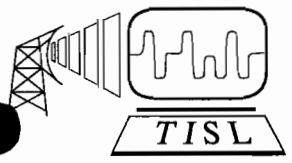
# Design and Implementation of Components for the RDRN Prototype System

**Thanh Mai**
**Y. Leo Lui**
**Benjamin J. Ewy**
**Craig Sparks**
**Joseph B. Evans**
**Victor S. Frost**

TISL Technical Report TISL-10920-10

July 1995




Telecommunications and Information Sciences Laboratory
The University of Kansas Center for Research, Inc.
2291 Irving Hill Drive Lawrence, Kansas 66045

# Design and Implementation of Components for the RDRN Prototype System

Thanh Mai
Y. Leo Lui
Craig Sparks
Benjamin J. Ewy
Joseph B. Evans
Victor S. Frost

Telecommunications and Information Sciences Laboratory
Department of Electrical and Computer Engineering
The University of Kansas
Lawrence, Kansas 66045

July 28, 1995

This paper documents the design of several circuits for the Rapidly Deployable Radio Network(RDRN) currently being implemented at the Telecommunications and Information Sciences Laboratory at the University of Kansas.

Some elements that will be implemented in the RDRN project are host interface card, digital beamforming and an ATM-based radio link. The prototype system will include an antenna element system housed in a VME rack, which will contain a Rack Control Card, an Oscillator Card, a QPSK Receiver Card, 8 Transmit Beamforming Cards, a transmit antenna array, and a single omnidirectional receive antenna.

The host will be responsible for differentially encoding the bit streams from various cells, and passing them to FIFOs in the Rack Controller Card. The host will also compute and download lookup tables for each of the Transmit Beamforming Cards based on information from the orderwire. The first section describes the timing of the major signals on the PCI card operating in the add-on mode and suggests a way to test the operation.

The second section is on a pseudo-random data generator to be used with a Stanford Telecom QPSK transmitter for the prototype of a simplex omnidirectional radio link. The data generator is based on the principle of pseudonoise(PN) sequence generation. It generates two data streams at 2 Mb/s, which provide the I and Q data for the QPSK modulator.

The prototype system will allow for 4 independent 2 Mb/s beams. The initial omnidirectional prototype receiver will be based on the STEL-9236/LB/CE, a 2048 kbps QPSK Digital Demodulator. The third section describes the serial-to-parallel conversion at the Demodulator-Receiver FIFO interface. The serial-to-parallel DMUX and control logic are implemented in a Xilinx FPGA, and both single-beam and quadruple-beam operations are supported.

The digital beamforming will be done in the transmit direction at baseband in the prototype. Lookup tables will be used to store the summed 10-bit I and Q's for each combination of input I and Q bits for each of the 4 beams, which is necessary for beam steering and the placement of

nulls. The Transmit Beamforming Cards are based around 4Kx8 dual port SRAMs used to build the lookup tables. The last section describes the circuit which simulates the downloading of data into the SRAMs and the transfer of data from the SRAMs to a Quadrature Modulator through D/A converters. This circuit will allow testing of the beamforming before the host interface is complete.

# 1 Testing the PCI Card

One of the major elements of the RDRN system is the Host Interface. The host interface card will use a PCI prototyping board from AMCC to develop a simple device driver to allow debugging of the RF VME system. To be able to design with the PCI, specifically to use the PCI in the add-on mode, one needs to understand the functions of all the signals that are involved and how they cooperate with each other. Included in the next pages are the timing diagrams of the read and write cycles of the add-on bus.

Signals that we need to pay good attention to are DQ[31:0], ADR[5:2], PTRDY#, PTATN#, PTNUM[1:0], PTBE[3:0]#, PTADR#, PTWR, SELECT#, RD#, WE#, and BE[3:0]#. The read, write cycles of the PCI and the add-on are in opposite. When the PCI writes data to the add-on, the add-on must read the internal pass-thru data register to source the data on the DQ[31:0] bus. When the PCI reads data from the add-on, the add-on must write to the internal pass-thru data register, so the PCI can read from there.

To test the operation of the read and write cycle of the add-on bus based on the understanding of the timing diagram, a suggested logic diagram is shown on the next figure. A register is used to hold the data written by the PCI to the add-on via the DQ bus and the outputs of this register will drive the DQ bus when the PCI reads from the add-on. If the read values are the same as the written values, the PCI card works ok. Notice that the register is clocked at the negation of the /RD signal (the data on the DQ bus must be hold valid longer than the propagation delay of the register), and the outputs of the register are valid when the /WR signal is asserted.

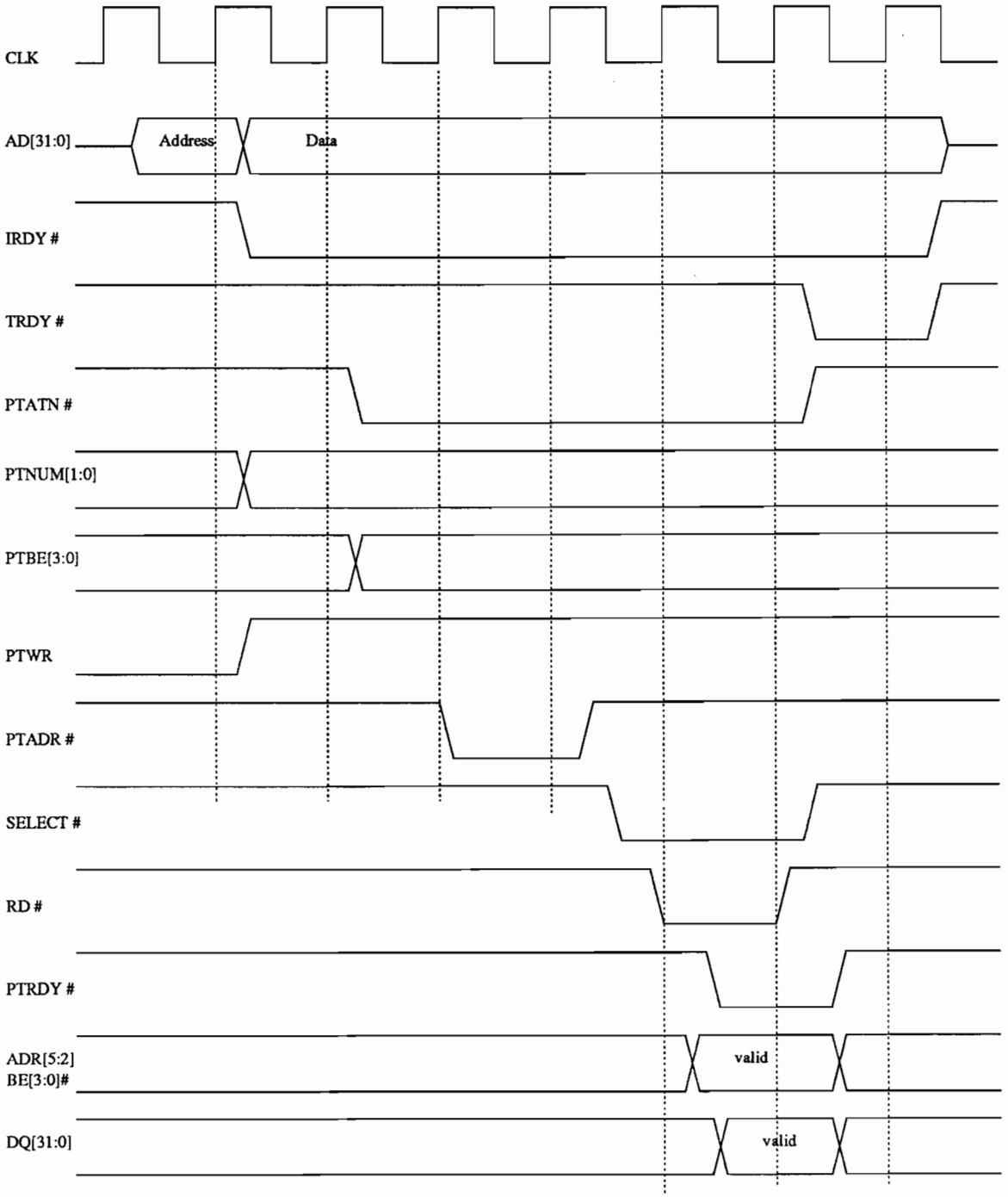



Figure 1: Timing Diagram of an Add-on Write Cycle (PCI reads from the Add-on)

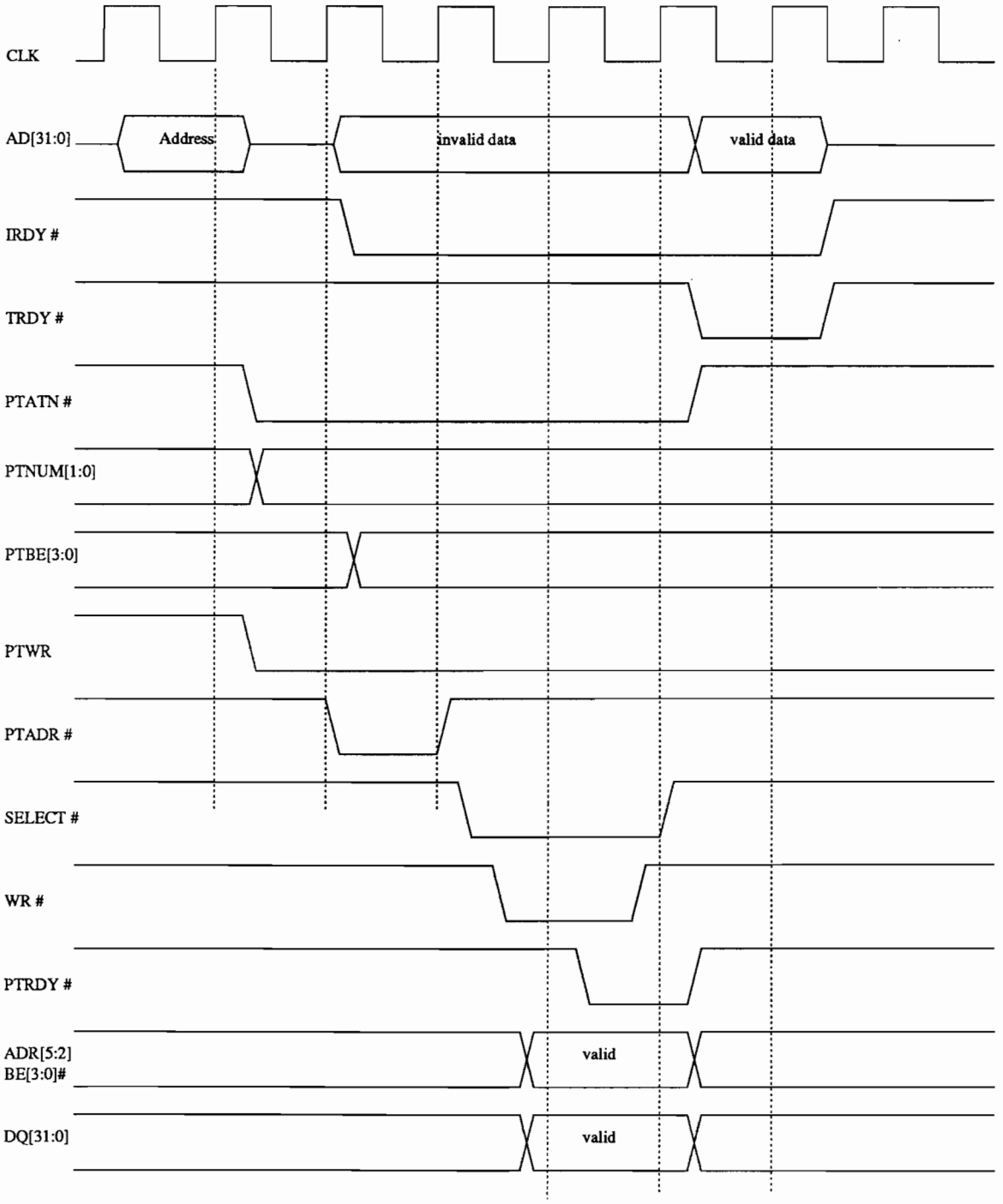



Figure 2: Timing Diagram of an Add-on Read Cycle (PCI writes to the Add-on)

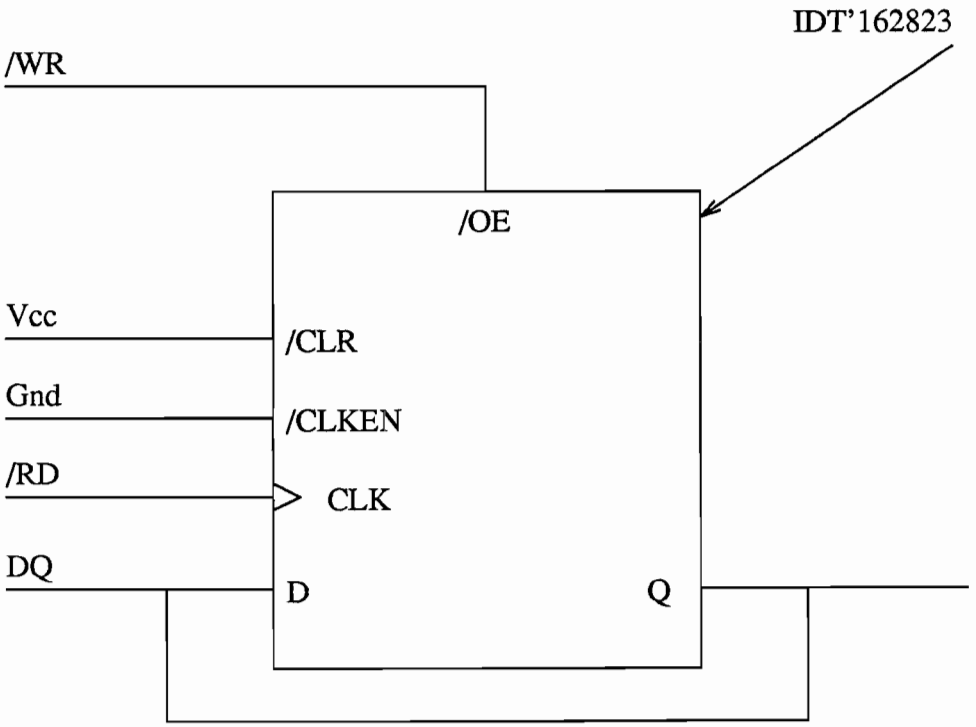



Figure 3: Suggested Logic Diagram of the PCI Test Circuit

# 2 Data Generator

## 2.1 Introduction

This section briefly describes the random data generator to be used with the Stanford Telecom modulator.

## 2.2 Background

The generator to be designed produces pseudonoise(PN) sequences, which are often generated with linear feedback shift registers(LFSRs). In its simplest form, an LFSR can be implemented by XORing some of the register bits and feeding the result back to the serial input of the shift register. The output sequence depends on the initial condition and feedback scheme of the LFSR and is periodic. For a maximum length output sequence, its cycle length is

$$2^n - 1$$

bits, where n is the number of stages in the feedback shift register. Maximum length sequences are best suited for PN sequences and can be obtained by choosing the right feedback schemes. The generation of PN sequences has been studied in detail, and the feedback schemes for generating maximum length sequences have also been well documented.

## 2.3 Implementation

A simple 10-bit LFSR is implemented in a 22V10 with the [10, 3] feedback scheme; i.e., the 3rd and 10th bits are XORed and the result is fed back into the 1st bit(serial input) of the register, as shown in Figure 4. This is one of the feedback schemes that generate maximum length sequences. DIP switches provide initial values and load/shift enable to the 22V10. The PN sequences are taken from the serial output of the LFSR from the 22V10 and then buffered. The feedback scheme

can be changed in the 22V10 easily by rewriting the equation for the serial input. Figure 5 shows the block diagram of the data generator. Actually two data generators are built on a circuit board to provide the I and Q data.

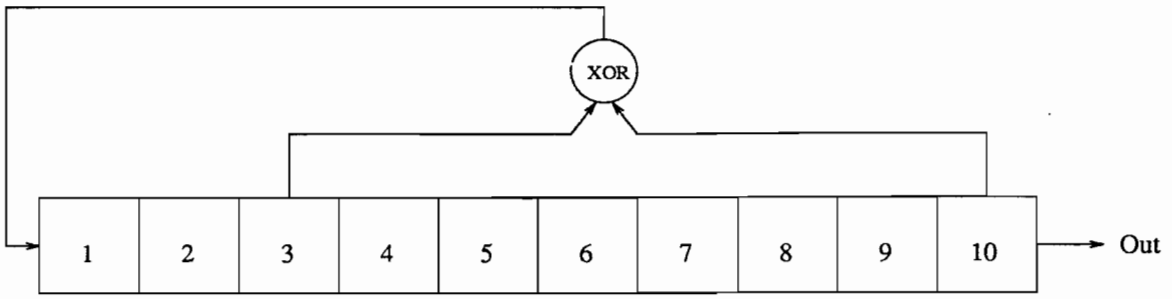



Figure 4: Linear Feedback Shift Register–[3,10] Feedback Scheme

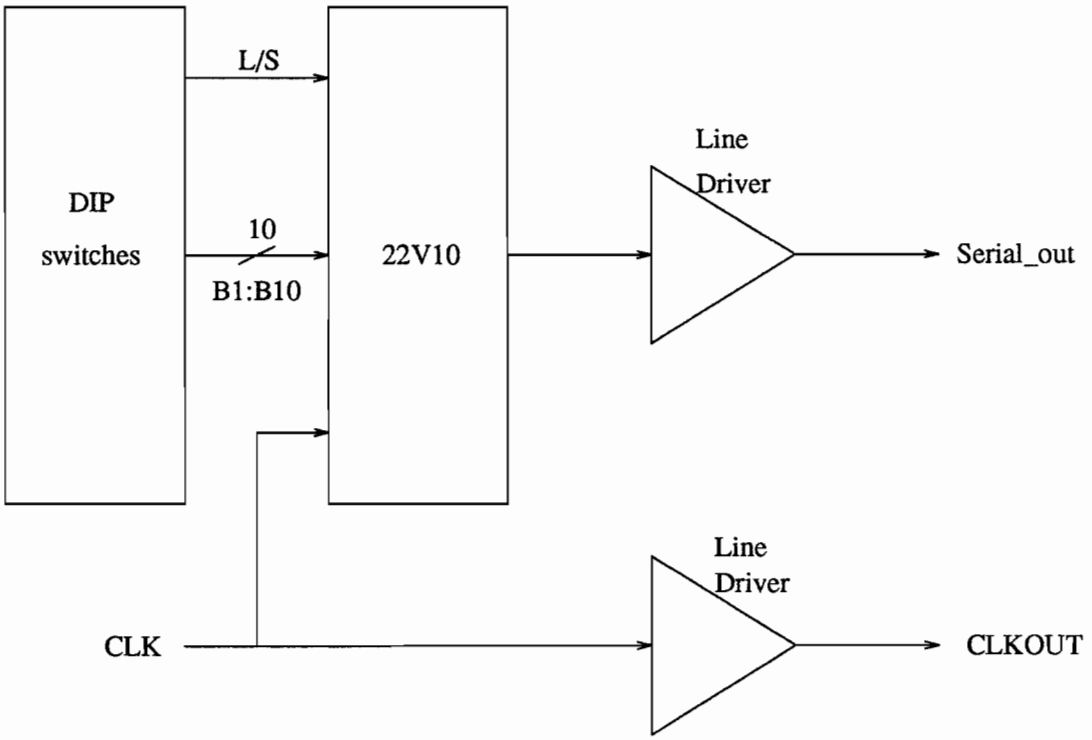



Figure 5: Block Diagram of Pseudo-random Data Generator

```
Title Pseudonoise Generator
Pattern pds
Revision 1
Author Y. Liu
Company REU Site
Date 060595
CHIP pn N85C22V10

;pin assignments

PIN 2 CLK ;clock pin
PIN 3 D0 ;input pins
PIN 4 D1
PIN 5 D2
PIN 6 D3
PIN 7 D4
PIN 9 D5
PIN 10 D6
PIN 11 D7
PIN 12 D8
PIN 13 D9
PIN 16 EN ;load/shift pin
PIN 17 Q0 ;output pins
PIN 18 Q1
PIN 19 Q2
PIN 20 Q3
PIN 21 Q4
PIN 23 Q5
PIN 24 Q6
PIN 25 Q7
PIN 26 Q8
PIN 27 Q9 ;serial output pin

;equations

EQUATIONS

Q0 := EN * D0 + /EN * (Q2 :+: Q9)
Q1 := EN * D1 + /EN * Q0
Q2 := EN * D2 + /EN * Q1
Q3 := EN * D3 + /EN * Q2
```

```
Q4 := EN * D4 + /EN * Q3
Q5 := EN * D5 + /EN * Q4
Q6 := EN * D6 + /EN * Q5
Q7 := EN * D7 + /EN * Q6
Q8 := EN * D8 + /EN * Q7
Q9 := EN * D9 + /EN * Q8

SIMULATION

VECTOR LFSR := [Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q9]
TRACE_ON CLK Q0 Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9
SETF EN /D0 /D1 /D2 /D3 /D4 /D5 /D6 /D7 /D8 D9 /CLK
PRLDF /Q0 /Q1 /Q2 /Q3 /Q4 /Q5 /Q6 /Q7 /Q8 /Q9
CLOCKF CLK
SETF /EN
FOR j := 1 TO 50 DO
BEGIN
CLOCKF CLK
END
TRACE_OFF

;end simulation
```

# 3 Demodulator-Receiver FIFO Interface

## 3.1 Introduction

This section focuses on the design and implementation of the interface between the Demodulator STEL9236 and the FIFO which is part of the receiver card circuit. The heart of the interface is the FPGA because it performs all the serial-to-parallel data conversion as well as generates all the needed control signals.

## 3.2 Design Specification

We will design the interface to operate in two different modes. The data are obtained from one receiver with the first mode and from four receivers with the second.

### 3.2.1 Operation and Timing of the first mode

First of all, the RxDATA and RxCLK streams from the Demodulator are fed into the 32-bit DMUX (implemented in FPGA) to be converted to 32-bit in parallel data on the outputs. Instead of writing the data directly to the FIFOs, the data are stored in the registers and the outputs from the registers are written to the FIFOs, the stable data from the registers have plenty of time to be written to the FIFOs before the next 32bit data are latched in.

All the components of the circuit operate synchronously. At the outputs of the Demodulator, the data are valid at the rising edges of the Clk. After 32 clocks, the 32-bit data are presented at the outputs of the DMUX, a control signal REG_EN generated from the FPGA will enable the registers to take the 32-bit data as inputs and store them shortly after the rising of 32th clock. After the rising of 33th clock, another control signal WR_EN also generated from the FPGA will enable the writing to the FIFO. A 5-bit counter is actually implemented in the FPGA to keep track of the clock counts.

### 3.2.2 Operation and Timing of the second mode

The operation in this mode is basicly the same as the first one. However, the timing is different. Since data are now received from four sources, they can be written to the FIFOs at a faster rate. Four bytes are received from four receiver (instead of one receiver) are latched into the registers and later written to the FIFOs. Therefore, we will use four 8-bit DMUXs to get parallel data and a 3-bit counter to keep track of the clock counts.

There is one technical problem that needs to be resolved. Since we use four different receivers (Demodulators), we have four different pairs of RxDATAs and RxCLKs. The RxCLKs essentially run at the same speed but have different phases, so we need to insert four small FIFOs before the four 8-bit DMUXs and use a common clock to get 4-bit of data out of the small FIFOs at the same time.

## 3.3 Implementation

The serial-to-parallel conversion and control logic are implemented in a Xilinx FPGA. Two modes of operation are supported, depending on the nubmer of data streams.

### 3.3.1 Single data stream

In the single data stream mode(Figures 6 and 7), 16 CLBs in the FPGA are cascaded to form a serial-in parallel-out shift register. A synchronous binary counter which cycles through 32 states repeatedly is also implemented. It is used to provide the write control to the 16823 registers and FIFO whenever a new set of 32-bit data is shifted into the FPGA. The shift register and the counter both use the same clock provided by the demodulator board.

The counter has a CE input, which must be asserted to enable counting. Five (5) D-type flip-flops are used in the counter, and they are divided to be fit into 3 CLBs on the FPGA as follows:

(1,2), (3), (4,5). The sixth output is the CEO for cascading. It is assumed that there exists a signal indicating the successful operation of the demodulator, and this signal is to be used to drive CE.

The control logic that has been implemented on the FPGA includes the write enables for the 16823 registers and FIFO. REG_EN (active LOW) is asserted whenever the count reaches 11111 (binary), and the 32-bit parallel data is to be clocked into the 16823s before the next rising edge of the system clock. The write operation to the FIFO takes place one clock cycle after the write to the 16823s, so FIFO_EN (active LOW) is asserted whenever the count reaches 00000 (binary). Similarly, the 32-bit data must be clocked into the FIFO before the next rising edge of the system clock because both REG_EN and FIFO_EN are pulses asserted for only one clock cycle. The clock inputs to the 16823s and FIFO come from another source and could be the inverted system clock. Other control logic that will be added on the FPGA includes possibly the power-on reset and initialization command string to the demodulator board.

### 3.3.2 Quadruple data stream

In the quadruple data stream mode(Figures 8 and 9), the 32-bit shift register mentioned above is broken down into four 1-byte shift registers, each accepting bits from a corresponding data stream. Since the four shift registers run in parallel, the throughput will be four times as much as before. The counter used in this mode cycles through 8 states instead of 32 because of the change in shift register length. It is implemented in 2 CLBs on the FPGA without CEO since cascading is not needed. The implementation of control logic changes correspondingly: REG_EN is asserted whenever the count reaches 111 (binary), and FIFO_EN is asserted at 000 (binary).

The synchronization of data bits from the four data streams is achieved by adding four small FIFOs at the inputs to the shift registers. Data from each demodulator output is clocked into its own FIFO with its accompanying clock signal, but a common system clock is used to transfer data from the four FIFOs to the 16823s. This eliminates the phase differences among the data streams.

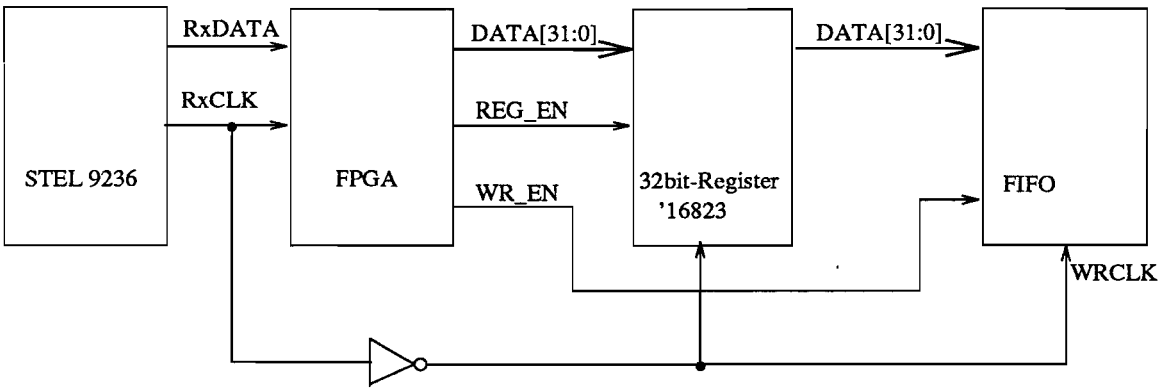



Figure 6: Block Diagram of the Demodulator and FIFO Interface

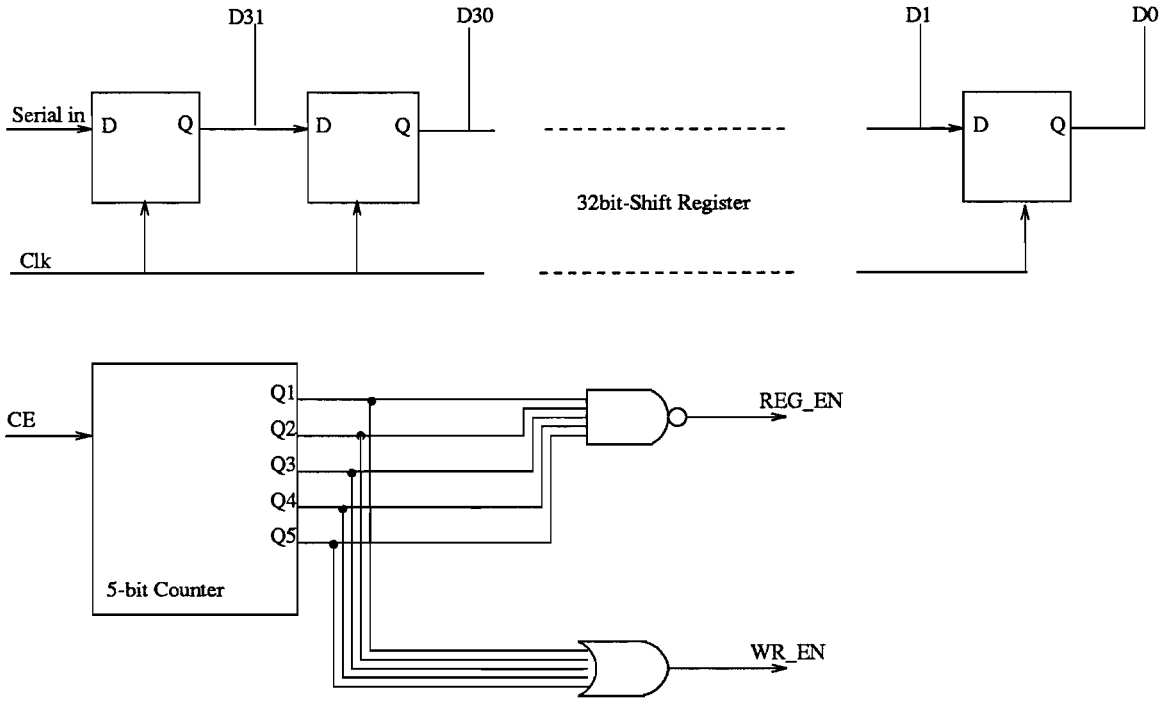



Figure 7: Logic Inside the FPGA

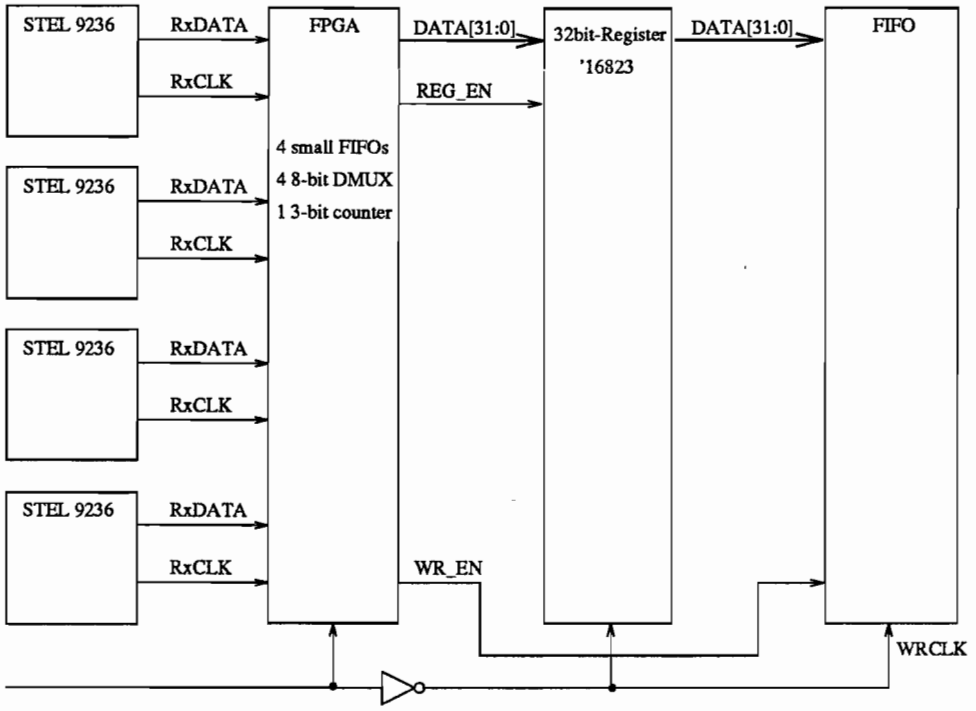



Figure 8: Block Diagram of the Demodulator and FIFO Interface, 2nd mode

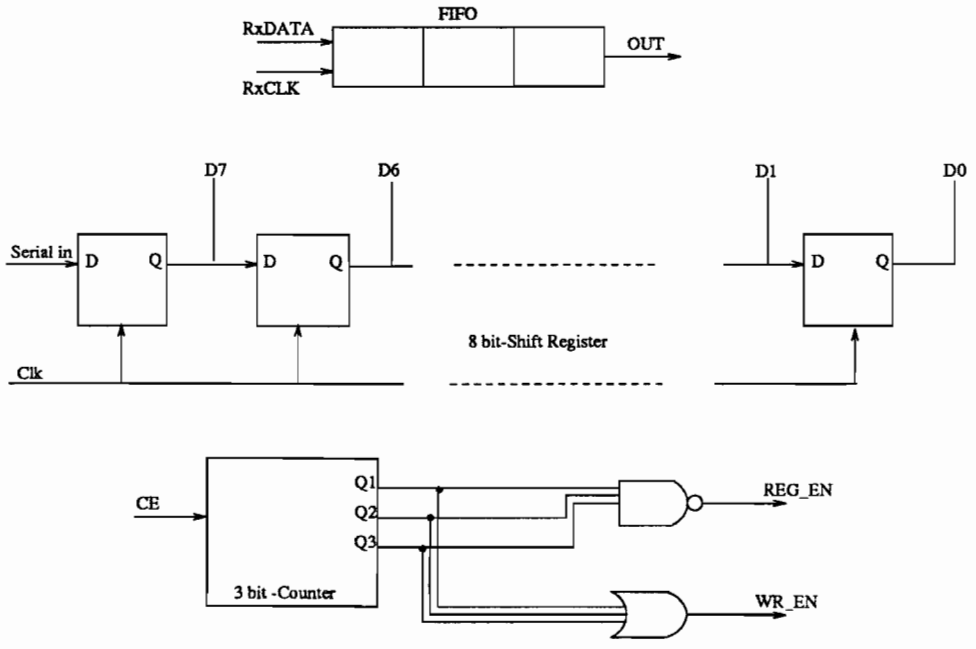



Figure 9: Logic Inside the FPGA, 2nd mode

```
// Program: Interface between STEL-9236 and FIFO (single data stream)
// Author: Y. Leo Liu
// Date: 6-22-95
// Version: 1.0
// All values in the create functions are character strings

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <stdio.h>
#include "lca.h"
#include "header.h"

int main(void)
{
// specify part
lca dmux("dmux","3195PQ208");
// the following two lines facilitate design – recommended usage
net NIL; // no net
char* NILS = "\0"; // a nil string
// implicit making of net - Long and Critical;
net Clock("Clock",'L','C');
// clock
net p2b("p2b"); // a net which connects pad to buffer
clkbuffer globclk(Clock,p2b,"GCLK");
// clbs
// The equation in the createclb should be enclosed in parantheses
// A set of parantheses can contain only one kind of operator other than
// a  (NOT)
// * – AND + – OR @ – XOR ! – NOT
char name[15], location[3], eq1[100], eq2[100], clk[20];
char input[15];
char output[20];
char tristate[20];
int i = 0;
// shift register
for(i = 2; i < 17; ++i)
{
sprintf(name, "regclb%d", i);
sprintf(location, "-1");
```

```
sprintf(eq1, "out%d = (out%d)", (2*i)-1, (2*i)-2 );
sprintf(eq2, "out%d = (out%d)", 2*i, (2*i)-1 );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
}
i = 1;
sprintf(name, "regclb%d", i);
sprintf(location, "-1");
sprintf(eq1, "out%d = (serial_in)", i);
sprintf(eq2, "out%d = (out%d)", 2*i, (2*i)-1 );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
// I/O pins
createiob("serial_input", "P48", "serial_in", NILS, NILS, NILS, NILS, NILS);
for (i = 1; i <= 32; ++i)
{
sprintf(name, "output_bit%d", i);
sprintf(location, "-1");
sprintf(output, "out%d", i);
createiob(name, location, NILS, output, NILS, NILS, NILS, NILS);
}
// clock and reset signals ... to be filled in... if necessary
// synchronous binary counter (00000b – 11111b)
sprintf(name, "counterclb1");
sprintf(location, "-1");
sprintf(eq1, "q1 = (q1 @ CE)" );
sprintf(eq2, "q2 = (q2 @ (q1 * CE))" );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
sprintf(name, "counterclb2");
sprintf(location, "-1");
sprintf(eq1, "CEO = (q3 * q2 * q1 * CE)" );
sprintf(eq2, "q3 = (q3 @ (q2 * q1 * CE))" );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
sprintf(name, "counterclb3");
sprintf(location, "-1");
sprintf(eq1, "q4 = (q4 @ CEO)" );
sprintf(eq2, "q5 = (q5 @ (q4 * CEO)" );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
createiob("count_en_input", "P49", "CE", NILS, NILS, NILS, NILS, NILS);
// control logic
sprintf(name, "reg_en_clb");
sprintf(location, "-1");
sprintf(eq1, "reg_en = (!(q1 * q2 * q3 * q4 * q5))" );
```

```
createclb(name, location, eq1, NILS, NILS, "ffx", "ffy", NILS, NILS);
sprintf(name, "fifo_en_clb");
sprintf(location, "-1");
sprintf(eq1, "fifo_en = (q1 + q2 + q3 + q4 + q5)" );
createclb(name, location, eq1, NILS, NILS, "ffx", "ffy", NILS, NILS);
createiob("reg_en_out", "P50", NILS, "reg_en", NILS, NILS, NILS, NILS);
createiob("fifo_en_out", "P51", NILS, "fifo_en", NILS, NILS, NILS, NILS);
dmux.writeXNF();
dmux.writeVHDL();
return 0;
}
```

```
// Program: Interface between STEL-9236 and FIFO (4 data streams)
// Author: Y. Leo Liu
// Date: 6-30-95
// Version: 1.0
// All values in the create functions are character strings

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <stdio.h>
#include "lca.h"
#include "header.h"

int main(void)
{
// specify part
lca dmux2("dmux2","3195PQ208");
// the following two lines facilitate design – recommended usage
net NIL; // no net
char* NILS = "\0"; // a nil string
// implicit making of net - Long and Critical;
net Clock("Clock",'L','C');
// clock
net p2b("p2b"); // a net which connects pad to buffer
clkbuffer globclk(Clock,p2b,"GCLK");
// clbs
// The equation in the createclb should be enclosed in parantheses
// A set of parantheses can contain only one kind of operator other than
// a  (NOT)
// * – AND + – OR @ – XOR ! – NOT
char name[15], location[3], eq1[100], eq2[100], clk[20];
char input[15];
char output[20];
char tristate[20];
int i = 0;
// four 1-byte shift registers
for(i = 2; i <= 4; ++i)
{
sprintf(name, "regclb%d", i);
sprintf(location, "-1");
```

```
sprintf(eq1, "out%d = (out%d)", (2*i)-1, (2*i)-2 );
sprintf(eq2, "out%d = (out%d)", 2*i, (2*i)-1 );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
}
for(i = 6; i <= 8; ++i)
{
sprintf(name, "regclb%d", i);
sprintf(location, "-1");
sprintf(eq1, "out%d = (out%d)", (2*i)-1, (2*i)-2 );
sprintf(eq2, "out%d = (out%d)", 2*i, (2*i)-1 );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
}
for(i = 10; i <= 12; ++i)
{
sprintf(name, "regclb%d", i);
sprintf(location, "-1");
sprintf(eq1, "out%d = (out%d)", (2*i)-1, (2*i)-2 );
sprintf(eq2, "out%d = (out%d)", 2*i, (2*i)-1 );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
}
for(i = 14; i <= 16; ++i)
{
sprintf(name, "regclb%d", i);
sprintf(location, "-1");
sprintf(eq1, "out%d = (out%d)", (2*i)-1, (2*i)-2 );
sprintf(eq2, "out%d = (out%d)", 2*i, (2*i)-1 );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
}
i = 1;
sprintf(name, "regclb%d", i);
sprintf(location, "-1");
sprintf(eq1, "out%d = (fifo1)", (2*i)-1);
sprintf(eq2, "out%d = (out%d)", 2*i, (2*i)-1 );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
i = 5;
sprintf(name, "regclb%d", i);
sprintf(location, "-1");
sprintf(eq1, "out%d = (fifo2)", (2*i)-1);
sprintf(eq2, "out%d = (out%d)", 2*i, (2*i)-1 );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
i = 9;
```

```
sprintf(name, "regclb%d", i);
sprintf(location, "-1");
sprintf(eq1, "out%d = (fifo3)", (2*i)-1);
sprintf(eq2, "out%d = (out%d)", 2*i, (2*i)-1 );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
i = 13;
sprintf(name, "regclb%d", i);
sprintf(location, "-1");
sprintf(eq1, "out%d = (fifo4)", (2*i)-1);
sprintf(eq2, "out%d = (out%d)", 2*i, (2*i)-1 );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
// I/O pins
createiob("serial_input1", "P48", "serial_in1", NILS, NILS, NILS, NILS, NILS);
createiob("serial_input2", "P47", "serial_in2", NILS, NILS, NILS, NILS, NILS);
createiob("serial_input3", "P46", "serial_in3", NILS, NILS, NILS, NILS, NILS);
createiob("serial_input4", "P45", "serial_in4", NILS, NILS, NILS, NILS, NILS);
for (i = 1; i <= 32; ++i)
{
sprintf(name, "output_bit%d", i);
sprintf(location, "-1");
sprintf(output, "out%d", i);
createiob(name, location, NILS, output, NILS, NILS, NILS, NILS);
}
// clock and reset signals ... to be filled in... if necessary
// the 4 small FIFOs for phase synchronization are not implemented yet
// synchronous binary counter (000b – 111b)
sprintf(name, "counterclb1");
sprintf(location, "-1");
sprintf(eq1, "q1 = (q1 @ CE)" );
sprintf(eq2, "q2 = (q2 @ (q1 * CE))" );
createclb(name, location, eq1, eq2, NILS, "ffx", "ffy", NILS, NILS);
sprintf(name, "counterclb2");
sprintf(location, "-1");
sprintf(eq2, "q3 = (q3 @ (q2 * q1 * CE))" );
createclb(name, location, NILS, eq2, NILS, "ffx", "ffy", NILS, NILS);
createiob("count_en_input", "P49", "CE", NILS, NILS, NILS, NILS, NILS);
// control logic
sprintf(name, "reg_en_clb");
sprintf(location, "-1");
sprintf(eq1, "reg_en = (!(q1 * q2 * q3))" );
createclb(name, location, eq1, NILS, NILS, "ffx", "ffy", NILS, NILS);
```

```
sprintf(name, "fifo_en_clb");
sprintf(location, "-1");
sprintf(eq1, "fifo_en = (q1 + q2 + q3)" );
createclb(name, location, eq1, NILS, NILS, "ffx", "ffy", NILS, NILS);
createiob("reg_en_out", "P50", NILS, "reg_en", NILS, NILS, NILS, NILS);
createiob("fifo_en_out", "P51", NILS, "fifo_en", NILS, NILS, NILS, NILS);
dmux2.writeXNF();
dmux2.writeVHDL();
return 0;
}
```

;PLDShell program for the first 22V10

Title Data Transfer Controller
Pattern pds
Revision 1
Author Leo and Thanh
Company TISL
Date July 12, 1995

CHIP DTC1 N85C22V10

;Pin Assignment

PIN 1 NC
PIN 2 CLK
PIN 3 Fill
PIN 4 Read
PIN 5 Q2
PIN 6 Q1
PIN 7 NC
PIN 8 NC
PIN 9 NC
PIN 10 NC
PIN 11 NC
PIN 12 NC
PIN 13 NC
PIN 14 GND
PIN 15 NC
PIN 16 NC
PIN 17 Stop
PIN 18 RdRom
PIN 19 A0
PIN 20 A1
PIN 21 A3
PIN 22 NC
PIN 23 A2
PIN 24 A5
PIN 25 A4
PIN 26 WrRam
PIN 27 RdRam
PIN 28 VCC

```
STRING COND1 ' ((/Q2 * /Q1 + /Q2 * Q1 + Q2 * /Q1) * Fill * /Read) '
STRING COND2 ' ((Q2 * Q1) * Fill * /Read) '
STRING COND3 ' ((/Q2 * /Q1 + /Q2 * Q1 + Q2 * /Q1) * /Fill * Read) '
STRING COND4 ' ((Q2 * Q1) * /Fill * Read) '

;Boolean equations for state variables (addresses)

EQUATIONS

A5 := COND1 * A5
+ COND2 * (A5 :+: (A4 * A3 * A2 * A1 * A0))
+ COND3
+ COND4
A4 := COND1 * A4
+ COND2 * (A4 :+: (A3 * A2 * A1 * A0))
+ COND3
+ COND4
A3 := COND1 * A3
+ COND2 * (A3 :+: (A2 * A1 * A0))
+ COND3 * ((A5 + A4 + A3 + A2 + A1 + A0) * A3)
+ COND4 * ((A5 + A4 + A3 + A2 + A1 + A0) * (A3 :+: (A2 * A1 * A0)))
A2 := COND1 * A2
+ COND2 * (A2 :+: (A1 * A0))
+ COND3 * ((A5 + A4 + A3 + A2 + A1 + A0) * A2)
+ COND4 * ((A5 + A4 + A3 + A2 + A1 + A0) * (A2 :+: (A1 * A0)))
A1 := COND1 * A1
+ COND2 * (A1 :+: A0)
+ COND3 * ((A5 + A4 + A3 + A2 + A1 + A0) * A1)
+ COND4 * ((A5 + A4 + A3 + A2 + A1 + A0) * (A1 :+: A0))
A0 := COND1 * A0
+ COND2 * /A0
+ COND3 * ((A5 + A4 + A3 + A2 + A1 + A0) * A0)
+ COND4 * ((A5 + A4 + A3 + A2 + A1 + A0) * /A0)


Stop = A5 * A4 * /A3 * /A2 * /A1 * /A0 * Fill
/RdRom = /Q2 * /Q1 + Q2 * Q1 + Fill * Read + /Fill * /Read
/WrRam = /Q2 + Q1 + /Fill + Read
/RdRam = /Q2 + Q1 + Fill + /Read
```

```
SIMULATION

VECTOR ADDR := [A5, A4, A3, A2, A1, A0]
TRACE_ON CLK Fill Read Q2 Q1 A5 A4 A3 A2 A1 A0 /RdRom /WrRam /RdRam Stop
SETF /Q2 /Q1 /CLK /Fill Read /A5 /A4 /A3 /A2 /A1 /A0 /Stop /RdRom /RdRam /WrRam
CLOCKF CLK
FOR i:=1 TO 49 DO
BEGIN
SETF /Q2 /Q1
CLOCKF CLK
SETF /Q2 Q1
CLOCKF CLK
SETF Q2 /Q1
CLOCKF CLK
SETF Q2 Q1
CLOCKF CLK
END
;SETF /Q2 /Q1
;CLOCKF CLK
;SETF /Q2 /Q1
;CLOCKF CLK
;FOR j:= 1 TO 10 DO
;BEGIN
;CLOCKF CLK
;END
;SETF /Fill
;CLOCKF CLK
;FOR j:= 1 TO 10 DO
;BEGIN
;CLOCKF CLK
;END
TRACE_OFF
```

;PLDShell program for the second 22V10

Title Data Transfer Controller
Pattern pds
Revision 1
Author Leo and Thanh
Company TISL
Date July 12, 1995

CHIP DTC2 N85C22V10

;Pin Assignment

PIN 2 CLK
PIN 3 Fill
PIN 4 Read
PIN 5 Stop
PIN 6 A5
PIN 7 A4
PIN 8 NC
PIN 9 NC
PIN 10 NC
PIN 11 NC
PIN 12 NC
PIN 13 NC
PIN 14 GND
PIN 15 NC
PIN 16 NC
PIN 17 NC
PIN 18 NC
PIN 19 CE1
PIN 20 CE2
PIN 21 CE3
PIN 22 NC
PIN 23 CE
PIN 24 Q2
PIN 25 Q1
PIN 26 CLKOUT
PIN 27 NC
PIN 28 VCC

```
;2-bit binary synchronous counter

STATE
MOORE_MACHINE
DEFAULT_BRANCH STATE1

;State Assignments

STATE1 = /Q2 * /Q1
STATE2 = /Q2 * Q1
STATE3 = Q2 * /Q1
STATE4 = Q2 * Q1

;State transitions

STATE1 := ENABLE -> STATE2
STATE2 := ENABLE -> STATE3
STATE3 := ENABLE -> STATE4
STATE4 := ENABLE -> STATE1

CONDITIONS

ENABLE = Fill * /Stop + Read * /Stop

EQUATIONS

/CE1 = A5 + A4 + Stop + Read + /Fill
/CE2 = A5 + /A4 + Stop + Read + /Fill
/CE3 = /A5 + A4 + Stop + Read + /Fill
/CE = /A5 + /A4 + /Read
CLKOUT = /CLK * Q2 * /Q1 * A5 * A4

SIMULATION

TRACE_ON CLK Q2 Q1 CLKOUT CE
SETF Read /Fill A5 A4 /Stop /CLK /Q1 /Q2
FOR j := 1 TO 30 DO
BEGIN
CLOCKF CLK
END
TRACE_OFF
```

# 4 Beamforming

## 4.1 Introduction

This section describes the circuit which will be used to test the downloading of lookup tables for the Transmit Beamforming Cards. The output of the circuit will also simulate the input to the Quadrature Modulator on the Transmit Beamforming Cards. Control logic is implemented in two 22V10s. Data are transferred from a 27C512 EPROM to 3 CY7B135 SRAMs sequentially to fill lookup tables and then read out in parallel from the SRAMs. Two HI5721 D/A converters receive the data, which simulate the I and Q combinations, and feed the output to the Quadrature Modulator.

## 4.2 Design & Implementation

Figure 10 shows the block diagram of the circuit. The basic control logic provides read, write, chip enable, output enable, and address lines for the EPROM and SRAMs, and clock signal for the D/A converters. Due to the timing constraints on these control signals, each data transfer is accomplished in 4 clock cycles. For downloading lookup tables into the SRAMs(FILL mode), the 4 steps are: assert address -> read from EPROM -> write to SRAM -> negate signals; for transferring data from the SRAMs to the D/A converters(READ mode), the 4 steps are: assert address -> read from EPROM -> read SRAMs and write to the DACs -> negate signals. However, the address is held constant in the 4 clock cycles.

A simple 2-bit binary counter (00b – 11b) is used to keep track of the 4 cycles. The read and write signals, specifically, /RdRom, /WrRam, and /RdRam, are implemented as functions of the states of the 2-bit counter. The 4 steps for each data transfer are fixed and repeat themselves every 4 clock cycles.

A state machine is used to provide EPROM addresses for writing to and reading from the

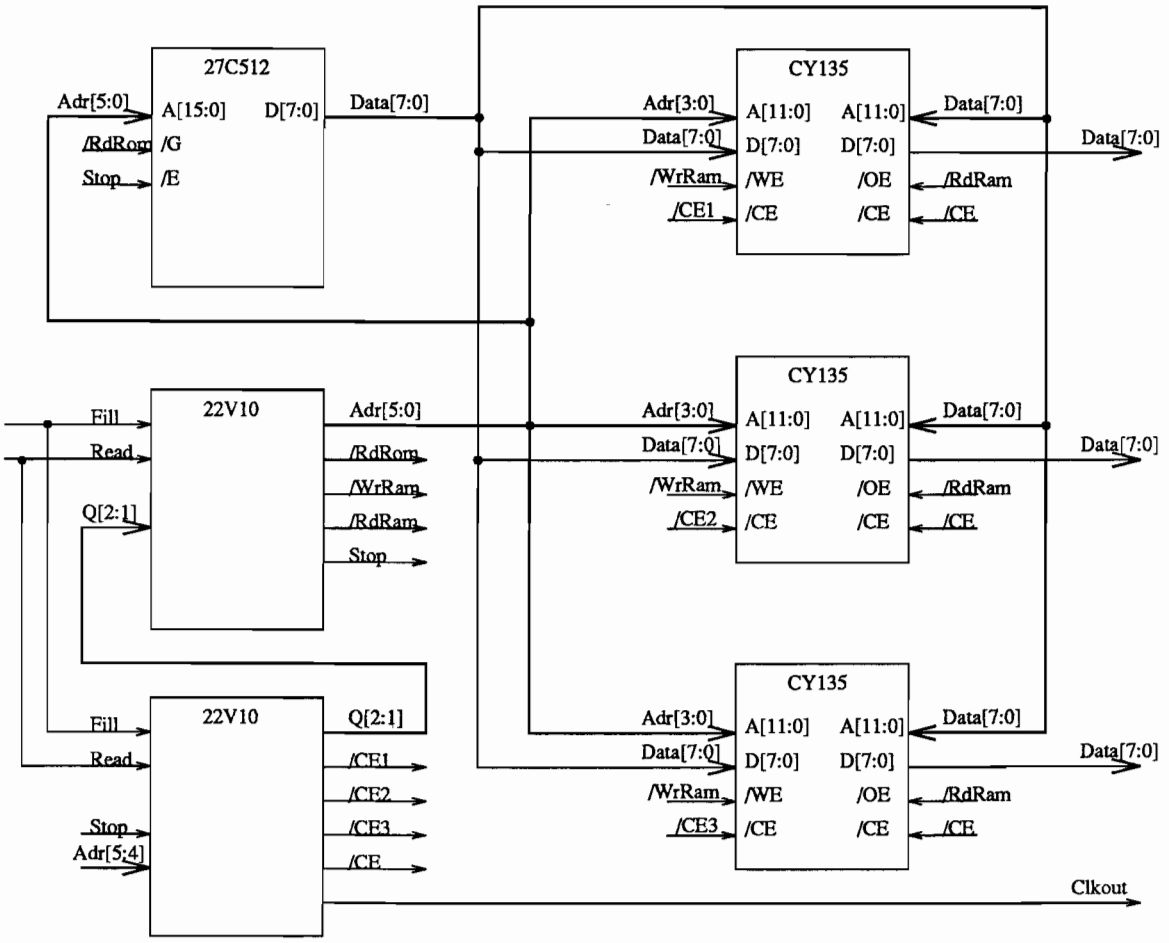
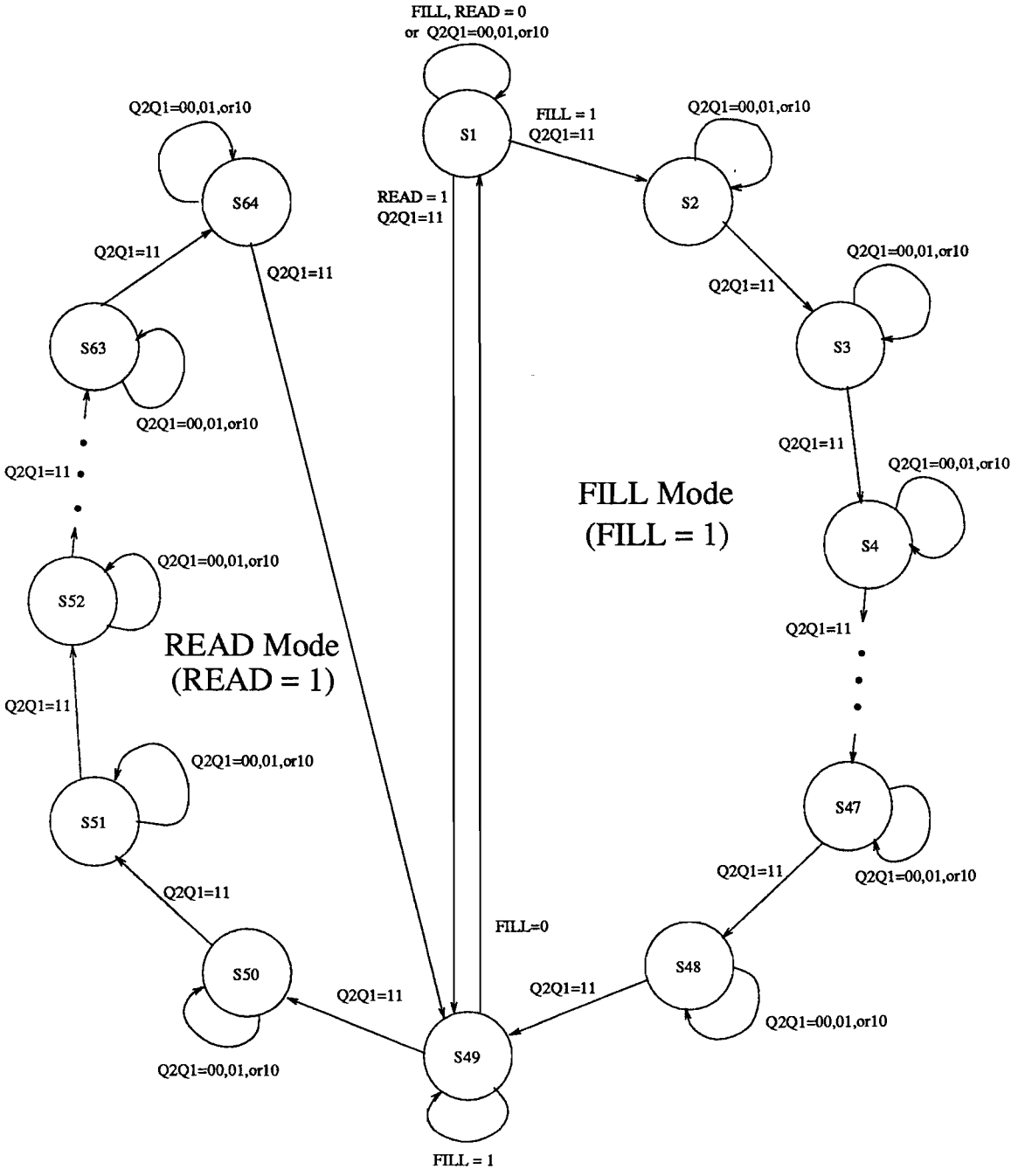



Figure 10: Block Diagram of Transmit Beamforming Lookup Table

SRAMs. It changes state every 4 clock cycles. The state transition diagram is shown in Figure 11. There are 64 states. States 1 through 48 are used for filling the SRAMs, and states 49 through 64 are used when reading from the SRAMs. This implementation determines the use of address space of the EPROM: 000000b – 101111b is for the data that will be transferred to the SRAMs, and 110000b – 111111b is for the SRAM addresses. When the FILL switch is turned on, the state machine goes from state 1 to 48 and finally enters the hold state 49. It stays in the hold state until the FILL switch is turned off. When the READ switch is turned on, the state machine jumps from the power-up state (S1) to state 49 at the rising clock edge when it's supposed to change state; i.e., when the 2-bit counter changes to 00b. Reading from the SRAMs is a continuous operation, and state machine counts from 110000b to 111111b and repeats indefinitely until READ is off.

The other control signals include output enable and chip enables. A STOP signal is asserted when FILL is completed to put the EPROM in standby mode and disable the chip enables of the SRAMs. There are 4 chip enables: 3 for the left ports of the SRAMs and 1 for all 3 right ports. They are functions of A5, A4, STOP, READ, and FILL. The clock signal for the D/A converters is also implemented in a 22V10 with its remaining capabilities. It is the complement of the circuit clock conditioned by the state of the 2-bit counter and current circuit mode. The signal is asserted once every 4 clock cycles and when the SRAM data are stable on the data bus.

It is assumed that writing to the SRAMs (FILL mode) and reading from them (READ mode) are never simultaneous. That is, the FILL and READ switches cannot both be on at the same time. Otherwise the circuit will stay idle and won't fill or read. It is also assumed that the clock rate is around 2MHz in this design. If the clock rate is too high, the long access time of the EPROM may require additional clock cycles for each data transfer.




Note: No matter where the state is in the Read mode, it will return to state S1 if the Read signal is turned off.

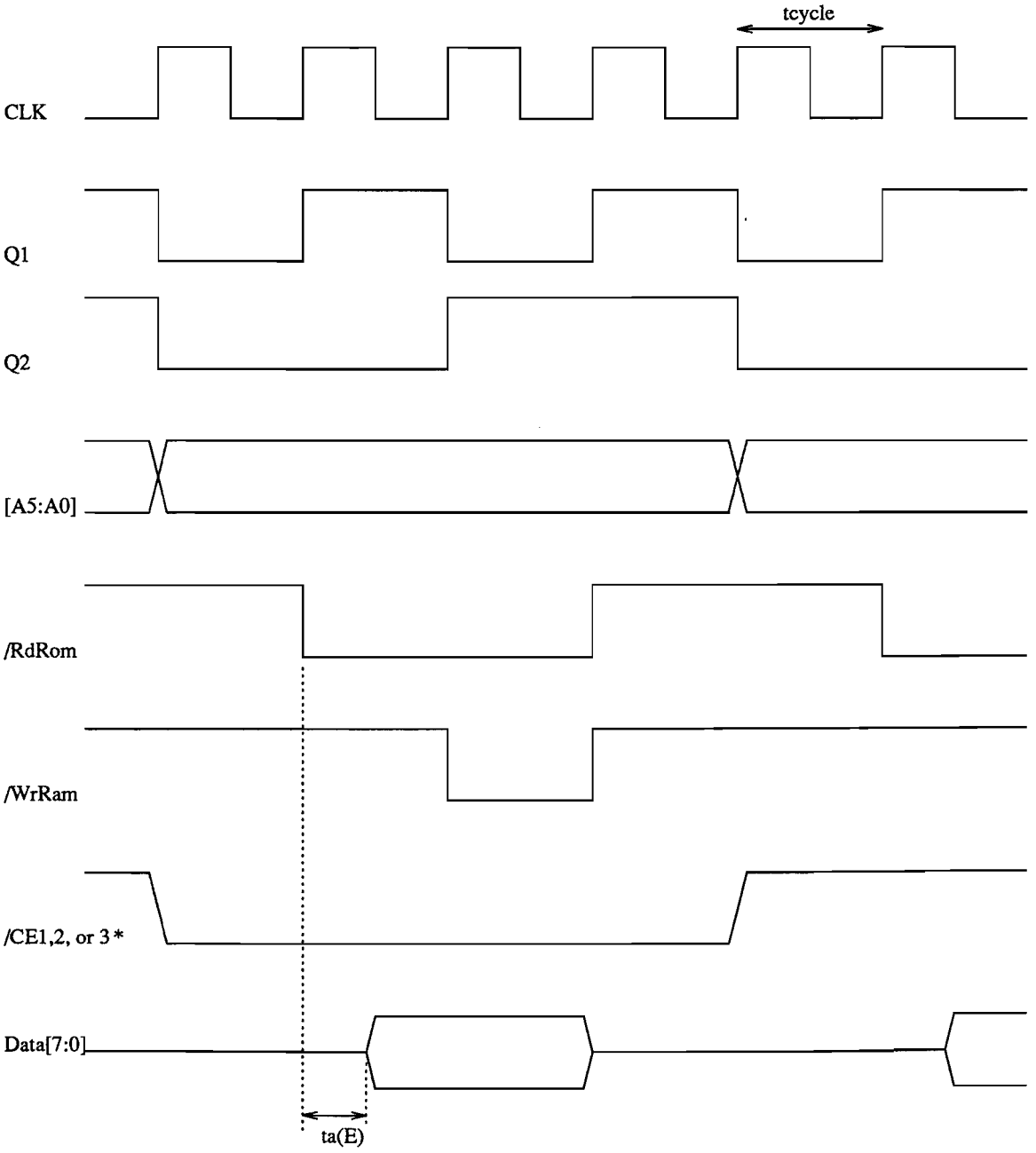
Figure 11: State Diagram of the State Machine implemented in two 22V10s

## 4.3 Operation

Two external switches provide the FILL and READ inputs to the circuit and control its operation mode. Both inputs are active HIGH and should be logic-0 when the circuit is not operating. Note that the two switches cannot both be ON at the same time because fill and read operations require different EPROM addresses while the state machine is designed to support one address at a time.

If the FILL switch is turned on, the circuit starts from the power-up state(S1) and increments to the next state every 4 clock cycles. Thus, each state(EPROM address) is held constant in the 4 clock cycles of each data transfer. The circuit enters the hold state(S49) and stays there when the fill operation is completed. The negation of the FILL input disables the 2-bit counter and resets the state machine. During the fill operation, the state machine functions as a 6-bit(A5:A0) binary counter at 1/4 of the clock frequency. For example, with a 2MHz clock, A0 would be oscillating at 500kHz, and A1 at 250kHz, ... etc. Figure 12 shows a typical FILL cycle.

After the FILL switch is turned off, the READ input can be asserted to read the SRAMs. In this mode, the circuit starts from the power-up state(S1) and jumps to state 49. It then stays in the loop between S49 and S64 and changes state every 4 clock cycles, as controlled by the simple 2-bit counter. Note that the actual read operation does not start until the circuit reaches state 49 because the /CEs to the right ports of the SRAMs are not asserted in the power-up state. In the READ mode the circuit is also a 6-bit counter, but with a different counting range. The other difference is that the read operation does not stop until the READ switch is turn off. The timing diagram for the READ mode is shown in Figure 13.

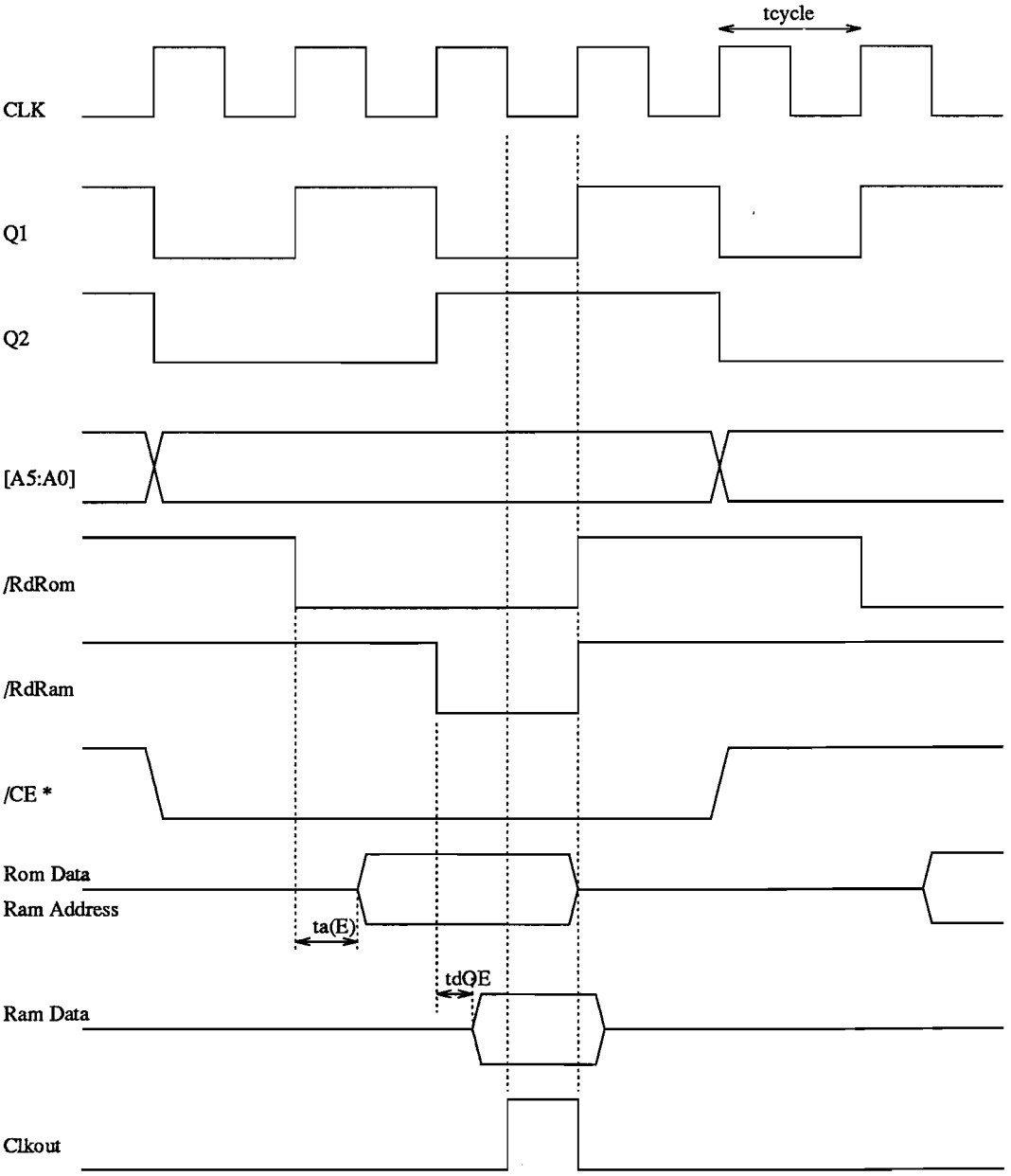



* : During the filling of one RAM, the chip select signal for that particular RAM is unchanged.

tcycle = 500ns

ta(E) max = 250 ns , Access time from chip enable.

Figure 12: Timing Diagram of a Fill Cycle



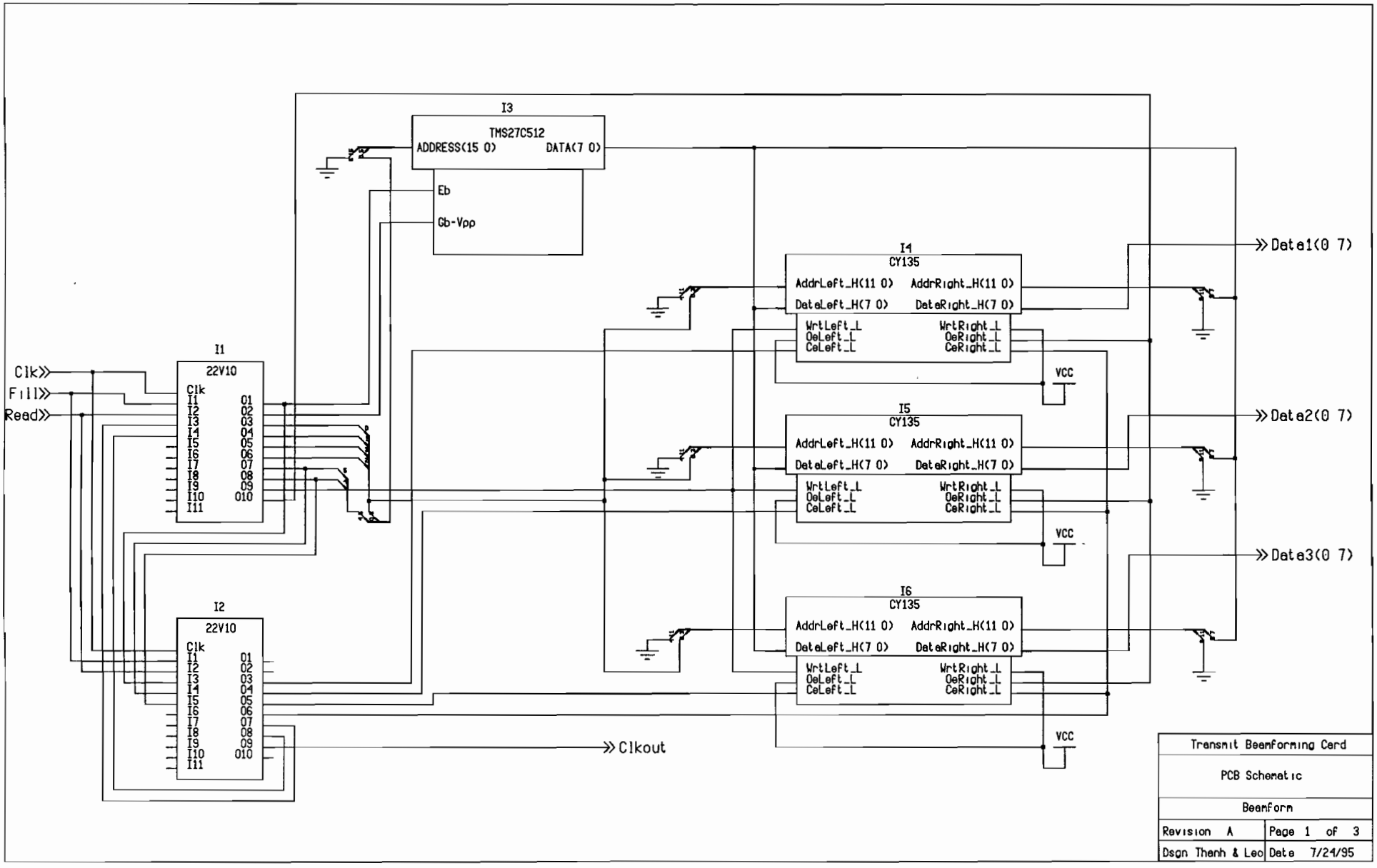

*: All three RAMs are selected on the right ports at the same time in the Read mode.

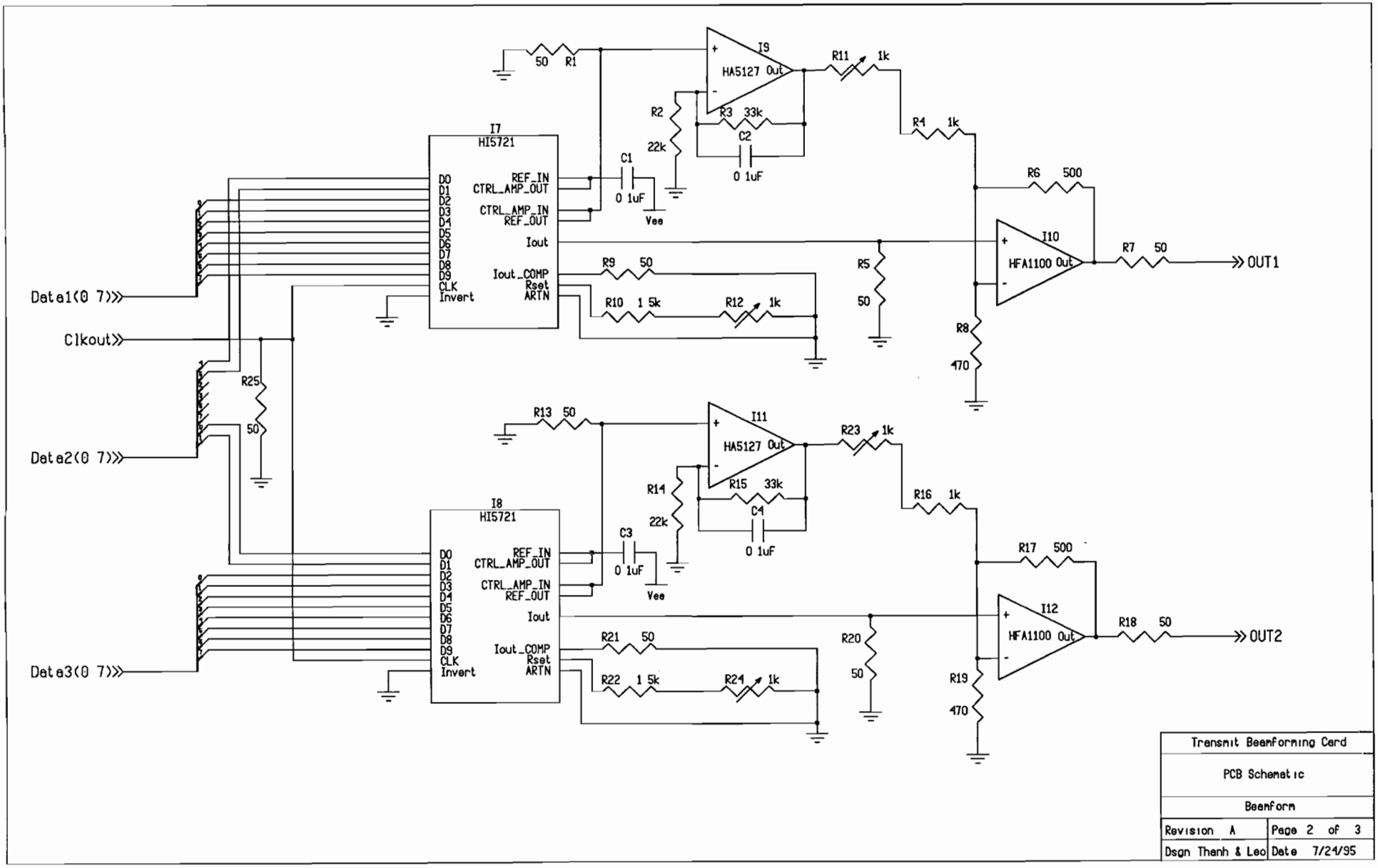
tcycle = 500ns

ta(E) max = 250 ns , Access time from chip enable.

tdOE max = 20ns

Figure 13: Timing Diagram of a Read Cycle


I3
TMS27C512
ADDRESS(15 0)
DATA(7 0)
Eb
Gb-Vpp
I4
CY135
AddrLeft_H(11 0)
AddrRight_H(11 0)
DataLeft_H(7 0)
DataRight_H(7 0)
WrtLeft_L
OeLeft_L
CeLeft_L
WrtRight_L
OeRight_L
CeRight_L
VCC
I5
CY135
I6
CY135
I1
22V10
I2
22V10
Clk
Fill
Read
Clkout
Data1(0 7)
Data2(0 7)
Data3(0 7)
Transmit Beamforming Card
PCB Schematic
Beamform
Revision A
Page 1 of 3
Dsgn Thanh & Leo
Date 7/24/95


50 R1
I9
HA5127 Out
R11 1k
R2
22k
R3 33k
C2
0 1uF
R4 1k
I7
HI5721
C1
0 1uF
Vee
R6 500
D0
D1
D2
D3
D4
D5
D6
D7
D8
D9
CLK
Invert
REF_IN
CTRL_AMP_OUT
CTRL_AMP_IN
REF_OUT
Iout
Iout_COMP
Rset
ARTN
I10
HFA1100 Out
R7 50
OUT1
R9 50
R10 1 5k
R12 1k
R5
50
R8
470
Data1(0 7)
Clkout
Data2(0 7)
Data3(0 7)
R25
50
R13 50
I11
HA5127 Out
R23 1k
R14
22k
R15 33k
C4
0 1uF
R16 1k
I8
HI5721
C3
0 1uF
Vee
R17 500
I12
HFA1100 Out
R18 50
OUT2
R21 50
R22 1 5k
R24 1k
R20
50
R19
470
Transmit Beamforming Card
PCB Schematic
Beamform
Revision A
Page 2 of 3
Dsgn Thanh & Leo
Date 7/24/95