];
LoadPhaseA = [];
for k = 1:nDopplerCell
    LoadPhaseA = [LoadPhaseA,PhaseA,zero3];
end
for k = 1:length(LoadPhaseA)+1
    if k==1'
        fprintf(f1,'%s','VinLD_Phase_SupTap_A LD_Phase_SupTap_A Gnd bit
        ({0');
    else
        fprintf(f1,'%d',LoadPhaseA(1,k-1));
    end
end
fprintf(f1,'%s','000000} on=5.0 off=0.0 pw=',num2str(pw),'n)');
fprintf(f1,'\r\n');
%load phase B
zero3 = zeros(1,nRangeCell+numzero-6);
PhaseB = [0,1,0,0,0,0];
LoadPhaseB = [];
for k = 1:nDopplerCell
    LoadPhaseB = [LoadPhaseB,PhaseB,zero3];
end
for k = 1:length(LoadPhaseB)+1
    if k==1'
        fprintf(f1,'%s','VinLD_Phase_SupTap_B LD_Phase_SupTap_B Gnd bit
        ({0');
    else
        fprintf(f1,'%d',LoadPhaseB(1,k-1));
    end
end
fprintf(f1,'%s','000000} on=5.0 off=0.0 pw=',num2str(pw),'n)');
fprintf(f1,'\r\n');
%load phase C
zero3 = zeros(1,nRangeCell+numzero-6);
PhaseC = [0,0,1,0,0,0];
LoadPhaseC = [];
for k = 1:nDopplerCell
    LoadPhaseC = [LoadPhaseC,PhaseC,zero3];
end
for k = 1:length(LoadPhaseC)+1
    if k==1'
        fprintf(f1,'%s','VinLD_Phase_SupTap_C LD_Phase_SupTap_C Gnd bit
        ({0');
    else
        fprintf(f1,'%d',LoadPhaseC(1,k-1));
    end
end
fprintf(f1,'%s','000000} on=5.0 off=0.0 pw=',num2str(pw),'n)');
```

276

```
fprintf(f1,'\r\n');
%load phase D
zero3 = zeros(1,nRangeCell+numzero-6);
PhaseD = [0,0,0,1,0,0];
LoadPhaseD = [];
for k = 1:nDopplerCell
    LoadPhaseD = [LoadPhaseD,PhaseD,zero3];
end
for k = 1:length(LoadPhaseD)+1
    if k==1'
        fprintf(f1,'%s','VinLD_Phase_SupTap_D LD_Phase_SupTap_D Gnd bit
        ({0');
    else
        fprintf(f1,'%d',LoadPhaseD(1,k-1));
    end
end
fprintf(f1,'%s','000000} on=5.0 off=0.0 pw=',num2str(pw),'n)');
fprintf(f1,'\r\n');
fprintf(f1,'%s','*');
fprintf(f1,'\r\n');
%use phase
zero3 = zeros(1,nRangeCell+numzero-6);
Phaseinc = [0,0,0,0,1,0];
UsePhaseinc = [];
for k = 1:nDopplerCell
    UsePhaseinc = [UsePhaseinc,Phaseinc,zero3];
end
for k = 1:length(UsePhaseinc)+1
    if k==1'
        fprintf(f1,'%s','Vinuse_Phase_inc use_Phase_inc Gnd bit
        ({0');
    else
        fprintf(f1,'%d',UsePhaseinc(1,k-1));
    end
end
fprintf(f1,'%s','000000} on=5.0 off=0.0 pw=',num2str(pw),'n)');
fprintf(f1,'\r\n');
fprintf(f1,'%s','*');
fprintf(f1,'\r\n');
%load gain AB
zero3 = zeros(1,nRangeCell+numzero-6);
GainAB = [0,0,0,0,1,0];
LoadGainAB = [];
for k = 1:nDopplerCell
    LoadGainAB = [LoadGainAB,GainAB,zero3];
end
for k = 1:length(LoadGainAB)+1
    if k==1'
        fprintf(f1,'%s','VinLD_Gain_SupTap_AB LD_Gain_SupTap_AB Gnd bit
        ({0');
    else
        fprintf(f1,'%d',LoadGainAB(1,k-1));
    end
end
fprintf(f1,'%s','000000} on=5.0 off=0.0 pw=',num2str(pw),'n)');
fprintf(f1,'\r\n');
```

```
%load gain CD
zero3 = zeros(1,nRangeCell+numzero-6);
GainCD = [0,0,0,0,0,1];
LoadGainCD = [];
for k = 1:nDopplerCell
   LoadGainCD = [LoadGainCD,GainCD,zero3];
end
for k = 1:length(LoadGainCD)+1
   if k==1'
      fprintf(f1,'%s','VinLD_Gain_SupTap_CD LD_Gain_SupTap_CD Gnd bit
      ({0');
   else
      fprintf(f1,'%d',LoadGainCD(1,k-1));
   end
end
fprintf(f1,'%s','000000} on=5.0 off=0.0 pw=',num2str(pw),'n)');
fprintf(f1,'\r\n');
fprintf(f1,'%s','*');
fprintf(f1,'\r\n');
%
fclose(f1);
%end of file
```

5. COMPARING MATLAB AND T-SPICE SIMULATIONS

Examining the outputs from T-Spice simulations requires some sort of post-data treatment. In this case a hard-limiter script-file converts output voltage levels from T-Spice simulations into binary and decimal representation. These values can thereafter be used i.e. by the compare script-file to compare Matlab simulation results with T-Spice outputs.

a. hard_limiter.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% hard_limiter.m
% Hard Limiter
% - reads modified text files generated from T-Spice output files
% - extracts and assigns values to variables
% - hard limiters values into binary representation
% - writes results in decimal form to text files
% - writes results in 2-complement binary form to text files
% Created by:
%   MAJ Stig Ekestorm, Nov -99, Modified Jan -00
%   Naval Postgraduate School
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** READ IN VALUES FROM T-SPICE OUTPUT FILES (in text file format)
***

clear                                           %clear all variables

%specify the number of rows and columns for the original text files
row=207;                                        %# of rows
%colin=6;                                       %# of columns for
input
colout=17;                                      %# of columns for
output

%open original text files to read values
%fin = fopen('       .txt','r');                %specify file name
foutI = fopen('Switch_I_OutputsMOD1d.txt','r');      %specify file
name
foutQ = fopen('Switch_Q_OutputsMOD1d.txt','r');      %specify file
name

%initalize
%tmpin=zeros(row,colin);
tmpoutI=zeros(row,colout);
tmpoutQ=zeros(row,colout);

%extract values from the original text files
%for idx2=1:row,                                %# of rows
%    for idx3=1:colin,                          %# of columns
%        tmpin(idx2,idx3)=fscanf(fin,'%f',1);   %reads in the values
%    end
%end
for idx2=1:row,                                 %# of rows of valid
result out
    for idx3=1:colout,                          %# of columns, OBS:
1st column is time
        tmpoutI(idx2,idx3)=fscanf(foutI,'%f',1);     %reads in the I
values
        tmpoutQ(idx2,idx3)=fscanf(foutQ,'%f',1);     %reads in the Q
values
    end
end

%close original text files
%fclose(fin);
fclose(foutI);
fclose(foutQ);

% *** EXTRACT/SEPARATE VARIABLES ***

%initalize
time=zeros(row,1);
%in=zeros(row,colin-1);
outI=zeros(row,colout-1);
outQ=zeros(row,colout-1);
%input=zeros(row,colin-1);
```

279

```
Iout=zeros(row,colout-1);
Qout=zeros(row,colout-1);

%assign values to correct variable
time=tmpoutI(:,1);
%in=tmpin(:,2:colin);
outI=tmpoutI(:,2:colout);
outQ=tmpoutQ(:,2:colout);

% *** HARD LIMITER ***

%hard limiter
for idx4=1:row,
    %for idx5=1:colin-1,
    %    if in(idx4,idx5)<=2.5,          %if less then 2.5V
    %        input(idx4,idx5)=0;         %set bit to "0"
    %    else                            %if higher then 2.5V
    %        input(idx4,idx5)=1;         %set bit to "1"
    %    end
    %end
    for idx5=1:colout-1,
        if outI(idx4,idx5)<=2.5,         %if less then 2.5V
            Iout(idx4,idx5)=0;           %set bit to "0"
        else          .                  %if higher then 2.5V
            Iout(idx4,idx5)=1;           %set bit to "1"
        end
    end
    for idx5=1:colout-1,
        if outQ(idx4,idx5)<=2.5,         %if less then 2.5V
            Qout(idx4,idx5)=0;           %set bit to "0"
        else                             %if higher then 2.5V
            Qout(idx4,idx5)=1;           %set bit to "1"
        end
    end
end

%flip matrices to get MSB to the left, and LSB to the right
%check the order of the values in the text file to confirm if this is
necessary
%input=fliplr(input);
Iout=fliplr(Iout);
Qout=fliplr(Qout);

% *** PRINT TO MATLAB COMMAND WINDOW ***

%print input and output matrices to MATLAB Command Window
%disp(' ')
%disp('Input vectors:')
%input
%disp(' ')
%disp('Output vectors:')
%Iout
%Qout
%disp(' ')

% *** PRINT TO TEXT FILES ***
```

```
%print I- and Q-output in decimal values to two new separate text files
f1=fopen('IoutputsDEC1d.txt','w');        %open text file to write I-
results to
f2=fopen('QoutputsDEC1d.txt','w');        %open text file to write Q-
results to
%print I- and Q-output in 2-complement binary representation to two new
separate text files
f3=fopen('IoutputsBIN1d.txt','w');        %open text file to write I-
results to
f4=fopen('QoutputsBIN1d.txt','w');        %open text file to write Q-
results to

Iout_dec=two2dec(Iout,7);                 %convert to decimal values
Qout_dec=two2dec(Qout,7);                 %convert to decimal values

fprintf(f1,'%12.8f\n',Iout_dec);           %write I-values as decimal
value
fprintf(f2,'%12.8f\n',Qout_dec);           %write Q-values as decimal
value
%fprintf(f1,'%d\n',Iout_dec);             %write I-values as decimal value
%fprintf(f2,'%d\n',Qout_dec);             %write Q-values as decimal value

for idx=1:row,
    fprintf(f3,'%d',dec2two(Iout_dec(idx),8,7)); %write I-values as 2-
complement binary
    fprintf(f3,'\r\n');
    fprintf(f4,'%d',dec2two(Qout_dec(idx),8,7)); %write Q-values as 2-
complement binary
    fprintf(f4,'\r\n');
end

fclose(f1);                               %close text file that results
has been written to
fclose(f2);                               %close text file that results
has been written to
fclose(f3);                               %close text file that results
has been written to
fclose(f4);                               %close text file that results
has been written to
%end of file
```

b. compare.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% compare.m
% Compare Matlab and T-Spice outputs
% - plots Matlab outputs for I and Q channel
% - plots T-Spice outputs for I and Q channel
% - plots difference between Matlab and T-Spice outputs for I and Q
%   channel
% Created by:
%   MAJ Stig Ekestorm, Nov -99, Modified Jan -00
%   Naval Postgraduate School
```

281

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% *** READ IN VALUES FROM T-SPICE OUTPUT FILES (in text file format)
***

%clear                                          %clear all variables

%specify the number of values to read from the original text files
%num=93;                                        %# of values, 1st radar
pulse
num=207;                                        %# of values, total # of
valid outputs from this T-Spice run

%open original text files to read values
fTI = fopen('IoutputsDEC1d.txt','r');           %specify file name
fMI = fopen('Iout.txt','r');                    %specify file name
fTQ = fopen('QoutputsDEC1d.txt','r');           %specify file name
fMQ = fopen('Qout.txt','r');                    %specify file name

%extract values from the original text files
tmpTI = fscanf(fTI,'%f'); %reads in the values
tmpMI = fscanf(fMI,'%f'); %reads in the values
TI = tmpTI(1:num);
MI = tmpMI(1:num);
tmpTQ = fscanf(fTQ,'%f'); %reads in the values
tmpMQ = fscanf(fMQ,'%f'); %reads in the values
TI = tmpTI(1:num);
MI = tmpMI(1:num);
TQ = tmpTQ(1:num);
MQ = tmpMQ(1:num);

fclose(fTI);
fclose(fMI);
fclose(fTQ);
fclose(fMQ);

%plot results
figure(1)
subplot(2,1,1)
plot(MI(62+32-1:2*(62+32-1)),'bo')
hold on
plot(TI(62+32-1:2*(62+32-1)),'rx')
hold off
grid
title('Comparing Matlab and T-Spice outputs - I-Channel')
xlabel('Data'), ylabel('Amplitude')
legend('Matlab','T-Spice')
axis([0 93 -100 100])
subplot(2,1,2)
plot(MI(62+32-1:2*(62+32-1))-TI(62+32-1:2*(62+32-1)),'g')
grid
title('Difference (Matlab and T-Spice) - I-Channel')
xlabel('Data'), ylabel('Amplitude')
legend('Difference')
axis([0 93 -1 1])
```

282

```
figure(2)
subplot(2,1,1)
plot(MQ(62+32-1:2*(62+32-1)),'bo')
hold on
plot(TQ(62+32-1:2*(62+32-1)),'rx')
hold off
grid
title('Comparing Matlab and T-Spice outputs - Q-Channel')
xlabel('Data'), ylabel('Amplitude')
legend('Matlab','T-Spice')
axis([0 93 -100 100])
subplot(2,1,2)
plot(MQ(62+32-1:2*(62+32-1))-TQ(62+32-1:2*(62+32-1)),'g')
grid
title('Difference (Matlab and T-Spice) - Q-Channel')
xlabel('Data'), ylabel('Amplitude')
legend('Difference')
axis([0 93 -1 1])

%end of file
```

6. BIT-VICE TRUNCATION OF TWO'S COMPLEMENT BINARY REPRESENTATION

A script file is presented that produces an example of how one can truncate values in two's complement binary representation for examining different effects.

a. truncate.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% truncate.m
% Test of how to truncate a decimal number representing a binary word
% Created by:
%    MAJ Stig Ekestorm, Mar -00
%    Naval Postgraduate School
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

format long

%a number, 8 integer bits and 7 fraction bits
aa_bit = zeros(1,15);
for idx = 0:7
    aa_bit(8+idx) = 2^idx;
end
for idx = 1:7
    aa_bit(idx) = 2^(idx-8);
end

%bit-by-bit value
aa_bit

%make the binary word into a decimal number
aa_dec = 0;
for idx = 1:15
    aa_dec = aa_dec + aa_bit(1,idx);
end

%*****************************************************************

%original decimal number
aa_dec

%convert original number from decimal to 2-complement binary
aa_bin = dec2two(aa_dec,8,7)

%truncate the binary word (i.e. take out the two least significant
bits)
aa_bin_trunc = aa_bin(1,1:length(aa_bin)-2)
```

284

```
%convert the truncated binary word from 2-complement to decimal
aa_dec_trunc = two2dec(aa_bin_trunc,5)


%************************************************************

%end of file

(Example-Printout from Matlab Command Window)

» clear
» truncate

aa_bit =

  1.0e+002 *

  Columns 1 through 4
    0.00007812500000    0.00015625000000    0.00031250000000
0.00062500000000

  Columns 5 through 8
    0.00125000000000    0.00250000000000    0.00500000000000
0.01000000000000

  Columns 9 through 12
    0.02000000000000    0.04000000000000    0.08000000000000
0.16000000000000

  Columns 13 through 15
    0.32000000000000    0.64000000000000    1.28000000000000

aa_dec =
    2.559921875000000e+002

aa_bin =
  Columns 1 through 12
      0    1    1    1    1    1    1    1    1    1    1    1

  Columns 13 through 16
      1    1    1    1

aa_bin_trunc =
  Columns 1 through 12
      0    1    1    1    1    1    1    1    1    1    1    1

  Columns 13 through 14
      1    1

aa_dec_trunc =
    2.559687500000000e+002
»
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. VISUAL BASIC CODES

1. VISUAL BASIC PROJECT TO RUN THE DIS CONCEPT DEMONSTRATOR

To be able to make the comparison between the Matlab simulation of the DIS and the DIS implemented using FPGA technology, one must add an intermediate step to the simulation flow described in Chapter V. After the Matlab file *mathostvX.m* has been executed, then all the necessary inputs are available in text files to run the hardware implementation of the DIS.

The interface with the FPGA computer board is a set of Visual Basic files composed into a Visual Basic project called FlexTest (Flextest.vbp). The files in the FlexTest project are included below. The files are

- file.bas

- Flecfunc.bas

- Global.bas

- Main.bas

- the_isar.bas

To be able to compile and run the project and the board properly, the necessary files have to be located in a file structure with the following path *c:\temasek\denise\thesis\final_design\vbfiles* due to hard coding issues. To run the Visual Basic project FlexTest, the user needs to open the project, open the the_isar.bas file, and

then run the file. A graphical user interface (GUI) will show up on the computer display

to visualize the signal processing taken place in the taps of the DIS.

a. file.bas

```
Attribute VB_Name = "fileio"
Public nRangeCell As Integer
Public nDopplerCell As Integer
Public targetExtent As Integer
Public Gain() As Integer
Public Phi() As Integer
Public DRFM() As Integer
Public sine() As Double
Public cosine() As Double


Public Sub readPara()
    ' This sub-routine reads the processing parameters generated by
Matlab (reading from paraVB.txt)
    Dim idx As Integer
    Dim idx1 As Integer

    Open "c:/temasek/denise/thesis/final_design/vbfiles/para.txt" For
Input As #1

    ' read in number of range cells (number of samples of the chirp
signal)
    Input #1, nRangeCell
    flexform.ParaText(0) = Val(nRangeCell)

    ' read in number of Doppler cells (ndop) first
    Input #1, nDopplerCell
    flexform.ParaText(1) = Val(nDopplerCell)

    ' next read in the target extent
    Input #1, targetExtent
    flexform.ParaText(2) = Val(targetExtent)

    ' read in gain values (number of gain values = targetExtent)
    ' ReDim Gain(targetExtent - 1) As Integer
    ReDim Gain(3) As Integer
    Gain(0) = 0
    Gain(1) = 0
    Gain(2) = 0
    For idx = 0 To targetExtent - 1
        Input #1, Gain(idx)
        ' adjustment to correct multiplication factors for the
amplitude (gain) value
        Select Case Gain(idx)
            Case Is = 1
            Gain(idx) = 0    ' no shift, multiplication by 1, hardware
bit "00"
            Case Is = 2
```

288

```
                Gain(idx) = 1    ' shift by 1, multiplication by 2, hardware
bit "01"
                Case Is = 4
                Gain(idx) = 2    ' shift by 2, multiplication by 4, hardware
bit "10"
                Case Is = 8
                Gain(idx) = 3    ' shift by 3, multiplication by 8, hardware
bit "11"
            End Select
        Next idx

        ' read in the next nDoppler values
        ReDim Phi(nDopplerCell - 1, targetExtent - 1) As Integer
        For idx = 0 To nDopplerCell - 1
            For idx1 = 0 To targetExtent - 1
                Input #1, Phi(idx, idx1)
            Next idx1
        Next idx
        Close #1
End Sub
Public Sub readRaw()
        ' This sub-routine reads the raw matlab-simulated ISAR data

        Dim idx1 As Integer
        Dim idx2 As Integer

        Open "c:/temasek/denise/thesis/final_design/vbfiles/rawint.txt" For
Input As #1

        ' Create the array dynamically
        ReDim DRFM(nDopplerCell - 1, nRangeCell + targetExtent - 1) As
Integer

        ' read in the first samples of the first pulse
        ' these values are phase values from the DFRM
        For idx1 = 0 To nDopplerCell - 1
            For idx2 = 0 To nRangeCell - 1
                Input #1, DRFM(idx1, idx2)
            Next idx2
        Next idx1
        Close #1
 End Sub
Public Sub readCosine()
        ' This sub-routine reads the raw matlab-simulated ISAR data

        Dim idx1 As Integer
        Dim tmp As String

        Open "c:/temasek/denise/thesis/final_design/vbfiles/cosine.txt" For
Input As #1

        ' Create the array dynamically
        ReDim cosine(32) As Double

        ' read in the first samples of the first pulse
        ' these values are phase values from the DFRM
```

289

```
      For idx1 = 0 To 31
          Line Input #1, tmp
          cosine(idx1) = Val(tmp)
      Next idx1
      Close #1
  End Sub
Public Sub readSine()
      ' This sub-routine reads the raw matlab-simulated ISAR data

      Dim idx1 As Integer

      Open "c:/temasek/denise/thesis/final_design/vbfiles/sine.txt" For
Input As #1

      ' Create the array dynamically
      ReDim sine(32) As Double
      Dim tmp As String

      ' read in the first samples of the first pulse
      ' these values are phase values from the DFRM
      For idx1 = 0 To 31
          Line Input #1, tmp
          sine(idx1) = Val(tmp)
      Next idx1
      Close #1
  End Sub
```

b. flecfunc.bas

```
Attribute VB_Name = "FlexFunc"
Option Explicit
'U4 Flex Programming Logic Control & Status Ports
'     Control
'     Port 380/382 - Write          A/B            A/B
'          D0  =    nConfig    U4-H10/B8   Flex U27/U28 - AU1
'          D1  =    nCS        U4-J11/F9   Flex U27/U28 - A35
'                   CS                     Flex U27/U28 - C33  (HI)
'          D2  =    nClr       U4-D11/F3   Flex U27/U28 - C17
'          D3  =    OutEn      U4-K3/K10   Flex U27/U28 - C19
'3
'
'     Port 380/382 - Read
'          D4  =    Init_Done  U4-L3/L2    Flex U27/U28 - R35
'          D5  =    Conf_Done  U4-L4/K4    Flex U27/U28 - C37
'          D6  =    nStatus    U4-L10/L9   Flex U27/U28 - AU37
'          D7  =    RDYnBSY    U4-J2/K11   Flex U27/U28 - N35
'
'     Port 381/383 - Write
'          D0-D7    Configuration Data for Flex chip A/B
'                   nWS        U4-H11/E11  Flex U27/U28 - E31
'
'     Port 381/383 - Read
'          D7  -    RDYnBSY for Flex chip A/B
'                   nRS        U4-G9/F10   Flex U27/U28 - A33
```

```
'
'
'    Port 384 -          FLEX User Control/Status Register
'        D0  = Enable DATA_DIR for READ Buffers (D0 on U4-A9, DATA_DIR
on U4-B11)
'        D1  = An/B  Select Flex A = 0, Select Flex B = 1
'        D2  = Int_CLRn
'        D3  = Unused
'        D4  = CFA1  Spare chip interconnect to FLEX A
'        D5  = CFB1  Spare chip interconnect ot FLEX B
'        D6  = TST_A Flex A circuit Test Point
'        D7  = TST_B FLEX B circuit TEst point
'
'
'    Port 386 -          FLEX A (left from component side) User Base
Addr. Register
'    Port 387 -          FLEX B (right from component side) User Base
Addr. Register

Declare Function write_port Lib "in_out" (paddr%, pdata%, byteword%) As
Integer
Declare Function read_port Lib "in_out" (paddr%, pdata%, byteword%) As
Integer
Declare Function InitClicks Lib "in_out" () As Integer
Declare Function Clicks Lib "in_out" () As Integer
Declare Function disable_int Lib "in_out" () As Integer
Declare Function enable_int Lib "in_out" () As Integer

Declare Function FlexConfig Lib "altera" (ByVal file As String, ByRef
plength As Long) As Integer
Declare Function FlexSend Lib "altera" (ByVal Cntrl_port%, ByVal
Status_port%, ByVal Data_port%) As Integer
Declare Function FlexSend10k50 Lib "altera" (ByVal Cntrl_port%, ByVal
Status_port%, ByVal Data_port%) As Long

'Memory Calls
Declare Function MemoryInit Lib "memory" (ByVal Start As Long, ByVal
Length As Long) As Integer
Declare Function MemoryRead Lib "memory" (ByVal Location As Long, Value
As Any) As Integer
Declare Function MemoryWrite Lib "memory" (ByVal Location As Long,
ByVal Value As Long) As Integer
Declare Function MemoryReadBuffer Lib "memory" (ByVal Location As Long,
ByVal Count As Long, ByRef Value As Integer) As Integer
Declare Function MemoryWriteBuffer Lib "memory" (ByVal Location As
Long, ByVal Count As Long, ByRef Value As Integer) As Integer

'LoadTgt DLL
Declare Function LoadTgtBuf Lib "loadtgt" (ByVal TgtNum%, ByVal param%,
ByVal Value&) As Integer
Declare Function TargetWrite Lib "loadtgt" (ByVal TgtNum%) As Integer
Declare Function WriteTargets Lib "loadtgt" (ByVal NumTgts%) As Integer
Declare Function InitializePorts Lib "loadtgt" () As Integer

'*** RES added variables ***
Public NumBds As Integer, offset As Integer
```

```
Public flex_user_ba_ctrl As Integer     ' set this port to determine
user design ba
Public user_ba(8) As Integer
Public BoardNum As Integer
Dim maxcount As Integer
Public present(8) As Integer, fstatus(8) As Integer
Public configdone(8) As Integer, Status(8) As Integer
Public num_bds_fnd As Integer
'**************************

' RMS 17 Sep 96
' Do I need to declare Function Delay(Dwell As Double) ?

Global filename As String
Global MSG As Integer
Global BoardType(8) As String      'Either "10K" or "8K"

' 5032 or 5192 Chip Addresses
Global FlexCtrlPortBA(8) As Integer
Global FlexCtrlPort As Integer
Global FlexStatusPort As Integer
Global FlexDataPort As Integer
Global FlexUserCtrlPort As Integer  'Used to Toggle between chips
Global FlexUserBasePort As Integer  'Used to Set base address in Flex
Global NumFlexes As Integer
Global FlexIndex As Integer
Global NumFlexFiles As Integer
Global FlexFileIndex As Integer
'RMS 17 Sep 96
'These are for 8K only, and maybe not there if we change the 5192
Global FlexONPort As Integer
Global FlexOFFPort As Integer

'Flex 10K50 Chip Addresses
Global FlexUserBA(8) As Integer

'Flex File Names and Documentation
Global flexfilename(10) As String
Global FlexMaxFiles As Integer
Global FlexFileDoc(10) As String

Global DP_MEM_Right_Addr_Lo As Integer
Global DP_MEM_Right_Addr_HI As Integer
Global DP_MEM_Right_Data As Integer
Global DP_MEM_Right_Ctrl As Integer

Global DP_MEM_Left_Addr_Lo As Integer
Global DP_MEM_Left_Addr_HI As Integer
Global DP_MEM_Left_Data As Integer
Global DP_MEM_Left_Ctrl As Integer

Global DP_MEM_Hand_Shake_Sim As Integer

Global DP_MEM_Left_addr_Mux As Integer
Global HP_Ctrl_Port As Integer      'Control Port to select HP connector
A data
```

```
Global HP_Ctrl_data As Integer        'Control data to Select HP connector
B data
                                      ' Data - HPA      HPB
                                      '   0    Toggle  Toggle
                                      '   1    Mem     Toggle
                                      '   2    Toggle  Mem
                                      '   3    Mem     Mem
Global AttenPortLO As Integer
Global AttenPortHI As Integer

Global Const nConfigLo = &H2
Global Const nConfigHI = &H3
Global Const nConfigHI_nCSLO = &H1
Global Const nConfigHI_nCSHI = &H3
Global Const nStatLO_RDYnBSYHI = &HC
Global Const ConfDone = &H18                    'Conf_Done & nSatus HI


Global altera(100000) As Integer
Global MaxAlteraPnts As Long

Global Const RngDef = &H100
Global Const PWDef = &H200

'***** internal addresses for the ISAR program *****
'Global Const phiAddr = &H10          ' for Doppler offset
'Global Const gainAddr = &H20         ' for gain
'Global Const tapAddr = &H30          ' for tap delay line
'Global Const modPulseAddr = &H40     ' for modulated pulse readback
'Global Const feedback = &H60          ' for reading back values

Function Delay(Dwell As Double)

' This routine creates a time delay that lasts untill Dwell seconds
' elapse from the time of call.
'
' It uses VBasic's Timer function, which returns the number of seconds
' since midnight on the system clock, rolling from 86400 to 0 at
midnight.
'
' This routine allows delays to begin before midnight and end after,
' or that span several days.

Dim SecPerDay As Double
Dim Start As Double, Done As Double, T As Double, LastT As Double

SecPerDay = 86400# ' = 60.0 * 60.0 * 24.0 seconds in a day

Start = Timer
Done = Start + Dwell

While (Done > SecPerDay) ' Midnight will come before the delay elapses.
    LastT = Start
    T = Timer
    While (T > LastT) ' Timer has not rolled over.
        LastT = T
```

```
            T = Timer
            DoEvents
        Wend
        ' It's midnight, so deduct the previous day's waiting
        ' and start a new day.
        Done = Done - (SecPerDay - Start)
        Start = 0
Wend

    ' The delay will elapse before midnight comes.
    While (Done > Timer)
        DoEvents
Wend

End Function
Public Function LoadFlex(filename As String, Index As Integer)

Dim MSG As Long
Dim dum1 As Integer
Static Status As Integer
Dim FileDate As String

'setup flex addresses
    FlxBaseAddr (BoardNum)

'Reset Flex Chip
MSG = write_port(FlexCtrlPort, nConfigLo, 1)   'Set nConfig (bit 0) LO

'Read & Send Data'
DoEvents

    MSG = FlexConfig(filename, MaxAlteraPnts)

If (MSG = 1) Then

        MSG = FlexSend10k50(FlexCtrlPort, FlexStatusPort, FlexDataPort)
        Status = Get10KStatus()

    FileDate = FileDateTime(filename)
    If Status = True Then
        DoEvents
        MSG = write_port(FlexCtrlPort, &HF, 1)      'Set nCONFIG, nCS,
nCLR, IO_ENB = 1
        LoadFlex = True
    Else
        LoadFlex = False
     End If

    Delay (0.2)                     'Delay .2 Second for Visual Effect

Else
    LoadFlex = False
End If

End Function
Public Sub InitFlexType()
```

```
Dim i As Integer
Dim ret As Integer
Dim memory_length As Long
Dim status_init As Integer

BoardType(BoardNum) = "10K"

  Call InitBoardBase(BoardType(BoardNum))     'Get Base Address

If BoardType(BoardNum) = "10K" Then InitFlx10KAddr (BoardNum)
'Set First Board as Default

End Sub

Function Get10KStatus()
Static MSG As Integer

    MSG = write_port(FlexUserCtrlPort, &H1, 1)        'Turn on Read
Buffer Capability

    MSG = read_port(FlexStatusPort, fstatus(BoardNum), 1)          'Get
Flex Status

    If fstatus(BoardNum) = &HFF Then
      'the board is off or not plugged in
      Get10KStatus = False
      present(BoardNum) = False
      Status(BoardNum) = False
      configdone(BoardNum) = False
      Exit Function
    End If

    If (&H40 And fstatus(BoardNum)) <> &H40 Then
      'nSTATUS bit = 0 -- an error occurred
      Get10KStatus = False
      Status(BoardNum) = False
      configdone(BoardNum) = False
      Exit Function
     Else
    End If

    If (&H60 And fstatus(BoardNum)) = &H60 Then
      'nSTATUS bit = 1 and CONF_DONE = 1 -- the FLEX programmed OK
''      ConfigDone.Value = 1
      Else
''      ConfigDone.Value = 0
      configdone(BoardNum) = False
      Get10KStatus = False
      Exit Function
    End If

    present(BoardNum) = True
    Status(BoardNum) = True
    configdone(BoardNum) = True
    Get10KStatus = True
```

```vb
End Function
Sub get_flex_ini()
Dim a As String, b As Integer

    On Error GoTo ini_err_handler

    Open "flex.ini" For Input As #1
        Input #1, NumBds, a
      For b = 1 To NumBds
        Input #1, BoardNum
        Input #1, FlexCtrlPortBA(BoardNum), crystal_clk(BoardNum)
        Input #1, user_ba(BoardNum), flexfilename(BoardNum)
        Input #1, a
      Next b
    Close #1
    Exit Sub

ini_err_handler:
    Exit Sub

End Sub

Function FlexSendVB(altera() As Integer, NumPnts As Long)

Static Status As Integer
Static j As Long
Static ConfigData As Integer

Debug.Print altera(5)
Debug.Print NumPnts

'Reset Flex Chip
    MSG = write_port(FlexCtrlPort, nConfigLo, 1)

    MSG = write_port(FlexCtrlPort, nConfigHI, 1)

'Check for FLEX Proper Response
    j = 0
    MSG = write_port(FlexCtrlPort, nConfigHI_nCSLO, 1)
    MSG = read_port(FlexStatusPort, Status, 1)
    While Status <> nStatLO_RDYnBSYHI
        MSG = read_port(FlexStatusPort, Status, 1)

        j = j + 1
        If j > 200 Then
            FlexSendVB = -2
            Exit Function
        End If
    Wend

    For j = 1 To NumPnts
        ConfigData = altera(j)
        MSG = write_port(FlexDataPort, ConfigData, 1)
    Next j

    MSG = write_port(FlexCtrlPort, nConfigHI_nCSHI, 1)
```

```
        Delay (0.5) 'Wait half a second before getting status
        MSG = read_port(FlexStatusPort, Status, 1)
        Status = Status And &H1C          'And out unused bits
            If Status <> ConfDone Then        'Is Conf_Done & nStatus HI
            FlexSendVB = -3
            Exit Function
        End If

        FlexSendVB = True

End Function


Function FlexConfigVB(filename As String) As Integer

Dim line As String
Static i As Long
Static CommaLeft As Integer
Static CommaRight As Integer
Static Token As Integer

'main.flexstatus.Text = "Reading Flex File " + filename
'main.flexstatus.BackColor = LtGray
DoEvents

Open filename For Input As #1
i = 1
Do While Not EOF(BoardNum) ' Loop until
    Line Input #1, line    ' Read data into two variables.
'    Debug.Print line  ' Print data to Debug window.
    CommaLeft = 1
    CommaRight = -1
    Do While True
        CommaRight = InStr(CommaLeft, line, ",", 1)
        If CommaRight = 0 Then  'If not at first character then exit
            If CommaLeft <> 1 Then Exit Do
            CommaRight = 10
        End If
        Token = Mid(line, CommaLeft, CommaRight - CommaLeft)
        altera(i) = Val(Token)
        i = i + 1
        CommaLeft = CommaRight + 1
    Loop
Loop

Close #1     ' Close file.
MaxAlteraPnts = i - 1 'Fix total number of bytes read
FlexConfigVB = True
End Function
Function Flx10KSetAddr(Index As Integer)

    Dim MSG As Integer, CBA As Integer, UBA As Integer

    CBA = FlexCtrlPortBA(Index)
    UBA = FlexUserBA(Index)
```

```
        FlexCtrlPort = CBA                       'Control Port
        FlexDataPort = CBA + 1                   'Data Programming Port
        FlexStatusPort = CBA                     'Status Port
        FlexUserCtrlPort = (CBA And &H3F0) + 4   'Flex User Control Port (A
or B)
        FlexUserBasePort = UBA

        ' RMS 17 Sep 96
        ' These do not make sense with the new (or old) U4 map.  Bob?
        DP_MEM_Right_Addr_Lo = CBA + 3
        DP_MEM_Right_Addr_HI = CBA + 5


End Function


Function FlexSendBuffer()
Dim MSG As Long

'main.flexstatus(0).Text = "Sending Data "
MSG = FlexSendVB(altera(), MaxAlteraPnts)
If MSG <> MaxAlteraPnts Then
'       main.flexstatus(0).Text = "Error Configuring Flex"
'       main.flexstatus(0).BackColor = red
    FlexSendBuffer = False
    Exit Function
End If

'main.flexstatus(0).Text = "Flex Configured"
'msg = GetFlexSDatus()

FlexSendBuffer = True
End Function

Sub InitFlx10KAddr(BoardNum As Integer)

    'Set Altera 5192 Base Addresses
        FlexCtrlPort = FlexCtrlPortBA(BoardNum) + 0
        FlexDataPort = FlexCtrlPortBA(BoardNum) + 1
        FlexStatusPort = FlexCtrlPortBA(BoardNum) + 2
        'optFlxBaseAddr(Board).Value = True.

    'Set FLEX Address for Left side of Dual Port Memory
        DP_MEM_Left_Data = FlexUserBA(BoardNum) + 0
        DP_MEM_Left_Addr_Lo = FlexUserBA(BoardNum) + 1
        DP_MEM_Left_Addr_HI = FlexUserBA(BoardNum) + 2
        DP_MEM_Left_Ctrl = FlexUserBA(BoardNum) + 3

    'Set FLEX Address for Right side of Dual Port Memory
        DP_MEM_Right_Data = FlexUserBA(BoardNum) + 4
        DP_MEM_Right_Addr_Lo = FlexUserBA(BoardNum) + 5
        DP_MEM_Right_Addr_HI = FlexUserBA(BoardNum) + 6
        DP_MEM_Right_Ctrl = FlexUserBA(BoardNum) + 7

        DP_MEM_Left_addr_Mux = FlexUserBA(BoardNum) + &HC
```

298

```vb
End Sub


'Note that 'FlexUserBA' is determined by .ttf design file
'FlexCtrlPortBA(BoardNum) is the board addr. determined by wire straps
to 5192
'
'This Routine returns the number of Files read
Public Sub InitBoardBase(BrdType As String)
 Dim i As Integer
 Dim dum As String
 Dim line As String
 Dim line2 As String
 Dim filename As String

'************
Exit Sub

 filename = "Win" + BrdType + ".ini"

Open filename For Input As #1

Input #1, NumFlexFiles, dum
For i = 0 To NumFlexFiles - 1
    Input #1, flexfilename(i), FlexFileDoc(i)
Next i

' Skip Three lines
Line Input #1, dum
Line Input #1, dum
Line Input #1, dum

i = 0
Do Until EOF(1)
    Input #1, FlexCtrlPortBA(i), FlexUserBA(i), dum        '5192 and FLEX
base addr
    i = i + 1
Loop


If (i > 8) Then
   NumFlexes = 8
   MsgBox ("File " + filename + " contains too many base addresses.")
Else
   NumFlexes = i
End If

Close #1

End Sub
Sub init_flex_param()

' 5032 or 5192 Addresses
        FlexCtrlPort = FlexCtrlPortBA(BoardNum) + 0
        FlexDataPort = FlexCtrlPortBA(BoardNum) + 1
```

```vb
        FlexStatusPort = FlexCtrlPortBA(BoardNum) + 2

        FlexOFFPort = FlexCtrlPortBA(BoardNum) + 6
        FlexONPort = FlexCtrlPortBA(BoardNum) + 7

        flex_user_ba_ctrl = FlexCtrlPortBA(BoardNum) + 3

End Sub



'I input parameter sets the address of the right port
'on dual port memory (0-4095)

Function SetAddrRight(i As Integer)
  Static LoAddr As Integer
  Static HiAddr As Integer

    If i > 4095 Then
        SetAddrRight = False: Exit Function
    End If

    LoAddr = i Mod 256
    HiAddr = i \ 256
    MSG = write_port(DP_MEM_Right_Addr_Lo, LoAddr, 1)
    MSG = write_port(DP_MEM_Right_Addr_HI, HiAddr, 1)

    SetAddrRight = True

End Function
Private Sub FlxBaseAddr(Index As Integer)
Dim a
Dim MSG As Integer, CBA As Integer, UBA As Integer

        'Definition of Ports used to program  and control the FLEX chip
        'on a 10K50 board

            Flx10KSetAddr (Index)
            MSG = Get10KStatus()

'           optFlxBaseAddr(Index).Value = True
        FlexIndex = Index

'

        'Get and Display Status of Current Flex Chip
        MSG = Get10KStatus()

End Sub



'I input parameter sets the address of the left port
'on dual port memory (0-4095)

Function SetAddrLeft(i As Integer)
  Static LoAddr As Integer
  Static HiAddr As Integer
```

```
    If i > 4095 Then
        SetAddrLeft = False
        Exit Function
    End If

    LoAddr = i Mod 256
    HiAddr = i \ 256           'Be Sure to Integer divide
    MSG = write_port(DP_MEM_Left_Addr_Lo, LoAddr, 1)
    MSG = write_port(DP_MEM_Left_Addr_HI, HiAddr, 1)

    SetAddrLeft = True

End Function


Function usecDelay(Dwell As Integer)
Dim initClick As Long
Dim currentClick As Long
Dim EndClick As Long
Dim icnt As Long
Dim ret As Integer

EndClick = Dwell / 0.8381     'Each click represents 0.8381 usec

ret = InitClicks
initClick = Clicks           'Sets Down counter to max value ( about
65,000)
If initClick < 0 Then
    initClick = 65535 + initClick
End If
currentClick = Clicks        'Reads current count
If currentClick < 0 Then
    currentClick = 65535 + currentClick
End If

icnt = 0

While (EndClick > (initClick - currentClick))
    currentClick = Clicks
    If currentClick < 0 Then
        currentClick = 65535 + currentClick
    End If
    icnt = icnt + 1
    If icnt > 1000000 Then
        usecDelay = False
        Exit Function
    End If
Wend

usecDelay = True
End Function


Sub Board_Bit()
Dim i As Integer
```

301

```
'   init_flex_param
    flexform.ini_num.Text = NumBds
    num_bds_fnd = 0

'** find # of boards that respond to ping **
For BoardNum = 1 To NumBds
    Flx10KSetAddr (BoardNum)
        flexform.addr(BoardNum).Text = Hex(FlexCtrlPortBA(BoardNum))
    ck_bd_present
    If present(BoardNum) = False Then
        flexform.present(BoardNum).BackColor = red
    Else
        flexform.present(BoardNum).BackColor = green
        num_bds_fnd = num_bds_fnd + 1
    End If
Next BoardNum

    flexform.found_num.Text = num_bds_fnd

End Sub
Sub ck_bd_present()
Static MSG As Integer

    MSG = write_port(FlexUserCtrlPort, &H1, 1)        'Turn on Read
Buffer Capability

    MSG = read_port(FlexStatusPort, fstatus(BoardNum), 1)          'Get
Flex Status

    If fstatus(BoardNum) = &HFF Then
      present(BoardNum) = False
    Else
      present(BoardNum) = True
    End If

End Sub
Sub test_boards()
Dim dum As Integer, dly As Long, invar As Integer

    flexform.Show
'** set initial state to gray in leds
    For dum = 1 To NumBds
        flexform.present(dum).BackColor = LtGray
        flexform.Bstatus(dum).BackColor = LtGray
        flexform.Bconfigdone(dum).BackColor = LtGray
        flexform.BBIT(dum).BackColor = LtGray
    Next dum
'** check presence of boards ***
    get_flex_ini     ' read basic flex addrs. & pri param. from
filename$.ini file
    reset_flexs

    Board_Bit        ' check # of boards and operational status

'** load flex chips & display status **
  For BoardNum = 1 To NumBds
```

302

```
      If present(BoardNum) = True Then
          InitFlexType
          flexform.Bconfigdone(BoardNum).BackColor = yellow
          MSG = LoadFlex(flexfilename(BoardNum), 1)
          Flx10KSetAddr (BoardNum)
          Get10KStatus
        If Status(BoardNum) = False Then
          flexform.Bstatus(BoardNum).BackColor = red
        Else
          flexform.Bstatus(BoardNum).BackColor = green
        End If
'*** write user base addr. to flex (a=3x6, b=3x7)
        offset = (0.5 * (FlexCtrlPortBA(BoardNum) And &H2) + 6) -
(FlexCtrlPortBA(BoardNum) And &H2)
        dum = write_port(FlexCtrlPortBA(BoardNum) + offset,
user_ba(BoardNum) \ &H4, 1) 'set Flex User BaseAddr
        If configdone(BoardNum) = False Then
          flexform.Bconfigdone(BoardNum).BackColor = red
        Else
          flexform.Bconfigdone(BoardNum).BackColor = green
        End If
          flexform.filename(BoardNum).Text = flexfilename(BoardNum)
          flexform.userBA(BoardNum).Text = Hex(user_ba(BoardNum))
'** check BIT register for proper load
          dum = write_data(&HFC, BoardNum * &H15A5, 2) 'write board
(internal addr (linear addr. 0 ->255 words) , data , word)
          dum = read_data(&HFC, invar, 2)
          If invar = BoardNum * &H15A5 Then
              flexform.BBIT(BoardNum).BackColor = green
          Else
              flexform.BBIT(BoardNum).BackColor = red
          End If
'          dum = write_port(user_ba(BoardNum), 0, 2)    ' zero BIT reg.
for noninterference w/next board
    End If    'end check for board present
  Next BoardNum

      For dly = 1 To 10000: DoEvents: Next dly
End Sub
Sub reset_flexs()    'reset flex chips (unload)
Dim dum As Integer

  get_flex_ini
  For BoardNum = 1 To NumBds
    MSG = write_port(FlexCtrlPortBA(BoardNum), nConfigLo, 1)   'Set
nConfig (bit 0) LO
'     MSG = write_port(FlexCtrlPortBA(BoardNum), nConfigHI, 1)   'Set
nConfig (bit 0) LO
  Next BoardNum

'** set initial state to gray in leds
    For dum = 1 To NumBds
        flexform.present(dum).BackColor = LtGray
        flexform.Bstatus(dum).BackColor = red
        flexform.Bconfigdone(dum).BackColor = LtGray
        flexform.BBIT(dum).BackColor = LtGray
```

303

```
        Next dum

        Board_Bit

End Sub
Function write_data(iaddr As Integer, idata As Integer, iword As
Integer) As Integer
     'MSG = write_port(user_ba(BoardNum) + 2, iaddr, 2) 'internal addr
of I/O - 256 addrs. per Flex chip
     'MSG = write_port(user_ba(BoardNum) + 0, idata, 2) 'internal data
of I/O - always 16 bit (word) write
    MSG = write_port(FlexUserCtrlPort, &H0, 1)
    MSG = write_port(user_ba(&H1) + 2, iaddr, 2) 'internal addr of I/O
- 256 addrs. per Flex chip
    MSG = write_port(user_ba(&H1) + 0, idata, 2) 'internal data of I/O
- always 16 bit (word) write
End Function
Function read_data(iaddr As Integer, idata As Integer, iword As
Integer) As Integer
    MSG = write_port(FlexUserCtrlPort, &H0, 1)
    MSG = write_port(user_ba(&H1) + 2, iaddr, 1) 'internal addr of I/O
- 256 addrs. per Flex chip
    MSG = write_port(FlexUserCtrlPort, &H1, 1)
    MSG = read_port(user_ba(&H1) + 0, idata, 2) 'internal data of I/O -
always 16 bit (word) write
End Function
```

c. global.bas

```
Attribute VB_Name = "global"
Global crystal_clk(8) As Single

'***** color definitions ****
Global Const red = &HFF&
Global Const blue = &HFF0000
Global Const green = &HFF00&
Global Const black = &H0
Global Const yellow = &HFFFF&
Global Const brown = &H80FF&
Global Const ltblue = &HFFFF00
Global Const LtGray = &H8000000F
Global Const DkGray = &H808080
Global Const Beige = &HC0FFFF
```

d. main.bas

```
Attribute VB_Name = "MainMod"
Option Explicit

Sub Main()
Dim dum As Integer, cnt As Integer, invar As Integer

'*** open running windows ***
```

```
    flexform.Show    'flexform.frm

'*** check the latch values ***
    test_boards  ' flexfunc.bas
     readPara      ' file.bas
     readRaw       ' file.bas
     readCosine    ' file.bas
     readSine      ' file.bas
     isar          ' the_isar.bas
     ' pep_test            'peptest.bas'
End Sub
```

e. the_isar.bas

```
Attribute VB_Name = "the_isar"
Option Explicit
'***** internal addresses for the ISAR program *****
Global Const phiAddr = &H10         ' for Doppler offset
Global Const gainAddr = &H20        ' for gain
Global Const tapAddr = &H30         ' for tap delay line
Global Const modPulseAddr = &H40    ' for modulated pulse readback
Global Const feedback = &H60        ' for reading back values

Public Sub isar()

Dim batchCnt As Integer
Dim pulseCnt As Integer
Dim intraPulseCnt As Integer
Dim tapCnt As Integer
Dim Tap(3) As Integer
Dim Phase(3) As Integer
Dim rgain(3) As Integer
Dim GainOutI(3) As Integer
Dim GainOutQ(3) As Integer
Dim Acc(3) As Integer
Dim Lut(6) As Integer
Dim finalAcc(4) As Integer
Dim tmp As Integer
Dim dummy1, dummy2 As Double
Dim LutI As Double
Dim LutQ As Double
Dim GainOutIdec, GainOutQdec As Double
Dim dummy3, dummy4 As Double
Dim idx As Integer
Dim sumI, sumQ As Double
Dim finalAccI, finalAccQ As Double
Dim finalAccI_NEW, finalAccQ_NEW As Double 'for test of Public Function
numFormConv
Dim number As Integer    'for test of Public Function
twoComplement2Float
Dim test As Double       'for test of Public Function
twoComplement2Float
Dim n As Integer         'for test of Public Function
twoComplement2Float
```

305

```
'load data files (file.bas)
readPara
readRaw

'Open file for write
Open "lets_check.txt" For Output As #1
Open "imagei.txt" For Output As #2
Open "imageq.txt" For Output As #3

'test of Public Function twoComplement2Float
'Print #1, "Test of Public Function twoComplement2Float: "
'n = 4 'number of bits
'For number = 0 To (2 ^ n - 1)
'    test = twoComplement2Float(number, n) '* 2 ^ (n - 1)
'    Print #1, number, "=", test 'print to lets_check
'Next number

'Initialize gain values
flexform.TGain(0).Text = Gain(0)
flexform.TGain(1).Text = Gain(1)
flexform.TGain(2).Text = Gain(2)

'Reset tap-delay line
MSG = write_data(tapAddr, 0, 2)
MSG = write_data(tapAddr, 1, 2)

'loop for batch
For batchCnt = 0 To nDopplerCell - 1
    '
    'loop for intra-pulse: repeat for number of range gates + target
extent
    For intraPulseCnt = 0 To (nRangeCell + targetExtent - 1)
        '----------------------------------------------------------
_
        'Write Phi and Gain values
        For tapCnt = 0 To targetExtent - 1
            '
            'load Doppler offset parameters
            'MSG = write_data(phiAddr + tapCnt, Phi(nDopplerCell -
batchCnt - 1, tapCnt), 2)
            MSG = write_data(phiAddr + tapCnt, Phi(batchCnt, tapCnt),
2)
            '
            Print #1, "batchCnt=", batchCnt
            Print #1, "intraPulseCnt=", intraPulseCnt
            Print #1, "tapCnt=", tapCnt
            Print #1, "Phi(batchCnt, tapCnt)=", Phi(batchCnt, tapCnt)
            '
            'load gain parameters
            MSG = write_data(gainAddr + tapCnt, Gain(tapCnt), 2)
            '
        Next tapCnt
        '
        '----------------------------------------------------------
_
```

```
        'Read back gain values
        For idx = 0 To 2
            MSG = read_data(gainAddr + idx, rgain(idx), 2)
            rgain(idx) = rgain(idx) And &H7
            flexform.TGain(idx).Text = rgain(idx)
            Print #1, "rgain=", rgain(idx)
        Next idx
        '------------------------------------------------------------
-

        'Read back phi values
        For idx = 0 To 2
            MSG = read_data(phiAddr + idx, Phase(idx), 2)
            Phase(idx) = Phase(idx) And &H1F
            flexform.TPhi(idx).Text = Phase(idx)
            Print #1, "Phase=", Phase(idx)
            Print #1, "Phase(idx)=", twoComplement2Float(Phase(idx), 5)
* (2 ^ 7), "converted from 2-complement"

        Next idx
        '------------------------------------------------------------
-

        ' Strobe into delay line
        MSG = write_data(tapAddr + 1, 0, 2) ' ripple data
        MSG = write_data(tapAddr + 2, DRFM(batchCnt, intraPulseCnt), 2)
' Strobe in new data
        Print #1, "DRFM(batchCnt, intraPulseCnt)=", DRFM(batchCnt,
intraPulseCnt)
        Print #1, "DRFM(batchCnt, intraPulseCnt)=",
twoComplement2Float(DRFM(batchCnt, intraPulseCnt), 5) * (2 ^ 7),
"converted from 2-complement"
            '
        'Read back tap values
        For idx = 0 To 2
            MSG = read_data(tapAddr + idx, Tap(idx), 2)
            Tap(idx) = (Tap(idx) And &H1F)
            flexform.TTap(idx).Text = Tap(idx)
            Print #1, "Tap=", Tap(idx)
        Next idx
        '------------------------------------------------------------
-

        'Read back ph_acc values (mod 32)
        For idx = 0 To 2
            MSG = write_data(feedback, idx, 2)
            MSG = read_data(feedback, Acc(idx), 2)
            Acc(idx) = Acc(idx) And &H1F
            flexform.TAcc(idx).Text = Acc(idx)
            Print #1, "Acc=", Acc(idx)
            Print #1, "Acc(idx)=", twoComplement2Float(Acc(idx), 5) *
(2 ^ 7), "converted from 2-complement"

        Next idx
        '------------------------------------------------------------
-

        'Read back LUT values
        tmp = 3
        'sumI = 0
```

```
          'sumQ = 0
          For idx = 0 To 2
              MSG = write_data(feedback, tmp + (idx * 2), 2)
              MSG = read_data(feedback, Lut(idx * 2), 2)
              MSG = write_data(feedback, tmp + (idx * 2 + 1), 2)
              MSG = read_data(feedback, Lut(idx * 2 + 1), 2)
              Lut(idx * 2 + 1) = (Lut(idx * 2 + 1) And &HFF)
              Lut(idx * 2) = (Lut(idx * 2) And &HFF)
              dummy1 = Val(Format(twoComplement2Float(Lut(idx * 2), 8),
"##.###"))
              dummy2 = Val(Format(twoComplement2Float(Lut(idx * 2 + 1),
8), "##.###"))
              LutI = twoComplement2Float(Lut(idx * 2), 8)
              LutQ = twoComplement2Float(Lut(idx * 2 + 1), 8)
              flexform.TLut(idx).Text = Str(dummy1) & "," & Str(dummy2)
              'Print #1, "LUT(idx*2)=", Lut(idx * 2), "dummy1=", dummy1,
"LutI=", LutI
              'Print #1, "LUT(idx*2+1)=", Lut(idx * 2 + 1), "dummy2=",
dummy2, "LutQ=", LutQ
              'Yeo's output of intermediate results
              'flexform.TPhi(idx).Text = Hex(Lut(idx * 2)) & "," &
Hex(Lut(idx * 2 + 1))
              'sumI = sumI + twoComplement2Float(Lut(idx * 2), 8) *
Gain(idx)
              'sumQ = sumQ + twoComplement2Float(Lut(idx * 2 + 1), 8) *
Gain(idx)
              Print #1, "idx =", idx
              Print #1, "LutI("; idx; ")=", " ", Lut(idx * 2), "LutI=",
LutI
              Print #1, "LutQ("; idx; ")=", " ", Lut(idx * 2 + 1),
"LutQ=", LutQ
              Print #1, "Gain("; idx; ")=", " ", Gain(idx)
          Next idx
          'flexform.TSum(0).Text = Str(sumI)
          'flexform.TSum(1).Text = Str(sumQ)
          'flexform.TSum(2).Text = Str(0)
          'flexform.TSum(3).Text = Str(0)
          'Print #2, sumI
          'Print #3, sumQ
          '--------------------------------------------------------------
-

          'Read back gain block outputs (I channel - 11 bits)
          tmp = 13
          For idx = 0 To 2
              MSG = write_data(feedback, (tmp + idx), 2)
              MSG = read_data(feedback, GainOutI(idx), 2)
              GainOutI(idx) = GainOutI(idx) And &H7FF
              'GainOutIdec =
Val(Format(twoComplement2Float(GainOutI(idx), 11), "##.###"))
              GainOutIdec = twoComplement2Float(GainOutI(idx), 11)
              Print #1, "GainOutI("; idx; ")=", GainOutI(idx),
"GainOutIdec=", GainOutIdec
          Next idx
          '
          'Read back gain block outputs (Q channel - 11 bits)
          tmp = 16
```

```
        For idx = 0 To 2
            MSG = write_data(feedback, (tmp + idx), 2)
            MSG = read_data(feedback, GainOutQ(idx), 2)
            GainOutQ(idx) = GainOutQ(idx) And &H7FF
            'GainOutQdec =
Val(Format(twoComplement2Float(GainOutQ(idx), 11), "###.###"))
            GainOutQdec = twoComplement2Float(GainOutQ(idx), 11)
            Print #1, "GainOutQ("; idx; ")=", GainOutQ(idx),
"GainOutQdec=", GainOutQdec
        Next idx
        '--------------------------------------------------------------
 -


        DoEvents
        '--------------------------------------------------------------
 -

        'Read back sum values (modified code by Stig, 2 Aug -99)
        tmp = 9
        finalAccI = 0
        finalAccQ = 0
        '
        'Read back sum values (I channel - 13 bits)
        MSG = write_data(feedback, tmp + 0, 2)
        MSG = read_data(feedback, finalAcc(0), 2)
        finalAcc(0) = finalAcc(0) And &H1FFF
        dummy3 = Val(Format(twoComplement2Float(finalAcc(0), 13),
"####.###"))
        'flexform.TSum(0).Text = Str(dummy3)
        finalAccI = twoComplement2Float(finalAcc(0), 13)
        finalAccI_NEW = numFormConv(finalAcc(0))
        flexform.TSum(0).Text = Str(finalAccI)
        Print #1, "finalAcc(0)=", " ", finalAcc(0), "finalAccI = ",
finalAccI
        Print #1, " - To test the new numFormConv function",
"finalAccIN = ", finalAccI_NEW
        Print #2, finalAccI
        '
        'Read back sum values (Q channel - 13 bits)
        MSG = write_data(feedback, tmp + 1, 2)
        MSG = read_data(feedback, finalAcc(1), 2)
        'Print #1, "TTTEEESSSTTT  finalAcc(1) =  ", finalAcc(1)
        finalAcc(1) = finalAcc(1) And &H1FFF
        dummy4 = Val(Format(twoComplement2Float(finalAcc(1), 13),
"####.###"))
        'flexform.TSum(1).Text = Str(dummy4)
        finalAccQ = twoComplement2Float(finalAcc(1), 13)
        finalAccQ_NEW = numFormConv(finalAcc(1))
        flexform.TSum(1).Text = Str(finalAccQ)
        Print #1, "finalAcc(1)=", " ", finalAcc(1), "finalAccQ=",
finalAccQ
        Print #1, " - To test the new numFormConv function",
"finalAccQN = ", finalAccQ_NEW
        Print #1, "--------------------------------------------------------------
-----------------"
        Print #3, finalAccQ
        '
```

```
            'flexform.TSum(2).Text = Str(0)
            'flexform.TSum(3).Text = Str(0)
            '
            'tmp = 9
            'For idx = 0 To 1
            '     MSG = write_data(feedback, tmp + idx, 2)
            '     MSG = read_data(feedback, finalAcc(idx), 2)
            '     finalAcc(idx) = finalAcc(idx) And &H1FFF
            '     dummy3 = Val(Format(twoComplement2Float(finalAcc(idx),
13), "##.###"))
            '     Print #1, "FINALACC(idx)=", finalAcc(idx), "dummy3=",
dummy3
            '     flexform.TSum(idx).Text = Str(dummy3)
            '     finalAc = sumQ + twoComplement2Float(GainOutQ(idx), 11)
            'Next idx
            'Print #2, finalAcc(0)
            'Print #3, finalAcc(1)
            '
            flexform.Clock.Text = Str(batchCnt) + "," + Str(intraPulseCnt +
1)
    Next intraPulseCnt     'end intra-pulse loop
Next batchCnt     'end batch loop
Close #1
Close #2
Close #3
End Sub
Public Function twoComplement2Float(a As Integer, nbits As Integer) As
Double 'modified by Stig Ekestorm, 4 Aug -99
Dim dummy1, dummy2, dummy3, dummy4 As Integer

dummy1 = 2 ^ (nbits - 1)
dummy4 = 2 ^ nbits
dummy2 = dummy1 - 1
If (a >= dummy1) Then          ' negative number test
    dummy3 = dummy1 - (a And dummy2)
    twoComplement2Float = -1 * dummy3 / (2 ^ 7) '/ dummy1 '(divide by
128 to put the decimal point at the right position)
Else
    twoComplement2Float = a / (2 ^ 7) '/ dummy1 '(divide by 128 to put
the decimal point at the right position)
End If

If a >= 2 ^ nbits Then
    twoComplement2Float = -1111
End If
End Function
Public Function numFormConv(a As Integer) As Double 'created by Prof.
Fouts, 4 Aug -99
Dim tempvar As Integer

tempvar = &H1FFF And a
If (tempvar >= 4096) Then          ' negative number test
    tempvar = tempvar Xor &H1FFF
    tempvar = tempvar + 1
    numFormConv = -1 * tempvar / (2 ^ 7)
Else
```

```
        numFormConv = tempvar / (2 ^ 7)
End If
End Function
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. SCHEMATICS AND SYMBOLS

This Appendix contains all elements created in the five different design levels. Every Figure has two parts. The upper part shows the circuit as build in S-Edit, where the lower part shows the corresponding symbol. The regular Tanner library elements are not listed and can be found in the Tanner tools Library manual.

1.   LEVEL 1 MODULES
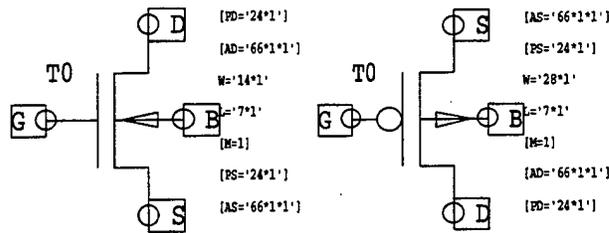
Figure 98. P-FET and N-FET Transistor Definition
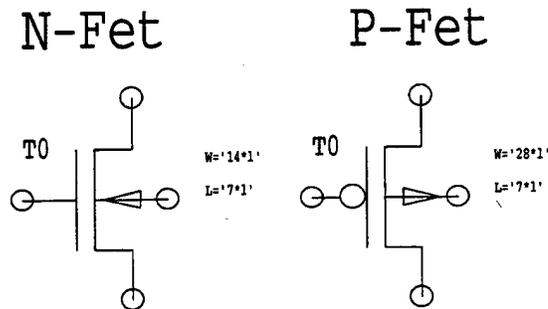
Figure 99. P-FET and N-FET Symbols
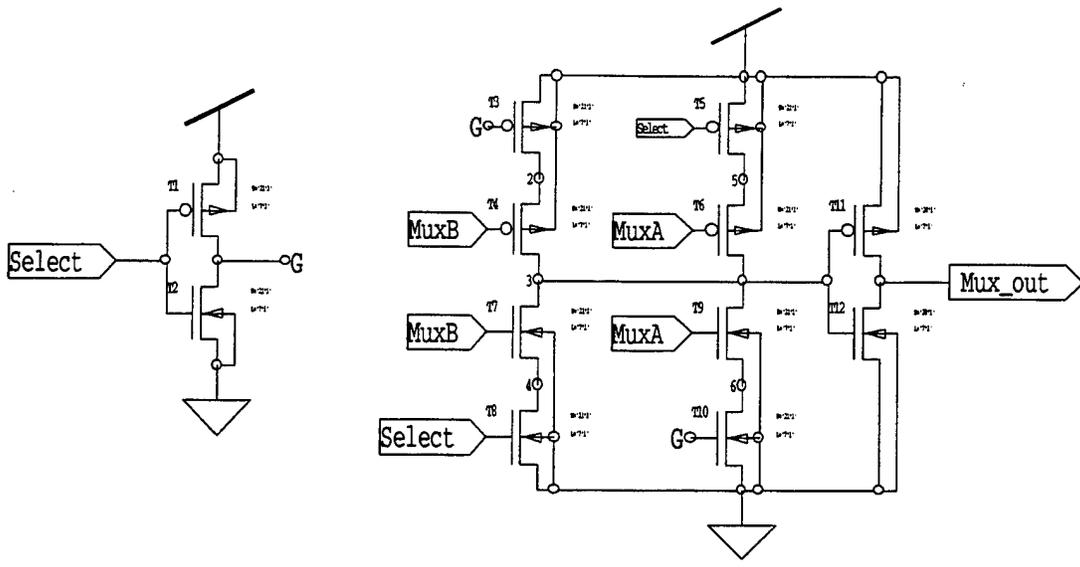
Logic Equation = (MuxA*notSelect + MuxB*Select)

Figure 100. Mux2 Circuit
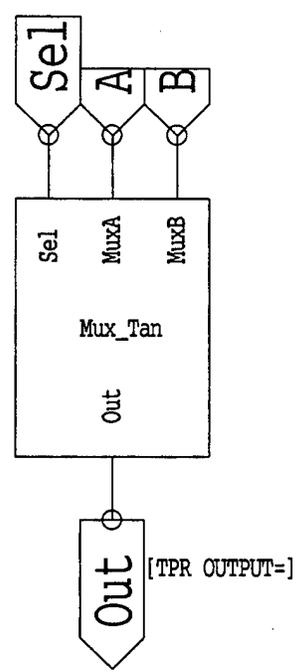
Figure 101. Mux2 Symbol (modified from Tanner's version)
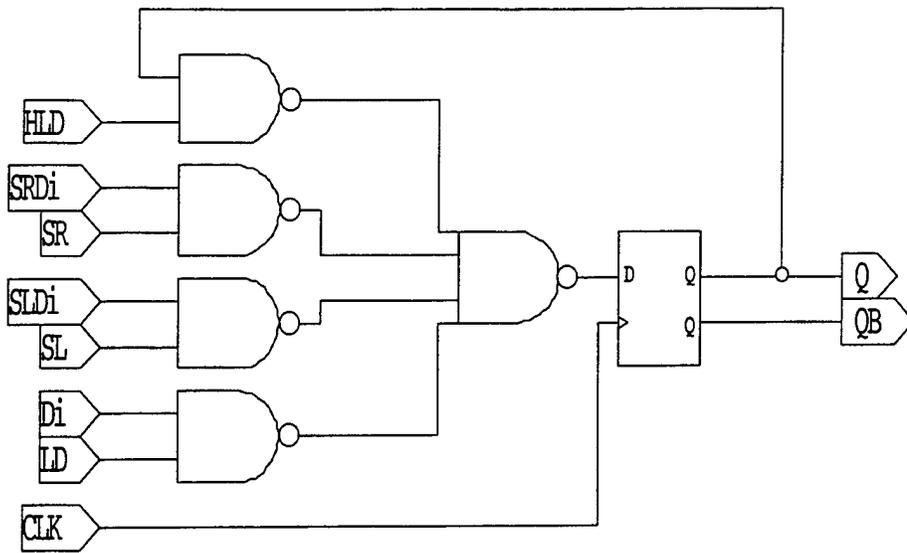
Figure 102. Register Cell Circuit
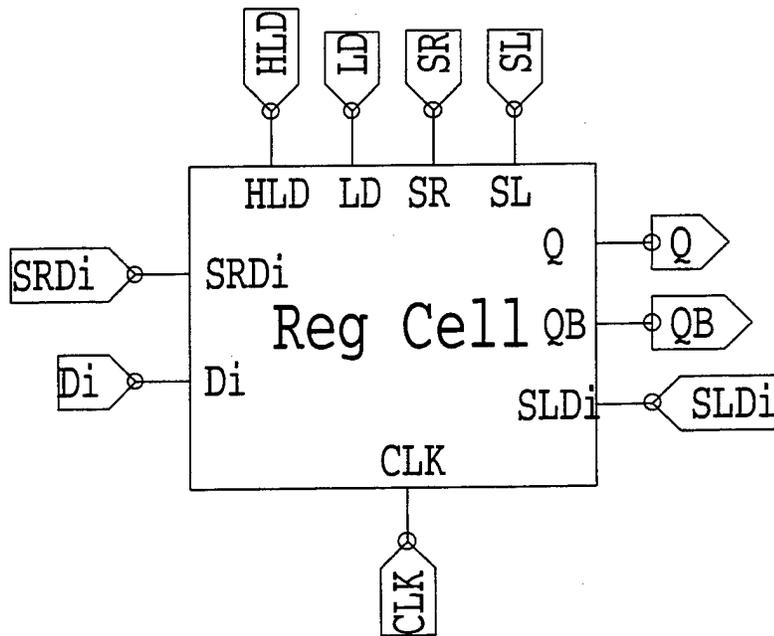
Figure 103. Register Cell Symbol

315

Register Cell with CLR   Master Latch⌐   ⌐Slave Latch

The combination of P-FET and N-FET is called a transmission gate.
P-FET pulls output high, N-Fet pulls output low
Point M is the transission between the master latch to the slave latch.
This setup is a positive-edge-triggered flip-flop that gets fed by a input tree.
The input tree produces only one valid output at a time as described in the operational notes.

Operational Note

When clock changes from low to high,
exactly one input of HLD, LD, SR, or
SL must be high, the other 3 must be low.
If all control inputs
are low, the output is cleared (low).

SRDo = shift right data out
SLDo = shift left data out
SLDi = shift left data input
SRDi = shift right data in
HLD = hold (do nothing, Q tx=Qt, notQ th=notQ t)
SR = shift right
SL = shift left
Di = data input
LD = load data
CLK = clock (positive edge triggered
Q = data output (consist of SLDo and SRdo)
QB = inverted data output

To Make N-Bit Register:

* use n cells
* connect Q of cell n to SRDi of cell n+1
* connect Q of cell n to SLDi of cell n-1
* connect HLD, SR, SL, LD, and CLK in parallel
* there will be n Di inputs; one to each cell
* there will be n Q and n QB outputs

Figure 104. D-Bit Register Cell with Synchronous Clear
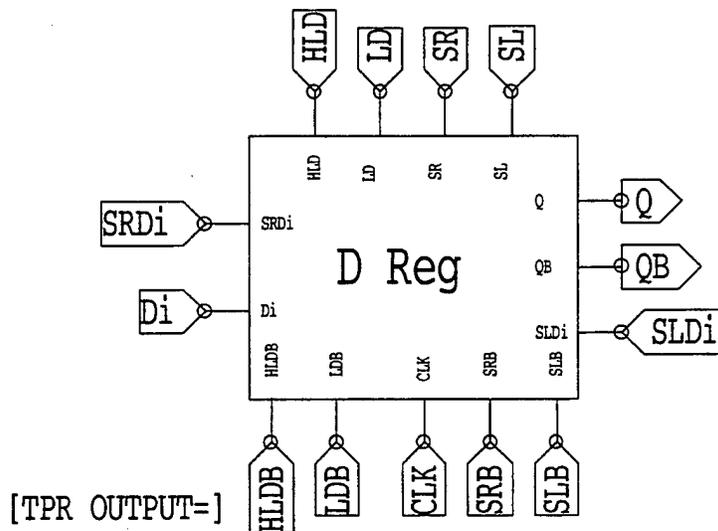
Figure 105. D-Bit Register Cell Symbol

316
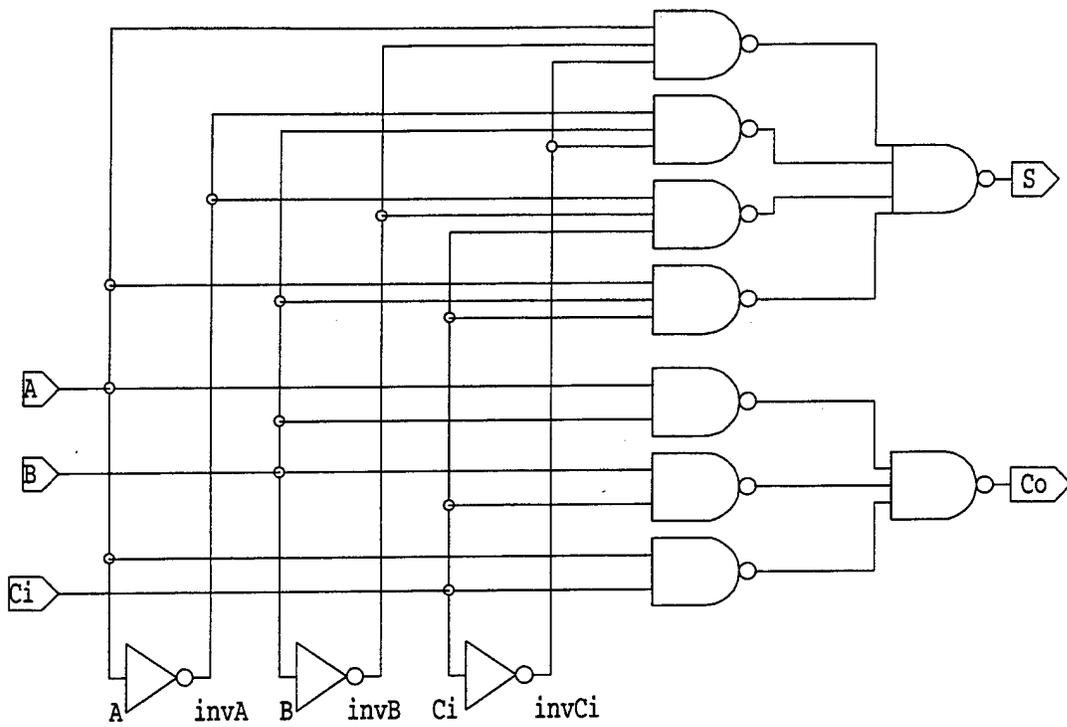
Figure 106. Adder Cell Circuit
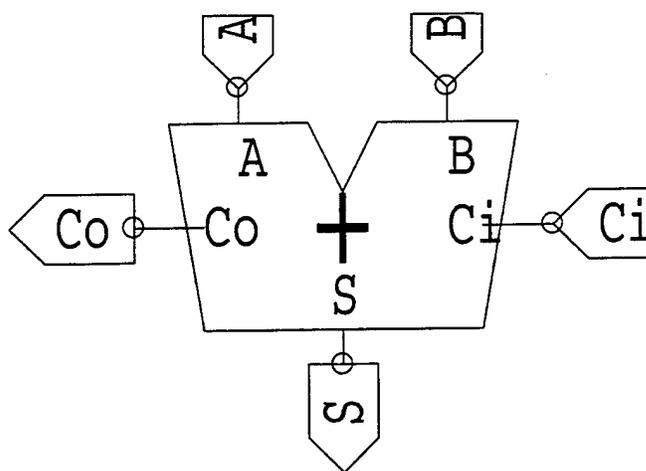
Figure 107. Adder Cell Symbol

2. LEVEL 2 MODULES
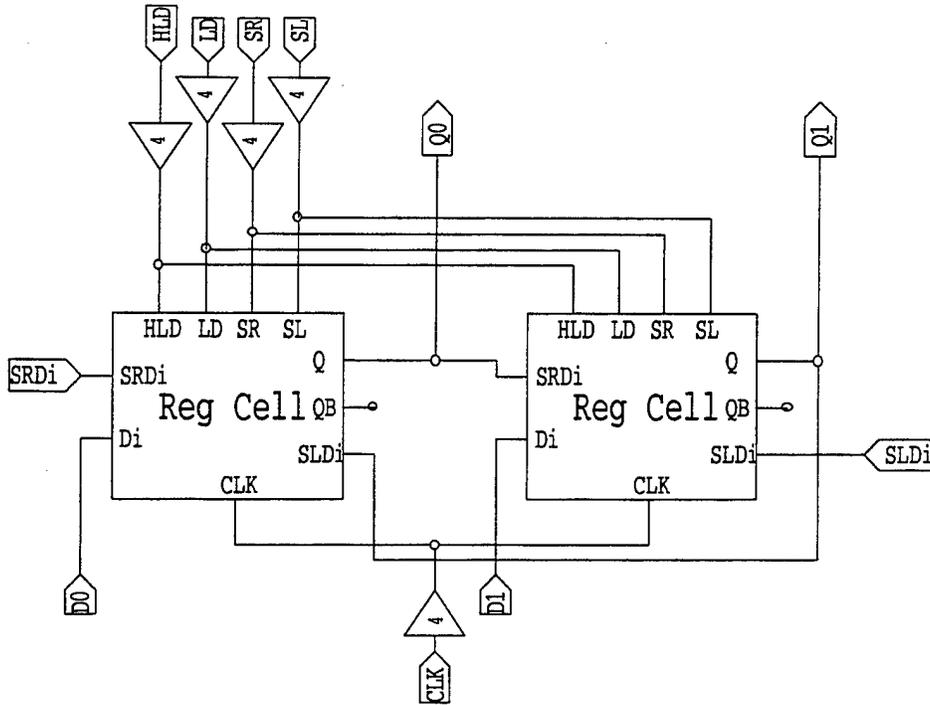
Figure 108. 2-Bit Register Circuit
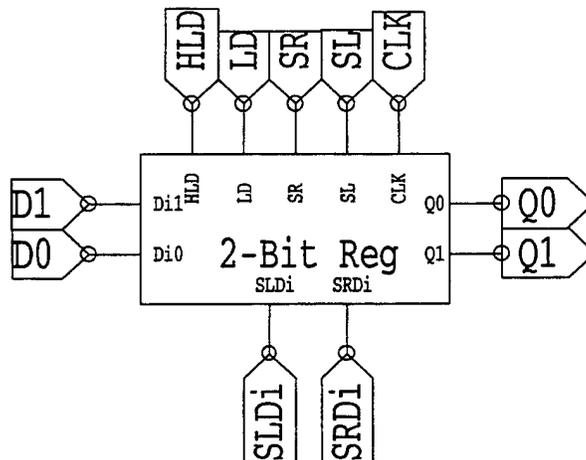
Figure 109. 2-Bit Register Symbol

318
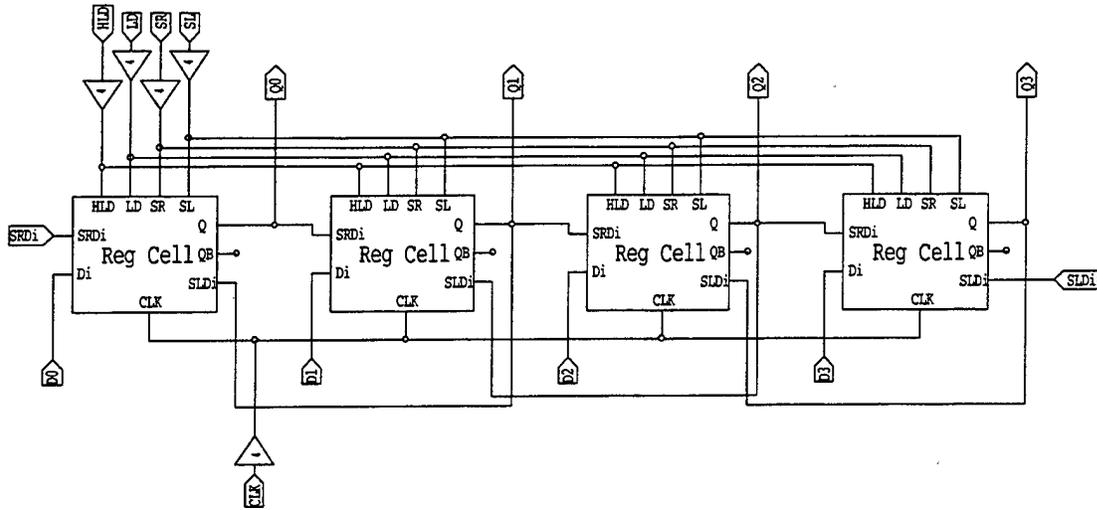
Figure 110. 4-Bit Register Circuit
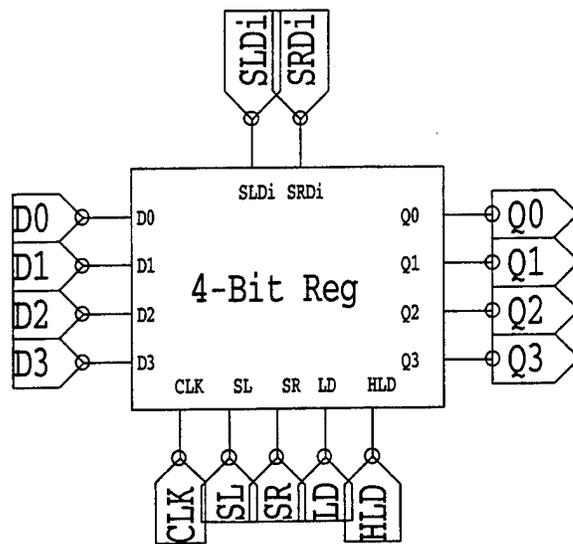
Figure 111. 4-Bit Register Symbol

319
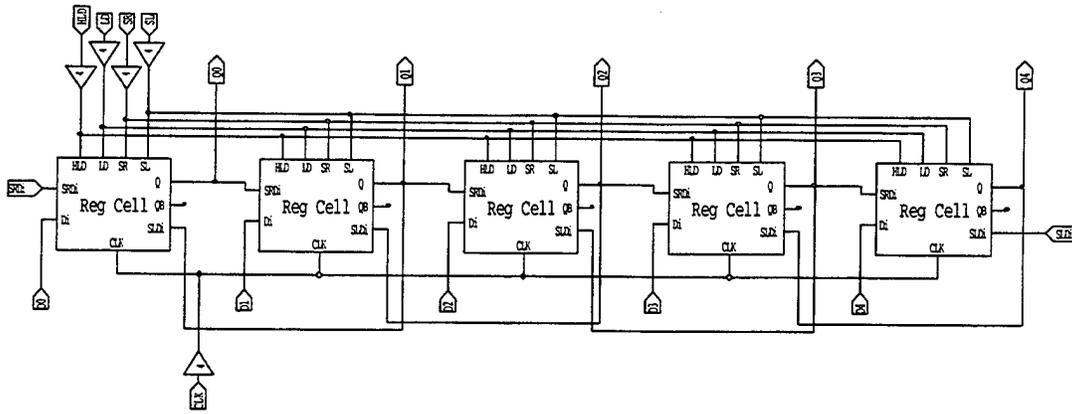
Figure 112. 5-Bit Register Circuit
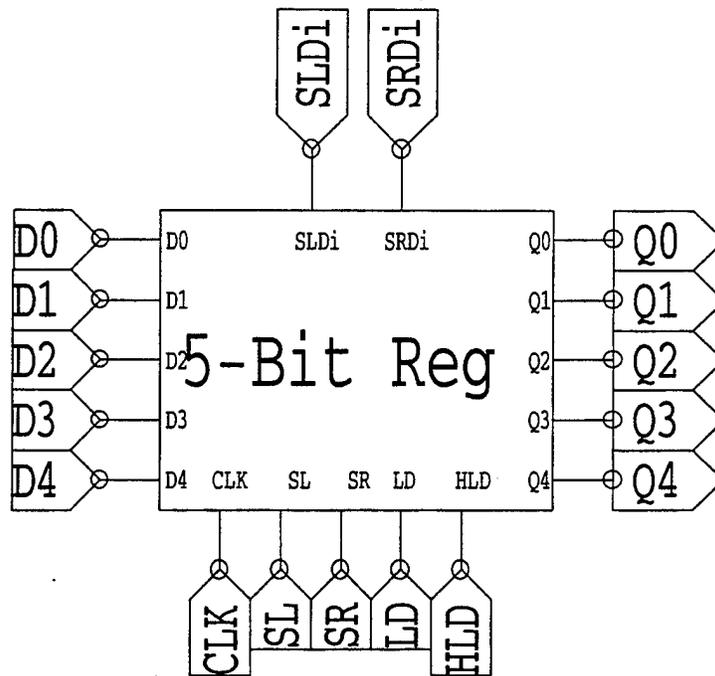
Figure 113. 5-Bit Register Symbol
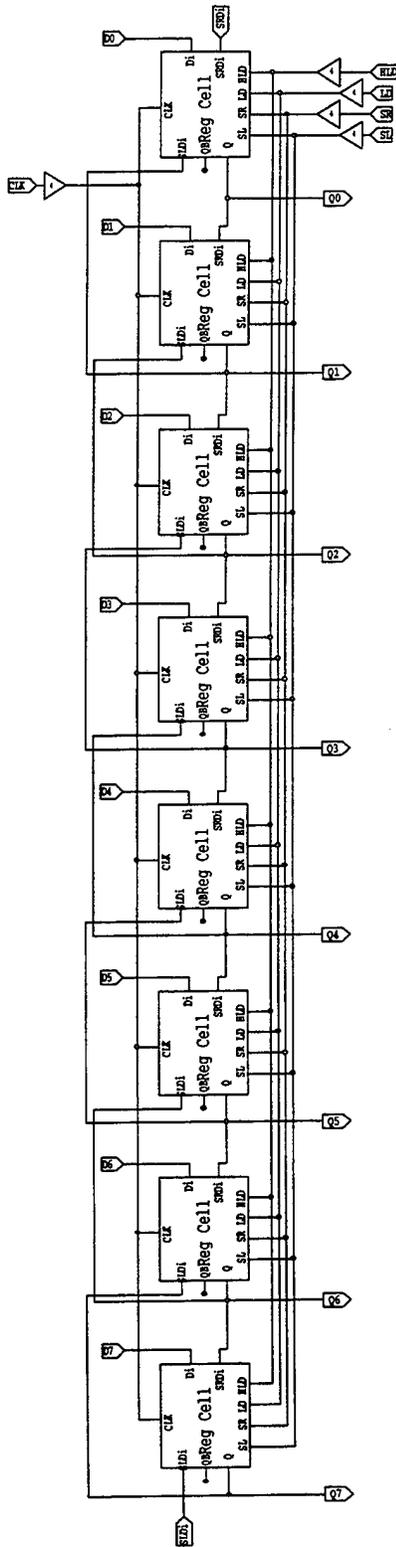
Figure 114. 8-Bit Register Circuit
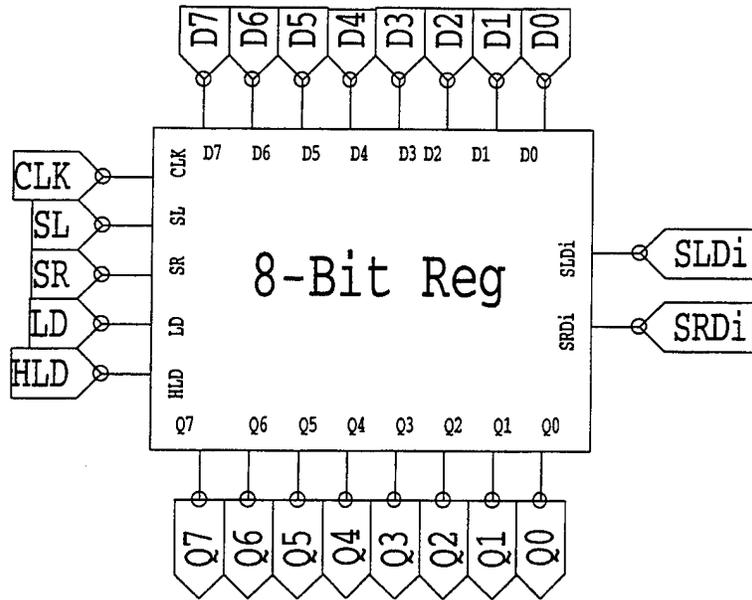
Figure 115. 8-Bit Register Symbol
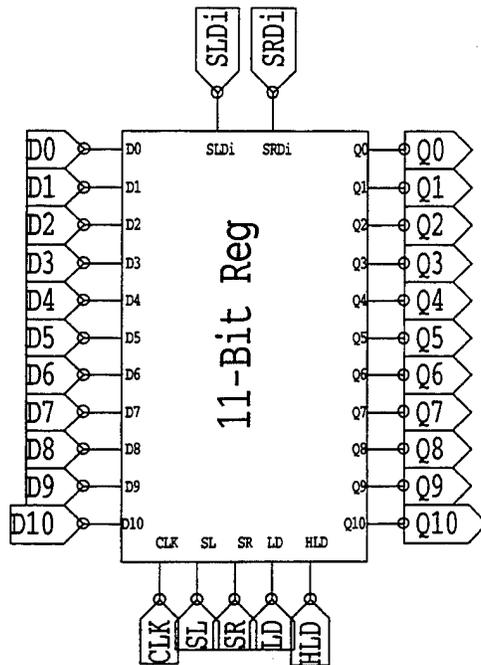
Figure 116. 11-Bit Register Symbol

322
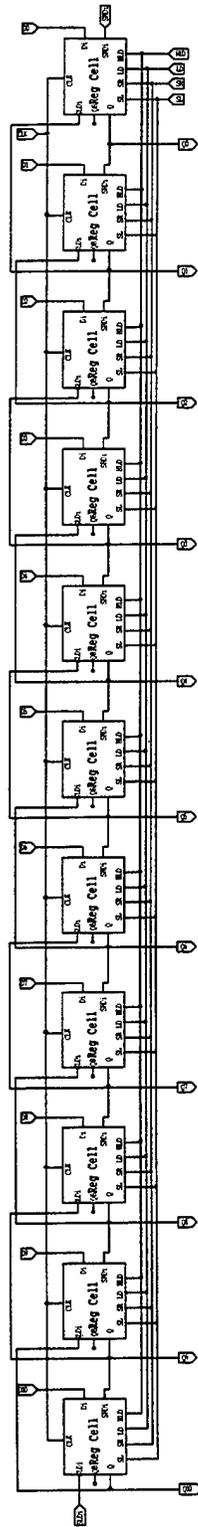
Figure 117. 11-Bit Register Symbol
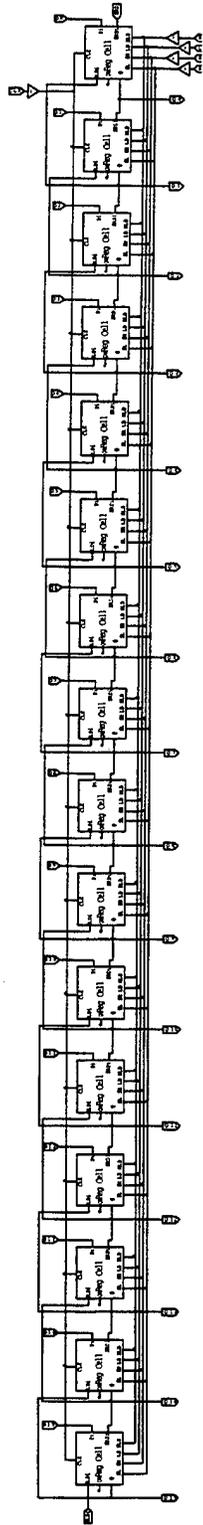
Figure 118. 16-Bit Register Circuit

324
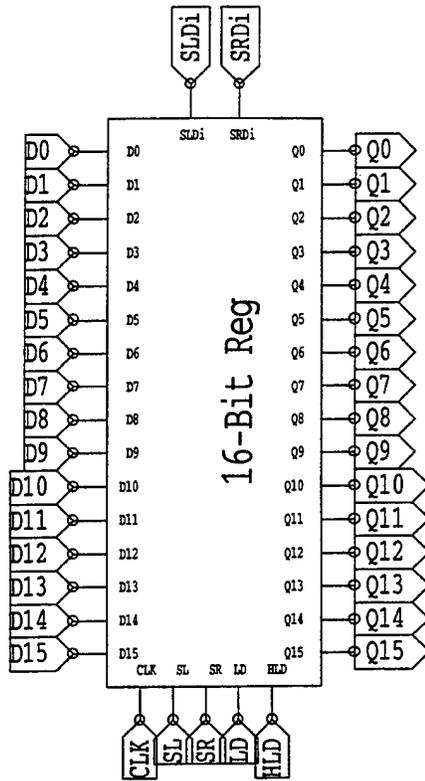
Figure 119. 16-Bit Register Symbol
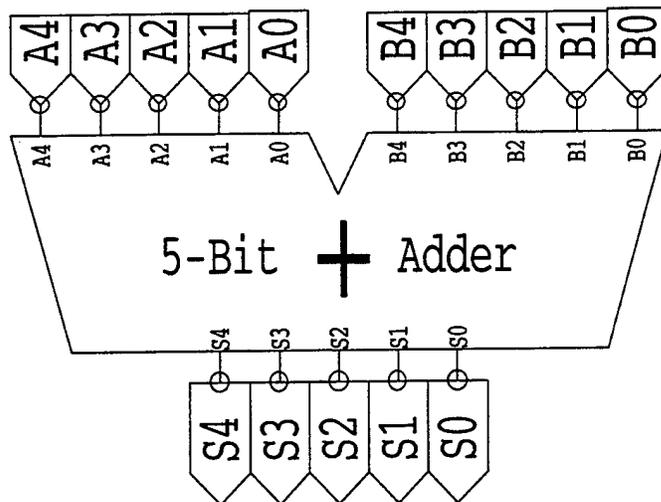
Figure 120. 5-Bit Adder Symbol
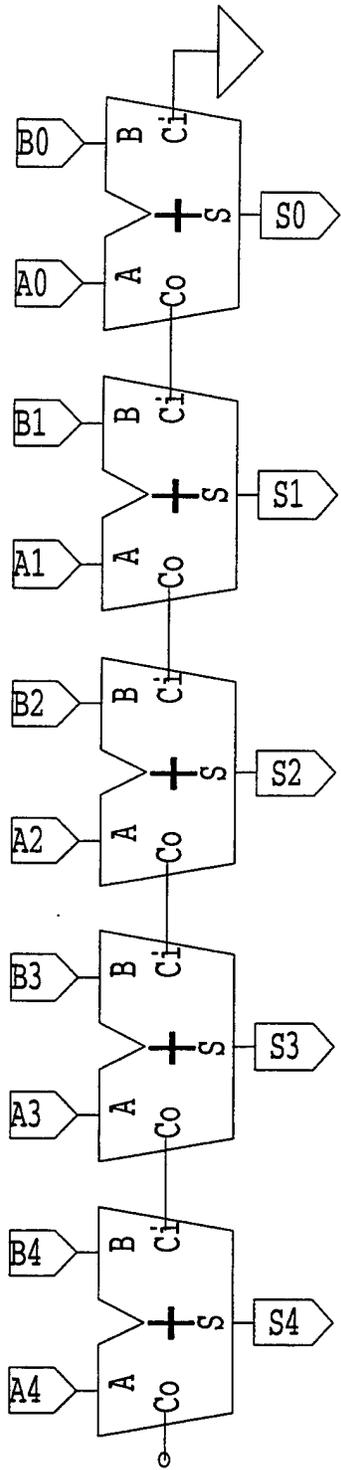
Figure 121. 5-Bit Adder Circuit
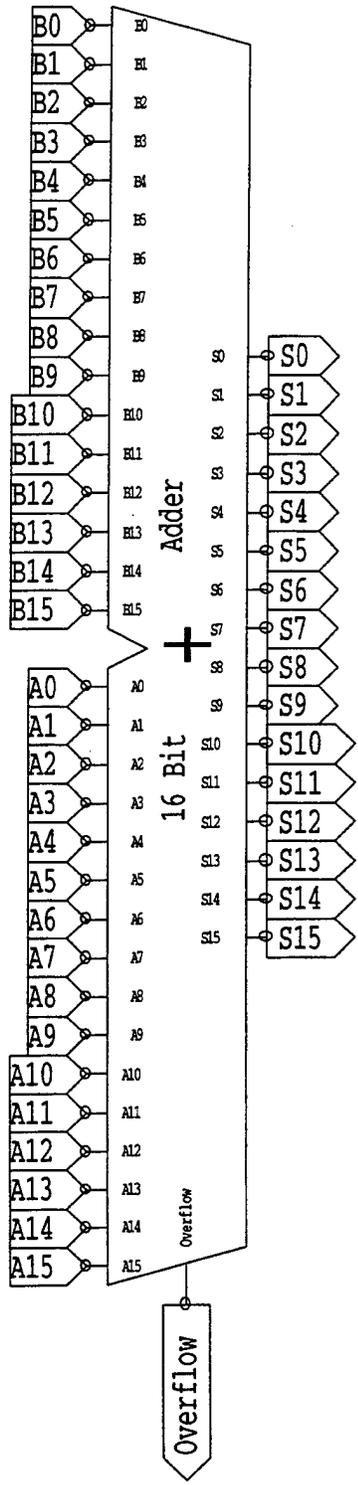
Figure 122. 16-Bit Adder Symbol
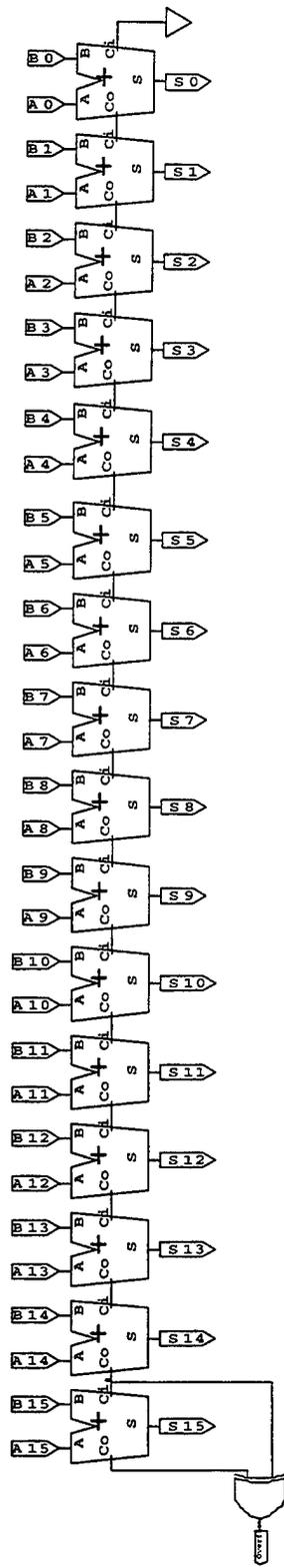
Figure 123. 16-Bit Adder Circuit
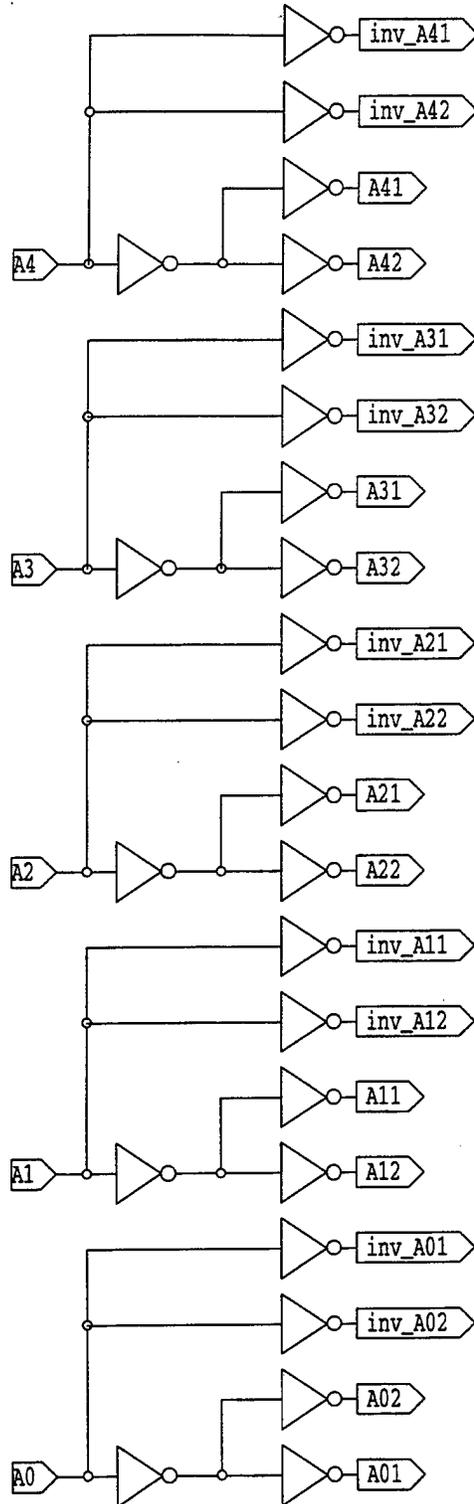
Figure 124. 5-to-32-Bit Decoder Part 1 Circuit
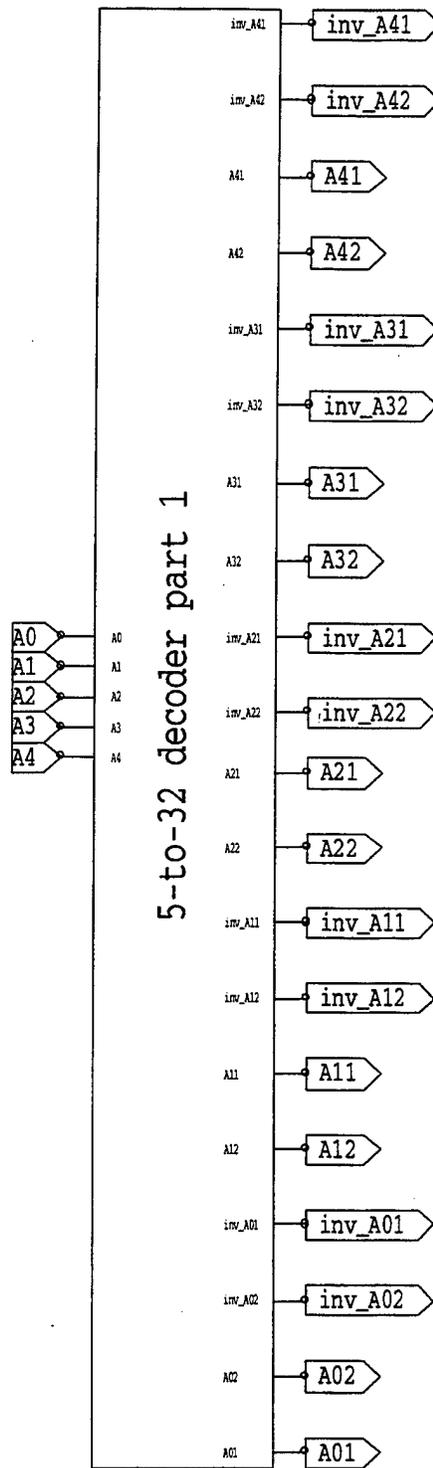
Figure 125. 5-to-32-Bit Decoder Part1 Symbol

330
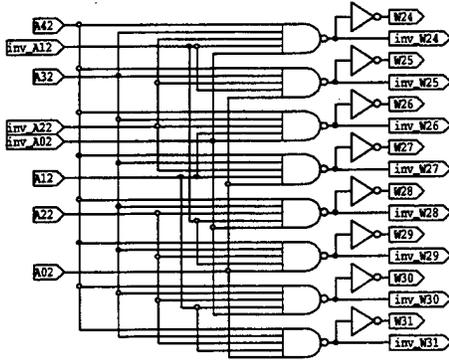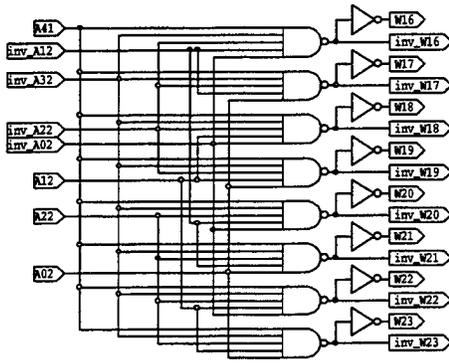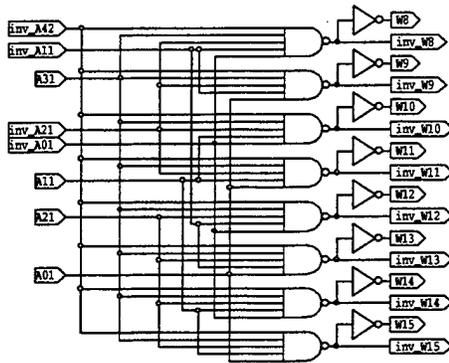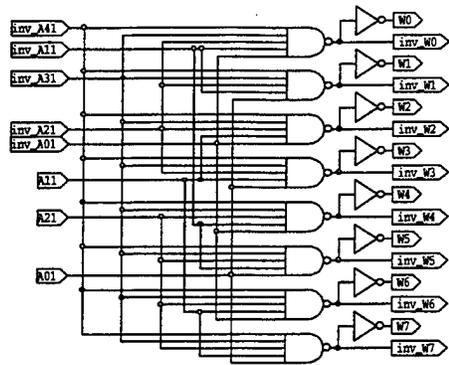
Figure 126. 5-to-32-Bit Decoder Part2 Circuit

331
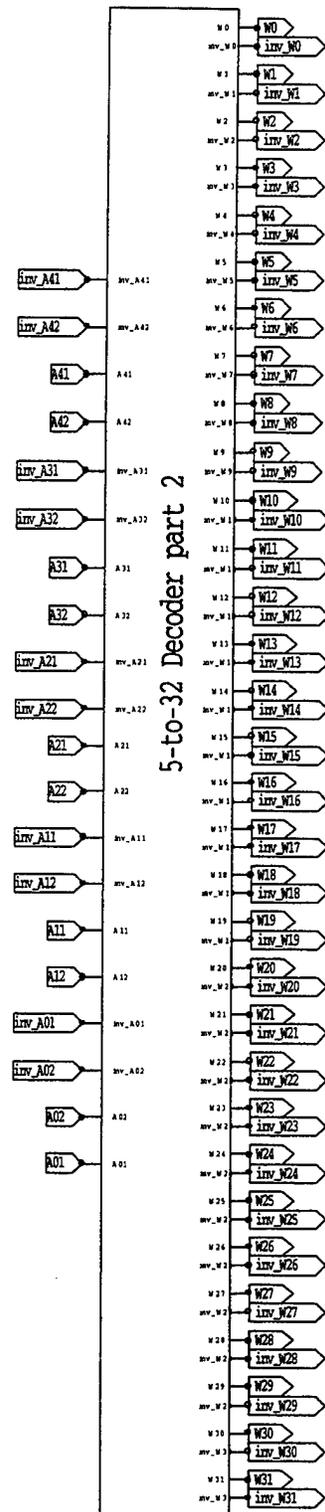
Figure 127. 5-to-32-Bit Decoder Part2 Symbol
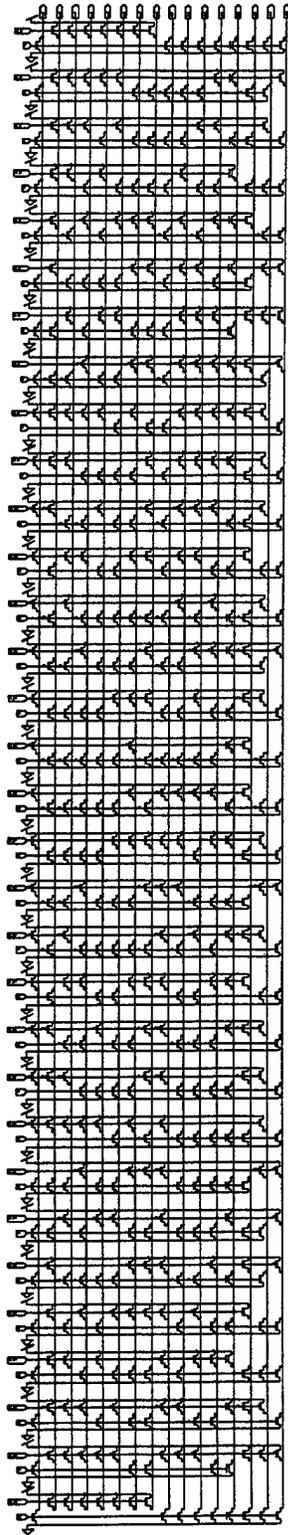
Figure 128. Programmed LUT Module Circuit

333
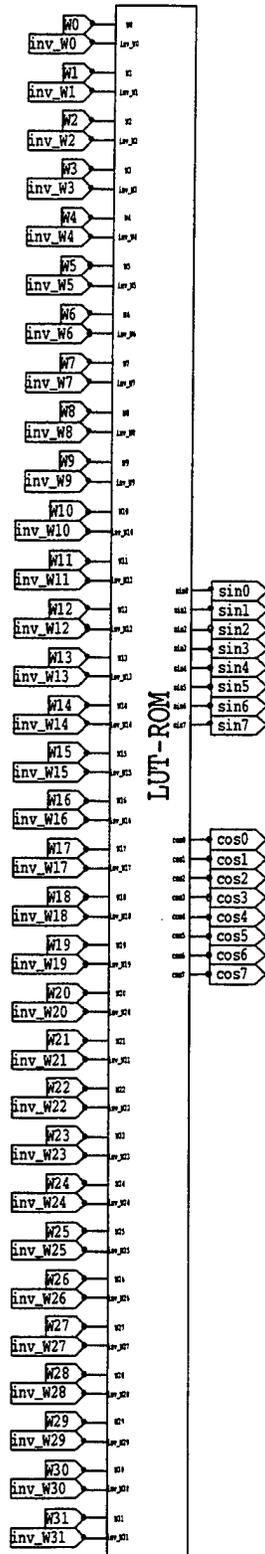
Figure 129. LUT Symbol

334
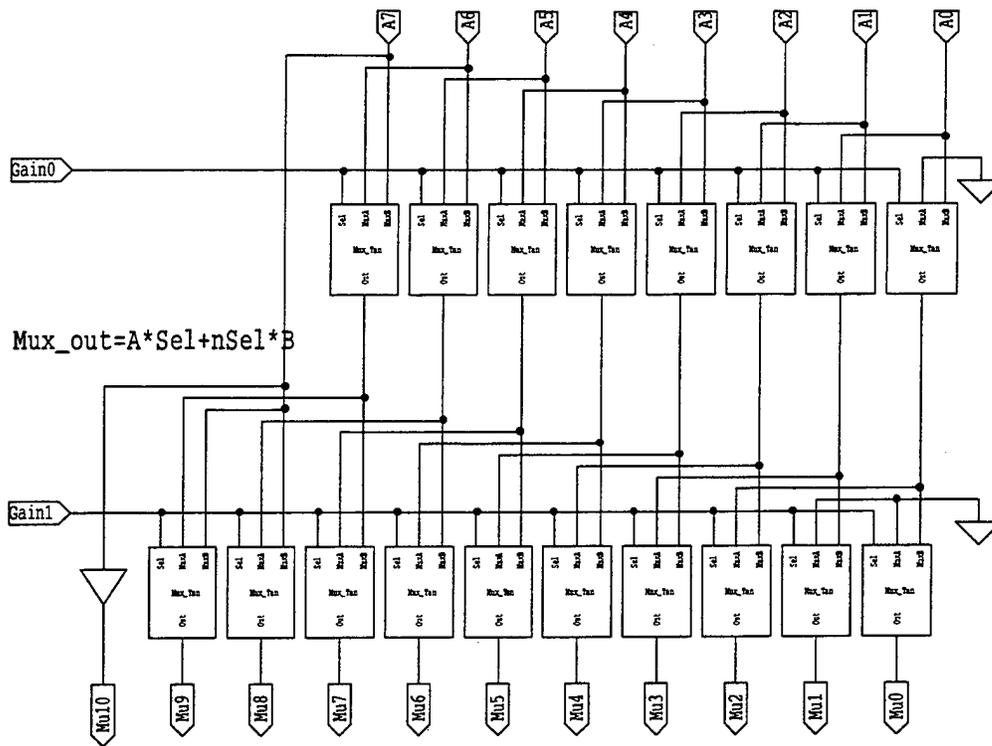
Figure 130. Gain-Shifter Circuit

Mux_out=A*Sel+nSel*B
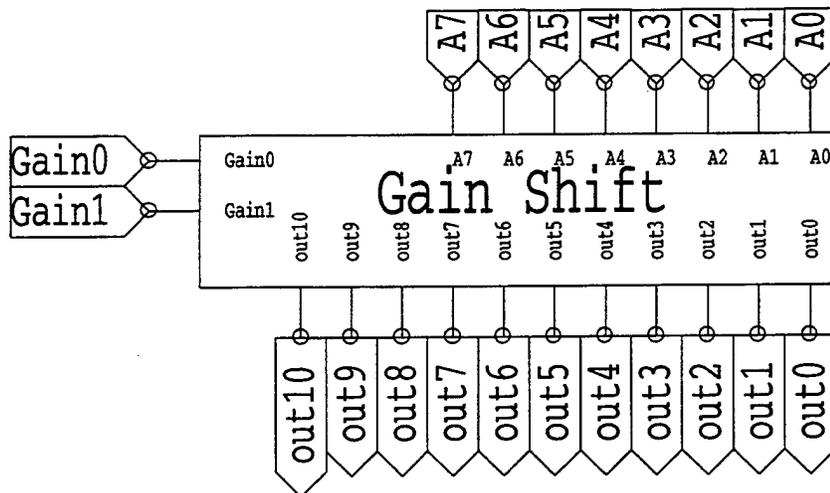
Figure 131. Gain-Shifter Symbol

335

3. LEVEL 3 MODULES
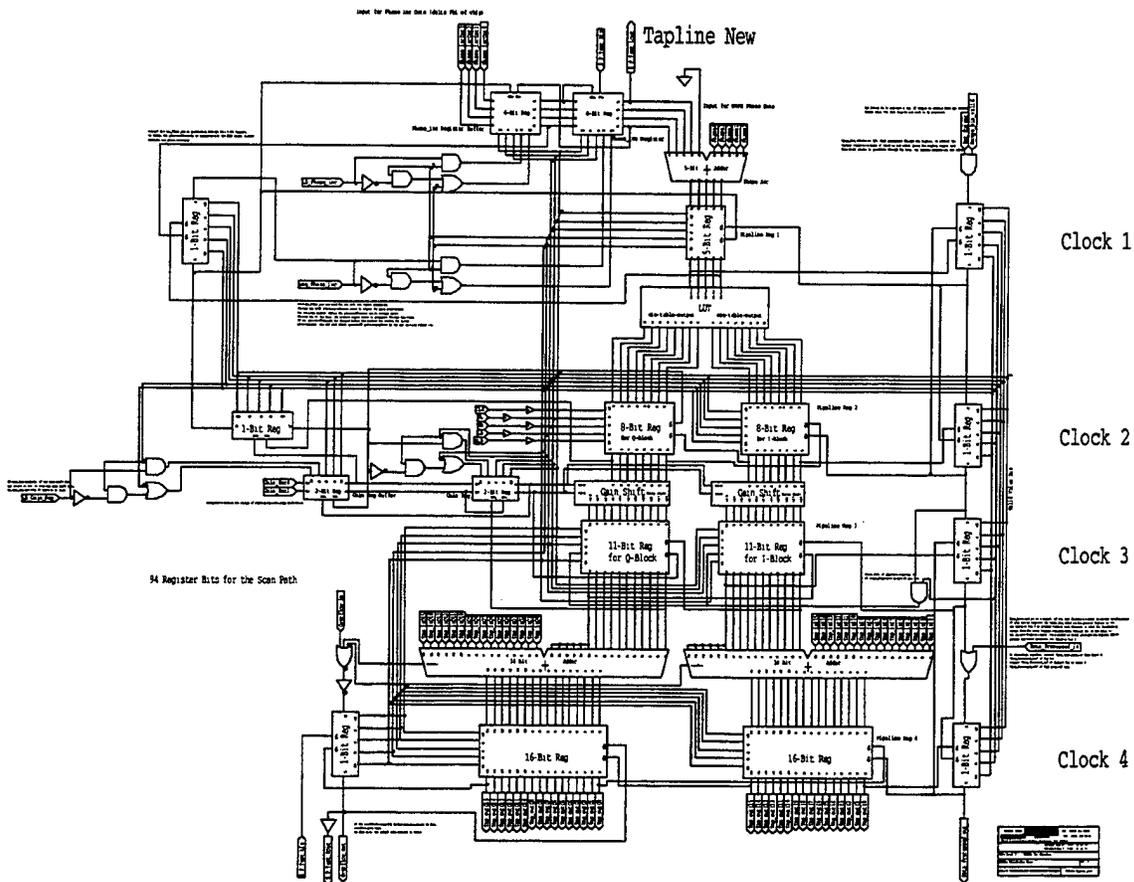
Figure 132. Tapline Circuit
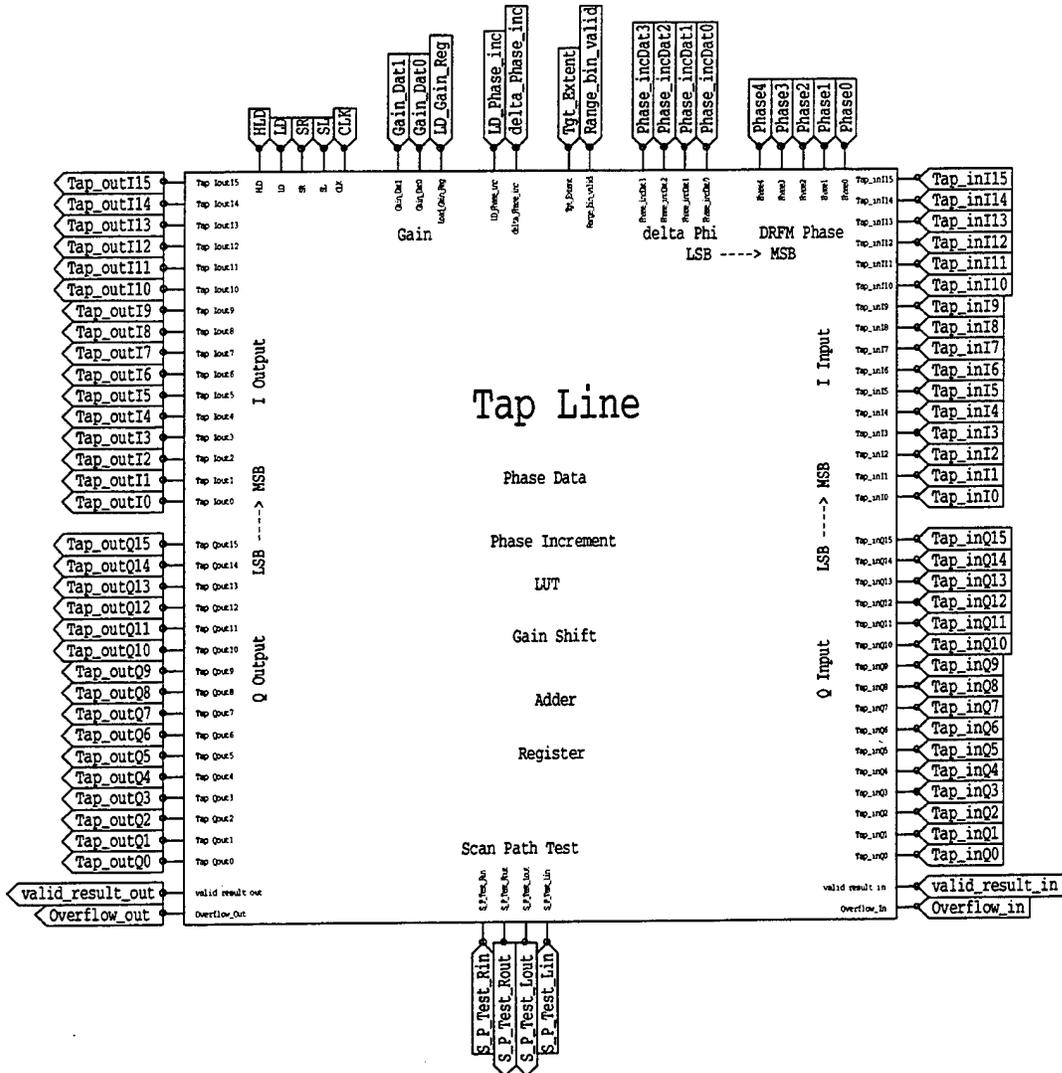
Figure 133. Tapline Symbol

4. LEVEL 4 MODULES
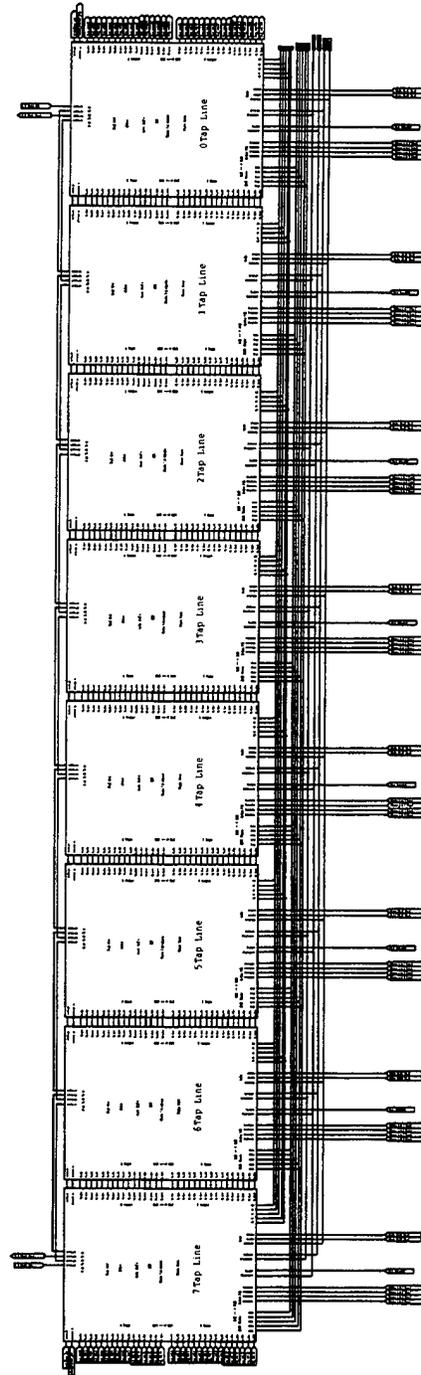
Figure 134. Supertap Circuit
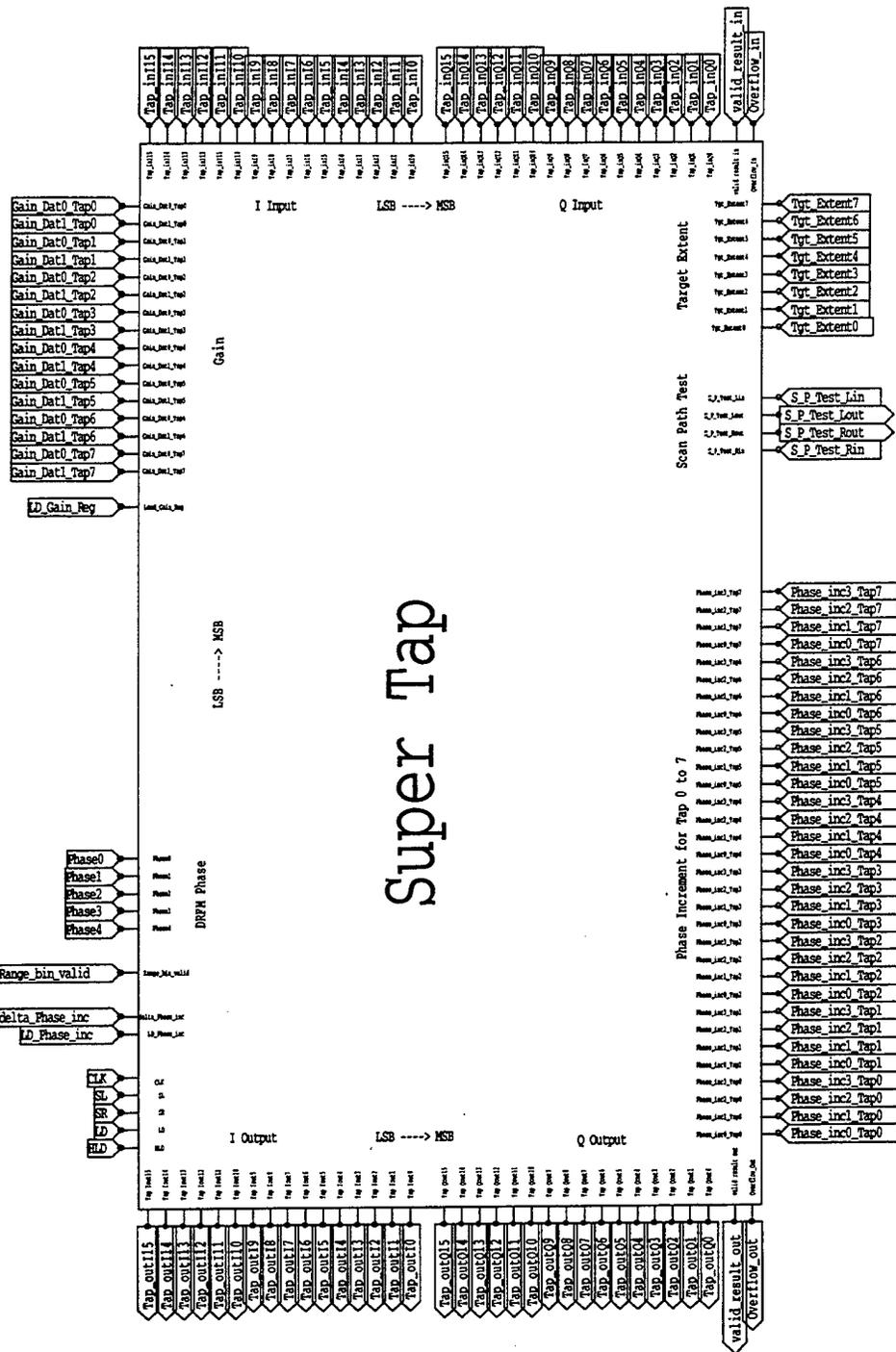
Figure 135. Supertap Symbol

339

5. LEVEL 5 MODULES
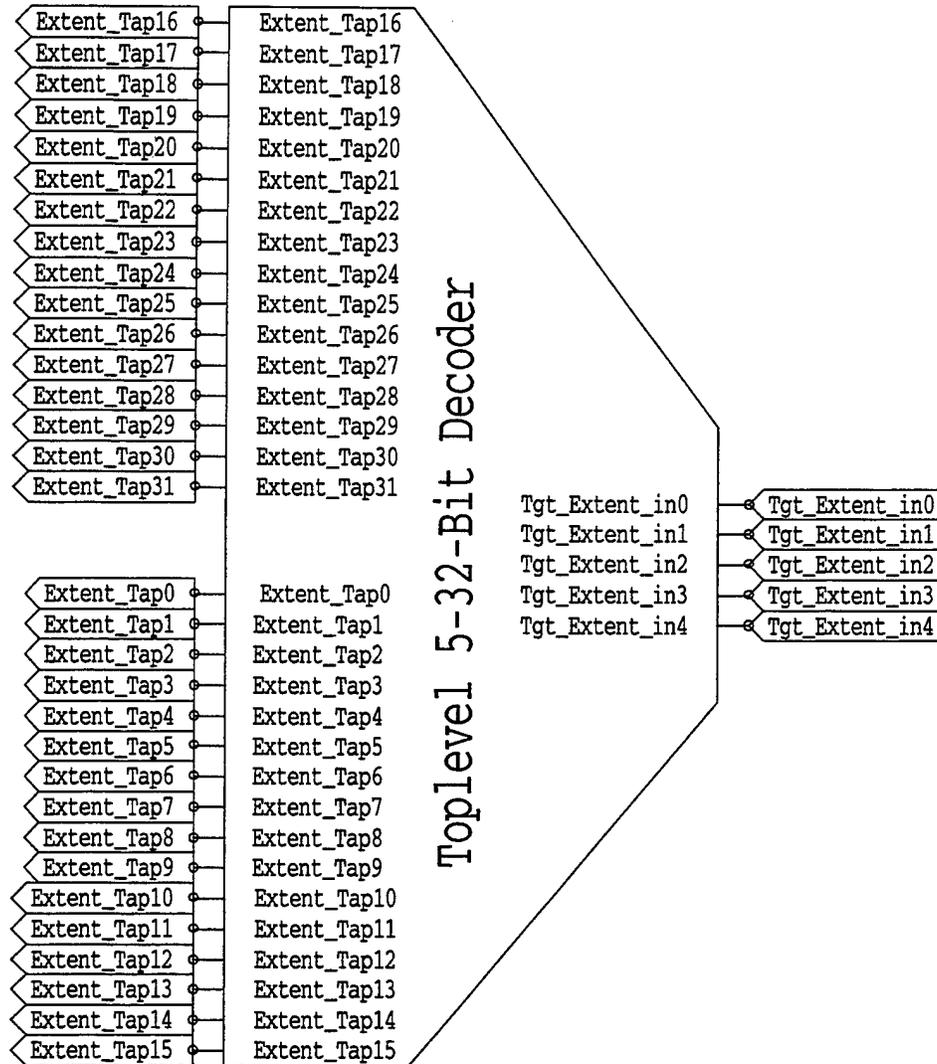
Figure 136. Toplevel 5-to-32 Decoder Symbol
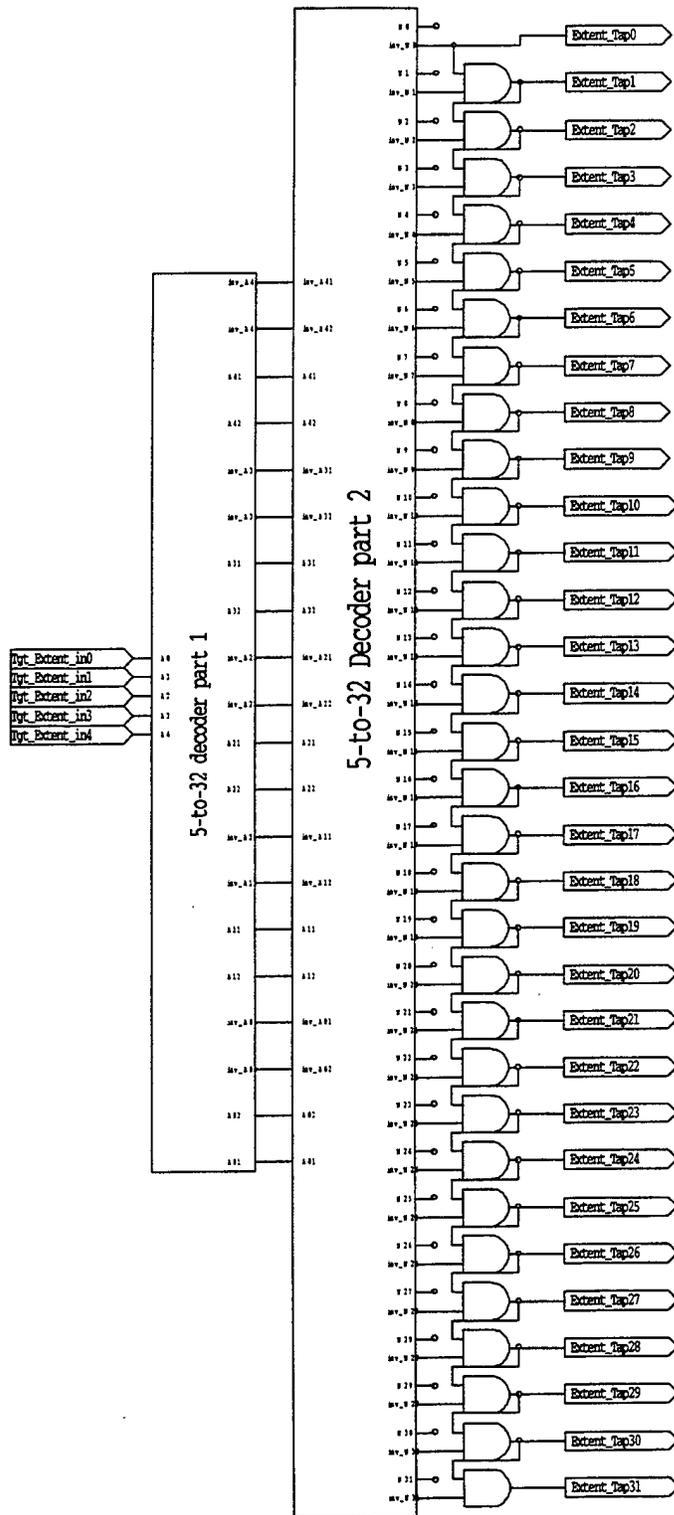
Figure 137. Toplevel 5-to-32 Decoder Circuit
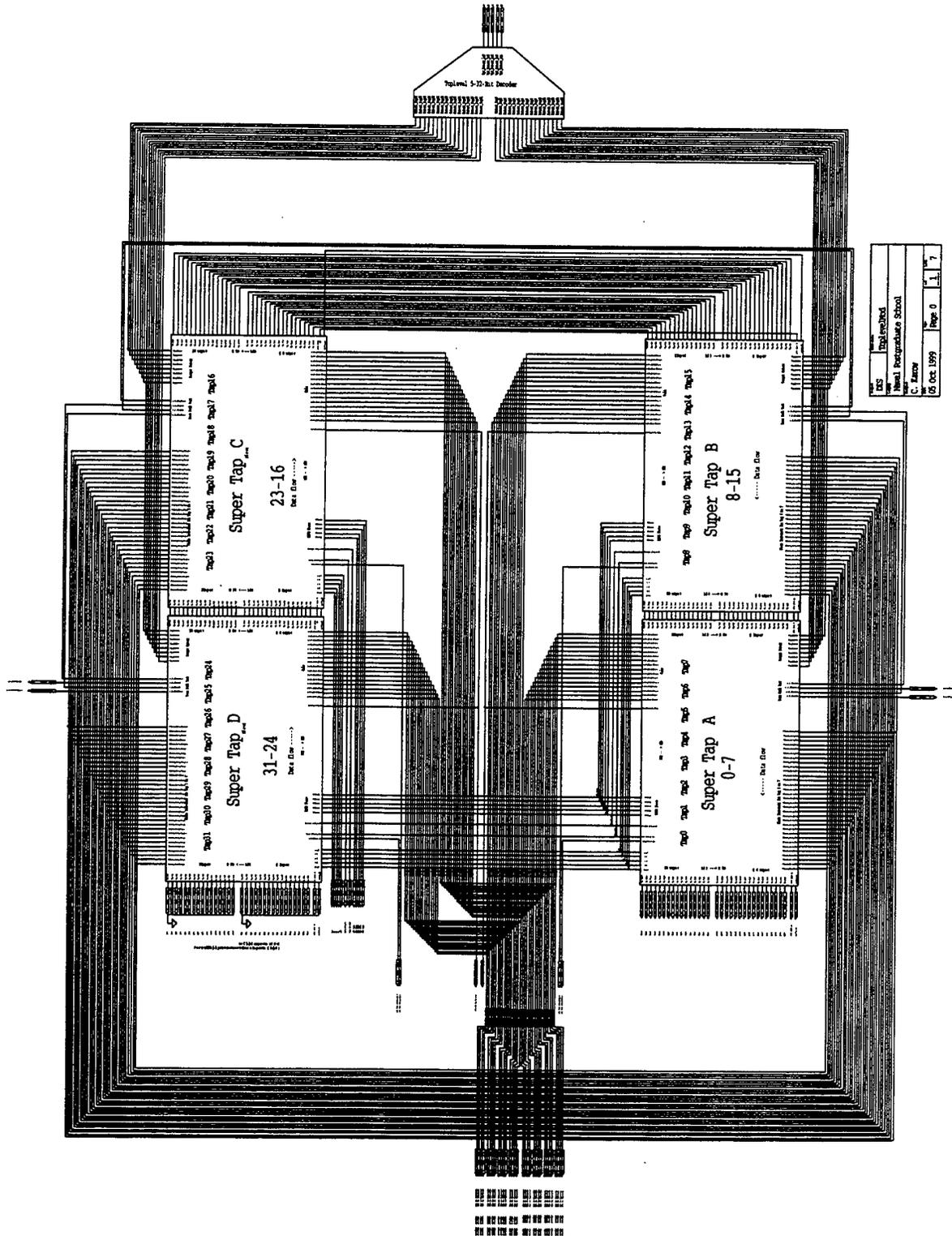
Figure 138. Toplevel Circuit

342

LIST OF REFERENCES

1. Donald R. Wehner, "High Resolution Radar," 2nd Edition.

2. R. M. Nuthalapathi, "High Resolution Reconstruction of ISAR Images," *IEEE Transactions on Aerospace and Electronic Systems,*" Vol. 28, No. 2, p. 462ff, April, 1992.

3. P. E. Pace, Surratt, R. E., Yeo, S.-Y., "Signal Synthesizer and Method Therefore," *Patent File Attorney* Docket No. 79,429, Sept. 1, 1999.

4. T. T. Vu, et al., "A GaAs Phase Digitizing and Summing System for Microwave Signal Storage," *IEEE Journal of Solid State Circuits*, Vol. 24, p. 104, February, 1989.

5. Mathwork Inc., Homepage for MATLAb, http://www.mathworks.com.

6. Yeo, Siew-Yam, "A Digital Image Synthesizer for ISAR Counter-Targeting," Master's Thesis, Naval Postgraduate School, Monterey, September 1998.

7. Naval Research Laboratory (NRL), http://radar-www.nrl.navy.mil/Areas/ISAR.

8. Raytheon Homepage, http://www.ueci.com/es/esproducts/ses137/ses137.htm.

9. MAX+PLUS II Getting Started version 8.1 (5.4 MB).

10. Altera Max+Plus II Online-manual.

11. Altera Homepage, http://www.altera.com/.

12. Visual Software Inc., Statecad 5.0 and Statebench printed manuals.

13. Visual Software Solutions Inc. Homepage, http://www.statecad.com.

14. SimGen Online manual.

15. Mentor Graphics Homepage, http://www.mentor.com.

16. AMI FPGA/ASIC Design Techniques Seminar, April 16, 1999.

17. Tanner Tools, printed manuals for LVS 8.02, Nettran, General Instructions for Tanner Tools.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Suite 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

3. Chairman, Code IW.. 1
   Information Warfare Academic Group
   Naval Postgraduate School
   Monterey, California 93943-5121

4. Chairman, Code EC.. 1
   Department of Electrical and Computing Engineering
   Naval Postgraduate School
   Monterey, California 93943-5121

5. Dr. Phillip Pace, Code EC/PC 2
   Department of Electrical and Computing Engineering
   Naval Postgraduate School
   Monterey, California 93943-5121

6. Dr. Douglas J. Fouts, Code EC/FS.. 1
   Department of Electrical and Computing Engineering
   Naval Postgraduate School
   Monterey, California 93943-5121

7. Commanding Officer Naval Research Laboratory 1
   Attn: Dr. John Montgomery
   Code 5700.00
   4555 Overlook Avenue, S.W.
   Washington, D.C. 20375-5339

8. Commanding Officer Naval Research Laboratory 1
   Attn: Mr. Alfred DiMattesa
   Code 5701.00
   4555 Overlook Avenue, S.W.
   Washington, D.C. 20375-5339