LOW POWER SPREAD SPECTRUM DEMODULATOR FOR WIDEBAND WIRELESS COMMUNICATIONS

by

Kevin M. Stone

Memorandum No. UCB/ERL M95/71

25 August 1995

cover page

LOW POWER SPREAD SPECTRUM DEMODULATOR FOR WIDEBAND WIRELESS COMMUNICATIONS

by

Kevin M. Stone

Memorandum No. UCB/ERL M95/71

25 August 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Contents

List of Figures

List of Tables

Acknowlegements

I would first and foremost like to express my thanks to Professor Brodersen for providing me with his guidance and support. Next, I would like to thank Samuel Sheng for his technical help, his guidance, and especially his friendship. Without him, my master's project would have never come into fruition. The two other people who also deserve tremendous thanks are Ian O'Donnell and Randy Allmon. They both contributed invaluable technical help. Randy worked on the initial specification and architecture for the chip and Ian designed the correlators as well as some other smaller cells.

To continue, I'd like to thank James Peroulas for providing me with a framework to test my chip and Craig Teuscher for explaining CDMA theory to me as well as letting me snarf many of his figures. I would like to thank Anantha Chandrakasan for providing with valuable design advice and for giving me a contact at Matsushita, Tom Burd for his help in Viewlogic and also for taking all my razzing about the Boys, and Brian Richards for helping lead me through the Lager maze. Some other people I would like to thank are Keith Onodera for his advice on CDMA demodulator design as well as for teaching me some basketball moves, Scarlett Wu for loaning me her Spice manuals and giving me food, and Tom Boot, Heather Brown, and Teresita Sarracino for all their administrative help. I would also like to thank Shankar Narayanaswamy, Dave Lidsky, Arthur Abnous, My Le, Renu Mehra, Andy Burstein, MacDaddy Adrian Isles, and Pramote Piriyapoksombut for making 550 such a fun place to work. Other people who made my life more enjoyable while I was at Cal were Sachin Deshpande, Andrew Liu, Dipanwita Deb, Erich Lichtblau, Peter Chen and the boys at ΔΤΔ. Finally, the three people who have had the most significant impact on my life, not only technically, but emotionally and spiritually at Cal are Lisa Guerra, Lapoe Lynn, and Sekhar Narayanaswami.

Lastly, I would like to dedicate this thesis to my parents. Without their undying love and kindness, generosity, charity, and support, I would have never gotten to where I am today.

Kevin Stone
27 March 1995

CHAPTER 1

Introduction

In recent years, there has been explosive growth in the computer, wireless communication, multimedia, and networking industries. As a result, computers are becoming more powerful and more networked-applications are harnessing this power by incorporating video, text/graphics, speech and/or handwriting recognition, and users are demanding that they be able to run these applications anytime and anyplace with no loss of processing power. However, in today's wireless computing environment, once the user breaks physical connection from the network, he is no longer able to run these types of high bandwidth applications.

The Infopad project [Sheng92] attempts to address this issue and more specifically, the radio project attempts to address one aspect of this equation, the high speed indoor wireless link. The goal of the Infopad CDMA radio project [Sheng95] is to support up to 50 users per picocell where each user can receive raw data at the rate of 2 Mb/s. This requires a chipping rate of 64 MHz and a sampling rate of 128MHz for timing recovery. The data is modulated using DQPSK modulation which encodes 2 bits/symbol and is transmitted at a symbol rate of 1MHz, thus achieving a 2Mb/s raw data rate.

The goal of this research is the development of the digital baseband circuitry used in the CDMA radio's receiver chip. The baseband circuitry must perform the following functions: initial coarse timing recovery to align the receiver and transmitter to within 1 chip, data recovery, detection of other nearby basestations, handoff, and fine (chip) timing recovery to keep the receiver and transmitter aligned to within 1/4 a chip. Since this chip will be used in a mobile, battery-operated receiver, minimizing power consumption is also of the utmost importance. Several techniques for reducing power were used, including architectural reconfiguration, use of multiple voltage levels, and logic design.

This paper is organized in the following manner: chapter 2 gives some background information and motivation for choosing CDMA, as well as a brief introduction to direct-sequence spread spectrum. Chapter 3 provides an overview of the chip as well as a more in-depth treatment of the backend's core functions: lock, digital phase lock loop, data recovery, and handoff. Chapters 4 and 5 provide external and internal documentation for the chip. Lastly, conclusions are drawn and future work is presented in Chapter 7.

This project attempts to show that it is possible to not only develop the required baseband DSP circuitry to support the above specifications, but also that they can be implemented as a low-power, low-cost custom-ASIC.

CHAPTER 2

Motivation/Background

This section is intended to help the reader understand where the DSP backend section fits into the overall radio system, and to motivate why certain hardware blocks exist and design decisions were made. This section first gives a motivation for choosing CDMA over other multiple access schemes, then gives some background information on spread sprectrum.

2.1 Why Choose CDMA?

When choosing a multiple access scheme to use, it is very important to consider what type of enviroment the system will be operating in. The environment in which the Infopad radio downlink will be used is an indoor office environment with a picocell radius of 5 meters. The primary interference in this type of environment is due to multipath interference. The difference in time between the arrival of a signal and it's last perceivable multipath arrival is called delay spread. From statistical measurements done in [Seid91], it was found that typical delay spreads range from 20ns to 60ns with Rician-distributed fading characteristics.

With this in mind, the viability of several different multiple-access techniques for the downlink was explored, including time-division multiple access, frequency-division multiple access, frequency hop spread spectrum, and direct sequence spread spectrum [Sheng91]--direct sequence spread spectrum was found to be the most suitable. Direct sequence is effective because of its ability provide mulitple access as well as its abiltiy to resolve multipath arrivals through ratio combining using a RAKE receiver [Proak89]. Initial simulations [Teus94] confirmed this effectiveness by showing that a RAKE can lower the BER by more than three orders of magnitude. This result can be seen in Figure 2.1. It should be noted that near-far effects do not have to be

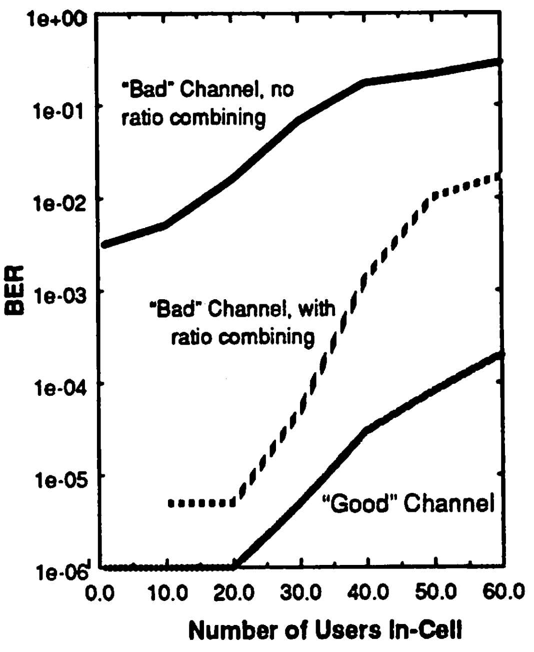


FIGURE 2.1 : BER vs. Number of users. This simulation assumed there was no noise in the channel [Wire94].

taken into consideration because the radio is in broadcast mode only.

A direct sequence spread spectrum system can not only support multiple user access, but it also inherently provides users with a high tolerance to interfering signals, and a high degree of resistance to narrowband nulls in the frequency response of the channel. Another advantage of

DS-CDMA, it the ability to provide variable quality of service (QOS) through power control. Algorithms have been developed [Yun94] which can accept QOS specifications for different substreams, assign them appropriate power levels, and can determine whether the system can add or must drop substreams to maintain the desired QOS.

2.2 Background on Spread Spectrum

Direct sequence spread spectrum is a technique which provide immunity to noise, resistance to jamming, and resistance to multipath fading. The term spreading refers to spreading of the signal over a frequency band.

The basic method which a direct sequence spread spectrum (DS-SS) system uses to "spread" the data is to multiply the symbol or data stream of rate $1/T_b$ by an independent, binary antipodal[1], pseudorandom (PN) sequence[2] of rate $1/T_c$ (called the chipping rate) where $T_c << T_b$. Any bit at the chipping rate is called a *chip* and the ratio of T_b to T_c is called the *processing gain* or *spreading factor*. The increase in performance obtained in a DS-SS system through the processing gain can be used to allow multiple users to occupy the same channel bandwidth provided that each user's signal has a unique pseudorandom signature.

One property, however, which PN sequences generally lack is good cross-correlation which is crucial in a multiaccess system to differentiate between the users' signals. One technique to guarantee that users' signals are differentiable, while mantaining the pseudorandomness of the PN sequence, is to overlay an additional user-specific code. One such code is called a Walsh code. The code sequence has rate $1/T_c$ and length equal to T_b/T_c. Walsh codes have the property of being perfectly orthogonal to one another. In other words, the cross-correlation/inner product of two

1. With binary antipodal signaling a "1" is represented as a "-1" and a "0" is repesented as a "1".
2. A PN sequence is also called a maximal length shift register sequence (see Appendix A for a description).

distinct Walsh codes should equal 0. An example set of 3-bit Walsh codes can be seen in Figure 2.2. To see the orthogonality, we take the inner product of Walsh code 1 and Walsh code 3. The

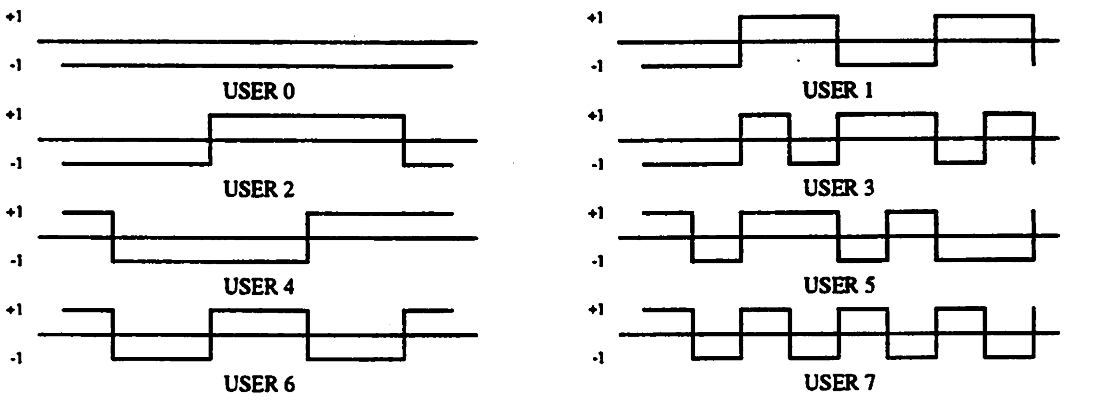


FIGURE 2.2 : An example 3-bit (8 chip) Walsh sequences.

result is 1 + 1 + 1 - 1 - 1 - 1 - 1 + 1 which sums to 0. The inner product of a Walsh code with itself is 2^n, where n is the number of bits in the walsh number. In this case, n equals 3, so the inner product is 8. Appendix B describes how Walsh codes are generated. This type of communication in which each user has their own distinct signature for transmitting over a common channel bandwidth is called code division multiple access (CDMA).

To recover the signal on the receiver side, the signal must be despread. This is done by multiplying the received signal by the same PN and Walsh sequence which were used to originally spread the signal in the transmitter, assuming here that the receiver and transmitter are synchronized. The result is then accumulated for the number of chips equal to the spreading factor. By looking at the accumulated result, the bit which was transmitted can be determined. In this ideal case, an accumulated result of $-(T_b/T_c)$ would mean a -1 was transmitted, while an accummulated result of (T_b/T_c) would mean a +1 was transmitted. Figure 2.3 illustrates this process for a spreading factor of $T_b/T_c = 8$. If there is no noise in the channel, then s(t) should equal s′(t)•PN•W1. This that means the accumation of s′(t)•PN•W1 over 8 chips will be either a -8 or +8 signifying a -1 or +1 respectively.

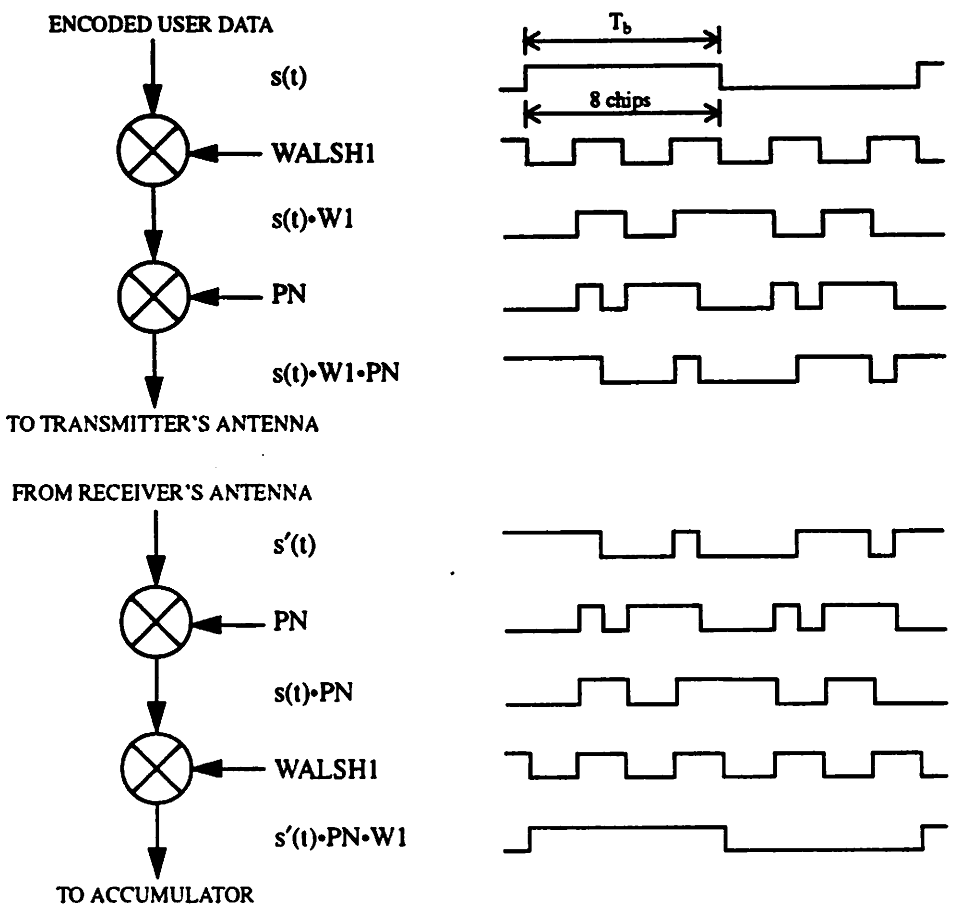


FIGURE 2.3 : Shows the process of spreading by a factor of 8 and despreading user data through multiplication by PN and Walsh sequences. This example assumes there is no noise in the channel.

In the real world, the received signal will not be ideal and will consist of the coherent data term and an incoherent error term. The error term is due to multipath interference, adjacent cell interference, thermal noise and ADC quantization noise. If the power of the data term is 1 and the noise power is σ^2, then the SNR is $1/\sigma^2$. When this signal is multiplied by the PN and Walsh and then correlated in the receiver, the coherent portion's power (the power of the user's data) will increase by the square of the spreading factor (n_{sf}^2) while the incoherent portion's (the error term) power will increase by the spreading factor (n_{sf}). Thus, the SNR after correlation will be $1 \cdot n_{sf}^2 / \sigma^2 \cdot n_{sf} = n_{sf}/\sigma^2$. In other words, the SNR after correlation has increased by a factor of n_{sf}. An example of the this noise reduction can be seen in Figures 2.4 and 2.5. For illustrative purposes, the error term is an analog sine wave and only a PN sequence is used. Figure 2.4 shows what would happen if the sine wave was integrated directly and Figure 2.5 demonstrates how the effect is diminished through the multiplication of a PN sequence.

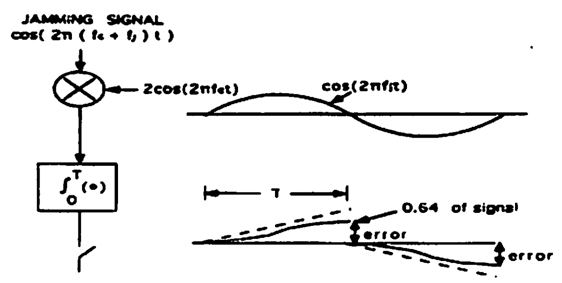


FIGURE 2.4 : Shows effect of error term.

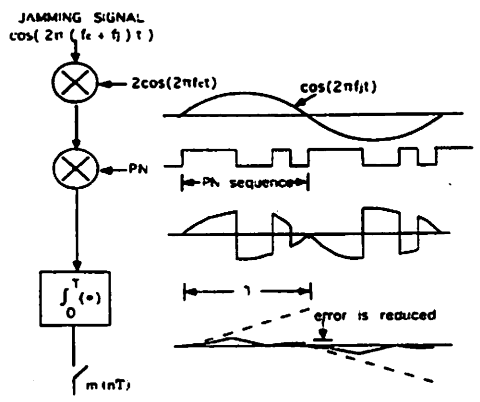


FIGURE 2.5 : Shows how error is reduced through multiplication by PN sequence.

2.3 Appendix A-Maximal Length Shift Register Sequences

A maximal length shift register (MLSR) sequence are pseudorandom (PN) because they easily synthesize repeating sequences that appear to be statistically white over short subsequences. A MLSR sequence is generated by a feedback shift register which is governed by the relation.

$$x_k = h_1 \cdot x_{k-1} \oplus \ldots \oplus h_n \cdot x_{k-n} \qquad \text{(EQ 2.1)}$$

where the output x_k and the coefficients are binary assuming values of "0" and "1". The zero coefficients correspond to no feedback tap, whereas the one coefficients correspond to the direct connection of the shift register output to the modulo-two summation. A representation of this above relation can be seen in Figure 2.6. If we add x_k to both sides of eqn. 2.1, remembering that

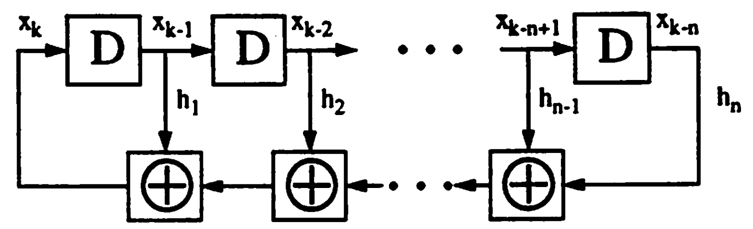


FIGURE 2.6 : A linear feedback shift register with binary input. The coefficients are binary, and the summation is modulo-two.

$x_k \oplus x_k = 0$, we get

$$x_k \oplus h_1 \cdot x_{k-1} \oplus \ldots \oplus h_n \cdot x_{k-n} = 0 \qquad \text{(EQ 2.2)}$$

In other words,

$$x_k * h_k \equiv 0 \ , \qquad \text{(EQ 2.3)}$$

if we define $h_0 = 1$ and $h_m = 0$ for $m < 0$ and $m > n$ and we interpret the summation as a convolution in the modulo-two sense. Thus, the D-transform of eqn. 2.3 is

$$h(D)X(D) = 0 \qquad \text{(EQ 2.4)}$$

where

$$h(D) = 1 \oplus h_1 D \oplus \ldots \oplus h_n D^n \qquad \text{(EQ 2.5)}$$

is the transfer function of the shift register. The D-transform is just like the Z-transform except that the additions are modulo-two and the symbol D is used instead of z^{-1}. The transfer function h(D) for the generator is a polynomial of degree n (we assume that $h_n = 1$) with binary coefficients is given the name *generator polynomial.*

Not all generator polynomials produce a maximal length sequence in which the output period is $r = 2^n - 1$. A generator polynomial of degree n will only produce a maximal length sequence if it does not divide any polynomial $(1 \oplus D^m)$ for $m < 2^n - 1$. A table of maximal length sequence generating polynomials can be seen in Table 2-1.

Table 2.1 : Generating polynomials of various orders. Each entry in the table when converted to binary specifies the coefficients of the polynomial h(D).

Order	Polynomial	Order	Polynomial
2	7	11	805
3	B	12	1053
4	13	13	201B
5	25	14	4443
6	43	15	8003
7	89	16	1100B
8	11D	17	20009
9	211	18	40081
10	409	19	80020

2.3.1 Properties of an PN Sequence

Ideally, a PN sequence of length "r" should have the following properties [1]:

- Relative frequencies of "-1" and "1" are each 1/2.
- Run lengths (of -1's or 1's) are as expected in a coin-flipping experiment. In other words, Prob(n -1's) = Prob(n 1's) = $(1/2)^n$.
- Will see all possible combinations of length $\log_2$"r" (except the all-zero) in sequence.
- If the random sequence is shifted by any non-zero number of elements, the resulting sequence will have an equal number of agreements and disagreements with the original sequence.
- Almost perfect autocorrelation properties:

$$\rho(\tau) = \begin{cases} r, & \tau = 0 \\ -\frac{1}{r}, & \tau \neq 0 \end{cases} \qquad \text{(EQ 2.6)}$$

The first four properties are necessary to make the resulting signal (the multiplication of the data stream by the PN sequence) look as Gaussian or white as possible. The whiter the transmitted signal, the more constant its power spectral density (PSD) will be over the band of interest. By having a constant PSD, no one narrowband null in the channel's frequency response will have more of a deleterious effect on the received signal than any other. The drawback to this technique is that the signal's SNR takes a hit[3]. The final property, almost perfect autocorrelation, is useful for synchronization between the transmitter and receiver. Unfortunately, in our system because we overlay the Walsh sequence on top of the PN sequence. This compromises the ideality of the PN sequence and correspondingly it's properties.

2.4 Appendix B-Walsh Code Generation

Walsh Codes are a set of orthogonal functions developed by Joe Walsh in 1973. A good discussion of the theory behind Walsh functions can be found in [Beau84]. The Walsh code number used in the Infopad is 6 bits, which produces a 64 bit length Walsh sequence. This appendix will only discuss how Walsh sequences are generated and will focus on three bit walsh numbers for simplicity.

The technique which the Infopad demodulator chip employs to generate Walsh sequences is to use the difference between successive Gray codes. Table 2-2 shows a three bit Gray code sequence and the difference between successive elements. A '1' in the difference entry signifies that bit in the

Table 2.2 : Describes generation of Walsh sequence for user 5 out of 8.

Grey Code Elements	Difference b/w successive gray code elements	Input of Toggle Flip-Flip (SOP with walnum = "101")	Output of Toggle Flip-Flop
000	100	1	1
001	001	1	0
011	010	0	1
010	001	1	1
110	100	1	0
111	001	1	1
101	010	0	0

3. This is in contrast to a frequency-hop spread spectrum system where the BER will be extremely high if the current frequency which has been hopped into lies in a null. However, a frequency hop signal's BER will be lower than that of a direct-sequence signal when the signal gets through.

Table 2.2 : Describes generation of Walsh sequence for user 5 out of 8.

Grey Code Elements	Difference b/w successive gray code elements	Input of Toggle Flip-Flip (SOP with walnum = "101")	Output of Toggle Flip-Flop
100	001	1	0

current Gray code entry is different from the previous entry. For example, the difference between Gray codes 010 and 110 is 100 which signifies that only the msb has changed. Thus, the difference entry is 100. Because Gray codes are designed so adjacent entries only differ by one bit, the difference code should have only one '1' for each entry.

The next step is to generate a sum-of-products with the Walsh number and use the result as the input to a toggle flip-flop. In this case, since we are using a three-bit walsh number, the equation would look like the following:

$$T_{FF,IN} = \text{walnum}_2 \cdot \text{diff}_2 + \text{walnum}_1 \cdot \text{diff}_1 + \text{walnum}_0 \cdot \text{diff}_0 \qquad \text{(EQ 2.7)}$$

The output of the toggle flip-flop is the Walsh sequence. Examples of 3-bit Walsh sequences can be seen in Figure 2-2 in Section 2.2 This algorithm can be implemented in various ways in hardware. One way can be seen in Figure 2.7.

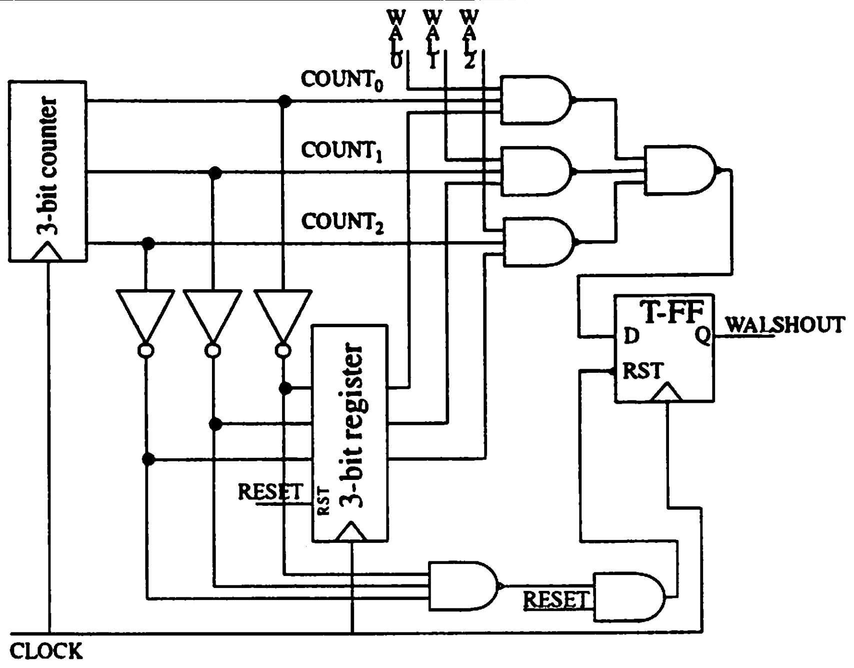


FIGURE 2.7 : Hardware implemention of a Walsh code generator. The user's Walsh number (WAL[2:0]) determines which Walsh sequence will be output [Walsh73].

CHAPTER 3

Chip Overview

As mentioned in the previous two chapters, the goal of the Infopad CDMA radio project is to provide for up to 50 users per basestation where each user can receive raw data at the rate of 2Mb/s. The radio uses orthogonal codes called Walsh codes to differentiate the users and a pseudorandom noise (PN) sequence to spread/despread the data as well as provide a pilot tone for synchonization and channel estimation. The data itself is encoded using DQPSK which provides protection against oscillator mismatch and slow channel variations. To meet these specifications, the radio requires a chipping rate of 64 Mchip/second and a transmit bandwidth of around 80-100 MHz.

The digital spread spectrum demodulator, the topic of this thesis, is only a part of the Infopad CDMA radio receiver. A block diagram of the radio can be seen in Figure 3.1. The receiver in its final form will be a hybrid or mixed-signal CMOS chip consisting of both analog and digital circuitry. The analog blocks which precede the demodulator will consist of an off-chip image filter, an LNA, an off-chip noise filter, a unity gain buffer, analog sampling demoduators, automatic gain control (AGC) circuitry, and a analog-to-digital converter (ADC). These five blocks plus the off-

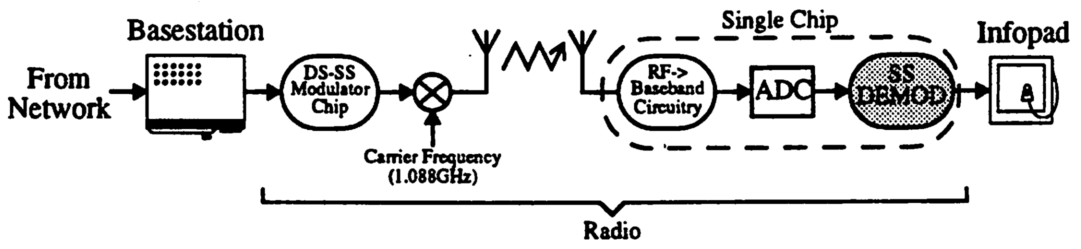


FIGURE 3.1 : Block diagram of the Infopad direct sequence spread spectrum radio.

chip filters convert the 1.088GHz RF signal down to two baseband four-bit sign-magnitude interleaved 128MHz streams. It was determined through simulation work [Teus94] that the quantization noise introduced by the ADC will begin to be the dominate contributor when the quantizer order is reduced below four bits. It was found that reducing the order below four bits significantly impacted the BER, while increasing the number of bits above four did little to improve performance. The reason why the data representation is sign-magnitude will become apparent when power issues are discussed. These two streams are then fed to the radio's digital backend/demodulator section. A block diagram of the digital backend can be seen in Figure 3.2.

Once the data gets to the demoduator, a variety of operations are performed. First, the two streams of interleaved 128MHz data from the ADC are converted into 2 parallel 64MHz in-phase and quadrature phase (I/Q) streams (total of 4 streams of 4 bits each) by the input data mux. Then, depending on the current mode of the demodulator, various operations may happen. Although the data recovery unit only requires one 64MHz I/Q stream, the data has been oversampled by a factor of two to achieve fine timing recovery which is why the ADC output is two 128MHz streams. The data fed to the lock, data recovery block and the adjacent cell scan block is considered the on-time data (I_{on}, Q_{on}), while the data sent to the DPLL is considered the off-time data (I_{off}, Q_{off}).

The first operation which must happen is the mobile must synchronize itself with the basestation. Because there is no carrier phase-locked recovery loop in the analog section, all of the synchronization, both initial lock and fine timing adjustment, must be done in the digital backend. The initial synchronization guarantees that the mobile is locked to within ±7.8ns ($T_{chip}/2$) to the

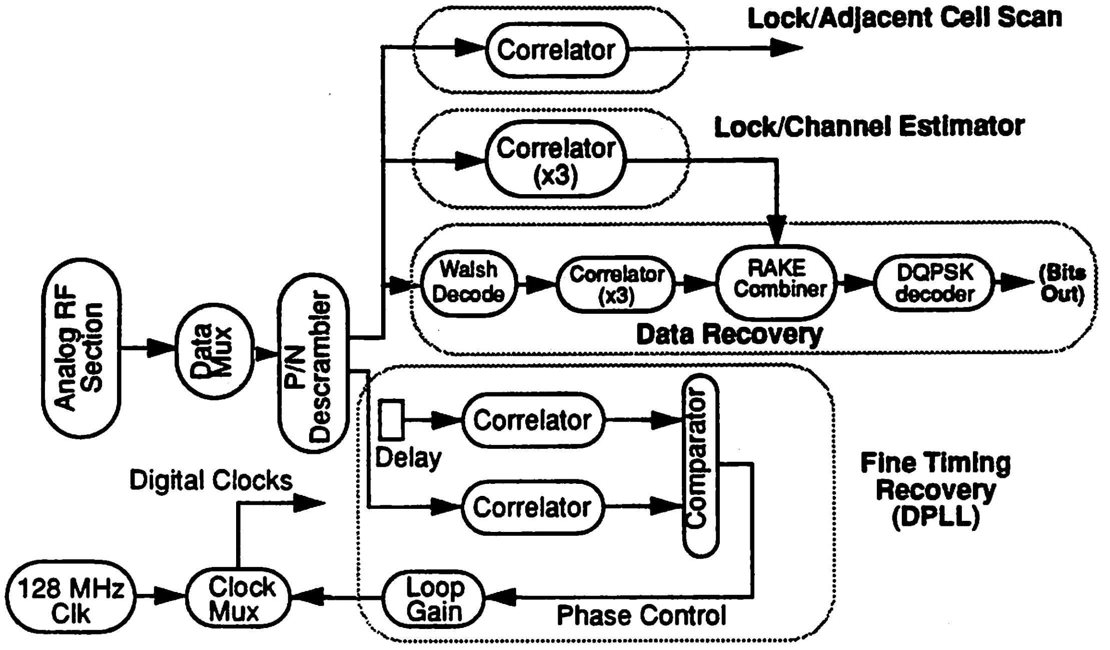


FIGURE 3.2 : Digital baseband receiver architecture Four of the correlators are shared between the lock block and the channel estimator/adjacent cell scan blocks [Wire94].

transmitter. After coarse lock, the fine timing adjustment, done by a digitial phase lock loop (DPLL), periodically attempts (every 1 us) to reduce this offset to within ±3.9ns ($T_{chip}/4$). The initial synchronization block is described in Section 3.1 and the DPLL is described in Section 3.2. One can see from Figure 3.2 that the lock block shares hardware with the adjacent cell scan block and the channel estimation block. Thus, before lock is acquired the four correlators are used for initial synchronization, and afterwards are used for adjacent cell scan and channel estimation. The DPLL has two dedicated correlators.

After lock has been achieved, the digital backend must then despread and recover the user data. This is done by first accumulating the data which has been multiplied by the appropriate bits from the PN and user's Walsh sequences for 64 T_{chips}. Then, the accumlated I and Q data is fed into a DQPSK decoder and user data is output at a rate of 2Mb/s. The data recovery block is described in Section 3.3.

Because the Infopad is operating in a mobile environment, there is no guarantee that it will only reside in one cell. A more likely scenario is it would remain in any given cell most of the time, but move into a new cell some of the time. As a result, the Infopad must constantly scan for adjacent cells and perform cell handoff when necessary. This is described in Section 3.4.

A more detailed description of the chip's individual blocks can be found in Chapter 5.

Low-Power Aspects

Beyond achieving functionality at the required thruput, minimizing power consumption was critical, give the portable nature of the application. The main source of power dissipation in CMOS circuits is dynamic power comsumed by the switching gates which can be expressed as

$$P_{dyn} = C_{total} \cdot V_{dd}^2 \cdot f_{clk} \quad \text{(EQ 3.1)}$$

where C_{total} is the amount of capacitance switched, V_{dd} is the supply voltage, and f_{clk} is the clock frequency [Chand92]. In our system, the clock frequency was fixed, so only the first three parameters could be varied to minimize power. One can see from eqn. 3.1, that reducing the supply voltage by a factor of two would result in a fourfold reduction in power consumption, whereas reducing the total switched capacitance by a factor of two would result only in a twofold reduction. In the design of the demodulator, both supply voltage reduction and minimization of switched capacitance techniques were used in order to reduce the demodulator's overall power dissipation.

Techniques to lower the supply voltage were explored on both a system and block level basis. On the system level, three different supply voltage levels, 5V, 3.3V, and 1.5V were used in an attempt to minimize power consumption. The 5V supply was used to power the clock generating circuitry which has a critical path of 4 ns, the 3.3V supply was used to power the control circuitry and the 1.5V supply was used to power the correlators. Both the control circuitry and correlators run at 64 MHz. The idea was to give each block of the chip only as much power as needed. These three voltage levels can be generated using off-chip low-power dc-dc converters [Strat94] and level converters are used internally to interface blocks with different power supplies. A block diagram of a level-converter can be seen in Figure 3.3.

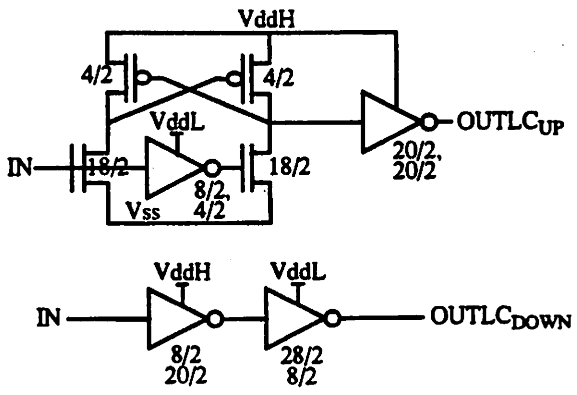


FIGURE 3.3 : Block diagram of a up/down level-converter.

The main block-level modification was done on the correlators. Originally, the correlators were designed to run off a 3.3V supply. However, it was identified early on that they contributed heavily to the demodulator's power consumption. Thus, they were redesigned to use a carry-save adder archictecture instead of the original ripple-carry adder architecture. This approach effectively bit-piplines the adder reducing the critical path down to a 1-bit adder and a register. By making this architectural change, the correlators were able to run off the aforementioned 1.5V supply.

The other strategy to reduce power consumption was to reduce the amount of capacitance being switched. Since the sign of the data is constantly being toggled due to multiplication with the PN and Walsh sequences, it was found in [Chand92] that using a sign-magnitude number representation will consume approximately 30% less power than a 2's complement number representation for this application.

3.1 Coarse Lock Acquisition

When the mobile unit is turned on, it must synchronize itself with the basestation before it can start receiving user data. This is done by locking onto a pilot tone which is being continuously transmitted by the basestation. The pilot tone, a PN sequence, is a periodic stream of bits or chips. It has pseudorandom noise-like properties and good auto-correlation properties. Figure 3.4(a)

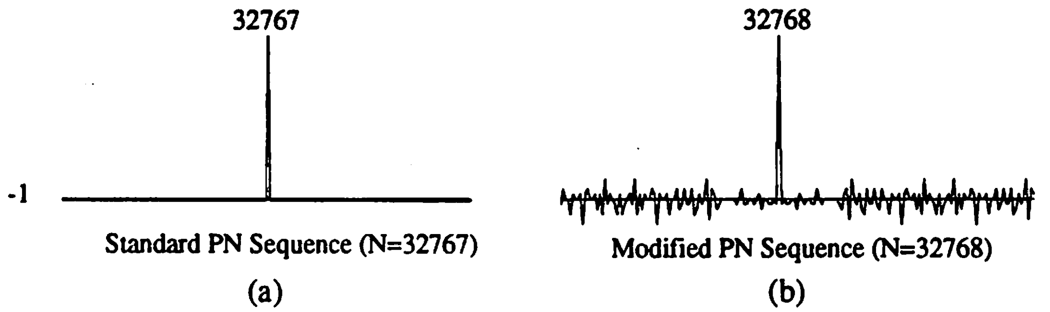


FIGURE 3.4 : PN Autocorrelation Functions (1-bit data) [Teus94].

shows an autocorrelation function for a PN sequence of length 32767 (N=32767). In terms of the transmitted PN sequence, PN_t and the PN sequence generated by the receiver, PN_r the autocorrelation function is shown in equation 3.1.

$$C_m = \sum_{n=0}^{N} PN_{t,n} \cdot PN_{r,n} \qquad \text{(EQ 3.2)}$$

If the mobile's PN generator is in sync with the basestation's PN generator, then ideally, the correlation C_m over one period would equal 32767. If the transmitter and receiver were not in sync, then ideally the correlation over one period would sum to -1. The reason why it is not ideal is because of multipath inteference (see Section 2.2) and quantization noise. The Infopad's spread spectrum radio uses a PN sequence of length $N=32768=2^{15}$ with a chipping rate of 64 MHz. One bit has been added to ease in the hardware implementjon and timing. The N=32768 sequence's autocorrelation properties are shown in Figure 3.4(b). Because of the excellent autocorrelation properties of the PN sequence we can compare C_m to a threshold register. With a certain degree of confidence, it can be assumed that if C_m is greater than the value stored in the threshold register then lock has been achieved.

Unfortunately, however, when the mobile is first turned on, the phase relationship between the receiver's and transmitter's PN sequence is unknown. In the worst case, the demodulator would have to correlate over all the phases of the PN sequence taking approximately 16.77 seconds ($2^{15} \bullet 2^{15} \bullet 1/64E6$) to acquire lock. This is too long! Fortunately, simulations run in Ptolemy determined that it was only necessary to correlate over 1024 chips before making the determination of whether lock has been achieved instead of correlating over the entire length of the PN sequence of 32768 chips. This reduces the acquisition time by a factor of 32. The acquisition time was further reduced by a factor of four by using three extra correlators running in parallel, in which each one is offset in time from its neighbors by one T_{chip}. Thus, with these changes, the worst case time-to-lock was reduced to approximately 131 ms. A diagram of the coarse acquistion block can be seen in Figure 3.5. A more detailed block diagram of the long

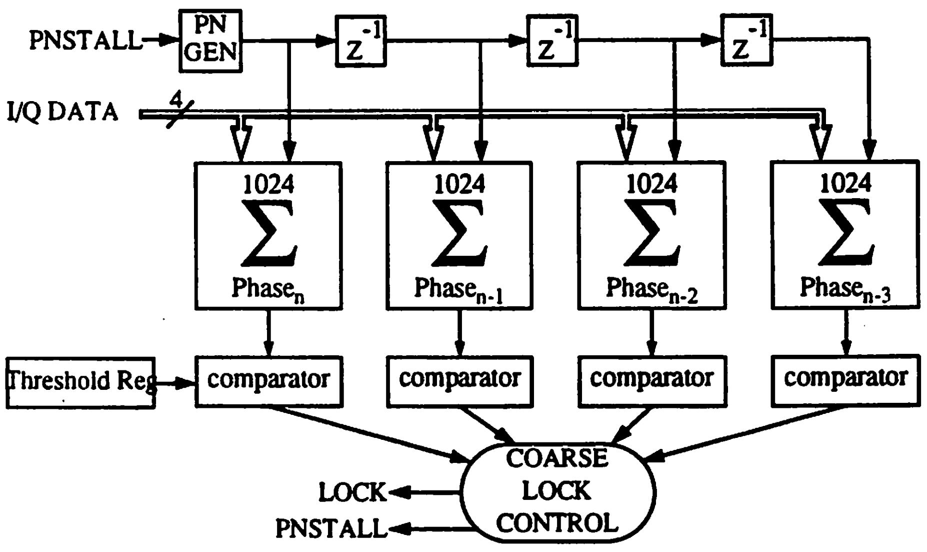

PNSTALL
PN GEN
z^{-1}
z^{-1}
z^{-1}
I/Q DATA
4
1024 Σ Phase_n
1024 Σ Phase_{n-1}
1024 Σ Phase_{n-2}
1024 Σ Phase_{n-3}
Threshold Reg
comparator
comparator
comparator
comparator
COARSE LOCK CONTROL
LOCK
PNSTALL

FIGURE 3.5 : Block diagram from coarse lock acquistion. The hollow arrow indicates 4-bit I/Q pair.

correlators can be seen in Figure 3.6. The signal CMPTH_L is fed from each of the long

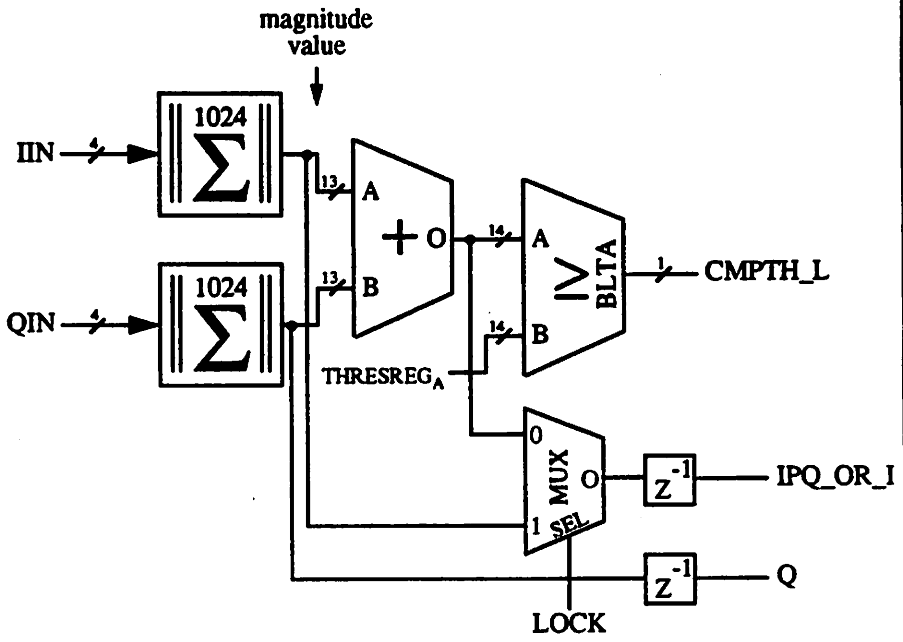


FIGURE 3.6 : Block diagram of a multipath correlator.

correlators to the lock circuitry. The other signals, IPQ_OR_I and Q are used for channel estimation (see Section 3.X). When testing coarse lock acquistion (LOCK = '0'), IPQ_OR_I can be used to evaluate whether the chip is receiving energy or not.

The coarse lock acquisition search algorithm is the following:

① Reset the PN generator to known state (i.e. Phase$_N$). Therefore, correlator$_0$ will correlate over phase$_N$, correlator$_1$ over phase$_{N-1}$, correlator$_2$ over phase$_{N-2}$, and correlator$_3$ over phase$_{N-3}$.

② Run correlators for 1024 cycles.

③ Dump output of the correlators to latches.

④ Compare the value stored in each latch to a threshold register which indicates how much energy is required to acquire lock. In our case, this step costs us 64 cycles (1 symbol period).

⑤ If one of the correlated energy values is greater than the threshold register, than goto ⑦, else goto ⑥.

⑥ Stall the PN generator 4 four clock cycles and clear our the correlators. This will shift correlator$_0$ from phase$_N$ to phase$_{N-4}$, correlator$_1$ from phase$_{N-1}$ to phase$_{N-5}$, correlator$_2$ from phase$_{N-2}$ to phase$_{N-6}$, and correlator$_3$ from phase$_{N-3}$ to phase$_{N-7}$. Goto . Note: Phase$_{-1}$=Phase$_{32767}$, Phase$_{-2}$=Phase$_{32766}$, etc.

⑦ STALL THE PN GENERATOR THE NUMBER OF CYCLES EQUAL TO THE CORRELATOR WHICH FOUND LOCK. C0->DO NOT STALL, C1->STALL 1 CYCLE, C2->STALL 2 CYCLES, C3->STALL 3 CYCLES. TABLE 3.1 ILLUSTRATES THIS POINT.

recv'd data	Phase of C0 PN	Phase of C1 PN	Phase of C2 PN	Phase of C3 PN	Stall
D1	4	3	2	1	
D2	5	4	3	2	
D3	6	5	4	3	
D4	6	5	4	3	X
D5	6	5	4	3	X
D6	6	5	4	3	X
D7	7	6	5	4	
D8	8	7	6	5	

Table 3.1 : Illustrates how stall is used to make C0 become the on-time correlator. The numbers represent the current bit # in the PN sequence (32768 total). In this example, C3 has found lock. Stalling the receiver's PN generator for 3 clock cycles causes C0 to become the on-time correlator. C1, C2, and C3 become multipath estimators after lock.

⑧ GOT LOCK -> START ACQUIRING DATA.

It should be noted that in our system, synchronization does not have to be guaranteed between mobile units, only between the mobile unit and the basestation.

3.2 Digital Phase Lock Loop

The role of the digital phase lock loop (DPLL) is to maintain synchronization between the receiver and transmitter after initial lock has been acquired. Misalignment between the receiver and transmitter can occur for two reasons: (1) coarse lock synchronization only guarantees that the receiver and transmitter are aligned to within $T_{chip}/2$, (2) due to the mobile nature of the Infopad, the channel can vary, and (3) due to oscillator offsets. To reduce the chance of bit errors, it is desirable to maintain as small an offset as possible.

A high-level block diagram of the DPLL can be seen in Figure 3.7. One of the DPLL correlators receives an I_{off}/Q_{off} pair, while the other receives an I_{off}/Q_{off} pair which has been delayed by T_{chip}. Both correlators extract the pilot tone information from the off-time data by multiplying the I_{off}/Q_{off} pair by a bit from the PN sequence, since the pilot tone is the PN sequence sequence by itself. The energy from the pilot tone is accumulated for 1024 cycles (16 μs) and then

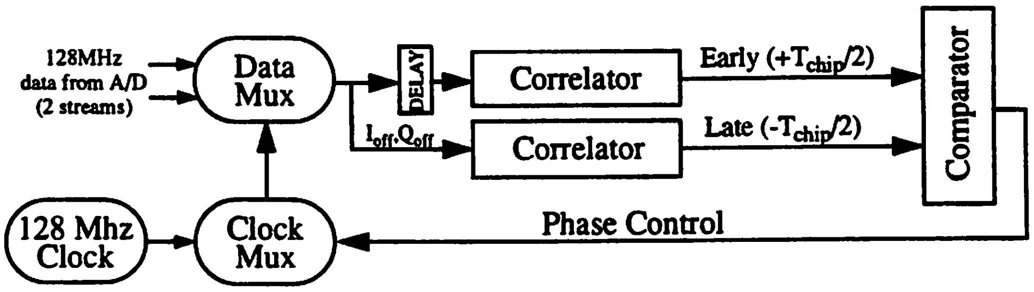


FIGURE 3.7 : Block diagram of the digital phase lock loop. Correlates and compares early and late pilot tone energy to see if clock's phase should be adjusted by $\pm T_{chip}/4$.

dumped into latches. We have assumed that a DPLL with an update period of 16 us will be sufficient to capture changes in the channel. A comparison is then made using the early and late pilot tone energies (called E_{early} and E_{late} respectively) in order to determine if and how the clock's phase should be adjusted. A block diagram of this comparison circuitry can be seen in Figure 3.8. The comparison compares the absolute value of the difference between E_{early} and E_{late} ($|E_{early}-E_{late}|$) to a threshold register. The signal CMPTH3 shows the result. If the value is greater than the threshold register, than E_{early} and E_{late} are far enough apart to warrant the clock's phase being adjusted. If the value is less, then the clock's phase should not be adjusted. This prevents needless ping-ponging when E_{early} and E_{late} have similar values. By looking at the sign of E_{early}-E_{late}, it can be determined how the clock's phase should be adjusted. If the sign is positive, then the system is sampling too late and the phase of the clock should be reduced. If the sign is negative, then the system is sampling too early and the phase of the clock should be extended. Figure 3.9 illustrates this point.

Ideally, one would like the DPLL's granularity for adjusting the phase of the sampling clock to be of infinite precision. However, this would be prohibitively expensive both in power and hardware complexity. Instead, a trade-off between granularity and power/hardware complexity needed to be made. Initially, the DPLL was designed to adjust the phase of the sampling clock by $\pm$ $T_{chip}/8$ which required an input clock of 256MHz. This design was soon found to be

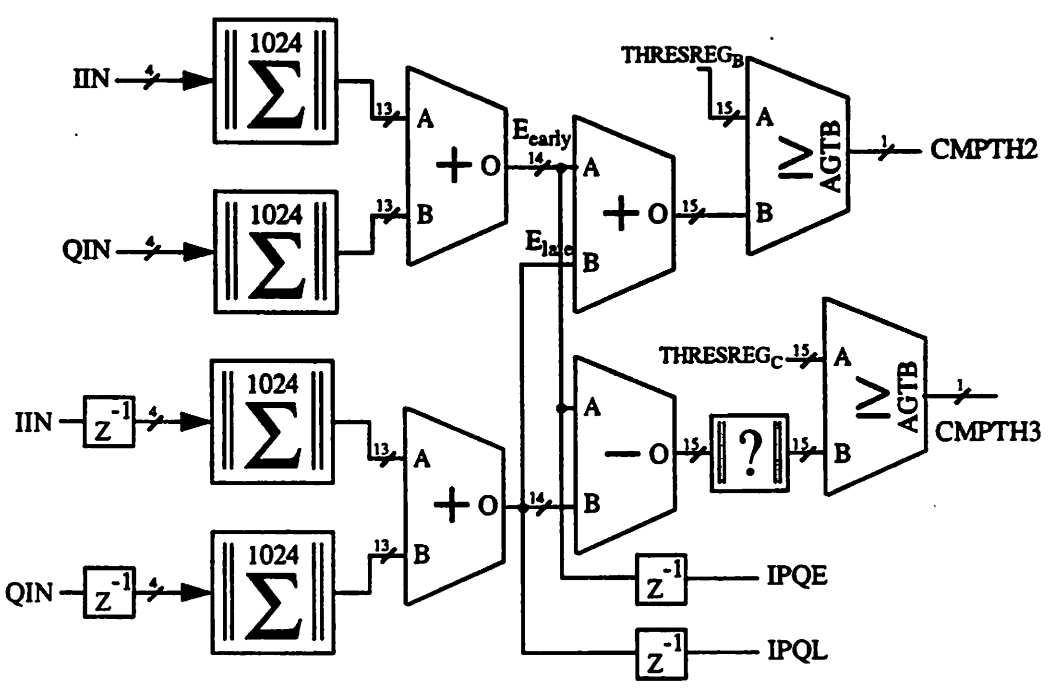


FIGURE 3.8 : Block diagram of DPLL correlators along with backend decision circuitry. CMPTH3 indicates whether the system clock's phase should be adjusted. CMPTH2 indicates whether the received tone energy is large enough to stay in lock. Note: CMPTH2 is used in the lock control circuitry, but not the DPLL.

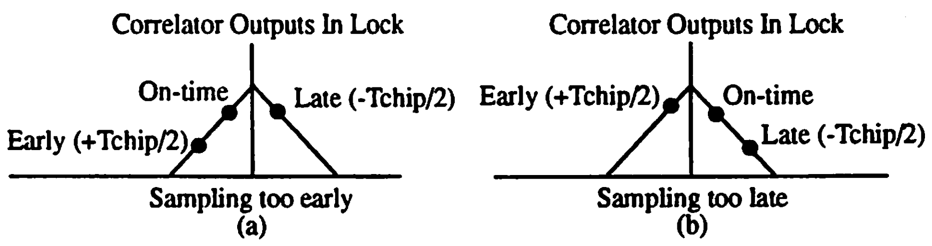


FIGURE 3.9 : Shows idealized pilot tone energy relationships between the early, late, and on-time correlators. (a) since E_{early} is less than E_{late}, we are sampling too early, therefore extend phase of the clock. (b) since E_{early} is greater than E_{late}, we are sampling too late, therefore reduce phase of the clock.

unacceptable because the analog section only required the positive and negative phases of a 128MHz clock. Since only one sinusoidal oscillator was being used, this meant the analog section needed to perform a divide by two in order to obtain its two 128MHz clocks. Because of the analog section had such tight phase offset requirements for the positive and negative phases of its clock (i.e. their edges had to overlap almost perfectly), the skew introduced by the clock dividers could not be tolerated. Thus, a more conservative DPLL granularity of $\pm T_{chip}/4$ was choosen which could be accomplished using a 128MHz input clock. Unfortuanately, due to limations with Ptolemy simulator [Buck93] using the SDF domain, the effect of different DPLL granularities was not simulated. There was one benefit, however, which resulted from this change--it was must easier to build a DQPSK decoder which could handle a ±90° shift in it's coordinate axis, than a ±45° degree shift.

3.2.1 Clock Generation

Although it will be discussed more in detail in Chapter 5, a brief overview of the clock generator will be presented here. As can be seen in Figure 3.10, there are four phases of a 64MHz

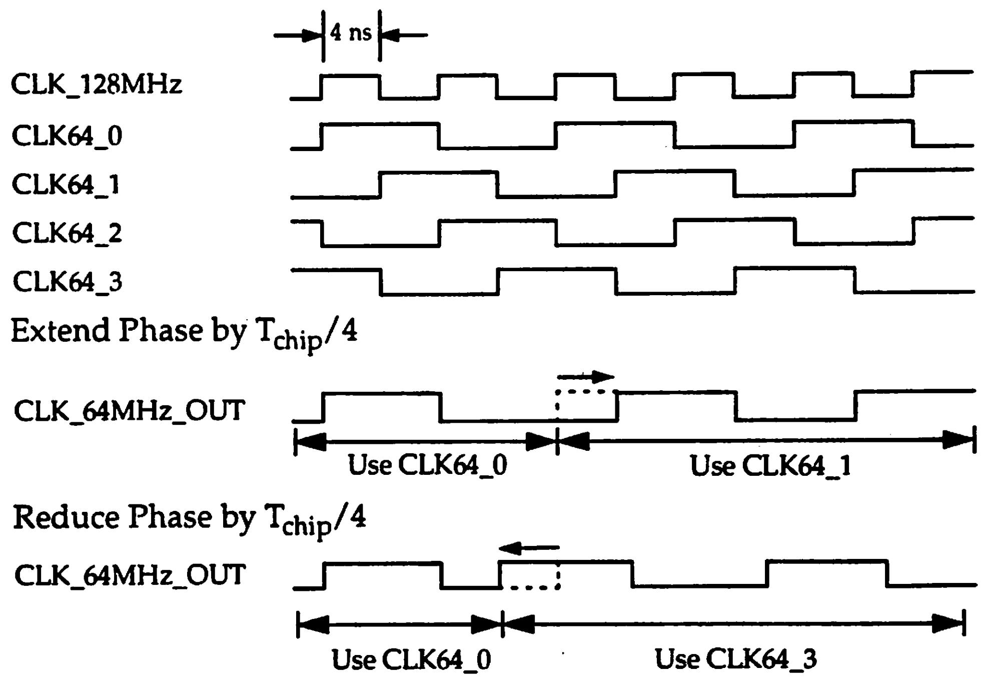


FIGURE 3.10 : Shows how the clock's phase is adjusted by $\pm T_{chip}/4$. For example purposes, it is assumed that CLK64_0 is used initially for both cases.

clock each offset from each other by $T_{chip}/4$ which are used to produce the final output, a 64MHz clock. By switching between these four clocks at the appropriate instances, the outputted clock's phase can be adjusted by $\pm T_{chip}/4$. To extend the phase of the clock, switch to $[(\text{NUM\_CLOCK}_{current}+1) \bmod 4]$; to reduce the phase of the clock, switch to $[(\text{NUM\_CLOCK}_{current}+3) \bmod 4]$. Care must be taken not to produce any glitches in the output clock.

3.3 Data Recovery

Once synchronization has been achieved, the mobile unit can begin recovering user data. Although this chip is only concerned with data recovery (i.e. despreading and decoding), is it worthwhile here to first discuss how the data is encoded and why this encoding scheme was choosen. A discussion of how the data is spread can be found in Section 2.2.

3.3.2 DQPSK Encoding

The oscillators choosen in our system have a typical frequency accuracy of 20 parts/million. This variation means there will be a slight frequency offset between the carrier frequency and the sampling frequency for both the in-phase and quadrature signals. Assuming an oscillator of 1.088GHz, the offset would be 21.76kHz. This offset can be viewed as a slowm relative to the symbol rate, rotation of the DQPSK constellation in symbol space. In other words, the constellation will rotate 7.8° in 1 µs (64 T_{chips}-symbol period), 125.2° in 16 µs (1024 T_{chips}), or 360° in 46µs (1/21.76kHz). Because of this frequency offset, differential quadrature phase shift keying (DQPSK) was choosen as the modulation scheme since it can tolerate slow rotations in the constellation symbol space and thus the use of a carrier phase-locked recovery loop can be avoided [Sheng94].. In our case, the symbol rate is 1 Mbaud, so there will be a 7.8° rotation in the symbol space, which is tolerable.

A DQPSK encoder converts two bits of user data into one complex symbol from the following set: 0 (1,0), $\pi/2$ (0, 1), π (-1,0), and $3\pi/2$ (0, -1). The real part is called the in-phase signal (I) and the imaginary part is called the quadrature signal (Q). It works by encoding two bits as a phase offset which is added to the previously transmitted symbol to determine the next transmitted symbol. The phase offset which should be added to the previous symbol for each combination of bits is described in the following table:

Bit_{n+1}	Bit_n	Δ_θ
0	0	0°
0	1	90°
1	0	270°
1	1	180°

Example. The following bit pairs need to be transmitted: (0, 0), (1, 1), (1, 0), (1, 0), (0, 0), (0, 1), (1, 0) (note: the order of transmission is assumed (bit_{n+1}, bit_n), (bit_{n+3}, bit_{n+2}), ...). The previous symbol transmitted was $\pi/2$. Thus, the next six symbols to be transmitted would be $\pi/2$, $3\pi/2$, π, $\pi/2$, $\pi/2$, π, and $\pi/2$.

3.3.3 Data Correlation

As discussed in Section 2.2, the method used to recover data is to first multiply the received signal by the PN sequence and the Walsh sequence assigned to the user, then correlate the results for 64 T_{chips}. In our system, there are 64 possible Walsh codes. Walsh code 0 is used for the pilot tone, Walsh code 1 is used for the control channel, and Walsh codes 2-63 are used for the user data channels. Because the signal is DQPSK encoded there are actually two streams which need to be correlated-one for I and one for Q. A block diagram of the data recovery block can be seen in Figure 3.11. The chips fed to both the I and Q data correlators are 4-bit sign-magnitude numbers. Since the PN and Walsh sequences are just streams of +1s and -1s, a multiplication by -1 is just a sign-bit toggle, and a multiplication +1 will cause no sign-bit change. The correlation of a 4-bit sign-magnitude number over 64 T_{chips} will produce a 10-bit sign-magnitude result.

3.3.4 DQPSK Decoding

The accumulated I stream value (I_{acc}) and the accumated Q stream value (Q_{acc}) is taken from the correlators and fed into the DQPSK decoder. Because I_{acc} and Q_{acc} are composed of the

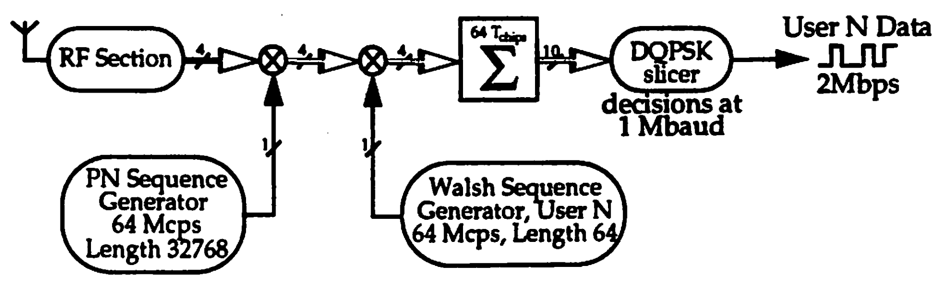

RF Section
4
4
4
64 T_chips
Σ
10
DQPSK slicer
decisions at 1 Mbaud
User N Data
2Mbps
1
1
PN Sequence Generator 64 Mcps Length 32768
Walsh Sequence Generator, User N 64 Mcps, Length 64

FIGURE 3.11 : Data recovery block diagram. Hollow arrow indicates complex valued data [Teus94].

despread user data plus the spread noise as opposed to only despread user data, their values will be non-ideal. The error term will be small, and the DQPSK slicer will correctly determine which bits were transfered.

The slicer makes it's decision based on the phase difference between the current symbol $\overline{S_n}$ which is defined by $I_n + jQ_n$ and the previous symbol $\overline{S_{n-1}}$ which is defined by $I_{n-1} + jQ_{n-1}$. The angle between the two can be found by performing the following division

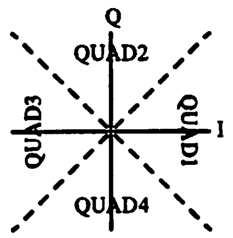
$$\frac{\overline{S_n}}{\overline{S_{n-1}}} = \frac{|S_n|}{|S_{n-1}|}\angle(\theta_n - \theta_{n-1}) = \frac{(I_n + jQ_n)}{(I_{n-1} + jQ_{n-1})}. \qquad \textbf{(EQ 3.3)}$$

The equation can be simplified by multiplying the top and bottom by $\overline{S_{n-1}}^*/\overline{S_{n-1}}^*$, thus obtaining the following:

$$\frac{\overline{S_n}}{\overline{S_{n-1}}} \cdot \frac{\overline{S_{n-1}^*}}{\overline{S_{n-1}^*}} = \frac{I_n + jQ_n}{I_{n-1} + jQ_{n-1}} \cdot \frac{I_{n-1} - jQ_{n-1}}{I_{n-1} - jQ_{n-1}} = \frac{(I_n I_{n-1} + Q_n Q_{n-1}) + j(I_{n-1}Q_n - I_n Q_{n-1})}{\text{Real Part}} \qquad \textbf{(EQ 3.4)}$$

The angle is defined by the numerator and thus the denominator is not needed. In actuality, we do not need to know the angle per se, but only the relationship between the absolute values of the real and imaginary parts of the numerator and their sign. The following table describes the decision regions and shows how we can use the absolute values and signs of the real and imaginary parts to make a decision:

Quadrant	Phase Diference	Check	Output (Bit_{n+1}, Bit_n)
1	$-45° < \Delta_\theta \le 45°$	$\lvert Re\rvert > \lvert Img\rvert$, Re+	(0,0)
2	$45° < \Delta_\theta \le 135°$	$\lvert Img\rvert > \lvert Re\rvert$, Img+	(0,1)
3	$135° < \Delta_\theta \le 225°$	$\lvert Img\rvert > \lvert Re\rvert$, Img-	(1,1)
4	$-135° < \Delta_\theta \le -45°$	$\lvert Re\rvert > \lvert Img\rvert$, Re-	(1,0)



Note: $Re = I_n I_{n-1} + Q_n Q_{n-1}$ and $Im = I_{n-1} Q_n - I_n Q_{n-1}$.

The one drawback of using differential encoding is the BER for DQPSK is 3dB lower than the SNR for coherent QPSK. This is because an error in the nth bit affects the difference between not just the nth bit and the n-1th bit, but also between the nth bit and the n+1th bit. This is illustrated in the following example:

Example. The symbols $\pi/2$, $\pi/2$, $3\pi/2$, π, $\pi/2$, $\pi/2$, π, and $\pi/2$ are transmitted. However, due to multipath interference, the symbols $\pi/2$, $\pi/2$, $3\pi/2$, $3\pi/2$, $\pi/2$, $\pi/2$, π, and $\pi/2$ are received. As a result, the outputted bits are (0,0), (1,1), (0,1), (0,1), (0,0), (0,1), and (1,0) in which bits 6and 7 are incorrect. If QPSK encoding was used, then only a 1 bit error would have occurred.

3.3.5 Can a RAKE[1] help?

The signal being transmitted to the users is coming from one source--the basestation. However, because of reflections from walls, ceilings, people, and furniture, there is attenuation and multipath delay of the arriving signal. Hence, at any instant of time, the received signal is not only composed of the primary line of sight component, but is also composed of several secondary components due to reflections. The time between when the transmitted signal first arrives at the receiver and the time when the last multipath arrival is received is called the *delay spread.* When the delay spread of the channel is greater than the symbol period, intersymbol interference (ISI)

1. The matched filter demodulator is called a RAKE correlator because of the resemblance of the tapped-delay-line matched filter to an ordinary garden rake. That is, the RAKE matched fileter/correlator resembles a garden rake in the way it collects the signal energy from all the resolvable multipath signal components [Proak89].

ensues. Measurements done by [Seid91] have shown the delay spread for an indoor channel to be between 20ns and 60ns. Since the signal is only resolvable at multiples of the chip time, T_{chip}, the number of resolvable multipaths is equal to the following equation:

$$N = T_{delayspread}/T_{chip} + 1 \qquad \text{(EQ 3.5)}$$

In our case, since $T_{delayspread}$ and T_{chip} equal 40ns and 16ns respectively, the number is resolvable multipaths is approximately three. If nothing is done, then the multipath arrivals will act as interference and lower the SNR of the received signal. The idea behind a RAKE receiver, however, is to somehow use the information contained in the multipath arrivals (essentially delayed and attenuated copies of the original signal) to boost the SNR.

The RAKE receiver uses the pilot tone to obtain an estimate of the channel's impulse response. For coherent detection, each finger's output is phase-corrected according to the carrier-phase shift indicated by the channel estimate as well as attenuated according to the relative magnitude of the multipath arrival versus the line-of-sight (LOS) component. All the phase-corrected and weighted outputs are then summed and fed to the DQPSK slicer. A block diagram of a RAKE receiver can be seen in Figure 3.12. For the case with two significant paths with path delay amplitudes of β_0 and β_1 and phase delays of θ_0 and θ_1, the decision variable would look like the following:

$$Y = Y_0 + \frac{\beta_1}{\beta_0} e^{-j(\theta_1 - \theta_0)} Y_1 \qquad \text{(EQ 3.6)}$$

This assumes additive white Gaussian noise (AWGN) is the dominant noise source. However, in the Infopad system, interference, not AWGN, is assumed to be the dominant noise source. It was shown in [Teus94], that the decision variable in an interference-limited system would instead look like the following

$$Y = Y_0 + \left(\frac{\beta_1}{\beta_0}\right)^3 e^{-j(\theta_1 - \theta_0)} Y_1 \qquad \text{(EQ 3.7)}$$

Notice that the weighting coefficient for the first multipath arrival increases as the cube of the amplitude ratio instead of just linearly as in Eqn. 3.5. In other words, if β_1, the amplitude of Y_1, is

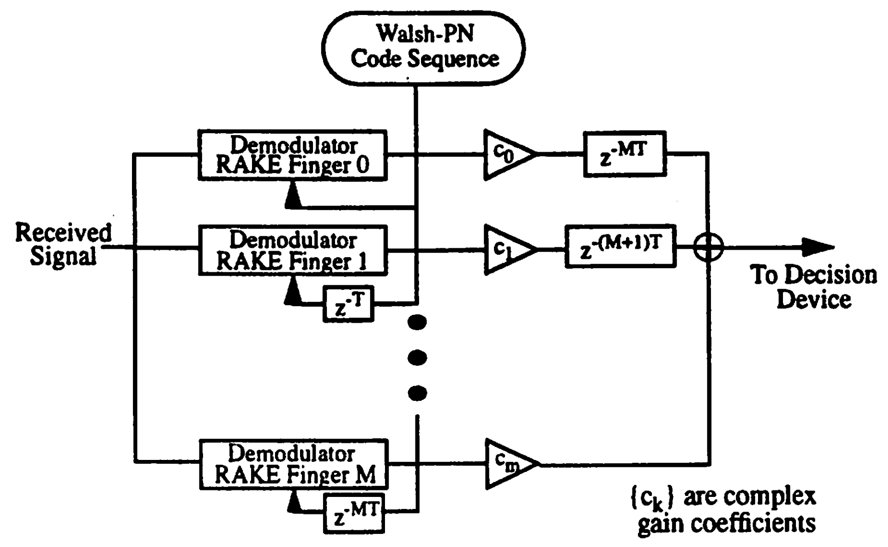

Walsh-PN Code Sequence
Demodulator RAKE Finger 0
Demodulator RAKE Finger 1
Demodulator RAKE Finger M
Received Signal
c_0
c_1
c_m
z^{-MT}
$z^{-(M+1)T}$
z^{-T}
z^{-MT}
To Decision Device
$\{c_k\}$ are complex gain coefficients

FIGURE 3.12 : RAKE receiver. Each finger is separated by one chip [Teus94].

smaller than β_0, the amplitude of Y_0, than the information contributed by the second term will be minimal. For example, if β_1 is half β_0 then the weighting coefficient will be 1/8. This scenario is likely because of the Rician distributed nature of the channel (i.e. one LOS component and several weaker multipath arrivals). Other techniques such as echo cancellation are being explored by Teuscher.

3.4 Adjacent Cell Scan/Handoff

One of the major benefits of the Infopad is its inherent mobility resulting from operating in a wireless LAN environment. The design specfication has placed no restrictions (within reason) on which basestations the Infopad may be connected to. In other words, as long as there is available capacity, the pad is free to wander from room to room, basestation to basestation and at all times it should be able to maintain connection with the backbone network.

Initially, when the Infopad is first turned on, it will lock onto the nearest basestation. This is done using the lock synchronization circuitry described in Section 3.1. However, it is very likely that as the pad roams around, it will receive pilot tones from other nearby basestations. As long as the current basestation's pilot tone energy is the strongest, then the Infopad will remain locked to the current basestation. However, if the Infopad nears the edge of the current cell or even enters an adjacent cell, then the pilot tone energy received from its current basestation may be weaker than the pilot tone energy received from the adjacent cell. In that case, it would be wise to stop receiving the data from the original basestation and start receiving data from the new cell's basestation. This process of determining which basestation's pilot tone signal is the strongest is called *adjacent cell scan* and the process of moving from one basesation to another is called *handoff*. This section focuses on these two functions.

3.4.6 Adjacent Cell Scan

At any given time, the mobile unit will be receiving information transmitted from the basestation it is currently locked onto as well as from adjacent cells. A part of each basestation's transmitted signal is the pilot tone which is just the periodic pseudorandom sequence of 32768 chips. Because the pilot tone is guaranteed to use at least 20% of the signal power, it can be used to provide a type of received signal strength indicator (RSSI). All that needs to be done is to correlate the received signal containing the pilot tone using a local PN sequence to do the despreading. If the transmitter's PN sequence (a.k.a. the pilot tone) and receiver's PN sequence are aligned to within one chip, then there will be a large peak in the correlated energy. If they are not aligned, then the correlated energy will be small. Because the property, the pilot tone can be used to distinguish between basestations.

If each basestation is synchronized with the others, then they can each be assigned their own unique phase of the pilot tone. For example, one cell, cell 0, could be assigned phase 0 and the adjacent six cells, cells 1-6, could be assigned phases 2048, 4096, 6144, 8192, 10240, and 12288 respectively. By correlating over all 32768 phases of the pilot tone and keeping track of the top three correlation energies and their phase offsets, one can determine which basestations are nearby. The adjacent cell scan circutry (ACS) uses the following algorithm:

① WAIT FOR LOCK ACQUISTION.

② RESET THE ACS'S PN GENERATOR TO PHASE$_0$ AND CLEAR OUT THE TOP ENERGY REGISTERS.

③ CORRELATE THE RECEIVED SIGNAL FOR 1024 CYCLES USING THE ACS'S PN GENERATOR AND WALSH CODE 0 (ALL ONES). THIS WILL EXTRACT THE PILOT TONE ENERGY FROM THE SIGNAL.

④ IF THE CORRELATED ENERGY IS ONE OF THE TOP THREE CORRELATED ENERGIES SO FAR, THEN SAVE THE CORRELATED ENERGY AND THE ACS'S PN GENERATOR'S CURRENT PHASE OFFSET.

⑤ CLEAR OUT THE ACCUMULATED ENERGY. IF THE CURRENT PHASE OF THE ACS'S PN GENERATOR IS LESS THAN 32767, THEN STALL THE PN GENERATOR FOR ONE T_{CHIP}. THIS WILL SHIFT THE PN GENERATOR'S PHASE BY ONE CHIP. GOTO ③. ELSE, IF THE CURRENT PHASE IS 32767, THEN GOTO ⑥.

⑥ DUMP THE TOP THREE ENERGIES AND THEIR CORRESPONDING PHASE OFFSETS TO ANOTHER SET OF REGISTERS AND EMIT A SIGNAL OFF-CHIP INDICATING THAT NEW RSSIS ARE AVAILABLE. GOTO ②.

For example, if the mobile is in cell 1 and near cells 5 and 6, then the output of the ACS circuitry might indicate that the top energy is at phase 10240, the next is at phase 2048, and the last is at phase 12288.

This technique of using pilot tone as an RSSI is used in the lock circuitry, the channel estimators, and the DPLL as well.

3.4.7 Handoff

The top three energies and their offsets are transmitted to the basestation. A decision can then be made by the controlling network software ("cellserver") whether or not to perform a handoff. The decision is not only based on the received RSSIs, but also on the current occupation of the cells. For example, the signal received from cell 5 may be stronger than the current cell, cell 1, but cell 5 is at capacity and cannot support any more users. Thus, even though, it would be desirable for the user to switch from cell 1 to cell 5, it may not be feasible. If the cellserver decided, however, to perform a handoff, then the following things must happen [Le95]:

① THE CELLSERVER DATABASE MUST BE UPDATED TO INDICATE THAT THE USER HAS MOVED TO A NEW CELL.

② THE CELLSERVER MUST ASSIGN A NEW WALSH CODE FOR THE USER'S DATA.

③ THE NEW CELL MUST NOW TRANSMIT THE USER'S DATA USING THE APPROPRIATE WALSH CODE.

④ THE OLD BASESTATION MUST TRANSMIT THE NEW WALSH CODE TO THE INFOPAD.

⑤ THE INFOPAD MUST LOAD IN THE NEW WALSH CODE AND BEGIN RECEIVING DATA FROM THE NEW BASESTATION.

⑥ THE OLD BASESTATION STOPS TRANSMITTING DATA FOR THE USER.

CHAPTER 4

External Documentation

4.1 I/O

The inputs, outputs, and power supply pins of the spread spectrum demodulator chip are described in Table 5.1 and Table 5.2.

Table 4.1 : Pinout of demodulator chip listed in order of pin number.

PIN	I/O/ SUPPLY	NAME	PIN	I/O/ SUPPLY	NAME	PIN	I/O/ SUPPLY	NAME
1	S	GND_1	45	S	VDD_11	89	I	DATAIN5
2	O	ODATA27	46	I	OSCH	90	O	ODATA10
3	S	VDD_1	47	I	CURRBIAS	91	S	VDD_6
4	I	ECRSTDUMP	48	I	OSCL	92	O	ODATA9
5	O	ODATA26	49	S	GND_5	93	S	PWR1_5
6	I	RESETL	50	S	VDD_4	94	I	QINX2
7	O	ODATA25	51	O	DUMP1024H	95	O	ODATA8
8	I	IINX3	52	I	DATAIN14	96	O	DUMP64H
9	S	PWR3_1	53	S	PWR5_4	97	I	ECDATA2
10	S	PWR5_1	54	S	PWR3_2	98	O	ODATA7
11	I	IIN3	55	I	IIN0	99	O	CMP3THL
12	I	ADDR1	56	I	DATAIN13	100	S	GND_8
13	I	ADDR0	57	O	ODATA17	101	O	ODATA6
14	O	ICLK	58	I	DATAIN12	102	S	VDD_7
15	O	QCLK	59	O	ODATA16	103	I	ECDATA1
16	S	PWR1_1	60	I	DATAIN11	104	O	ODATA5

Table 4.1 : Pinout of demodulator chip listed in order of pin number.

PIN	I/O/ SUPPLY	NAME	PIN	I/O/ SUPPLY	NAME	PIN	I/O/ SUPPLY	NAME
17	S	PWR5_2	61	I	IINX0	105	S	PWR1_6
18	S	VDD_2	62	O	ODATA15	106	O	ODATA4
19	I	CSL	63	S	PWR1_8	107	I	QINX1
20	I	WRL	64	I	ECW	108	S	PWR3_5
21	I	EC64CLK	65	O	ODATA14	109	S	VDD_8
22	I	IIN2	66	O	CMP2THH	110	I	QIN1
23	S	PWR1_2	67	S	VDD_5	111	I	DATAIN4
24	O	ODATA24	68	O	ODATA13	112	I	DATAIN3
25	I	TSTMODE0	69	S	PWR5_5	113	I	CLKRST
26	O	ODATA23	70	I	ECDATA3	114	S	VDD_9
27	I	TSTMODE1	71	O	ODATA12	115	S	PWR5_6
28	I	IINX2	72	I	DATAIN10	116	S	PWR3_6
29	O	ODATA22	73	O	ODATA11	117	S	PWR1_7
30	S	GND_2	74	I	QINX3	118	S	PWR5_7
31	I	ECDUMP	75	I	DATAIN9	119	S	PWR3_7
32	O	ODATA21	76	S	GND_6	120	S	GND_9
33	O	LOCK	77	I	QIN3	121	I	QIN0
34	S	PWR5_2	78	I	DATAIN8	122	I	DATAIN2
35	O	ODATA20	79	S	PWR1_3	123	O	ODATA3
36	I	OMODE2	80	O	DUMPRST	124	I	DATAIN1
37	I	ECPN	81	O	STALLL	125	O	ODATA2
38	O	ODATA19	82	S	PWR1_4	126	I	DATAIN0
39	S	VDD_3	83	S	PWR3_3	127	I	QINX0
40	O	ODATA18	84	I	DATAIN7	128	O	ODATA1
41	I	IINX1	85	S	PWR3_4	129	S	VDD_10
42	I	OMODE1	86	S	GND_7	130	I	ECDATA0
43	I	OMODE0	87	I	DATAIN6	131	O	ODATA0
44	I	IIN1	88	I	QIN2	132	O	STRETCHSAMP

Table 4.2 : Pinout for demodulator chip grouped by signal name.

NAME	PIN COUNT	DESCRIPTION	PIN(S)
CLKRST	1	Reset signal for clock generator	113
RESETL	2	Chip reset signal	6
OSCH	3	Positive phase of 128MHz sinusoidal oscillator	46
OSCL	4	Negative phase of 128MHz sinusoidal oscillator	48
CURRBIAS	5	Bias voltage for clock pad	47
DATAIN[14:0]	20	Bus used to load values into registers	52,56,58,60,72,75,78,84,87, 89,111,112,122,124,126
ADDR[1:0]	22	Register address lines	12,13
CSL	23	Chip select	19
WRL	24	Write signal for register	20
OMODE[2:0]	27	Select which signals to output	36,42,43
TSTMODE[1:0]	29	Determines the type of input to the chip	27,25
IIN[3:0]	33	Input data from ADC or transmitter chip	11,22,44,55
QIN[3:0]	37	Input data from ADC or transmitter chip	77,88,110,121

Table 4.2 : Pinout for demodulator chip grouped by signal name.

NAME	PIN COUNT	DESCRIPTION	PIN(S)
IINX[3:0]	41	Input data from transmitter chip (for testing only)	8,28,41,61
QINX[3:0]	45	Input data from transmitter chip (for testing only)	74,94,107,127
ODATA[27:0]	73	Output data bus	2,5,7,24,26,29,32,35,38,40, 57,59,62,65,68,71,73,90,92, 95,98,101,104,106,123,125, 128,131
ICLK	74	unused-connected to ground	14
QCLK	75	internal system clock	15
LOCK	76	high when mobile has acquired lock	33
STALLL	77	low when PN generator is being stalled	81
STRETCHSAMP	78	0: Adjusted phase by - $T_{chip}/4$ 1: Adjusted phase by + $T_{chip}/4$	132
CMP2THH	79	0: No adjustment made to phase of clock 1: Adjusted phase of clock	66
CMP3THL	80	0: Not enough energy to be in lock 1: Still have energy to be in lock	99
DUMPRST	81	Indicates that the MP1 correlator was cleared	80
DUMP64H	82	Indicates when output of data recovery correlator is valid	96
DUMP1024H	83	dicates when output of long correlators are valid	51
ECRSTDUMP	84	Extra correlator reset latches while dumping	4
EC64CLK	85	Extra correlator clock	21
ECDUMP	86	Extra correlator dump output	31
ECPN	87	Extra correlator PN input	37
ECW	88	Extra correlator walsh input	64
ECDATA[3:0]	92	Extra correlator data	70,97,103,130
PWR1[	100	1.5V internal power supply	16,23,63,79,82,93,105,117
PWR3	107	3.3V internal power supply	9,54,83,85,108,116,119
PWR5	114	5V internal power supply	10,17,34,53,69,115,118
VDD	125	5V pad power supply	3,18,39,45,50,67,91,102, 109,114,129
GND	132	ground	1,30,49,76,86,100,120

4.2 Programmable Registers

The registers in the demodulator are two-level latches. This allows the front half of the register to be externally written into without affecting the internal operation of the chip which depends on the registers' values. A block diagram of a register can be see in Figure 5.1 along with its control signals. The signal CLKX (X signifies the register number) is externally controlled by the CS_L, WR_L and ADDR signals. Since there is only one set of the ADDR, CS_L and WR_L lines, only one register can be updated at a time. The backend of the register is updated when CLK_BACK goes high. This occurs every rising edge of CLK64 during reset and when the PN generator's

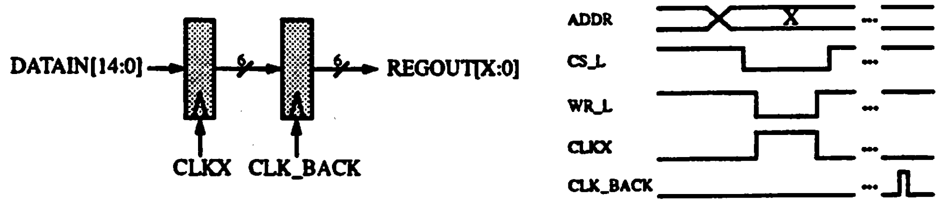


FIGURE 4.1 : Block diagram of two level registers along with timing diagram.

output is all ones during normal operation. There are four registers in the demodulator whose functions and bit-widths are described in Table 5.3.

Table 4.3 : Describes the demodulator's four writable registers.

ADDR	NAME	BIT WIDTH	FUNCTION
00	WALSH	6	Walsh number
01	THRESA	14	Used to determine whether coarse lock has been acquired
10	THRESB	15	Used to determine whether the Infopad is still in lock
11	THRESC	15	Used to determine whether clock phase should be adjusted by $T_{chip}/4$

4.3 Observation Modes

The OMODE signal is a 3-bit signal that controls which signals will be driven onto the 28-bit output bus. A description of all eight possibilities along with an explanation of the signals outputted can be seen in Figure 4.2.

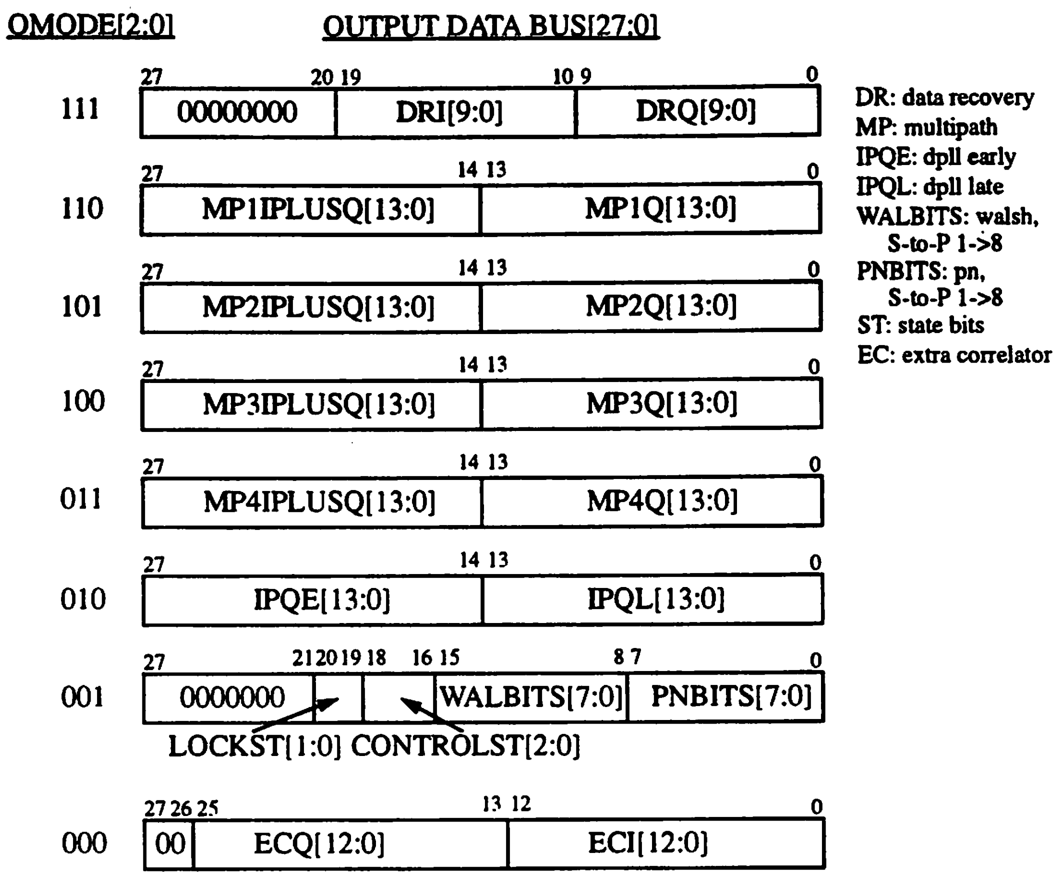


FIGURE 4.2 : Describes the different signals which can be driven onto the output data bus.

4.4 Test Modes

The demodulator has three functional modes (see Table 5.4) which are controlled by the 2-bit

TSTMODE signal. One mode is used for regular operation and two are used for testing purposes.

Table 4.4 : Description of the three operating modes for the demodulator.

TSTMODE	MODE	Inputs	Comments
00	normal	IIN,QIN	Data is unmodified
01	test	IIN,QIN	Data converted from 4-bit binary (0 to 15) to 4-bit sign magnitude (-7 to 7)
10	test	IIN,QIN, IINX,QINX	Data converted from 4 parallel 2's complement (-8 to 7) streams at 1/2 OSCH's frequency to two interleaved sign-magnitude (-7 to 7) streams at OSCH's frequency
11	UNDEF.	UNDEF.	UNDEF.

Independent of the test mode, after the data has passed through the testmode block, it will consist of two 4-bit sign-magnitude interleaved streams at a frequency equal to one-half OSCH's frequency.

CHAPTER 5

Internal Documentation

This chapter gives a high-level overview of each block in the chip; more detailed schematics can be found in Appendix A.

5.1 Correlator

The heart of the demodulator chip is the correlator which can be thought of as an accumulator. Its role is to accumulate the 64 MHz 4-bit sign-magnitude data whose sign bit has been multiplied by a bit from the PN and from the Walsh sequence. It accumulates data until an external dump signal is received, whereupon, the final sum is latched, the correlator's internal registers are cleared and the accumulation process begin again.

5.1.1 Basic Architecture

The basic correlator block can be seen in Figure 5.1 It should be noted that the correlator shown in Figure 5.1 is actually one-half of a complex I/Q correlator. The exact same circuitry is

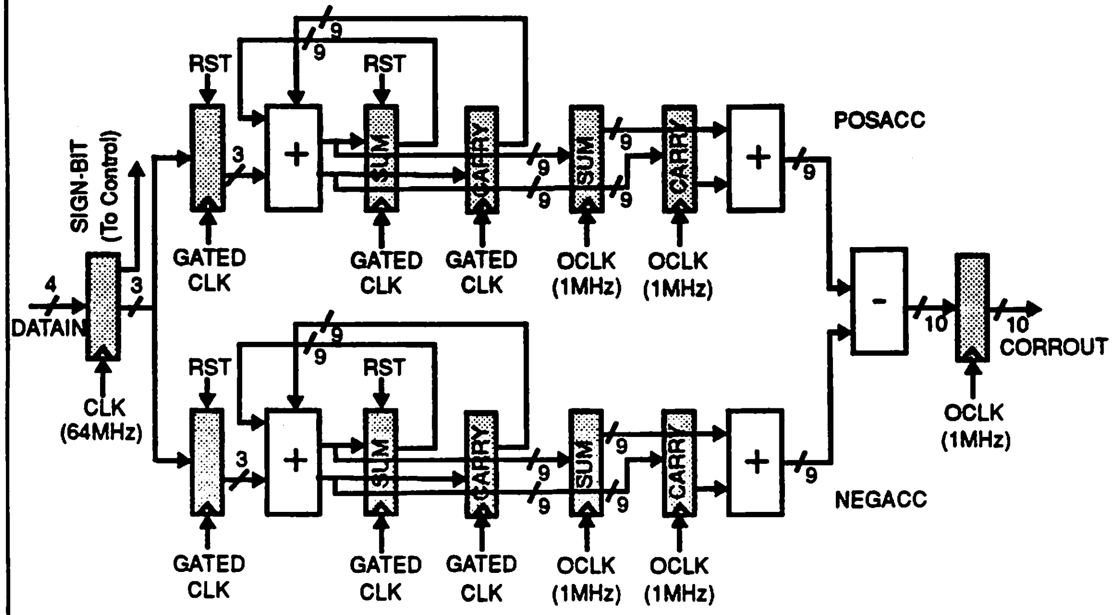


FIGURE 5.1 : Datapath for correlator (one-half of complex I/Q).

used for the I path and the Q path. The correlator works in the following manner: an incoming 4-bit sign-magnitude piece of data is latched by a 4-bit register. Then, depending on the data's sign bit which has already been multiplied by bits from the PN and Walsh sequences, it will either be latched by the first register in the POSACC datapath or the NEGACC datapath. If, for example, the data's sign was positive, then the data will be latched by the first POSACC register. At the same time, the previously accumulated SUM and CARRY vectors are latched and fed back into the adder. Thus, this new piece of data is added with to the previously accumlated positive data. In actuality, the addition is broken up into a 3-bit add and a 6-bit accumulate. The clocks to the NEGACC datapath are gated off, so no switching activity occurs in the NEGACC datapath. Likewise, if data is negative, the NEGACC datapath is used and the POSACC datapath remains idle. By using a carry-save architecture instead of the more traditional ripple-carry approach, the critical path was reduced to a half-adder and a register delay. Effectively, the adder becomes pipelined at the bit-level. A bit-slice of the carry-save adder can be seen in Figure 5.2.

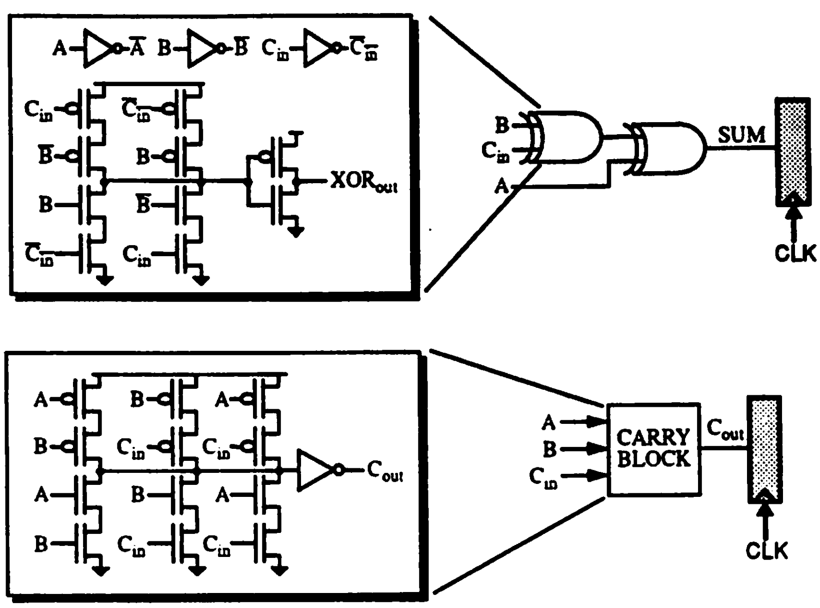


FIGURE 5.2 : Bit slice of the carry-save adder.

The data is accumlated until a dump signal is received, whereupon, the final sum and carry vectors are latched for each datapath and summed together to produce the negative and positive accumulated data. The negative accumulated data is then subtracted from the positive and the output is determined. Both the adders and the subtractor use a 2's-complement ripple-carry architecture. If these final adds and subtract were a part of the critical path, then the maximum speed of the carry-save adder would be limited by these slow blocks. Fortunately, in our system, this is not the case. In our design, 64 chips of data are accumulated before being dumped. Thus, at the same time the correlator is accumulating a new set of 64 samples, the backend circuitry performs the final addition and subtraction, operating at 1MHz. As a result, the backend processing only needs to operate at 1MHz.

The correlator just described above can only correlate 64 samples at a time. For the data recovery block, no additional circuitry is needed since only correlations over 64 chips are needed. However, for the remainder of the blocks, 1024 sample correlations are desired. For these longer correlations, the outputted data, CORROUT, can be feed into a 14 bit accumulator which would be updated 16 times (once every 1 μs) before producing the final output.

5.1.2 Constellation Rotation

While the 21.76kHz constellation rotation described in Section 3.3.2 may be tolerable for the DQPSK decoder, it is not tolerable for the channel estimation correlators. This is because they correlate the pilot tone over 1024 T_{chips}, instead of 64 T_{chips}, which results in a 125.2° rotation. Each correlator sums for 1024 T_{chips} both the in-phase and quadrature phase components of the received pilot tone. The equations can be expressed as

$$Q = \sum_{i=0}^{1023} S_i \sin\theta_i, I = \sum_{i=0}^{1023} S_i \cos\theta_i \qquad \text{(EQ 5.1)}$$

where i represents the number of chips, S_i is the ith data sample received (assumed already multiplied by the bits from the PN and Walsh sequences) and θ_t is the amount the constellation has rotated (in degrees) since the beginning of the correlation. Again, the angle θ_t varies from 0° when i=0 to 125.2° when i=1023 or increases 0.12° per T_{chip}. If the correlators operated in this manner without being modified, then this rotation would produce severe errors in the estimation of the pilot tone energy (used for lock and the DPLL) as well as errors in the estimation of the channel's impulse response (used in the RAKE receiver).

The first pass of the demodulator only used the on-time data and did not try to improve the SNR through ratio combining with the multipath arrivals. As a result, a solution was proposed which attempted to fixed the rotation problem for pilot tone energy estimation, but not for estimating the channel's impulse response since channel estimation was not being performed. In actuality, the currently implemented solution prohibits estimation of the channel's impulse response--the reason why will be explained shortly.

The modified correlators still correlate over 1024 T_{chips}, but instead of correlating continuously for

1024 T_{chips}, they are broken into 16 correlations of 64 T_{chips}. If this were the only change, then the correlators would be functionally identical to the original ones in Equation 5.1. The key modification is that the absolute value of each individual correlation over 64 T_{chips} is taken *before* being added into the running accumulation of 1024 T_{chips}. Thus, the new correlation equations look like the following:

$$Q = \sum_{j=0}^{15} \left| \sum_{i=0}^{63} S_{ij} \sin\theta_{ij} \right|, I = \sum_{j=0}^{15} \left| \sum_{i=0}^{63} S_{ij} \cos\theta_{ij} \right| \qquad \text{(EQ 5.2)}$$

Taking the absolute value has the same effect as beginning a new correlation. Thus, since the absolute value is taken every 64 T_{chips}, it is like the constellation only rotates 7.8° which is tolerable. Unfortunately, though, taking the absolute value does not come without a price. The price is that the magnitude information is maintained, but the phase information is lost. This is acceptable for pilot tone energy estimation since it is only concerned with magitudes, but not for channel impulse response measurements which require both magnitude and phase information. In other words, using this technique prevents channel impulse response measurements from being performed.

The channel impulse response measurement determines the nature of the channel for the on-time data and the data which has been delayed by one and two T_{chips} due to multipath. This will be used in the RAKE receiver. The details of the RAKE are discussed in Section 3.3.5, but it suffices to say that only the relative phase information between the on-time channel reponse and the two multipath channel responses is needed. One technique to maintain the relative phase information is shown in the following equations:

$$I_n = \mathrm{Re}\left\{ \sum_{j=0}^{15} \left(\frac{\sum_{i=0}^{63} S_{ij,n} \cos\theta_{ij,n} + j \sum_{i=0}^{63} S_{ij,n} \sin\theta_{ij,n}}{\sum_{i=0}^{63} S_{ij,\text{on-time}} \cos\theta_{ij,\text{on-time}} + j \sum_{i=0}^{63} S_{ij,\text{on-time}} \sin\theta_{ij,\text{on-time}}} \right) \right\} \qquad \text{(EQ 5.3)}$$

$$Q_n = \mathrm{Im}\left\{\sum_{j=0}^{15}\left(\frac{\sum_{i=0}^{63} S_{ij,n}\cos\theta_{ij,n} + j\sum_{i=0}^{63} S_{ij,n}\sin\theta_{ij,n}}{\sum_{i=0}^{63} S_{ij,\text{on-time}}\cos\theta_{ij,\text{on-time}} + j\sum_{i=0}^{63} S_{ij,\text{on-time}}\sin\theta_{ij,\text{on-time}}}\right)\right\} \quad \text{(EQ 5.4)}$$

where n is the nth multipath bounce. This technique should remove the effect of the constellation rotation, but it will distort the magnitude. The difference between the magnitude of I_n in Equation 4.4 and $\sum_{i=0}^{1023} S_{i,n}\sin\theta_{i,n} / \sum_{i=0}^{1023} S_{i,\text{on-time}}\sin\theta_{i,\text{on-time}}$ is similar to the difference between $\frac{a+b}{c+d}$ and $\frac{a}{c}+\frac{b}{d}$, which is small if $a \approx b$ and $c \approx d$.

5.1.3 Power Consumption

The correlator was designed with low power consumption in mind. Two important architectural features were incorporated to achieve this goal. The first was that the correlator uses a sign-magnitude adder--there are separate adders for the positive and negative data; the second was that each adder uses a carry-save architecture. The benefits of these are explained below.

Sign-magitude vs. Ripple-carry adder

Simulations done in [Chand94] compared the power consumption of a 2's complement versus a sign-magnitude adder for different input patterns (see Table 5.1). Both adders used a ripple-carry architecture. It was shown that for random data, which characterizes the correlator's input pattern, a sign-magnitude adder consumes approximately 30% less power than a 2's complement adder. This is due to reduced switching activity.

Table 5-1 : Comparison of the power dissipation of 2's complement adder vs. sign-magnitude adder.

Input Pattern (1024 cycles)	2's Complement Power (3V)	Sign-Magnitude Power (3V)
constant (IN=7)	1.97 mW	2.25 mW
ramp (-7, -6, ..., 6, 7)	2.13 mW	2.43 mW
random	3.42 mW	2.51 mW
min→max→min (-7, 7, -7, 7, ...)	5.28 mW	2.46 mW

Carry-save architecture

By using the carry-save adder, instead of the slower ripple-carry, it was possible to reduce the supply voltage to 1.5V from 3V and almost a fourfold reduction in power consumption was achieved. Power reduction wasn't quite fourfold because the carry-save adder uses twice the number of registers as the ripple-carry, and as a result there is more switching and the clock is more heavily loaded.

5.1.4 Estimation of Pilot Tone Energy

The long correlators are used by the demodulator to estimate the received signal's energy by measuring the energy of the pilot tone. Ideally, one would like to use the magnitude of the complex pilot tone signal, $\sqrt{I^2 + Q^2}$, or even $I^2 + Q^2$ as a measure of the energy. However, in order to avoid using multipliers, the pilot tone energy was approximated as $|I| + |Q|$. Figure 5.3, shows the error which is incurred from using this approximation. The approxmation is plotted as $|\sin\theta| + |\cos\theta|$, where θ varies from 0° to 360°. The normalized error ranges from 0 to 0.41. By chosing proper values for the threshold registers, the impact of this error can be minimized.

5.2 Clock Mux

The role of the clock mux is twofold. One is to synthesize the clocks for the chip and the second is to convert the incoming interleaved data at rate OSCH into four parallel streams at rate OSCH/2.

5.2.5 Clock Generator

The Infopad's spread spectrum demodulator simultaneously requires 4 phases of a 64 MHz where each phase is separtated from one another by 4 ns. Another requirement imposed by the chip's digital phase lock loop is that the phase of the clocks must be adjustable by ± 4ns. This section explains how the 4 phases are generated as well as how their phase can be adjusted.

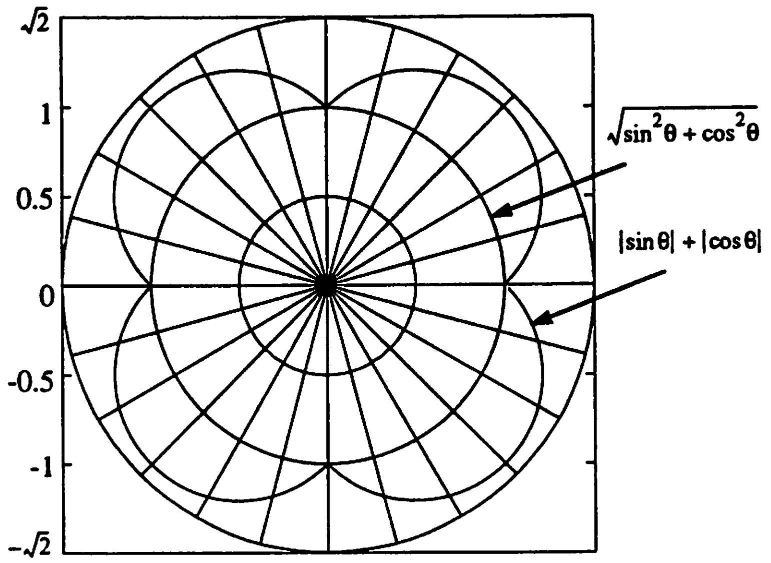


FIGURE 5.3 : Graph comparing the absolute value magnitude approximation vs. angle against the true magnitude calculation. The values have been normalized.

Input/Outputs

Table 5.2 shows the input and output signals for the clock generator.

Table 5.2 : I/Os for clock generator

Signal Name	Description	Direction
clk128_l	clock	Input
clk128_h	clock	Input
clkrst_h	reset clock generator flip-flops to known state	Input
reset_l	reset signal for control flip-flips	Input
valid_data	indicates when clock's phase should be adjusted; high for one 64 Mhz clock cycle every 1024 Tchips	Input
extend_phase	0 - reduce phase, 1 - extend phase	Input
change_phase_l	0 - change phase 1 - keep phase the same	Input
lock	0 - out of lock 1 - got lock	Input

Table 5.2 : I/Os for clock generator

Signal Name	Description	Direction
shr_l_1d (killpulse)	This signal "kills" one rising edge of the clock that is fed to the correlators. Used when reducing phase of clock by $-T_{chip}/4$.	
clk64	64 Mhz clock	Output

Implementation

The heart of the clock generating circuitry is two flip-flops (FF) whose outputs are inverted and used as their inputs (see Figure 5.4). One FF is clocked by the positive phase of a 128MHz

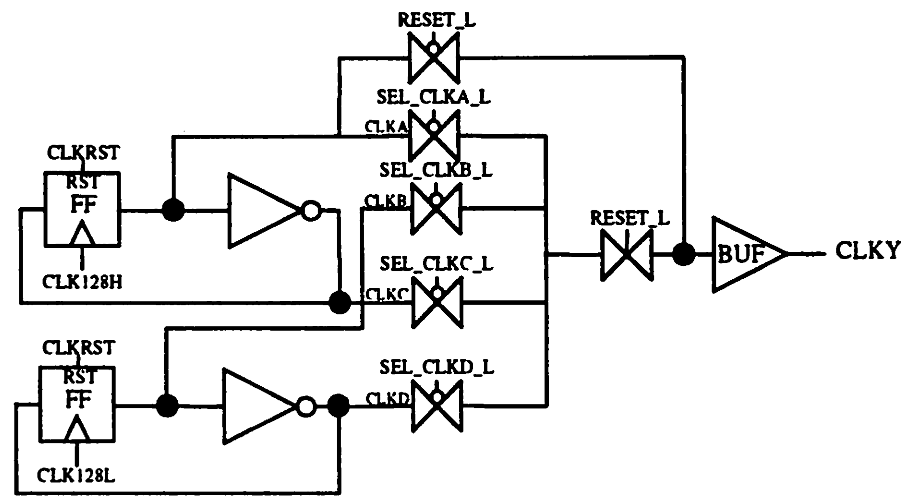


FIGURE 5.4 : Generates four phases of a 64MHz clock (clka, clkb, clkc, and clkd) which are offset from each other by 4 ns. Only one of the clocks is outputed as clkx.

clock and the other is clocked by the negative phase. The output of these FFs and the output of their trailing inverters generate the four phases of a 64 Mhz clock each offset in time by 4 ns (see Figure 5.5). At any given time, only one of the four active-low pass gates controlled by the sel_clk signals will be activated--the rest will be tristated.

There are two reset signals used in the circuitry shown in Figure 5.4. CLKRST is used to set the two FFs into a deterministic state. This signal only needs to be low for 10ns. RESET_L is the

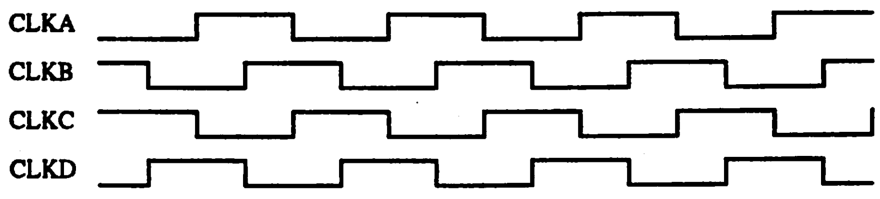


FIGURE 5.5 : Four phases of 64MHz clock generated inside clock generator.

global reset signal. When RESET_L is low CLKA is selected (the SEL_CLK pass gates are bypassed). Because the clock generator is supplying the clocks for the chip, we must guarantee that it is outputting a clock even during the reset mode in order to reset/clear the internal latches. The SEL_CLK signals depend on the clock which is produced by the clock generator in Figure 5.4. If we did not have the bypass pass-gates for reset, then the select lines would have to be initialized in order to produce the internal clock. However, this is impossible since they themselves depend on the internal clock to be initialized. When RESET_L is high, the output of the four SEL_CLK pass-gates is passed to CLKY. The output of this circuit, CLKY, is fed to a block of dual edge-triggered flip-flops (see Section 5.1.3) which generates the four phases of the 64MHz clock needed by the chip. The correlators have been designed to run at 64MHz (15.6ns period) when operating at 1.5 volts. However, when the clock's phase is reduced, for one cycle, the clock's period is reduced by 4 ns down to 11.7ns. Unfortunately, if this 11.7 ns period clock were passed to the correlators, then non-deterministic results would ensue. In order to combat this problem, we "kill off" a rising edge of the clock by setting killpulse_l low for one cycle. Thus, the effective period of the clock for this cycle becomes 27.3ns (see Figure 5.8). The penalty which is paid, is that the correlators lose one sample every time the clock's period is reduced. Therefore, the worst case is that one sample is lost every 1024 samples; this effect is tolerable. Unfortunately, the control circuitry cannot lose any rising-edges. This is because once we have lock, the transmitter's and receiver's PN generators are synchonized. If the receiver's PN generator loses a cycle due to having a rising-edge of it's clock removed, then it would no longer be synchronized with the

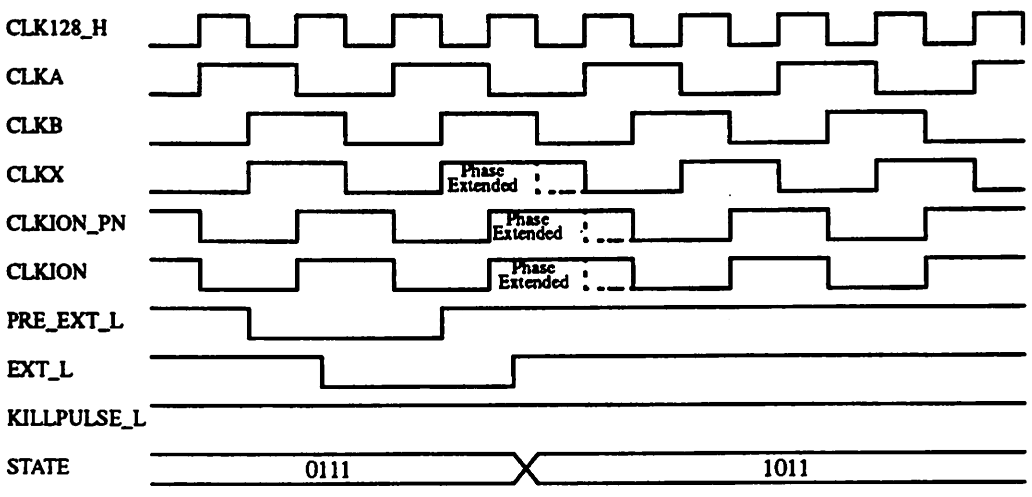


FIGURE 5.7 : Timing diagram for extending the clock's phase.

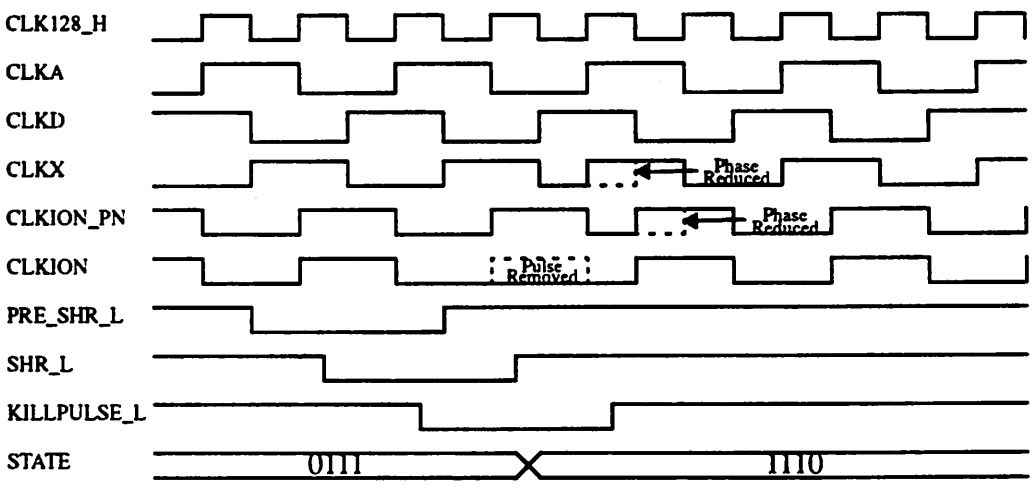
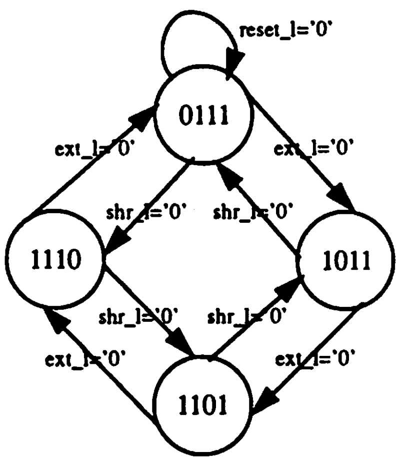


FIGURE 5.8 : Timing diagram for reducing the clock's phase.

Timing

A critical issue in design of the clock generator was making sure the critical path was met everywhere. For most of the circuitry the critical path was 16 ns. However, for several parts, the critical path was 4 ns. Because of the relatively large propagation time and setup time of the D-FF (about 2.0 ns), only about 1-2 levels of combinational logic could be tolerated. This 4ns critical path time consisted of the propagation time through a D-FF, the propagation time through two pass transmitter's PN generator. This would be catastrophic since the mobile would lose lock with the transmitter. As a result, there are two clocks which are generated; one for the control logic (CLK_ION_PN) and one for the correlators (CLK_ION).

The CLK_SEL signals are actually the state bits of a four-bit one-hot encoded state machine. Initially, the state is set to "0111". This means the CLKA pass-gate will be transparent while the others will be tristated. The three other valid states are 1011, 1101, and 1110 which cause CLKB, CLKC, or CLKD to be output respectively. The state transistion diagram can be seen in Figure 5.6.



$$\text{pre\_ext\_l} = \overline{\overline{\text{changephase\_l}} \cdot \text{lock} \cdot \text{valid\_data} \cdot \text{extend\_phase}}$$

$$\text{pre\_shr\_l} = \overline{\overline{\text{changephase\_l}} \cdot \text{lock} \cdot \text{valid\_data} \cdot \overline{\text{extend\_phase}}}$$

$$\text{ext\_l} = \text{latch(pre\_ext\_l)}$$

$$\text{shr\_l} = \text{latch(pre\_shr\_l)}$$

Note: state remains the same if ext_l='1' and shr_l='1' and reset_l='1'

FIGURE 5.6 : State transition diagram for SEL_CLK signals. Each state bit corresponds to a control bit for clock generator's pass-gates. Thus, bit3(msb)=SEL_CLKA, bit2=SEL_CLKB, bit3=SEL_CLKC, and bit4(lsb)=SEL_CLKD.

Normally, ext_l and shr_l equal '1'. The only time their values can change is once every 1024 cycles, when the digital phase lock loop may assert a change in the clock phase. At this time, valid_data will go high for one clock period. If changephase_l is low and lock is high, then either ext_l or shr_l will go low for one clock cycle depending on extend_phase. Example timing diagrams can be seen in Figure 5.7 and Figure 5.8.

gates, and the setup time for a D-FF. It was also found necessary to run the clock generator circuitry with a 5V supply to meet the timing requirements.

5.2.6 Dual-edge Trigger Flip-flop (DETFF) Block

This block generates the four phases of the 64MHz clock which are needed in the chip. In order to generate these four phases which are offset from one another by $T_{chip}/4$ or ~ 4ns, it is necessary to use a 128MHz clock to drive the DETFFs.

Inputs/Outputs

Table 5.3 : I/Os for DETFF block

Signal Name	Description	Direction
CLKIN	128MHz clock used to drive DETFFs	Input
DATAIN	64MHz clock from clock generator circuitry	Input
KILLPULSE_L	0 - keeps clk_ion, clk_qon, clk_ioff, clk_qoff low 1 - clk_ion, clk_qon, clk_ioff, clk_qoff are unmodified	Input
CLK_ION_PN	64 MHz clock used to drive control logic	Output
CLK_ION	64 MHz clock used to drive correlators. Also, used to sample data from the ADC.	Output
CLK_QON	64 MHz clock offset by $+T_{chip}/4$ from CLK_ION. Used to sample data from the ADC.	Output
CLK_IOFF	64 MHz clock offset by $+T_{chip}/8$ from CLK_ION. Used to sample data from the ADC.	Output
CLK_QOFF	64 MHz clock offset by $+T_{chip}/12$ from CLK_ION. Used to sample data from the ADC.	Output

Implementation

A block diagram of the DETFF block can be seen in Figure 5.9. The dual-edge triggered flops are taken from [Afgh91]. The basic schematic can be seen in Figure 5.10. One can see that the upper-half of the schematic is a standard rising-edge TSPCR (true-single phase clocking register) which can be found in [Burd94]. The lower-half is a falling-edge triggered TSPCR (fetFF) which is simply the dual of the rising-edge TSPCR (retFF). One can obtain the fetFF from the retFF by

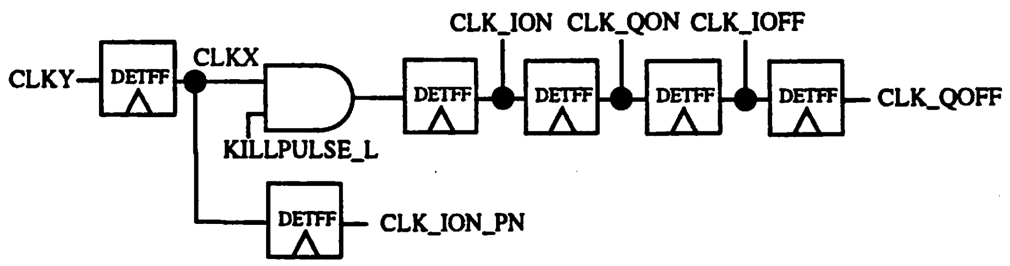


FIGURE 5.9 : Block diagram of DETFF block.

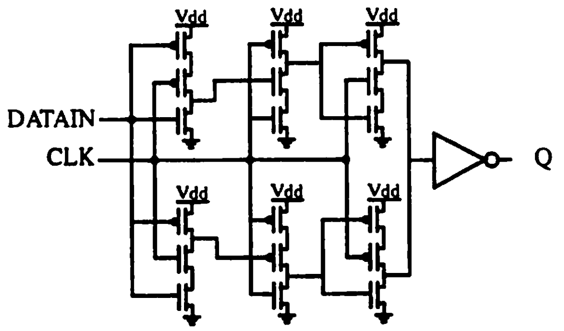


FIGURE 5.10 : Schematic for dual-edge triggered flip-flop.

flipping over each leg, then swapping NMOS for PMOS transistors and vice-versa. At any given time, either the retFF or the fetFF's output will be active--the other will be tri-stated.

5.2.7 Data Multiplexor

In the final incarnation of the radio, all aspects of the receiver, both analog and digital, will be integrated onto a single die. The interface between these two sections is, of course, the A/D converter (ADC) . By design, the ADC produces two 4-bit sign-magnitude interleaved streams of data running at 128MHz. The data multiplexor was added to convert this stream to 4 parallel non-interleaved (aligned) 4-bit streams running at 64MHz to simplify implementation and to provide $T_c/4$ timing resolution. Because the data has been oversampled by 2x, there is both ontime I and Q (ION and QON) data which is used for data recovery and lock acquisition and offtime I and Q (IOFF and QOFF) data which is used in the digital phase lock loop for fine timing adjustment.

Inputs/Outputs

Table 5.4 : I/Os for clkmux_datamux

Singnal Name	Description	Bit Width	Direction
CLK1A	Clock used to latch DATA1A signal.	1	Input
CLK1B	Clock used to latch DATA1B signal	1	Input
CLKR2_1X	Clock signal used to align I_{on}/Q_{on} or I_{off}/Q_{off}.	1	Input
CLKR3_1X	Clock signal used to align $I_{on}/Q_{on}/I_{off}/Q_{off}$.	1	Input
CLKR4_1X	Clock signal used to help synchronize datamux and correlators.	1	Input
MUXSEL1A	Used to select between DATA1A and DATA1B	1	Input
DATA1A	ADC stream which is 2x CLK1A's frequency	4	Input
DATA1B	ADC stream which is 2x CLK1A's frequency, 180° out of phase with DATA1A	4	Input
CLK2A	Clock used to latch DATA2A signal.	1	Input
CLK2B	Clock used to latch DATA2B signal	1	Input
CLKR2_2X	Clock signal used to align I_{on}/Q_{on} or I_{off}/Q_{off}.	1	Input
CLKR3_2X	Clock signal used to align $I_{on}/Q_{on}/I_{off}/Q_{off}$.	1	Input
CLKR4_2X	Clock signal used to help synchronize datamux and correlators.	1	Input
MUXSEL2A	Used to select between DATA1A and DATA1B	1	Input
DATA2A	ADC stream which is 2x CLK2A's frequency	4	Input
DATA2B	ADC stream which is 2x CLK2A's frequency, 180° out of phase with DATA2A	4	Input
DATA1OUT	Synch'd with DATA2OUT; either I_{on} or I_{off}.	4	Output
DATA2OUT	Synch'd with DATA1OUT; either Q_{on} or Q_{off}.	4	Output

Implementation

The circuitry which performs the parallelization can be seen in Figure 5.11. It consists of 4 identical 4-bit wide data paths. The first thing to note is that the two interleaved 128MHz streams of 4-bit data are called xdata and ydata, instead of I and Q. This is because there is no notion of I and Q yet in the two input streams. Depending on the sampling clocks, at any point in time X0 could be considered ION, QON, IOFF, or QOFF. As an example, in Figure 5.11, we see the even XDATA samples are first considered as the ION data. However, after the clock's phase is extended, the even XDATA samples become the QOFF data. The second thing to note is that only half of the R1 clocks are active. All the R1 clocks could be continuously running, but since only

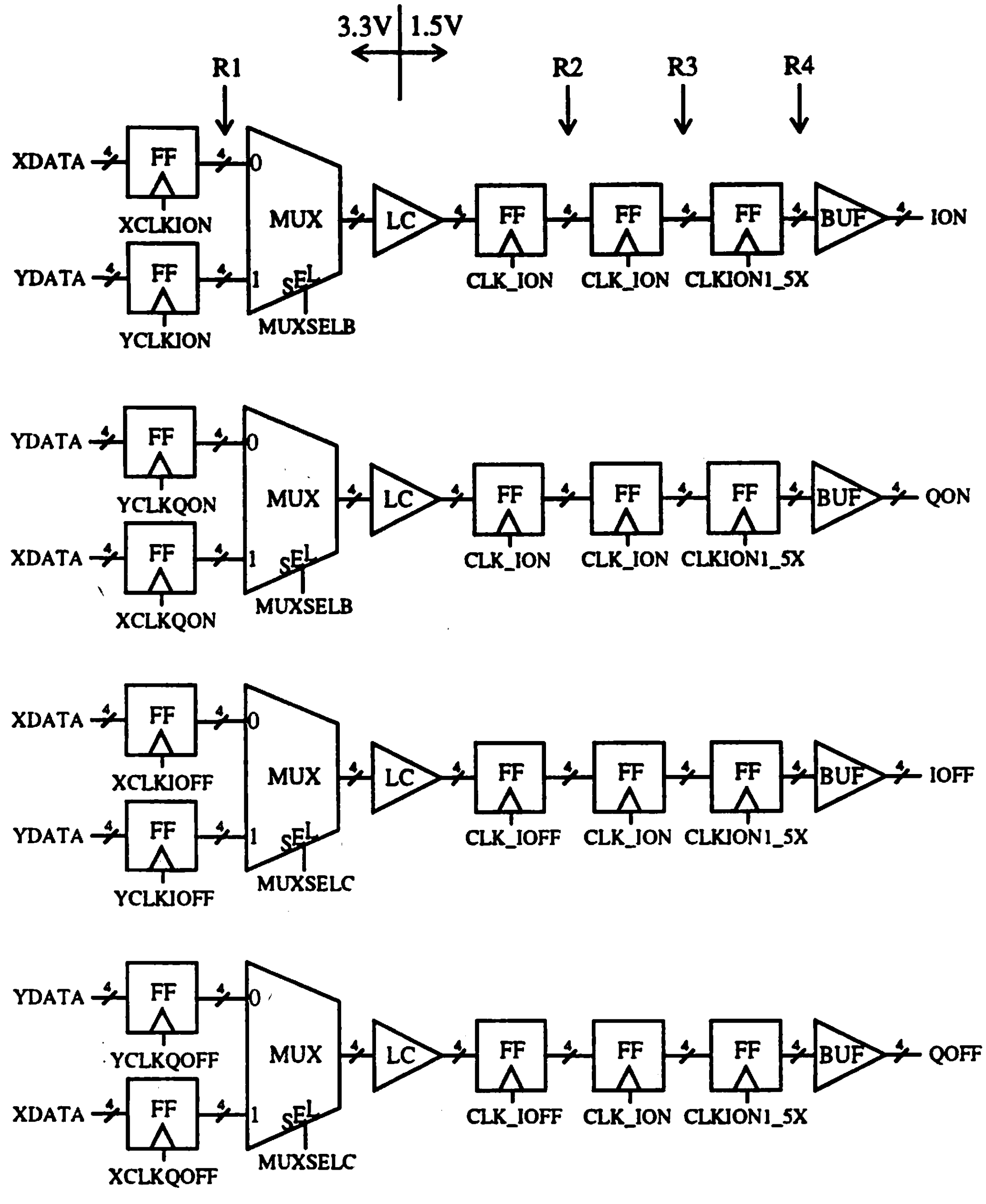


FIGURE 5.11 : Block diagram of the data multiplexor.

half of the outputs are used, power was saved by shutting off the clocks of the unused registers. However, the R2, R3, and R4 clocks can never be turned off because these registers need to be latched every cycle. Lastly, it should be noted that there are two voltages levels being used in the datapath shown in Figure 5.11. This is done because the data coming off the ADC is at 3.3V, but the correlators are running at 1.5V. The correlators could have been fed with 3.3V data, but this would mean we are driving the large capacitance wires which connect the correlators to the data block; thus, burning unnecessary power. The ION and QON outputs are fed to the data recovery correlator and the channel estimation correlators. IOFF and QOFF as well as IOFF and QOFF delayed by one clock cycle are fed to the early and late correlators used in the digital phase lock loop.

A timing diagram for the data multiplexor block can be seen in Figure 5.12. The R1 registers latch the incoming data, the R2 registers align ION with QON and IOFF with QOFF, and the R3 registers align ION, QON, IOFF and QOFF. The R4 registers are clocked by CLKION1_5X, instead of CLKION. By relatching the data with the correlators' clock, the data has more time to propagate from the data multiplexor to the correlators.

5.3 Clock Buffers

There are three main clock buffers in the demodulator. The first is a differential amplifier embedded in the pad frame which used to convert an off-chip differential sinusoidal oscillator's signal into two 50% duty cycle square-wave clock signals which are 180° out of phase. The second buffer is used to drive the control circuitry and the last is used to drive the correlators' clocks.

5.3.8 Clock Pad Buffer

The clock pad buffer converts an differential sinusoidal signal into differential 50% duty cycle square waves of equal frequency. A schematic and layout of the clock pad can be seen in Figure 5.13. The signals OSCL, OSCH, and CURRBIAS come from off-chip. OSCL and OSCH are from the sinusoidal oscillator and CURRBIAS is used to set the current down the two legs of the pre-

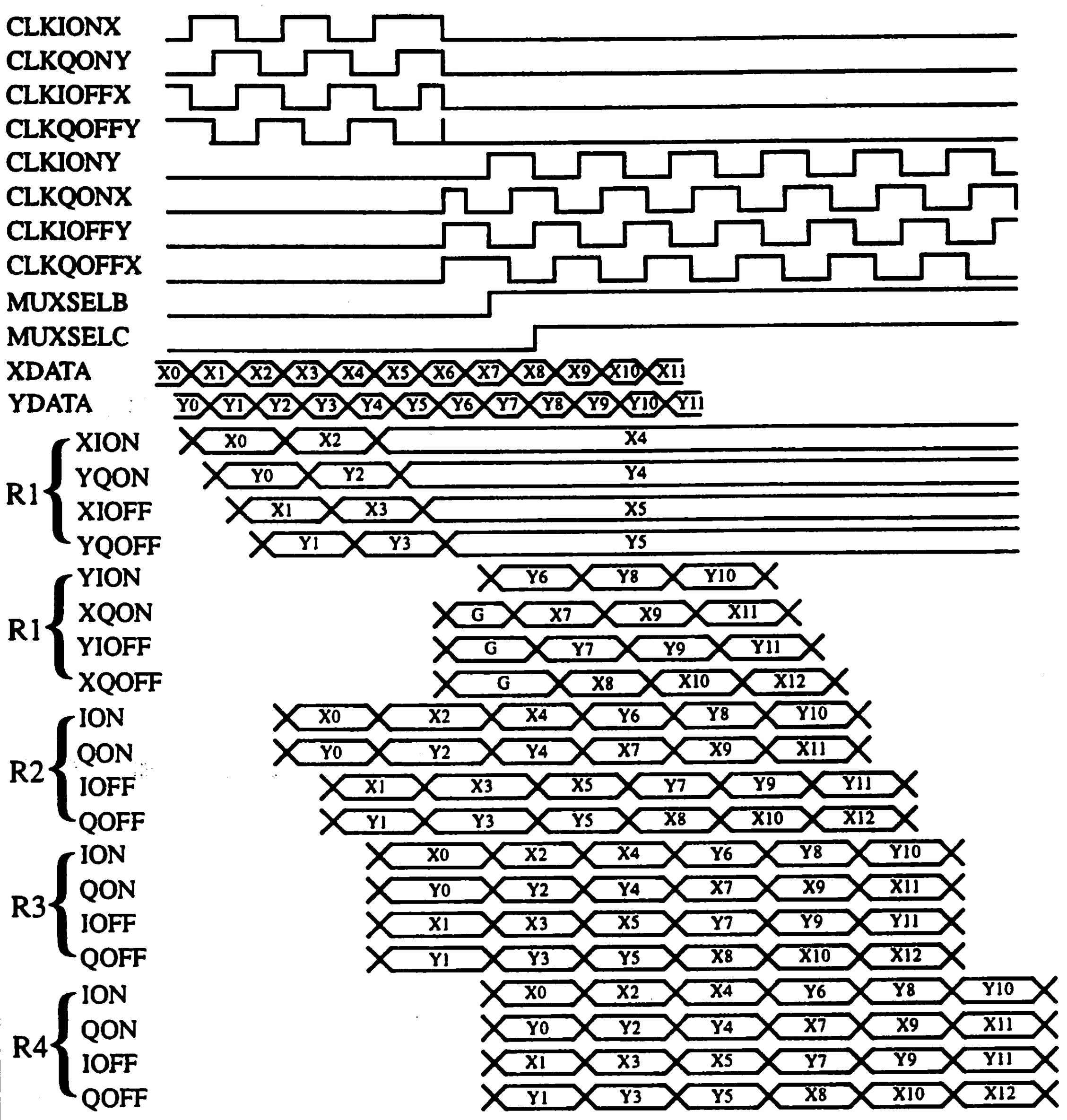


FIGURE 5.12 : Timing diagram for the data multiplexor. G= Garbage.

amplifier. CURRBIAS should be adjusted so the current down each leg is approximately 1 mA. The outputs of the clock pad buffer are CLK_L and CLK_H. One can set CURRBIAS by connecting the CURRBIAS pin to a potentiometer or an offchip precision current source. The potentiometer's other input should be connected to ground.

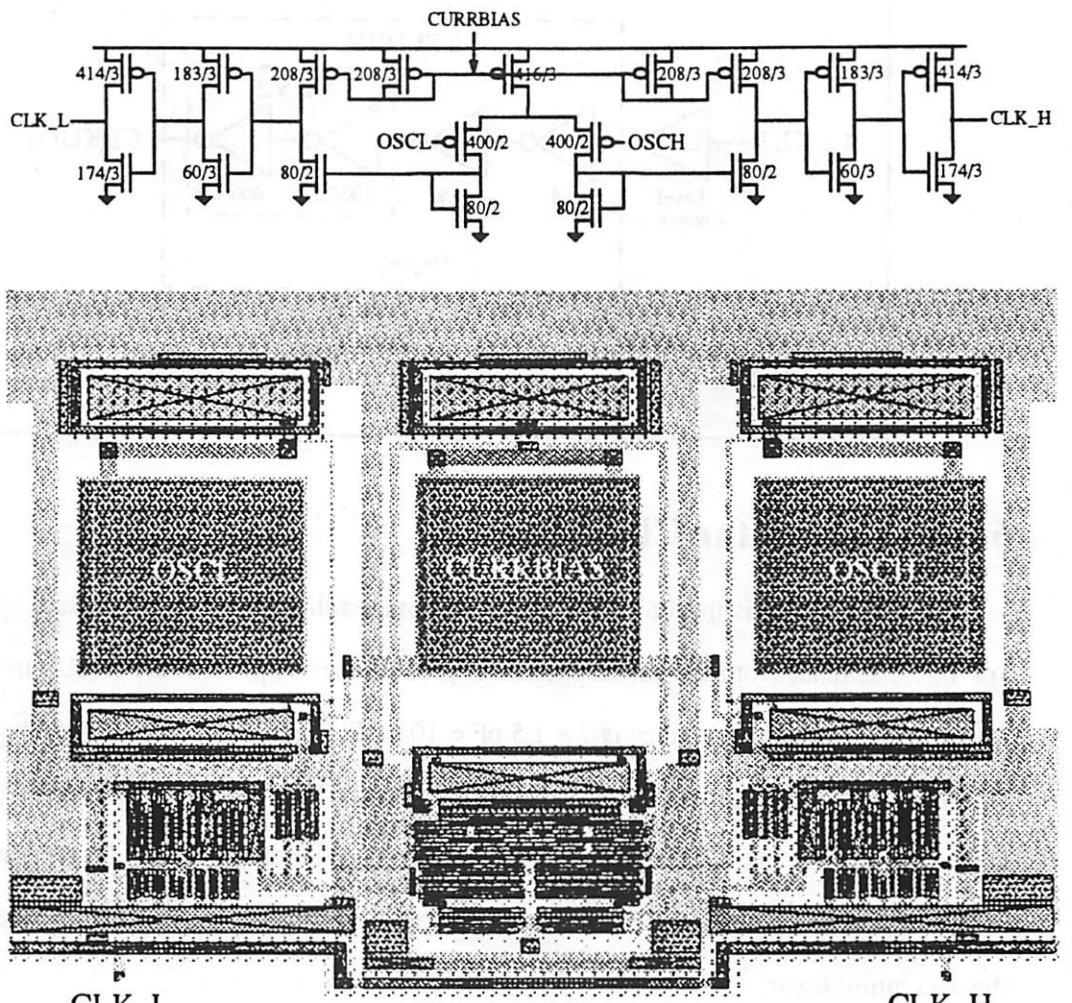


FIGURE 5.13 : Schematic and layout of differential input clock buffer. From the layout, one can see that the input clock buffer uses three pads, two for the oscillator's inputs and one for biasing the current down the preamplifer's two legs.

5.3.9 Control Logic Clock Buffer

The output of this clock buffer is used to drive the 3.3V control logic. The clock buffer has been designed to drive up to 2 pF with a 2ns worst-case rise/fall time. A high-level block diagram of the control logic clock buffer can be seen in Figure 5.14.

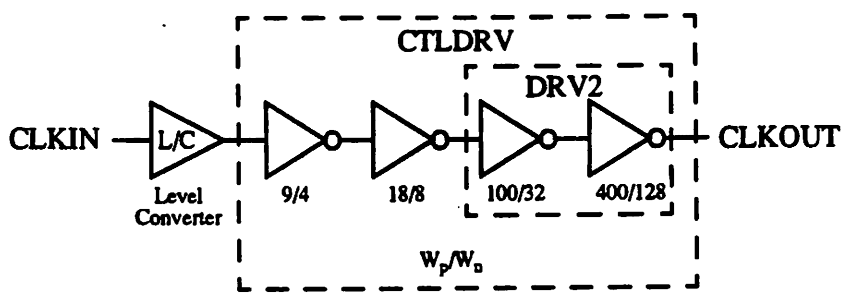


FIGURE 5.14 : Block diagram of the control logic clock buffer.

5.3.10 Correlator Clock Buffer

The clock buffer is used to drive the correlators. Each correlator's clock sees approximately 1pF of capacitance from the gate capacitances plus another 0.5pf from the clock wire capacitance. As a result the total capacitance is 7×1.5 pF = 10.5 pF. It was decided to design the clock buffer as a tree with 7 branches, one for each correlator. Thus, each branch or leg must drive approximately 1.5 pF. A block diagram of the correlator clock buffer can be seen in Figure 5.15 below. One extra branch was added to drive miscellaneous logic such as the final set of registers in the data multiplexor.

5.4 Control Logic

This section describes most of the control logic used in the demodulator. It is recommended that the reader read Section 2, Background on Spread Spectrum, before proceeding to read this section.

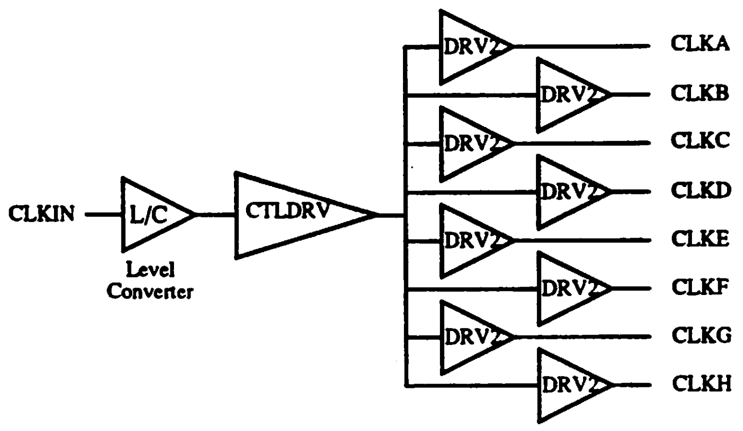


FIGURE 5.15 : Block diagram of the correlator clock buffer tree. More detailed diagrams of the CTLDRV and DRV2 bufffers can be seen in Figure 5.14.

5.4.11 Walsh Generator

Walsh codes are orthogonal codes used to differentiate an individual user's signals from other people's signals. Each user is assigned their own unique Walsh number which is loaded into the walshnum register by the Infopad. There is no default Walsh number, so upon power-up, the Infopad must load the walshnum register before the Walsh generator/data recovery unit can function correctly. During startup, Walsh number 1 is used as a control channel. After proper handshake and initialization has been achieved, the user will be assigned a Walsh number between 2 and 63. The Walsh codes are generated by using the difference between successive elements of a Gray code sequence plus the Walsh number to control a toggle flip-flop. This section describes the I/O for the Walsh unit and it's implementation.

Inputs/Outputs

Table 5.5 : I/Os for Walsh block

Signal Name	Description	Bit Width	Direction
CLK64	64MHz clock	1	Input
RESET_L	Global reset signal	1	Input
STALL_L	When 0 gates the clock which keeps the Walsh generator in the same state.	1	Input
PN_ALLONES_L	When 0 resets the Walsh generator to some initial state. Used to synchronize the Walsh generator and the PN generator	1	Input
WALSHNUM	User number-controlled by the externally loadable walshnum register.	6	Input
WALSHOUT	Current bit of the Walsh sequence corresponding to user WALSHNUM	1	Output
WALSHCNT	Current state of the walsh counter	6	Output

Implementation

The Walsh code generator followed the basic architecture shown in Section 2.4, except its datapath is 6 bits instead of 3 bits. The majority of the Walsh generator was implemented as synthesized VHDL which can be seen in Appendix B. However, in order to meet critical path timing the 6-bit counter and the nand block were designed using schematic capture. The counter was designed to exploit the fact that the msb's of the counter are stable for a long time and can thus pass through more gates before reaching the flip-flop input. Thus, the critical path only involves the countout0 signal passing through an and-gate, an xor-gate and a flip-flop. A generic version of this counter can be seen in Figure 5.16. This technique was also used to design the 11-bit counter found in the update control block. The 6-bit and 11-bit counters are slightly modified from the one shown in Figure 5.16 because reset and enable signals were added. This change, however, only added one extra gate to the critical path.

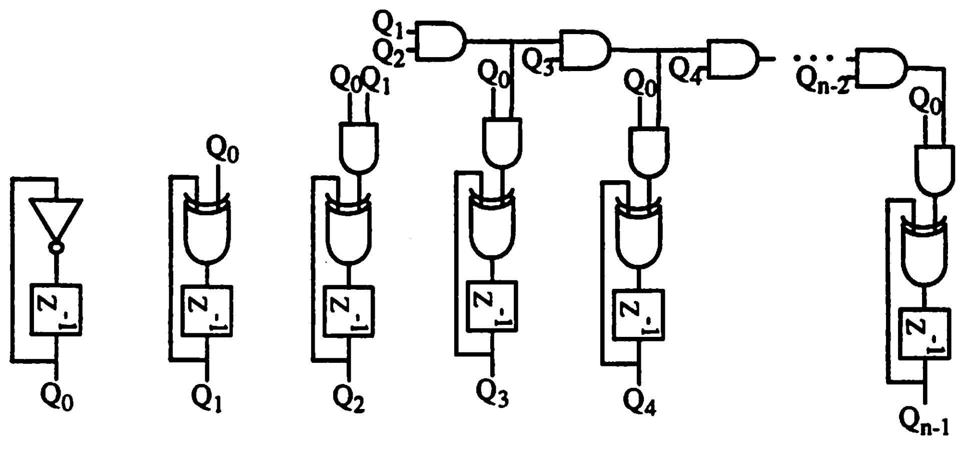


FIGURE 5.16 : Generic n-bit counter implementation designed to minimize the critical path. In this counter, the critical path is $t_{prop,ff} + t_{prop,and} + t_{prop,xor} + t_{setup,ff}$.

5.4.12 PN Generator

This block generates the PN sequence used for despreading the data. This section describes the I/Os as well as the PN generator's implementation.

Inputs/Outputs

Table 5.6 : I/Os for the PN Generator.

Signal Name	Description	Bit Width	Direction
CLK64	64MHz clock	1	Input
RESET_L	Global reset signal, loads seed into PN generator's shift register.	1	Input
STALL_L	When 0 gates the clock which keeps the PN generator in the same state. Used in lock acquisition to shift the phase of the receiver's PN seuqence with the transmitter's PN sequence.	1	Input
SEED	The initial starting point for the PN generator. Reloaded every time the PN generator wishes to begin again its 32768 chip sequence.	16	Input
PN_ALLONES_L	Indicates that the all the bits in the PN generator's shift regisister all ones. The next clock will start the PN sequence over again by loading in the SEED value. This signal is used to synchronize the Walsh generator with the PN generator.	1	Output

Table 5.6 : I/Os for the PN Generator.

Signal Name	Description	Bit Width	Direction
PN_OUT	Output of PN generator. Considered the "on-time" phase (i.e. should be aligned with the transmitter's PN sequence). Feed to correlator$_0$, DPLL correlators, and the data recovery correlator.	1	Output
PN_OUT1D	PN_OUT delayed by 1 clock cycle. Feed to correlator$_1$.	1	Output
PN_OUT2D	PN_OUT delayed by 2 clock cycles. Feed to correlator$_2$.	1	Output
PN_OUT3D	PN_OUT delayed by 3 clock cycles. Feed to correlator$_3$.	1	Output

Implementation

The system specification for the Infopad radio called for a PN sequence of length 32768 chips. The normal method is just to use the lsb of a feedback shift register as the PN sequence. However, the closest sequences in length are 32767 chips produced by a 15-bit feedback shift register and 66535 produced by a 16-bit feedback shift register (note: length = $2^n - 1$). There were two options considered: (1) Implement it as a 15-bit feedback shift register with one extra bit added at the end, or (2) implement it as a 16-bit feedback shift register, but only use 32768 of the 65536 chips before resetting the sequence. Option (2) was chosen.

The PN generator which was designed using VHDL (see Appendix B) and schematic capture is implemented as a 16-bit feedback shift register. The feedback which provides the next msb (bit 15) for the shift register is composed of the xor of bits 15, 13, 4 and 0 together. The seed was choosen so that when the 32768th bit was outputted all the taps of the shift register would be all ones. Thus, when the all ones case is detected, the seed is loaded into the shift register on the clock's next rising edge and the PN sequence starts anew. In actuality, "1111111111111110" is detected instead of all ones because there was not enough time to check for the all ones case as well as setup the seed as the next value to be loaded into the shift register in one clock cycle. Thus, "1111111111111110" is detected in one clock cycle. In the next cycle, the seed is setup to be loaded into the shift register. During reset, the seed is loaded into the shift register and the PN generator is synchronized with the Walsh generator.

5.4.13 Lock State Block

This block keeps track of whether the pad is in lock or not. It does not control the initial coarse synchronization. This is done in the update control block (Section 5.4.14).

Upon startup, the lock block is in state '00' indicating that lock has not been acquired yet. The output LOCK and LOCKRESET are both '0'. Every 1088 cycles, the VALID_DATA signal will go high indicating that C_1_L, C_2_L, C_3_L, and C_4_L are valid. The C_X_L signal indicates whether or not the accumlated energy in correlator$_X$ is greater than/equal to or less than the value stored in threshold register 1--'0' indicates greater than/equal to and '1' indicates less than. If any C_X_L signal is '0' and VALID_DATA is '1', then lock has been achieved, the new state is '01', and the LOCK signal is set to '1'.

Once in lock, the system will stay there unless the summation of the energies accumulated in the early and late correlators falls below the value stored in threshold register 3. This will be indicated when T_RESET and VALID_DATA both equal '1'. If the system does fall out of lock, then the lock status state machine will first progress to state '10' and set LOCK to '0' and LOCKRESET to '1', then return to state '00' where both LOCK and LOCKRESET are '0'.

Inputs/Outputs

Table 5.7 : I/Os for lock control block

Signal Name	Description	Bit Width	Direction
CLK64	64MHz clock	1	Input
RESET_L	Global reset signal, sets the lock statemachine in the "need to acquire lock" state.	1	Input
T_RESET	Signal comes from DPLL block. A '1' indicates that the radio fell out of lock. Determined by comparing the summation of the correlated energy in the early and late correlators with threshold register 2.	1	Input
VALID_DATA	Indicates C1_L, C2_L, C3_L, C4_L are valid.	1	Input

Table 5.7 : I/Os for lock control block

Signal Name	Description	Bit Width	Direction
C1_L	0-Accumulated energy in correlator$_0$ is greater than or equal to threshold regisiter 1. 1-Accumulated energy in correlator$_0$ is less than threshold regisiter 1.	1	Input
C2_L	0-Accumulated energy in correlator$_1$ is greater than or equal to threshold regisiter 1. 1-Accumulated energy in correlator$_1$ is less than threshold regisiter 1.	1	Input
C3_L	0-Accumulated energy in correlator$_2$ is greater than or equal to threshold regisiter 1. 1-Accumulated energy in correlator$_2$ is less than threshold regisiter 1.	1	Input
C4_L	0-Accumulated energy in correlator$_3$ is greater than or equal to threshold regisiter 1. 1-Accumulated energy in correlator$_3$ is less than threshold regisiter 1.	1	Input
LOCK	0-no lock 1-lock	1	Output
LOCKSTBITS	State bits of the lock state machine. Used for debugging purposes only.	2	Output

Implementation

The lock control block was implemented as a state machine in VHDL and then synthesized down to standard cells. Figure 5.17 shows the lock block's state transition diagram. The

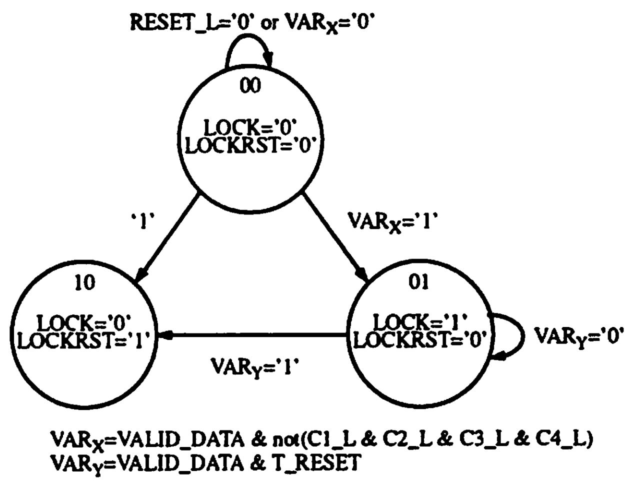

RESET_L='0' or VAR$_X$='0'
00
LOCK='0'
LOCKRST='0'
'1'
VAR$_X$='1'
10
LOCK='0'
LOCKRST='1'
01
LOCK='1'
LOCKRST='0'
VAR$_Y$='0'
VAR$_Y$='1'
VAR$_X$=VALID_DATA & not(C1_L & C2_L & C3_L & C4_L)
VAR$_Y$=VALID_DATA & T_RESET

FIGURE 5.17 : State machine for lock status block. VAR$_X$ and VAR$_Y$ have been added for legibility purposes only.

VALID_DATA signal is controlled by the update control block. It is valid approximately once every 1088 cycles.

5.4.14 Update Control

The heart of the demodulator's control is the update control block. The functionality of this block can be divided into four areas: coming out of reset, coarse lock acquisition, data recovery and channel estimation, and falling out of lock.

Inputs/Outputs

Table 5.8 : I/Os for Update Control Block

Signal Name	Description	Bit Width	Direction
CLK64	64MHz clock	1	Input
RESET_L	Global reset signal, sets the lock statemachine in the "need to acquire lock" state.	1	Input
C1_L	0-Accumulated energy in correlator$_0$ is greater than or equal to threshold regisiter 1. 1-Accumulated energy in correlator$_0$ is less than threshold regisiter 1.	1	Input
C2_L	0-Accumulated energy in correlator$_1$ is greater than or equal to threshold regisiter 1. 1-Accumulated energy in correlator$_1$ is less than threshold regisiter 1.	1	Input
C3_L	0-Accumulated energy in correlator$_2$ is greater than or equal to threshold regisiter 1. 1-Accumulated energy in correlator$_2$ is less than threshold regisiter 1.	1	Input
C4_L	0-Accumulated energy in correlator$_3$ is greater than or equal to threshold regisiter 1. 1-Accumulated energy in correlator$_3$ is less than threshold regisiter 1.	1	Input
LOCK	0-no lock 1-lock	1	Input
VALID_DATA (UPD1087)	Indicates C1_L, C2_L, C3_L, C4_L are valid. Also, tells the backend section of the correlators to latch in the new accumulated data.	1	Output
DR_RST	Clear out the contents of the data recovery correlators	1	Output
CORR_RST	clear out the contents of the carry-save adder's "dump-into" registers	1	Output
UPD63	When '1' indicates counted for 64 valid T_{chips}	1	Output
PNSTALL_L	When '0' stalls the PN and Walsh generators	1	Output

Table 5.8 : I/Os for Update Control Block

Signal Name	Description	Bit Width	Direction
UPDSTBITS	State bits of the update state machine. Used for debugging purposes only.	3	Output

Implementation

The updctrl block was written in VHDL (see Appendix XXX) and then synthesized via standard cells. It's main components are a 3-bit (8 state) state machine, an 11-bit counter, and 8 latches used to delay the control signals. The counter was designed using the same techniques as the one described in Section 5.3.X.

The minimum clock period of the control block is T_{chip} - $T_{chip}/4$ or $3T_{chip}/4$. This will happen for one clock cycle when the DPLL reduces the clock's phase by $T_{chip}/4$. Because of this strict timing requirement, the outputted control signals must be pipelined and thus are generated several clocks in advance of when they are needed. This is shown in Figure 5.18. The pipelining is problematical when one forgets that it takes several clock cycles for the signal to change, causing the old value to be around for several clocks longer than desired. For example, when coming out of reset, care had to be taken to prevent lingering values from causing errors.

After coming out of reset, updctrl enters the coarse lock aquisition mode. As presented in Section 3.1, the controller effectively counts for 1088 cycles before checking the C_X_L signals to see if lock has been acquired. The reason why updctrl counts for 1088 cycles instead of 1024 is due to the way the correlators operate (see Section 5.1). In order to prevent the constellation rotation error for becoming too great, the correlators are dumped every 64 cycles and the absolute value of their outputs are accumulated. Thus, at cycle 1024, the correlators are dumped for the 16th time. The outputs of the I and Q correlators must then be summed and compared to threshold register 1. This takes time, so the C_X_L are not ready until approximately cycle 1050. To make it easier on the control, the C_X_L signals are checked at cycle 1088 instead of around cycle 1050. At the same time the C_X_L signals are checked, the 11-bit counter is reset. If lock has been achieved, then the

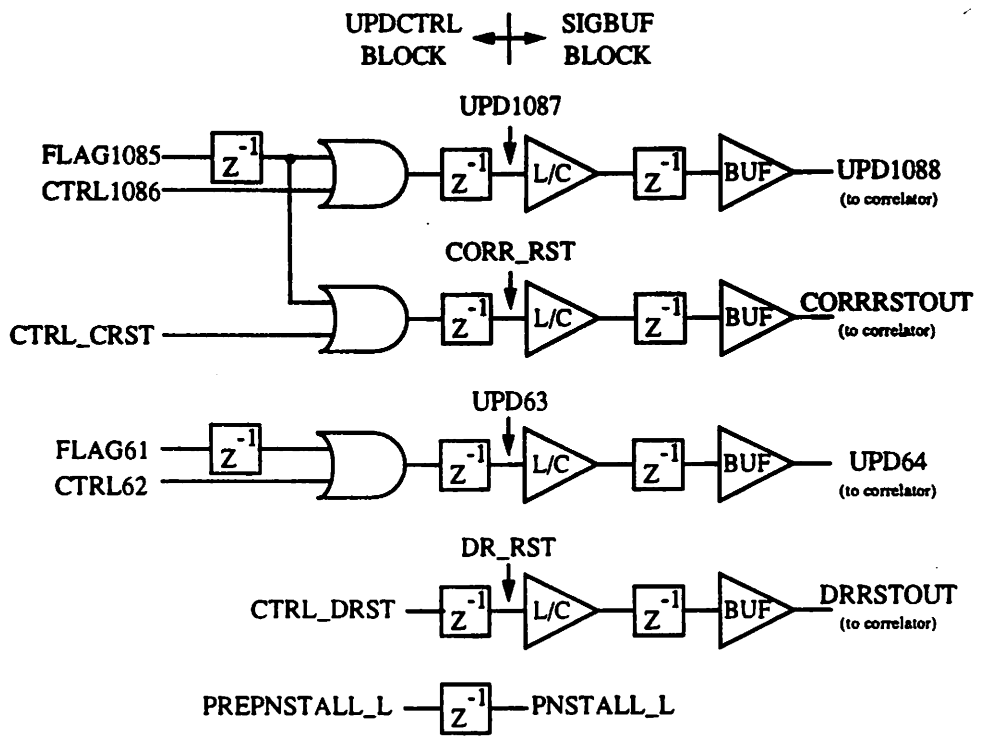


FIGURE 5.18 : Describes how the correlators' control signals are generated. The CTRL signals are outputs of the state machine and the FLAG signals are formed from the outputs of the 11-bit counter.

PN and Walsh generators are stalled the appropriate number of clock cycles (0, 1, 2 or 3) and the update control state machine enters the data acquisition state. If lock is not achieved, then the PN and Walsh generators are stalled for four clocks cycles, the correlators are cleared out, and the process repeats. In either case, the counter will have proceeded 0, 1, 2, or 3 cycles before the new data has begun to accumulate. This means that there will be up to three chips worth of invalid data incorporated into the next accumulation and only 1083 chips worth of valid data. This was necessary because the counter could only be reset on multiples of 64 in order to stay in synchronization with the PN and Walsh generators. During the lock mode, updctrl must generate control signals to update the correlators every 64 cycles as well as every 1088 cycles.

CHAPTER 6

Conclusion

6.1 Summary

This report describes the overall architecture for the Infopad's CDMA radio's demodulator chip as well as details the implemention of the first version. The first version was an 80K transistor chip fabricated in HP's psuedo 0.8μm process. It occupied 56.56 mm^2 (7.69 mm × 7.36 mm) and was packaged in a large cavity 132 PGA. It implemented all the functions described in the architecture description, except for the DQPSK decoder and the adjacent cell scan circuitry. A die photo of the chip can be seen in Figure 6-1. As of the date of this thesis' filing, the first version demodulator chip was only functionally tested using Tektronix's DAS9200 system. It was found that all functions performed (mostly) correctly at low speeds (up to 25 MHz).

6.2 Future Work

There is still plenty of work left to be done. There were two blocks which were not implemented in the first version of the demodulator chip: the DQPSK decoder and the adjacent

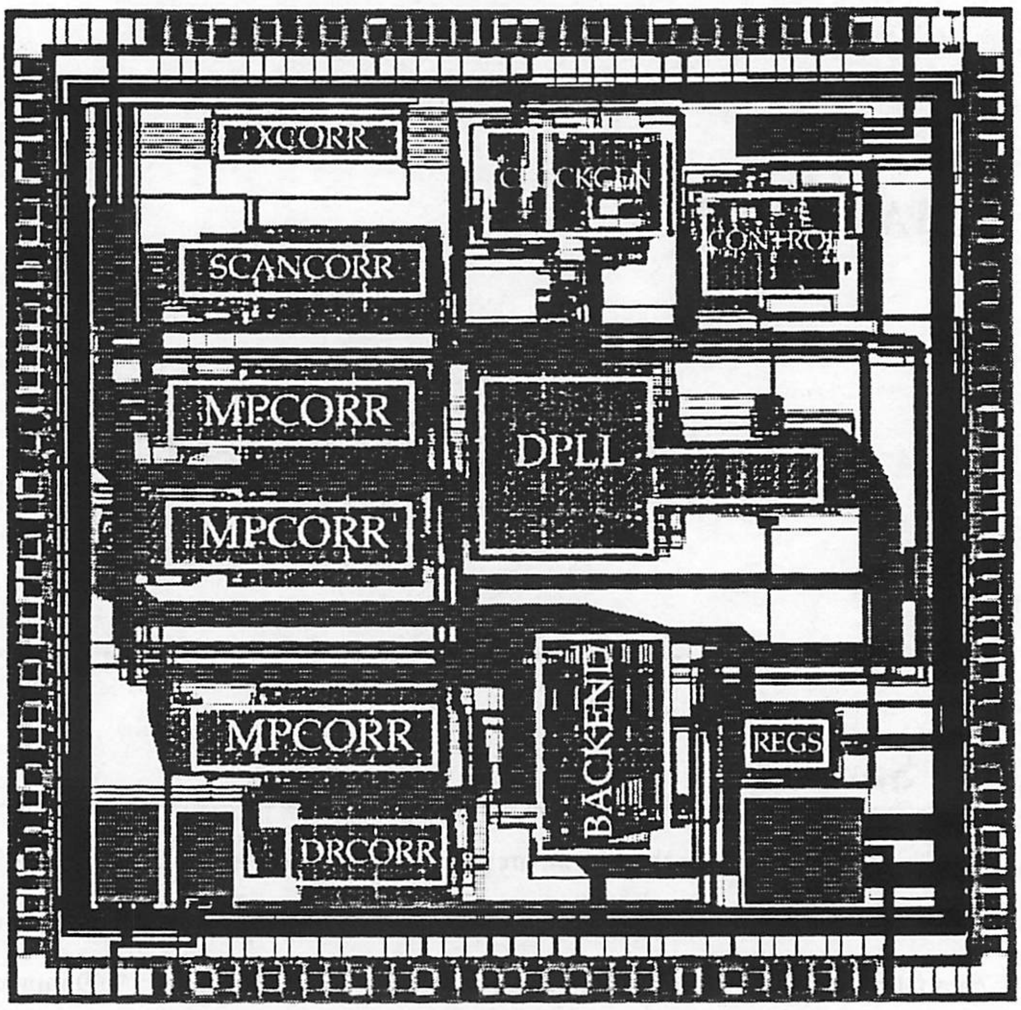


Figure 6-1 : Micrograph of the spread spectrum demodulator chip.

cell scan and handoff circuitry. The DQPSK decoder can be implemented either as another on-chip block or since it is running at 1 MHz, as a separate chip. The adjacent cell scan and handoff circuitry can be realized using an extra correlator with some different backend logic and another PN generator.

The demodulator chip was implemented as a stand-alone chip. However, the since final implemention of the Infopad's radio receiver was envisaged as a single chip solution (i.e. both the analog and digital sections integrated onto a single die), someone needs to look into the effect of having digital and analog circuitry on the same die to determine whether a single chip solution is practical and/or feasible versus something like and MCM or chip-on-board solution. The current version of the demodulator was designed to perform hard handoff. Looking at how to do a soft handoff can also be explored.

Techniques are being explored by Teuscher which focus on reducing the SNR through equalization and interference cancellation techniques. If some type of combining strategy is implemented, then the long correlations must be modified to handle the phase correction while maintaining the phase information.

At the higher level, what type of numbers can/should be provided to the network folks (i.e. SNR, RSSI, BER estimates) can be explored. Currently, only RSSI numbers are provided.

BIBLIOGRAPHY

[Afgh91] M. Afghahi, J. Yuan, "Double Edge-Triggered D-Flip-Flops for High-Speed CMOS Circuits", IEEE Journal of Solid-State Circuits, Vol. 26, No. 8, August 1991.

[Beau84] K. G. Beauchamp, Applications of Walsh and Related Functions, Orlando, Academic Press, 1984.

[Buck93] J. Buck, "The Ptolemy Kernel: A Programmer's Companion for Ptolemy 0.4", UCB/ERL Memorandum No. M93/8, January 19, 1993.

[Burd94] T. Burd, Low-power Cell Library, M.S. Thesis, U.C. Berkeley, June 1994.

[Chand94] A. Chandrakasan, "Low Power Digital CMOS Design", Ph.D. Thesis, Memorandum No. UCB/ERL M94/65, August 30, 1994.

[Le95] M. Le, S. Seshan, F. Burghardt, J. Rabaey, "Software Architecture of the Infopad System", UC Berkeley EECS Department, January 19, 1995.

[Proak89] J. Proakis, Digital Communications, USA: McGraw-Hill, 1989.

[Seid91] S.Y. Seidel, T.S. Rappaport, "914 MHz Path-Loss Prediction Models for Indoor Wireless Communications," submitted to IEEE Trans. Antennas and Propagation, May 17, 1991.

[Sheng91] S. Sheng, Wideband Digital Portable Communications: A System Design, M.S. Thesis, Memorandum No. UCB/ERL M91/108, December 3, 1991.

[Sheng92] S. Sheng, A. P. Chandrakasan, R. W. Brodersen, "A Portable Multimedia Terminal", IEEE Communications Magazine, pp. 64-75, December 1992.

[Sheng94] S. Sheng, R. Allmon, L. Lynn, I. O'Donnell, K. Stone, R. Brodersen, "A Monolithic CMOS Radio System for Wideband CDMA Communications", Proceedings to Wireless '94 Conf., June 1994.

[Sheng95] S. Sheng, Wideband Digital Portable Communications, Ph.D. Thesis, U.C. Berkeley, to be published in 1995.

[Strat94] A. Stratakos, S. Sanders, R. Brodersen, "A Low-voltage CMOS DC-DC Converter for a Portable Low-Powered Battery-Operated System," PESC 1994.

[Teus94] C. Teuscher, Software Simulation for the Infopad Downlink, M.S. Thesis, U.C. Berkeley, December 1994.

[Walsh73] Theory and applications of Walsh functions and other non-sinusoidal functions, Proceedings of a colloquium held on June 28th and 29th at the Hatfield Polytechnic.

[Yun94] L. Yun, D. Messerschmitt, "Power Control for Variable QOS on a CDMA Channel", Proc. IEEE MILCOM, Fort Monmouth, NJ, Oct 2-5, 1994, pp. 178-182.

Appendix A-Schematics

The actual schematics are located in the demodulator's schematic directory. They can be viewed using Viewlogic's viewdraw schematic capture program. The following is a list of schematics with their corresponding hierarchy:

corrpad - contains core of chip along with pad ring (schematic can be seen below)

- **corrcore** - contains all the blocks of the chip (schematic can be seen below)
  - **mpcorr** - multipath correlator with backend logic
    - **i_basecorr** - actual 13-bit correlator block
    - **mpback** - compares |I| + |Q| against thresreg1; outputs Q and either I or IPLUSQ depending on whether lock has been acquired or not.
    - **xbuffer** - dpp buffer of size 'H' which is used to drive output signals
  - **t1t2corr** - early and late correlators with backend logic
    - **t1t2base** - contains early and late I/Q correlator pairs
      - **i_basecorr** - actual 13-bit correlator block
    - **t1t2back** - compares $[(|I| + |Q|)_{early} + (|I| + |Q|)_{late}]$ with $thresreg_b$ to see if energy energy is being received to stay in lock; compares $[(|I| + |Q|)_{early} - (|I| + |Q|)_{late}]$ with $thresreg_c$ to see if the clock's phase lock loop should be adjusted, and if so, whether it should be extended or shrunk.

t1t2ss - converts 15-bit sign-magnitude number to 15-bit one's complement number

drcorr - data recovery correlator with backend logic

i_frontend - actual 9-bit correlator block

drback - latches and buffers output of I/Q correlators

extracorr - extra correlator used for debugging and power measurements

i_basecorr - actual 13-bit correlator block

regblk - registers with control logic

regbufs - registers and buffers

regctrl - control logic for registers

datablock - generates 64MHz clock; converts two 128MHz interleaved data streams to four parallel 64MHz streams as well as adjusts data depending on which testmode is selected.

testmode - adjusts data depending on which testmode is selected

bin2sm - converts binary number (0 to 15) to sign-magnitude

two2smx - converts two's complement number to sign-magnitude

xmtr_int - interface to transmitter chip. Converts four parallel streams to 2 interleaved streams at twice the input frequency

muxbuf - selects which type of data to send to the rest of the chip depending on the testmode bits

clkmux - generates 64MHz clock; converts two 128MHz interleaved data streams to four parallel 64MHz streams

clkmux_nodetff - generates 64 MHz clock which can be adjusted by $\pm T_{chip}/4$; generates control signals for clkmux_datamux

clkmux_detff - bank of dual-edge-triggered flip-flops which generate four phases of the 64MHz clock from a single 64MHz input clock.

clkmux_bufs - buffers for the clock signals

clkmux_datamux - converts 2 interleaved 128MHz data streams into 4 parallel 64MHz streams

clkmux_delay - delay element and buffer for data (used to create I_{off} and Q_{off}.

clkcntrldrv - buffers clock for control logic

lvlcvt - level converter

- **clk3_3driver** - clock buffer for control circuitry
- **clkcorrdrv** - buffers clock for correlators
  - **lvlcvt** - level converter
  - **clk1_5driver** - generates 8 clocks to drive 7 correlators + miscellaneous logic

ctrlblk - control circuitry

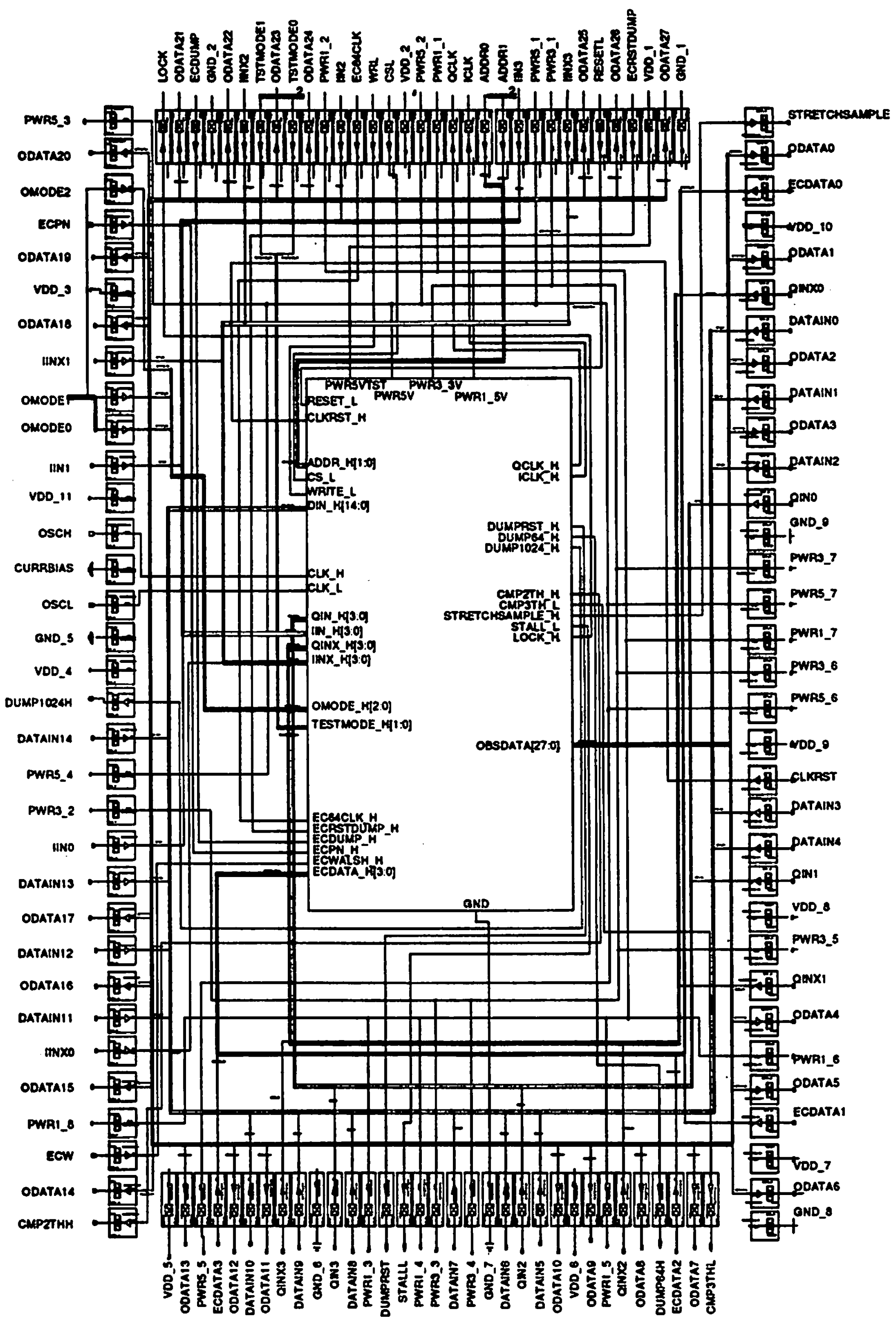
- **walsh** - generates Walsh sequence according to Walsh user number
- **pn** - generates PN sequence
- **lock** - lock control circuitry
- **updctrl** - generates update control signals for the correlators
- **sigbufs** - level converts, latches, and buffers signals
- **upddly** - latches and buffers signals
- **clk8mhzgen** - latches bit2 of walshcnt to generate 8Mhz clock
- **rstbufx** - latches and buffers signal
- **xbuffer** - buffers signals

osb - observation block which is used to select which outputs to view on the odata bus

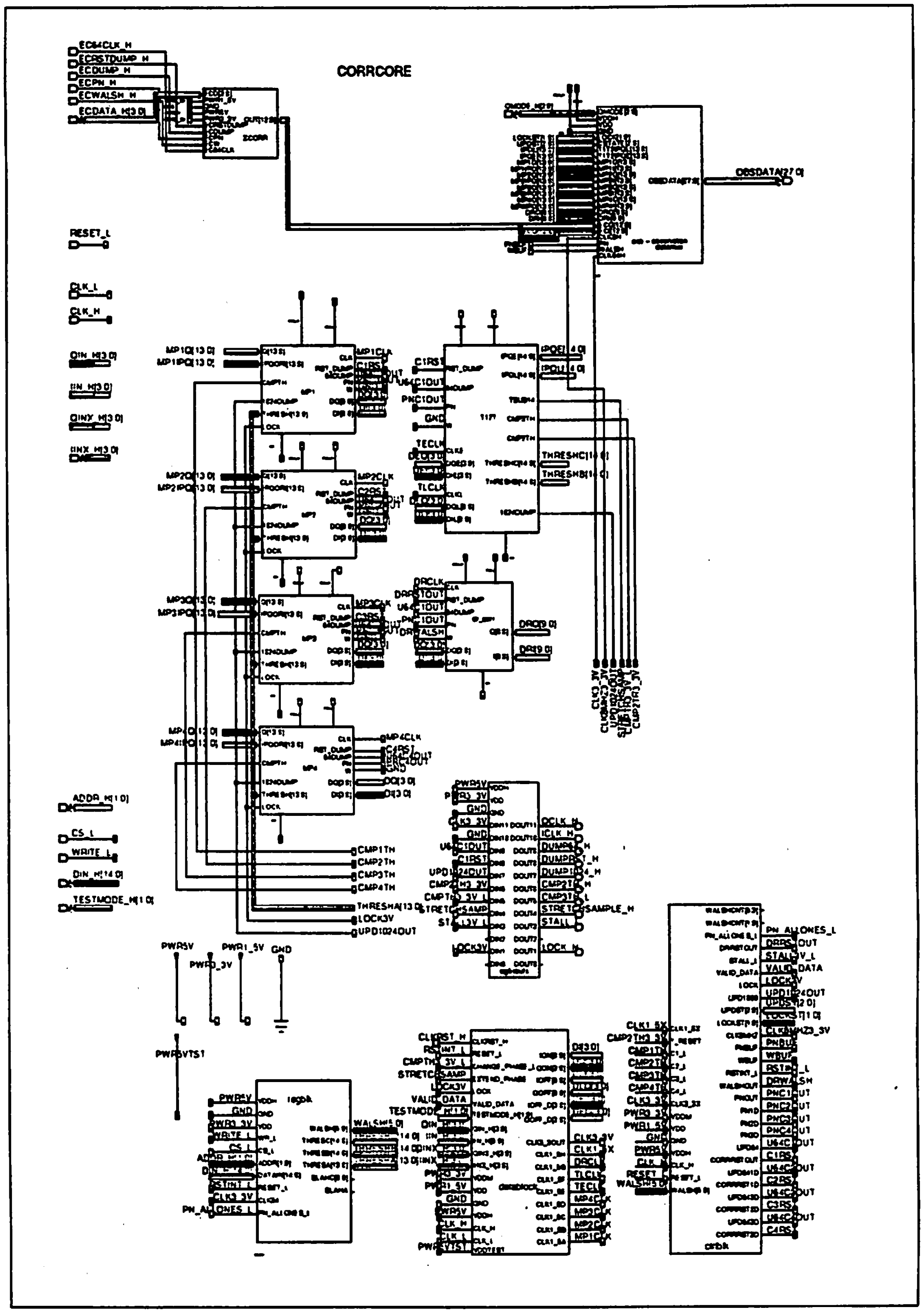
- **osbscan** - converts 1->8 serial to parallel converter
- **osbss** - generates grounds which are used for some of the msbs of the inputs to osbback
- **osbback** - 28-bit wide 8:1 mux

Two example schematics are shown on the following two pages: corpad and corrcore.

corrpad


LOCK
ODATA21
ECDUMP
GND_2
ODATA22
IINX2
TSTMODE1
ODATA23
TSTMODE0
ODATA24
PWR1_2
IIN2
EC64CLK
WRL
CSL
VDD_2
PWR5_2
PWR1_1
OCLK
ICLK
ADDR0
ADDR1
IIN3
PWR5_1
PWR3_1
IINX3
ODATA25
RESETL
ODATA26
ECRSTDUMP
VDD_1
ODATA27
GND_1
PWR5_3
ODATA20
OMODE2
ECPN
ODATA19
VDD_3
ODATA18
IINX1
OMODE1
OMODE0
IIN1
VDD_11
OSCH
CURRBIAS
OSCL
GND_5
VDD_4
DUMP1024H
DATAIN14
PWR5_4
PWR3_2
IIN0
DATAIN13
ODATA17
DATAIN12
ODATA16
DATAIN11
IINX0
ODATA15
PWR1_8
ECW
ODATA14
CMP2THH
STRETCHSAMPLE
ODATA0
ECDATA0
VDD_10
ODATA1
QINX0
DATAIN0
ODATA2
DATAIN1
ODATA3
DATAIN2
QIN0
GND_9
PWR3_7
PWR5_7
PWR1_7
PWR3_6
PWR5_6
VDD_9
CLKRST
DATAIN3
DATAIN4
QIN1
VDD_8
PWR3_5
QINX1
ODATA4
PWR1_6
ODATA5
ECDATA1
VDD_7
ODATA6
GND_8
VDD_5
ODATA13
PWR5_5
ECDATA3
ODATA12
DATAIN10
ODATA11
QINX3
DATAIN9
GND_6
QIN3
DATAIN8
PWR1_3
DUMPRST
STALLL
PWR1_4
PWR3_3
DATAIN7
PWR3_4
GND_7
DATAIN6
QIN2
DATAIN5
ODATA10
VDD_6
ODATA9
PWR1_5
QINX2
ODATA8
DUMP64H
ECDATA2
ODATA7
CMP3THL
PWR5VTST
PWR3_3V
PWR5V
PWR1_5V
RESET_L
CLKRST_H
ADDR_H[1:0]
CS_L
WRITE_L
DIN_H[14:0]
CLK_H
CLK_L
QIN_H[3:0]
IIN_H[3:0]
QINX_H[3:0]
IINX_H[3:0]
OMODE_H[2:0]
TESTMODE_H[1:0]
EC64CLK_H
ECRSTDUMP_H
ECDUMP_H
ECPN_H
ECWALSH_H
ECDATA_H[3:0]
GND
OCLK_H
ICLK_H
DUMPRST_H
DUMP64_H
DUMP1024_H
CMP2TH_H
CMP3TH_L
STRETCHSAMPLE_H
STALL_L
LOCK_H
OBSDATA[27:0]

corrcore


CORRCORE
ECDATA_H[3:0]
RESET_L
CLK_L
CLK_H
OBSDATA[27:0]
CMP1TH
CMP2TH
CMP3TH
CMP4TH
LOCK3V
UPD1024OUT
PWR5V
PWR1_5V
PWR3_3V
GND
PWR5VTS1
ADDR_H[1:0]
CS_L
WRITE_L
DIN_H[14:0]
TESTMODE_H[1:0]
regblk
ctrlblk

Appendix B-VHDL

List of VHDL files

```
--------------------------------------------------------------

-- Cell:        i_basecorr
-- Description: base block for correlator
-- Author:      Ian O'Donnell
-- Date:
-- Modified: 8/1/94-Modified Kevin Stone's previous base_dp.vhd to this.
-- Note:  Does not model hardware, only functionality at
-- input and output.  Internal nodes do not correspond.
-- Do not synthesize this and then expect it to work.
----------------------------------------------------------------------


entity i_basecorr is
  port (
    clock :         in  vlbit;
    rst_dump :      in  vlbit;
    dump :          in  vlbit;
    pn :            in  vlbit;
    walsh :         in  vlbit;
    datain :        in  vlbit_vector(3 downto 0);
    vdd :           in  vlbit;
    gnd :           in  vlbit;
    dataout :       out vlbit_vector(12 downto 0)
  );
end i_basecorr;

architecture behavior of i_basecorr is

  signal rst_dump1d, dump1d, pn1d, walsh1d : vlbit;
  signal data1d : vlbit_vector(3 downto 0);
  signal rst_dump2d, dump2d : vlbit;
  signal data2d, data3d : vlbit_vector(3 downto 0);
  signal datasum, dataacc, dumpdata : vlbit_vector(9 downto 0);
  signal dumpsum, dumpacc : vlbit_vector(12 downto 0);

begin
  process(clock)
    variable dumpsum_temp : vlbit_vector(13 downto 0);
    variable all0 : vlbit_vector(12 downto 0);
    variable datasum_temp, dump_temp : vlbit_vector(10 downto 0);
    variable datasum0 : vlbit_vector(9 downto 0);
    variable data2d_temp : vlbit_vector(3 downto 0);
  begin
    assert (vdd='1' and gnd='0')
       report "Vdd and GND not connected properly!"
       severity Failure;

    all0 := "0000000000000";
    datasum0 := "0000000000";

    if prising(clock) then
       if dump2d='1' then
          if rst_dump2d='1' then
             dumpacc <= all0(12 downto 0);
             dumpdata <= all0(9 downto 0);
     else
             dumpacc <= dumpsum(12 downto 0);
            if datasum(9)='1' then
              dump_temp := sub2c(datasum0, datasum);
                else
                   dump_temp := add2c(datasum0, datasum);
                end if;
             dumpdata <= dump_temp(9 downto 0);
          end if;
```

```
           dataacc <= all0(9 downto 0);
        else
           dataacc <= datasum( 9 downto 0);
        end if;

        rst_dump2d <= rst_dump1d;
        dump2d <= dump1d;
        data3d <= data2d(3 downto 0);
        data2d(3) <= (pn1d xor walsh1d xor data1d(3));
        data2d(2 downto 0) <= data1d(2 downto 0);
        rst_dump1d <= rst_dump;
        dump1d <= dump;
        pn1d <= pn;
        walsh1d <= walsh;
        data1d <= datain(3 downto 0);
     end if;

     if pfalling(clock) then
        dumpsum_temp := add2c(dumpacc, dumpdata);
        dumpsum <= dumpsum_temp(12 downto 0);
        dataout <= dumpsum_temp(12 downto 0);

        if data3d(3)='1' then
------------ This step is necessary to convert the input data (which
------------ is signed mag.) to 2's complement (which is how this vhdl
------------ model keeps the sum).
           data2d_temp(3) := all0(0);
           data2d_temp(2 downto 0) := data3d(2 downto 0);
------------ If the sign bit is high, we subtract the magnitude from
------------the current sum.
           datasum_temp := sub2c(dataacc, data2d_temp);
           datasum <= datasum_temp(9 downto 0);
        else
           datasum_temp := add2c(dataacc, data3d);
           datasum <= datasum_temp(9 downto 0);
        end if;
     end if;

   end process;

end behavior;


-------------------------------------------------------------------------
-- Cell:        i_frontend
-- Description: front end block of correlator
-- Author:      Ian O'Donnell
-- Date:
-- Modified: 8/1/94-Modified Kevin Stone's previous base_dp.vhd to this.
-- Note:  Does not model hardware, only functionality at
-- input and output.  Internal nodes do not correspond.
-- Do not synthesize this and then expect it to work.
-------------------------------------------------------------------------

entity i_frontend is
  port (
    clock :          in  vlbit;
    rst_dump :       in  vlbit;
    dump :           in  vlbit;
    pn :             in  vlbit;
    walsh :          in  vlbit;
    datain :         in  vlbit_vector(3 downto 0);
    vdd :            in  vlbit;
    gnd :            in  vlbit;
    bufdump :        out vlbit;
    datasign :       out vlbit;
    dataout :        out vlbit_vector(8 downto 0)
  );
end i_frontend;

architecture behavior of i_frontend is

  signal rst_dump1d, dump1d, pn1d, walsh1d : vlbit;
  signal data1d : vlbit_vector(3 downto 0);
  signal rst_dump2d, dump2d : vlbit;
  signal data2d, data3d : vlbit_vector(3 downto 0);
  signal datasum, dataacc : vlbit_vector(9 downto 0);

begin
  process(clock)
    variable all0 : vlbit_vector(12 downto 0);
    variable datasum_temp, dump_temp : vlbit_vector(10 downto 0);
    variable datasign_temp : vlbit;
    variable datasum0 : vlbit_vector(9 downto 0);
    variable data2d_temp : vlbit_vector(3 downto 0);
  begin
    assert (vdd='1' and gnd='0')
       report "Vdd and GND not connected properly!"
       severity Failure;

    all0 := "0000000000000";
    datasum0 := "0000000000";

    if prising(clock) then
       if dump2d='1' then
          if rst_dump2d='1' then
             dataout <= all0(8 downto 0);
             datasign <= '0';
     else
             if datasum(9)='1' then
           dump_temp := sub2c(datasum0, datasum);
                datasign_temp := '1';
             else
                dump_temp := add2c(datasum0, datasum);
                datasign_temp := '0';
             end if;
             dataout <= dump_temp(8 downto 0);
             datasign <= datasign_temp;
          end if;
          dataacc <= all0(9 downto 0);
       else
          dataacc <= datasum( 9 downto 0);
```

```
        end if;

        rst_dump2d <= rst_dump1d;
        dump2d <= dump1d;
        data3d <= data2d(3 downto 0);
        data2d(3) <= (pn1d xor walsh1d xor data1d(3));
        data2d(2 downto 0) <= data1d(2 downto 0);
        rst_dump1d <= rst_dump;
        dump1d <= dump;
        pn1d <= pn;
        walsh1d <= walsh;
        data1d <= datain(3 downto 0);
     end if;

     if pfalling(clock) then
        bufdump <= dump2d;
        if data3d(3)='1' then
------------ This step is necessary to convert the input data (which
------------ is signed mag.) to 2's complement (which is how this vhdl
------------ model keeps the sum).
           data2d_temp(3) := all0(0);
           data2d_temp(2 downto 0) := data3d(2 downto 0);
------------ If the sign bit is high, we subtract the magnitude from
-------------the current sum.
           datasum_temp := sub2c(dataacc, data2d_temp);
           datasum <= datasum_temp(9 downto 0);
        else
           datasum_temp := add2c(dataacc, data3d);
           datasum <= datasum_temp(9 downto 0);
        end if;
     end if;

   end process;

end behavior;

------------------------------------------------------------------------
--  Cell:        walsh
--  Description: walsh generator
--  Author:      Kevin Stone
--  Date:        3/24/94
--
--  NOTE:  FOR CORRECT SYNTHESIS MUST SYNTHESIZE WALSH BEFORE WALSH_NAND.
--          Counter was hand designed -> do not synthesize
--  6/20/94: Changed counter from asynch reset to synch reset
------------------------------------------------------------------------


entity walsh is
  port (
    reset_l :        in  vlbit;
    pn_allones_l :   in  vlbit;
    stall_l :        in  vlbit;
    walshnum :       in  vlbit_vector(5 downto 0);
    pnsynch :        in  vlbit;
    clk64 :          in  vlbit;
    walshout :        out vlbit;
    walshcount :      out vlbit_vector(5 downto 0)
  );
end walsh;

architecture behavior of walsh is

component count64sync
  port (
    clk :             in vlbit;
    rst :             in vlbit;
    cnt :             in vlbit;
    countout :        out vlbit_vector (5 downto 0)
  );
end component;

component walsh_nand
  port (
    walshnum :        in  vlbit_vector(5 downto 0);
    walcnt_int :      in  vlbit_vector(5 downto 0);
    walcnt_d1 :       in  vlbit_vector(5 downto 0);
    nandout :        out vlbit
  );
end component;

  signal walcnt_int : vlbit_vector(5 downto 0);
  signal walcnt_d1_temp : vlbit_vector(5 downto 0);
  signal walcnt_d1 : vlbit_vector(5 downto 0);
  signal tempnand6 : vlbit;
  signal walshout_int : vlbit;
  signal walshout_temp : vlbit;
  signal walcnt_rst : vlbit;
  signal walsh_rst : vlbit;
  signal stall_l_b : vlbit;
  signal reset_l_b : vlbit;
  signal pn_allones_l_b : vlbit;

begin

walcounter : count64sync -- synchronous reset, no load
  port map(
    clk => clk64,
    rst => walcnt_rst,
    cnt => stall_l,
    countout => walcnt_int
  );

walnand : walsh_nand
  port map(
    walshnum => walshnum,
    walcnt_int => walcnt_int,
    walcnt_d1 => walcnt_d1,
    nandout => tempnand6
  );

  walcnt_rst <= reset_l_b or pn_allones_l_b or pnsynch;
  pn_allones_l_b <= not(pn_allones_l);
  reset_l_b <= not(reset_l);
  stall_l_b <= not(stall_l);
  walshout <= walshout_int;
  walshcount <= walcnt_int;

  process(reset_l, pn_allones_l, stall_l, walcnt_int, walshout_int,
          tempnand6, walcnt_d1, walsh_rst)
    variable all0 : vlbit_vector(5 downto 0);
  begin
    all0 := "000000";
    if ((walcnt_int = all0) or reset_l='0' or pn_allones_l='0') then
        walsh_rst<='0';
    else
        walsh_rst<='1';
    end if;

    if walsh_rst='0' then
        walshout_temp <= '0';
    elsif stall_l='0' then
        walshout_temp <= walshout_int;
    elsif (tempnand6 = '1') then -- flip bit
        walshout_temp <= not (walshout_int);
    else
        walshout_temp <= walshout_int;
    end if;

    if (reset_l='0' or pn_allones_l='0') then
        walcnt_d1_temp <= all0;
    elsif stall_l='0' then
        walcnt_d1_temp <= walcnt_d1;
    else
        walcnt_d1_temp <= not(walcnt_int);
    end if;

  end process;

  latch_proc1 : process
  begin
    wait until prising(clk64);
    walshout_int <= walshout_temp;
  end process latch_proc1;

  latch_proc2 : process
  begin
    wait until prising(clk64);
    walcnt_d1 <= walcnt_d1_temp;
  end process latch_proc2;

end behavior;

------------------------------------------------------------------------
--  Cell:         walsh_nand
--  Description:
--  Author:       Kevin Stone
--  Date:         6/12/94
```

```
--
--
-----------------------------------------------------------------------


entity walsh_nand is
  port (
    walshnum :      in  vlbit_vector(5 downto 0);
    walcnt_int :    in  vlbit_vector(5 downto 0);
    walcnt_d1 :     in  vlbit_vector(5 downto 0);
    nandout :      out vlbit
  );
end walsh_nand;

architecture behavior of walsh_nand is

  signal tempnand0 : vlbit;
  signal tempnand1 : vlbit;
  signal tempnand2 : vlbit;
  signal tempnand3 : vlbit;
  signal tempnand4 : vlbit;
  signal tempnand5 : vlbit;

begin

  tempnand0 <= not (walshnum(0) and walcnt_int(0) and walcnt_d1(0));
  tempnand1 <= not (walshnum(1) and walcnt_int(1) and walcnt_d1(1));
  tempnand2 <= not (walshnum(2) and walcnt_int(2) and walcnt_d1(2));
  tempnand3 <= not (walshnum(3) and walcnt_int(3) and walcnt_d1(3));
  tempnand4 <= not (walshnum(4) and walcnt_int(4) and walcnt_d1(4));
  tempnand5 <= not (walshnum(5) and walcnt_int(5) and walcnt_d1(5));
  nandout <= not (tempnand0 and tempnand1 and tempnand2 and
                  tempnand3 and tempnand4 and tempnand5);

end behavior;

-----------------------------------------------------------------------
--  Cell:        count64sync
--  Description: sync rst, no load
--  Author:      Kevin Stone
--  Date:        6/20/94
--
--
-----------------------------------------------------------------------

--library synth;
--use synth.stdsynth.all;

entity count64sync is
  port (
    clk :           in vlbit;
    rst :           in vlbit;
    cnt :           in vlbit;
    countout :      out vlbit_vector (5 downto 0)
  );
end count64sync;
```

```
architecture behavior of count64sync is
  signal countout_int : vlbit_vector(5 downto 0);
  signal count_temp : vlbit_vector(5 downto 0);
  constant all0 : vlbit_vector(5 downto 0)
    := "000000";
  constant one : vlbit_vector(5 downto 0)
    := "000001";
begin

  count_proc : process(cnt, count_temp, countout_int, rst)
    variable count_temp2 : vlbit_vector(6 downto 0);
  begin
    count_temp2 := addum(countout_int, one);

    if rst='1' then
        count_temp <= all0;
    elsif cnt='1' then
        count_temp <= count_temp2(5 downto 0);
    else
        count_temp <= countout_int;
    end if;
  end process count_proc;

  latch_proc1 : process
  begin
    wait until prising(clk);
    countout_int <= count_temp after 100 ps;
  end process latch_proc1;

  countout <= countout_int;
end behavior;

------------------------------------------------------------------------
--  Cell:         lock
--  Description:
--  Author:       Kevin Stone
--  Date:         7/6/94

--    Inputs
--        CLK64
--        RESET_L:  chip reset
--        T_RESET:  Indicates radio fell out of lock
--        VALID_DATA:  Occurs approx every 1024 cycles
--        C1
--        C2
--        C3
--        C4
--    Outputs
--        LOCK
--        LOCKRESET:  High for one cycle, right after radio falls
--                    out of lock
--
--
------------------------------------------------------------------------
```

```
entity lock is
  port (
    clk64 :         in  vlbit;
    reset_l :       in  vlbit;
    t_reset :       in  vlbit;
    valid_data :    in  vlbit;
    c1_l :          in  vlbit;
    c2_l :          in  vlbit;
    c3_l :          in  vlbit;
    c4_l :          in  vlbit;
    lock :          out vlbit;
    lockreset:      out vlbit;
    statebits :     out vlbit_vector(1 downto 0) -- take out state for
                                                 -- debugging purposes
  );
end lock;

architecture behavior of lock is
  signal nextstate : vlbit_vector(1 downto 0);
  signal presentstate : vlbit_vector(1 downto 0);
  signal nextlock : vlbit;
begin

  statebits <= presentstate;

  process(clk64, presentstate, reset_l, valid_data, c1_l, c2_l, c3_l,
c4_l, t_reset)
    variable statetemp : integer;
  begin

    statetemp := vld2int(presentstate);
    case statetemp is
        when 0 =>  -- no lock
            lockreset <= '0';
           nextstate(0) <= (valid_data and not(c1_l and c2_l and c3_l and
c4_l));
            nextstate(1) <= '0';
             nextlock <= (valid_data and not(c1_l and c2_l and c3_l and
c4_l));
        when 1 =>  -- got lock
            lockreset <= '0';
            nextstate(0) <= not(valid_data and t_reset);
            nextstate(1) <= valid_data and t_reset;
            nextlock <= not(t_reset and valid_data);
        when 2 =>  -- lost lock
            lockreset <= '1';
            nextstate <= "00";
            nextlock <= '0';
        when others =>  -- invalid state
            lockreset <= '0';
            nextstate <= "00";
            nextlock <= '0';
    end case;
  end process;
  latch1Proc : process
  begin
    wait until prising(clk64) or reset_l='0';
    if reset_l='0' then
        presentstate <= "00";
    else
        presentstate <= nextstate;
    end if;
  end process latch1Proc;

  latch2Proc : process
  begin
    wait until prising(clk64) or reset_l='0';
    if reset_l='0' then
        lock <= '0';
    else
        lock <= nextlock;
    end if;
  end process latch2Proc;

end behavior;

-------------------------------------------------------------------------
--  Cell:        pn
--  Description: pn generator
--  Author:      Kevin Stone
--  Date:        3/23/94
--
-------------------------------------------------------------------------

--library synth;
--use synth.stdsynth.all;

entity pn is
  port (
    reset_l :        in  vlbit;
--    pnsynch :        in  vlbit;
    clk64 :          in  vlbit;
    stall_l :        in  vlbit;
    pn_ld :          in  vlbit;
    seed :           in  vlbit_vector(15 downto 0);
    pncode :         out vlbit_vector(15 downto 0);
    pn_out :         out vlbit;
    pn_out1d :       out vlbit;
    pn_out2d :       out vlbit;
    pn_out3d :       out vlbit;
    pn_xorout :      out vlbit;
    pn_allones_l :   out vlbit
  );
end pn;

architecture behavior of pn is

component mux15_3to1
  port (
    A :              in  vlbit_vector(15 downto 0);
    B :              in  vlbit_vector(15 downto 0);
    C :              in  vlbit_vector(15 downto 0);
    sel :            in  vlbit_vector(1 downto 0);
    Q :              out vlbit_vector(15 downto 0)
  );
end component;

component delay1
  port (
    input :          in  vlbit;
    reset_l :        in  vlbit;
    clk :            in  vlbit;
    output :         out vlbit
  );
end component;

  signal pncode_int : vlbit_vector(15 downto 0);
  signal load_l_temp, load_l_d1 : vlbit;
  signal pn_xorout_int : vlbit;
  constant temp1 : vlbit_vector(15 downto 0)
        := "1111111111111110";
  signal pnload_shift : vlbit_vector(15 downto 0);
  signal pnload_temp : vlbit_vector(15 downto 0);
  signal mux15sel : vlbit_vector(1 downto 0);
  signal pn_reset_l : vlbit;
  signal tmp1 : vlbit;
  signal pn_out1d_int : vlbit;
  signal pn_out2d_int : vlbit;
begin

mux15 : mux15_3to1
  port map(
    A => pncode_int,
    B => seed,
    C => pnload_shift,
    sel => mux15sel,
    Q => pnload_temp
  );

dly1 : delay1
  port map(
    input => pncode_int(0),
    reset_l => tmp1,
    clk => clk64,
    output => pn_out1d_int
  );

dly2 : delay1
  port map(
    input => pn_out1d_int,
    reset_l => tmp1,
    clk => clk64,
    output => pn_out2d_int
  );

dly3 : delay1
```

```
  port map(
    input => pn_out2d_int,
    reset_l => tmp1,
    clk => clk64,
    output => pn_out3d
  );

--  pn_reset_l <= reset_l and not(pnsynch);
  tmp1 <= '1';
  pn_out1d <= pn_out1d_int;
  pn_out2d <= pn_out2d_int;
  pn_reset_l <= reset_l;
  pn_xorout_int <= pncode_int(15) xor pncode_int(13) xor
                   pncode_int(4) xor pncode_int(0);
  pn_xorout <= pn_xorout_int;
  pncode <= pncode_int;
  pn_out <= pncode_int(0);
  pn_allones_l <= load_l_d1;
  pnload_shift <= pn_xorout_int & pncode_int(15 downto 1);

   load_proc : process(pncode_int, reset_l, pn_xorout_int, seed,
load_l_d1,
                         stall_l, pn_ld)
  begin
    -- 4/2/94: different from original sdl/bds file.  Changed polarity of
load
    -- Note: checking for all ones case one cycle ahead
    if stall_l='0' then
        load_l_temp <= load_l_d1;
    elsif (pncode_int = temp1) then
        load_l_temp <= '0';
    else
        load_l_temp <= '1';
    end if;

    -- load_l_d1 goes when pncode_int is all ones or when resetting
    if stall_l='0' then
        mux15sel <= "00";
    elsif load_l_d1='0' or pn_ld='1' then
        mux15sel <= "01";
    else
        mux15sel <= "10";
    end if;
  end process load_proc;

  latch_proc1 : process
  begin
    wait until prising(clk64);
    pncode_int <= pnload_temp;
  end process latch_proc1;

  latch_proc2 : process
  begin
    wait until prising(clk64) or pn_reset_l='0';
    if pn_reset_l='0' then
        load_l_d1 <= '0';
    else
        load_l_d1 <= load_l_temp;
    end if;
  end process latch_proc2;

end behavior;

-----------------------------------------------------------------------
--  Cell:        mux15_3to1
--  Description: 15 bit wide 3 to 1 mux
--  Author:      Kevin Stone
--  Date:        6/13/94
--
-----------------------------------------------------------------------

--library synth;
--use synth.stdsynth.all;

entity mux15_3to1 is
  port (
    A :             in  vlbit_vector(15 downto 0);
    B :             in  vlbit_vector(15 downto 0);
    C :             in  vlbit_vector(15 downto 0);
    sel :           in  vlbit_vector(1 downto 0);
    Q :             out vlbit_vector(15 downto 0)
  );
end mux15_3to1;

architecture behavior of mux15_3to1 is
begin
  muxProc : process(sel, A, B, C)
    variable muxsel_temp : integer;
  begin
    muxsel_temp := vld2int(sel);
    case muxsel_temp is
        when 0 =>  -- select seed
            Q <= A;
        when 1 =>  -- select pnsc_a
            Q <= B;
        when 2 =>  -- select pnsc_b
            Q <= C;
        when others => -- should never get here
            Q <= "XXXXXXXXXXXXXXXX";
    end case;
  end process muxProc;
end behavior;

-----------------------------------------------------------------------
-- Cell:        updctrl
-- Description:
-- Author:      Kevin Stone
-- Date:        7/5/94
--
-- Note: upd63, upd1087, and corr_rst need to be level
-- converted to 1.5V, then latched to produce upd64, upd1088, corr_rst1_5
-- 7/6/94: changed pn_stall_l to pnstall
-- 7/17/94: using upd1087 as valid_data signal.  This means that
-- the outputs from the correlators must be before the final
-- latch instead of after.
-- 8/1/94: add flop create a delay reset_1 signal.  This is used to
-- make sure the proper thing is done when coming out of reset
------------------------------------------------------------------------

library synth;
use synth.stdsynth.all;

entity updctrl is
  port (
    clk64 :         in  vlbit;
    reset_1 :       in  vlbit;
    c1_1 :          in  vlbit;
    c2_1 :          in  vlbit;
    c3_1 :          in  vlbit;
    c4_1 :          in  vlbit;
--    valid_data :    in  vlbit;
    lock :          in  vlbit;
    pnstall_1 :     out vlbit;
--    pnsynch :       out vlbit; -- used to keep counter/pn/walsh in synch
when
--                                 -- coming out of reset
    upd63 :         out vlbit; -- counted for 64 valid Tchips
    upd1087 :       out vlbit; -- tells backend of corr's to latch data
     corr_rst :       out vlbit; -- clear out contents of carry-save
adder's
                                 -- "dump-into" registers
    dr_rst :        out vlbit; -- clear out contents of dr_corr regs.
    statebits :     out vlbit_vector(2 downto 0) -- take out state for
                                                 -- debugging purposes
  );
end updctrl;

architecture behavior of updctrl is

component cnt2048lds
  port (
    countin :       in  vlbit_vector(10 downto 0);
    clk :           in  vlbit;
    rst :           in  vlbit;
    ld :            in  vlbit;
    cnt :           in  vlbit;
    countout :      out vlbit_vector(10 downto 0)
  );
end component;

component delay1
  port (
    input :         in  vlbit;
    reset_1 :       in  vlbit;
    clk :           in  vlbit;
    output :        out vlbit
  );
end component;
```

```
  signal presentstate : vlbit_vector (2 downto 0);
  signal nextstate : vlbit_vector (2 downto 0);
  signal count : vlbit_vector (10 downto 0);
  signal tmp64 : vlbit_vector (10 downto 0); -- used as constant input to
cnt2048
  signal valid_data : vlbit; -- tells control that output from correla-
tors is valid
  signal rstflag : vlbit; -- indicates that we are in state=000\b
  signal flag61 : vlbit; -- based only on value of count
  signal flag62 : vlbit; -- latched flag61
  signal ctrl62 : vlbit; -- control signal generated by state machine
                         -- which can also set upd63
  signal preupd63 : vlbit; -- input to flop where upd63 is the output
  signal flag1085 : vlbit; -- based only on value of count
  signal flag1086 : vlbit; -- latched flag1085
  signal ctrl1086 : vlbit; -- control signal generated by state machine
                           -- which can also set upd1087
  signal preupd1087 : vlbit; -- input to flop where upd1087 is the output
  signal upd63_int : vlbit;
  signal upd1087_int : vlbit;
  signal corr_rst_int : vlbit;
  signal ctrl_crst : vlbit; -- control signal generated by state machine
                            -- which can also set corr_rst
  signal ctrl_drst : vlbit; -- control signal generated by state machine
                            -- which can also set dr_rst
  signal precorr_rst : vlbit; -- input to flop where corr_rst is the out-
put
  signal predr_rst : vlbit; -- input to flop where dr_rst is the output
  signal pnstall_l_int : vlbit;
  signal prepnstall_l : vlbit; -- input to flop where pnstall is the out-
put
  signal rst_count : vlbit;
  signal cnt_count : vlbit;
  signal ld_count : vlbit;
  signal tmp1 : vlbit;
  signal tmp0 : vlbit;
  signal rstflag_d1 : vlbit;
begin

tmp0 <= '0';
tmp1 <= '1';
tmp64 <= "00001000000";
preupd63 <= flag62 or ctrl62;
preupd1087 <= flag1086 or ctrl1086;
precorr_rst <= flag1086 or ctrl_crst;
upd63 <= upd63_int;
upd1087 <= upd1087_int;
valid_data <= upd1087_int;
corr_rst <= corr_rst_int;
pnstall_l <= pnstall_l_int;
statebits <= presentstate;
cnt_count <= pnstall_l_int; -- keep count in sync with data/pn/walsh
-- pnsynch <= not(presentstate(2) or presentstate(1) or present-
state(0));
rstflag <= presentstate(2) or presentstate(1) or presentstate(0);
```

```
cnt2048 : cnt2048lds
  port map(
    countin => tmp64,
    clk => clk64,
    rst => rst_count,
    ld => tmp0,
    cnt => cnt_count,
    countout => count
  );

-- pipeline control signals to minimize the number of levels of
-- logic between flops.  Want only 3 levels of logic, 4 at most.
dly1 : delay1
  port map(
    input => flag61,
    reset_l => tmp1,
    clk => clk64,
    output => flag62
  );

dly2 : delay1
  port map(
    input => preupd63,
    reset_l => tmp1,
    clk => clk64,
    output => upd63_int
  );

dly3 : delay1
  port map(
    input => flag1085,
    reset_l => tmp1,
    clk => clk64,
    output => flag1086
  );

dly4 : delay1
  port map(
    input => preupd1087,
    reset_l => tmp1,
    clk => clk64,
    output => upd1087_int
  );

dly5 : delay1
  port map(
    input => precorr_rst,
    reset_l => tmp1,
    clk => clk64,
    output => corr_rst_int
  );

dly6 : delay1
  port map(
    input => ctrl_drst,
    reset_l => tmp1,
    clk => clk64,
    output => dr_rst
  );

dly7 : delay1
  port map(
    input => prepnstall_l,
    reset_l => tmp1,
    clk => clk64,
    output => pnstall_l_int
  );

dly8 : delay1
  port map(
    input => rstflag,
    reset_l => tmp1,
    clk => clk64,
    output => rstflag_d1
  );

  flag_proc : process(count)
    variable tmp61 : vlbit_vector(5 downto 0);
    variable tmp1085 : vlbit_vector(10 downto 0);
  begin
    tmp61   :=      "111101";
    tmp1085 := "10000111101";
    if (count(5 downto 0) = tmp61(5 downto 0)) then
        flag61 <= '1';
    else
        flag61 <= '0';
    end if;

    if (count = tmp1085) then
        flag1085 <= '1';
    else
        flag1085 <= '0';
    end if;
  end process flag_proc;

  stmachProc : process(lock, c1_l, c2_l, c3_l, c4_l, presentstate,
                       upd1087_int, valid_data, rstflag_d1)
    variable statetemp : integer;
  begin
    statetemp := vld2int(presentstate);
    case statetemp is
        -- note: wierd things happen in reset because upd63, upd1087,
        --       corr_rst are flopped values.  Thus, after reset_l
        --       goes high, upd1087 will be high for one extra cycle.
        --       This means the controller will be in state 1 and see
        --       upd1087 is high and thus want to reset the counter.
        --       This will screw up the counter's synch with pn/walsh.
        --       Therefore, the kluge is to generate a signal which
        --       will keep counter/pn/walsh in synch when coming out
        --       of reset.
        when 0 =>  -- reset state
        nextstate <= "001";
        prepnstall_l <= '1';
        ctrl62 <= '1';  -- reset correlators
        ctrl1086 <= '1'; -- clear out backend
        ctrl_crst <= '1';
        ctrl_drst <= '1';
        rst_count <= '1'; -- reset count
        ld_count <= '0';
    when 1 => -- trying to acquire lock
        prepnstall_l <= '1';
        ctrl62 <= '0';
        ctrl1086 <= '0';
        ctrl_crst <= '0';
        ctrl_drst <= '0';
        ld_count <= '0';

        if upd1087_int='1' and rstflag_d1='1' then
            rst_count <= '1';
        else
           rst_count <= '0'; -- used to keep pn, walsh, count in sync
        end if;

        -- use rstflag_d1 make that the correlators'
        -- comparators' outputs are not used right
        -- after coming out of lock
        if valid_data='1' and rstflag_d1='1' then
            if c1_l='0' then
                nextstate <= "110"; -- stall 0 cycles
            elsif c2_l='0' then
                nextstate <= "101"; -- stall 1 cycle
            elsif c3_l='0' then
                nextstate <= "100"; -- stall 2 cycles
            elsif c4_l='0' then
                nextstate <= "011"; -- stall 3 cycles
            else
                nextstate <= "010"; -- stall 4 cycles
            end if;
        else -- don't do anything
            nextstate <= "001";
        end if;
    when 2 => -- stall 4 cycles
        prepnstall_l <= '0';
        nextstate <= "011";
        ctrl62 <= '0';
        ctrl1086 <= '0';
        ctrl_crst <= '0';
        ctrl_drst <= '0';
        rst_count <= '0';
        ld_count <= '0';
    when 3 => -- stall 3 cycles
        prepnstall_l <= '0';
        nextstate <= "100";
        ctrl62 <= '0';
        ctrl1086 <= '0';
        ctrl_crst <= '0';
        ctrl_drst <= '0';
        rst_count <= '0';
        ld_count <= '0';
    when 4 => -- stall 2 cycles
        prepnstall_l <= '0';
        nextstate <= "101";
        ctrl62 <= '0';
        ctrl1086 <= '0';
        ctrl_crst <= '0';
        ctrl_drst <= '0';
        rst_count <= '0';
        ld_count <= '0';
    when 5 => -- stall 1 cycle
        prepnstall_l <= '0';
        nextstate <= "110";
        ctrl62 <= '0';
        ctrl1086 <= '0';
        ctrl_crst <= '0';
        ctrl_drst <= '0';
        rst_count <= '0';
        ld_count <= '0';
    when 6 =>
        if lock='1' then
            nextstate <= "111";
        else
            nextstate <= "001";
        end if;

        prepnstall_l <= '1';
        ctrl62 <= '1';  -- clear out correlator
        ctrl1086 <= '0';
        ctrl_crst <= '1'; -- clear out correlator
        ctrl_drst <= '0';
        rst_count <= '0';
        ld_count <= '0';
    when 7 => -- got lock
        prepnstall_l <= '1';
        ctrl62 <= '0';
        ctrl1086 <= '0';
        ctrl_crst <= '0';
        ctrl_drst <= '0';
        ld_count <= '0';

        if upd1087_int ='1' then
            rst_count <= '1';
        else
            rst_count <= '0';
        end if;

        if lock='1' then
            nextstate <= "111";
        else
            nextstate <= "000";
        end if;
    when others => -- should never get here
        prepnstall_l <= '1';
        nextstate <= "000";
```

```
            ctrl62 <= '0';
            ctrl1086 <= '0';
            ctrl_crst <= '0';
            ctrl_drst <= '0';
            rst_count <= '0';
            ld_count <= '0';
    end case;
  end process stmachProc;

  latch1Proc : process
  begin
    wait until prising(clk64) or reset_l='0';
    if reset_l='0' then
        presentstate <= "000";
    else
        presentstate <= nextstate;
    end if;
  end process latch1Proc;

end behavior;

------------------------------------------------------------------------
--  Cell:        cnt2048lds
--  Description: 11 bit counter with synchronous load and reset
--  Author:      Kevin Stone
--  Date:        7/4/94
--
--
------------------------------------------------------------------------

--library synth;
--use synth.stdsynth.all;

entity cnt2048lds is
  port (
    countin :       in vlbit_vector (10 downto 0);
    clk :           in vlbit;
    rst :           in vlbit;
    ld :            in vlbit;
    cnt :           in vlbit;
    countout :      out vlbit_vector (10 downto 0)
  );
end cnt2048lds;

architecture behavior of cnt2048lds is
  signal countout_int : vlbit_vector(10 downto 0);
  signal count_temp : vlbit_vector(10 downto 0);
  constant all0 : vlbit_vector(10 downto 0)
    := "00000000000";
  constant one : vlbit_vector(10 downto 0)
    := "00000000001";
begin

  count_proc : process(rst, ld, cnt, countin, count_temp, countout_int)
    variable count_temp2 : vlbit_vector(11 downto 0);
  begin
```

```
    count_temp2 := addum(countout_int, one);

    if rst='1' then
        count_temp <= all0;
    elsif ld='1' then
        count_temp <= countin;
    elsif cnt='1' then
        count_temp <= count_temp2(10 downto 0);
    else
        count_temp <= countout_int;
    end if;
  end process count_proc;

  latch_proc1 : process
  begin
    wait until prising(clk);
    countout_int <= count_temp after 100 ps;
  end process latch_proc1;

  countout <= countout_int;
end behavior;

------------------------------------------------------------------------
--  Cell:        regctrl
--  Description: Registers for thres #1, #2, #3 and walsh number
--  Author:      Kevin Stone
--  Date:        3/28/94
--               9/13/94: Added pnclk
--
--
------------------------------------------------------------------------

library synth;
use synth.stdsynth.all;

entity regctrl is
  port (
    reset_l :        in  vlbit;
    clk64 :          in  vlbit;
    pn_allones_l :   in  vlbit;
    cs_l :           in  vlbit;
    wr_l :           in  vlbit;
    addr :           in  vlbit_vector(1 downto 0);
    clka :           out vlbit;
    clkb :           out vlbit;
    clkc :           out vlbit;
    clkd :           out vlbit;
    clkback :        out vlbit -- low when pn shiftreg is allones
  );
end regctrl;

architecture behavior of regctrl is
begin
  clka <= not(cs_l or wr_l) and not(addr(1)) and not(addr(0));
  clkb <= not(cs_l or wr_l) and not(addr(1)) and    (addr(0));
  clkc <= not(cs_l or wr_l) and    (addr(1)) and not(addr(0));
  clkd <= not(cs_l or wr_l) and    (addr(1)) and    (addr(0));
  clkback <= (not(reset_l) and clk64) or (reset_l and pn_allones_l);

end behavior;

------------------------------------------------------------------------
-- Cell:          two2smx
-- Description: convert two's complement numbers (-8 to 7) to
-- sign magnitude (-7 to 7).  Sign of INB and INC are inverted.
-- This is done to interface with the transmitter chip.
-- Author:        Kevin Stone
-- Date:          10/18/94
--
------------------------------------------------------------------------


entity two2smx is
  port (
    ina :         in  vlbit_vector(3 downto 0);
    inb :         in  vlbit_vector(3 downto 0);
    inc :         in  vlbit_vector(3 downto 0);
    ind :         in  vlbit_vector(3 downto 0);
    outa :        out vlbit_vector(3 downto 0);
    outb :        out vlbit_vector(3 downto 0);
    outc :        out vlbit_vector(3 downto 0);
    outd :        out vlbit_vector(3 downto 0)
  );
end two2smx;

architecture behavior of two2smx is

component buff104_4
  port (
    input :             in  vlbit_vector (3 downto 0);
    output :            out vlbit_vector (3 downto 0)
  );
end component;

signal outa_temp : vlbit_vector(3 downto 0);
signal outb_temp : vlbit_vector(3 downto 0);
signal outc_temp : vlbit_vector(3 downto 0);
signal outd_temp : vlbit_vector(3 downto 0);

begin
bufa : buff104_4
  port map(
    input => outa_temp,
    output => outa
  );

bufb : buff104_4
  port map(
    input => outb_temp,
    output => outb
  );
```

```
bufc : buff104_4
  port map(
    input => outc_temp,
    output => outc
  );

bufd : buff104_4
  port map(
    input => outd_temp,
    output => outd
  );


  process(ina)
    variable ina_temp : integer;
  begin
    ina_temp := v1d2int(ina);
    case ina_temp is
        when 0 =>
            outa_temp <= "0000";
        when 1 =>
            outa_temp <= "0001";
        when 2 =>
            outa_temp <= "0010";
        when 3 =>
            outa_temp <= "0011";
        when 4 =>
            outa_temp <= "0100";
        when 5 =>
            outa_temp <= "0101";
        when 6 =>
            outa_temp <= "0110";
        when 7 =>
            outa_temp <= "0111";
        when 8 =>
            outa_temp <= "1111";
        when 9 =>
            outa_temp <= "1111";
        when 10 =>
            outa_temp <= "1110";
        when 11 =>
            outa_temp <= "1101";
        when 12 =>
            outa_temp <= "1100";
        when 13 =>
            outa_temp <= "1011";
        when 14 =>
            outa_temp <= "1010";
        when 15 =>
            outa_temp <= "1001";
        when others =>
            outa_temp <= "XXXX";
    end case;
  end process;

  process(inb)
```

```
    variable inb_temp : integer;
  begin
    inb_temp := v1d2int(inb);
    case inb_temp is
         when 0 =>
             outb_temp <= "1000";
         when 1 =>
             outb_temp <= "1001";
         when 2 =>
             outb_temp <= "1010";
         when 3 =>
             outb_temp <= "1011";
         when 4 =>
             outb_temp <= "1100";
         when 5 =>
             outb_temp <= "1101";
         when 6 =>
             outb_temp <= "1110";
         when 7 =>
             outb_temp <= "1111";
         when 8 =>
             outb_temp <= "0111";
         when 9 =>
             outb_temp <= "0111";
         when 10 =>
             outb_temp <= "0110";
         when 11 =>
             outb_temp <= "0101";
         when 12 =>
             outb_temp <= "0100";
         when 13 =>
             outb_temp <= "0011";
         when 14 =>
             outb_temp <= "0010";
         when 15 =>
             outb_temp <= "0001";
         when others =>
             outb_temp <= "XXXX";
   end case;
 end process;

 process(inc)
   variable inc_temp : integer;
 begin
   inc_temp := v1d2int(inc);
   case inc_temp is
        when 0 =>
            outc_temp <= "1000";
        when 1 =>
            outc_temp <= "1001";
        when 2 =>
            outc_temp <= "1010";
        when 3 =>
            outc_temp <= "1011";
        when 4 =>
            outc_temp <= "1100";
        when 5 =>
            outc_temp <= "1101";
        when 6 =>
            outc_temp <= "1110";
        when 7 =>
            outc_temp <= "1111";
        when 8 =>
            outc_temp <= "0111";
        when 9 =>
            outc_temp <= "0111";
        when 10 =>
            outc_temp <= "0110";
        when 11 =>
            outc_temp <= "0101";
        when 12 =>
            outc_temp <= "0100";
        when 13 =>
            outc_temp <= "0011";
        when 14 =>
            outc_temp <= "0010";
        when 15 =>
            outc_temp <= "0001";
        when others =>
            outc_temp <= "XXXX";
    end case;
end process;

process(ind)
    variable ind_temp : integer;
begin
    ind_temp := v1d2int(ind);
    case ind_temp is
        when 0 =>
            outd_temp <= "0000";
        when 1 =>
            outd_temp <= "0001";
        when 2 =>
            outd_temp <= "0010";
        when 3 =>
            outd_temp <= "0011";
        when 4 =>
            outd_temp <= "0100";
        when 5 =>
            outd_temp <= "0101";
        when 6 =>
            outd_temp <= "0110";
        when 7 =>
            outd_temp <= "0111";
        when 8 =>
            outd_temp <= "1111";
        when 9 =>
            outd_temp <= "1111";
        when 10 =>
            outd_temp <= "1110";
        when 11 =>
            outd_temp <= "1101";
        when 12 =>
            outd_temp <= "1100";
        when 13 =>
            outd_temp <= "1011";
        when 14 =>
            outd_temp <= "1010";
        when 15 =>
            outd_temp <= "1001";
        when others =>
            outd_temp <= "XXXX";
    end case;
  end process;

end behavior;
```