University of Arkansas, Fayetteville
ScholarWorks@UARK

Theses and Dissertations

5-2012

Silicon Germanium SRAM and ROM Designs for Wide Temperature Range Space Applications

Matthew Barlow
University of Arkansas, Fayetteville

Follow this and additional works at: http://scholarworks.uark.edu/etd

Part of the VLSI and Circuits, Embedded and Hardware Systems Commons

Recommended Citation
Barlow, Matthew, "Silicon Germanium SRAM and ROM Designs for Wide Temperature Range Space Applications" (2012). *Theses and Dissertations*. 362.
http://scholarworks.uark.edu/etd/362

This Thesis is brought to you for free and open access by ScholarWorks@UARK. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of ScholarWorks@UARK. For more information, please contact scholar@uark.edu, ccmiddle@uark.edu.

SILICON GERMANIUM SRAM AND ROM DESIGNS FOR WIDE TEMPERATURE RANGE SPACE APPLICATIONS

SILICON GERMANIUM SRAM AND ROM DESIGNS FOR WIDE TEMPERATURE RANGE SPACE APPLICATIONS

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Electrical Engineering

By

Matthew Barlow
University of Arkansas
Bachelor of Science in Electrical Engineering, 2007

May 2012
University of Arkansas

Abstract

This thesis presents a design flow from specifications and feature requirements to embeddable blocks of SRAM and ROM designs from 64 bytes to 1 kilobyte that are suitable for lunar environments. The design uses the IBM SiGe 5AM BiCMOS 0.5 micron process for a synchronous memory system capable of operating at a clock frequency of 25 MHz. Radiation mitigation techniques are discussed and implemented to harden the design against total ionizing dose (TID), single-event upset (SEU), and single-event latch-up (SEL). The memory arrays are also designed to operate over the wide temperature range of -180 ℃ to 125 ℃. Design, simulation, and physi cal layout are evaluated throughout the process. Modeling of the memory arrays for static timing analysis (STA) is done to allow easy integration of the design into a typical RTL design flow. System simulation data is incorporated into block-level simulations to validate the memory timing models. Hardware testing over five iterations of the memory array designs demonstrates the functionality of the design as well as validates the design specifications.

This thesis is approved for recommendation
to the Graduate Council.

Thesis Director:

____

Dr. H. Alan Mantooth

Thesis Committee:

____

Dr. Jia Di

____

Dr. Scott Smith

Thesis Duplication Release

I hereby authorize the University of Arkansas Libraries to duplicate this thesis when needed for research and/or scholarship.

Agreed ____

Matthew Barlow

Refused ____

Matthew Barlow

Acknowledgements

The author is grateful for the guidance, support, and dedication of the thesis committee, consisting of Dr. H. Alan Mantooth, Dr. Jia Di, and Dr. Scott Smith. The work described here is a part of the Silicon-Germanium Integrated Electronics for Extreme Environments project. The experience shared by Richard Berger, Adam Bumgarner, and Jason Ross of helped this work get started on the right track and maintain momentum. Additionally, the author would like to thank Guoyuan Fu, Brent Hollosi, Chris Lee, and Aaron Arthurs for their parts helping out with the design, simulation, and testing involved with the entire project

Dedication

I would like to dedicate this thesis to my wife Kathryn, my daughter Abigail, and my son Ethan.

Table of Contents

List of Figures

List of Tables

Chapter 1 – Introduction and Overview

Data storage is a crucial part of almost all digital systems. Without some form of memory, circuits are limited in their responses to a set of input conditions. The design of reliable, compact, high-performance memories can present a significant design challenge. Memories designed for extreme environment applications, such as for operation in the rigors of space, present a wholly unique set of constraints and challenges that must be met. In this work the design of memory arrays for application to the lunar environment has been undertaken.

1.1 Digital Data Storage for Microcontroller Applications

Digital data storage can be put into special arrays of optimized structures to improve the density and performance of a system. The traditional data storage elements such as the latch and flip-flop are simplified to store a voltage in an array. These simplifications either store a single voltage dynamically, or with feedback reinforcing the stored voltage.

Memory arrays can be used to store many different types of digital data. Microcontrollers and computers are primary users of memory arrays, although custom digital systems can also use memory. Typical microcontrollers such as the 8051 have program memory where in-order executable operation codes, or programs, are stored [1]. Depending on the architecture of the microcontroller core, the program memory may share address space with the scratchpad or data storage memory, which follows the Von Neumann architecture [2]. The other dominant processor architecture type is the Harvard architecture, which has separate address spaces for program memory and data memory, and is used by the 8051 microcontroller family [2].

The first few lines executed after the microcontroller powers on or resets are typically stored in a non-volatile memory array such as a ROM. In simple memory architectures, a single ROM bank is sufficient to store the instructions required to accomplish the intended task. Larger systems, such as a computer, or systems requiring greater flexibility may start with an initial ROM subroutine which then loads executable code into volatile RAM from a disk or remote host.

Data acquired from sensors or other measurements can be stored in memory arrays. Many operations benefit from large sets of measurements or periodic measurements. A simple use of such data is for playback or historical logging, such as with a digital storage oscilloscope, where a large memory array is filled with sequential digitized samples from an analog input that is displayed to a user afterwards. Other uses include temporary storage of measured data until the data can be transmitted to a master controller or a permanent storage device.

For any mathematical operation more complicated than a binary operation, most processors require an intermediate step. Typical microcontrollers, such as the 8051, require the calculation and storage of intermediate terms for calculations as simple as adding three numbers. Many simple microcontrollers lack complicated instructions found in computer processors such as floating point operations, multiplication, or division. These complicated instructions require multi-step algorithms with different state variables to complete. From a programming perspective, dynamic variables must be stored in RAM.

1.2 Project Overview

The NASA Silicon Germanium Exploration Technology Development Program (ETDP) team was formed to do research on robust electronics for extreme environments such as space. During the first year of the project, the ETDP team embarked upon the challenge of creating an integrated replacement for a remote health node (RHN) design by BAE Systems as a demonstration of its capabilities. The RHN module was originally designed to be placed throughout the proposed X-33 space plane as a sensor data acquisition device [3]. This module contained a large number of parts within a warm box to maintain a suitable temperature during operation. The ETDP project aim is to develop a two-chip solution that essentially replicates the functionality of the original RHN. This new design consists of a mixed-signal sensor interface chip, the Remote Sensing Interface (RSI), and a digital control chip, the Remote Digital Controller (RDC). These two chips, along with a voltage regulator and oscillator, replicate the functionality of the original RHN. The shielding and heating of the warm box are not necessary for the more robust proposed ICs, and thus even greater power, size, and weight savings over the original design are achieved. The proposed use of this device is to function over long-term lunar missions. The temperatures on the moon can reach values as low as -180 ℃ duri ng a lunar night, and even down to -230 ℃ in permanently shadowed polar craters. Temperatures can reach up to +120 ℃ in direct lunar sunlight [4]. The elimination of environmental controls requires the electronics to function over this significantly wider temperature range.

1.3 Silicon Germanium Process

The IBM SiGe5AM BiCMOS process has been used for the NASA ETDP project. This process combines a 0.5 μm minimum gate length CMOS process with a silicon germanium hetero-junction bipolar transistor. A local interconnect layer is available in addition to 4 layers of metal. The top layer of metal is a thick analog metal layer [5].

Several challenges arise when dealing with an extreme temperature range. Validation of any design before manufacture is typically done through simulation. However, the models supplied with the process design kit only cover a subset of the required temperature range, from -55 ℃ to +125 ℃. Transistor models for the extended temperature range of -180 ℃ to -55 ℃ were developed using measured data, allowing verification of designs through simulation over the desired temperature range of -180 ℃ to +125 ℃ [6].

Additional issues arise at low temperatures, where hot carrier injection can cause a gradual degradation of short channel transistors over time. As temperatures decrease, electrons are more likely to become trapped in the gate oxide [7]. This trapped charge decreases the effective V_t of the device, eventually making the device inoperable. Experimental data shows that this is primarily an issue for the NFET devices; PFET devices are largely unaffected. Two different techniques can be used to mitigate this effect at temperatures down to -180 ℃, and bo th aim to reduce the density of the electric field in the transistor. The first method is to reduce the maximum voltage experienced by the transistor. In this application, the default process voltage of 3.3 V would need to be reduced to 2.5 V to provide adequate protection. The other method for mitigating the hot carrier injection effect is to maintain the process voltage of 3.3 V, and increase the channel length from the minimum of 0.5 μm up to 1.0 μm for all NFET devices [8]. To meet the project power supply requirement of 3.3 V, all NFETs in the RDC have gate lengths of 1.0 μm or longer.

Radiation effects are also major concerns for a given design. On a transistor device level, the total ionizing dose received by a device can degrade the performance of the device. Experimental data shows that NFET devices are particularly vulnerable to a decrease in V_t due to trapped charges in the shallow trench isolation around the transistor gate accumulated during exposure to radiation [4]. Other radiation effects are typically caused by a single heavy-ion strike on the circuit. When an ion strikes a transistor, the deposited charge may temporarily change the output of the circuit until the charge is dissipated. This effect may be a temporary glitch on a data signal, or it may change the stored value in a memory element. Depending on the nature of the single event upset, the result can cause a disruption in the circuit operation [9]. Mitigation of single event upsets has obvious implications in the design of memories, and requires both logical and physical design considerations.

Another single event effect which must be considered is latch-up. This is caused by the activation of the parasitic bipolar transistors inherent in the bulk CMOS process [10]. A heavy ion strike to the substrate can cause the activation of one of the many parasitic transistors formed by PNP/NPN junctions formed between substrate, wells and contacts. If one such transistor turns on, it can turn on another parasitic transistor, which then results in the latch-up chain reaction. Once activated, latch-up cannot be deactivated without removing the power supply from the circuit. Latch-up prevents the circuit from operating, and can cause excessive current draw resulting in device failure. In order to reduce the risk of latch-up in a CMOS process, the circuit designer must add guard rings, well contacts, and substrate contacts. These layout structures decrease the resistance between the base of the parasitic NPN/PNP and the power supply rails, making it harder for the pair to activate.

Chapter 2 – Background on Memory Design

Memory comes in all shapes and sizes. Depending on the intended use, there are several design parameters to emphasize. From simple one-bit designs to complex, high-density arrays, the tradeoffs focus on power, speed, and area.

2.1 Latch

The latch is the simplest form of digital data storage [11]. A latch holds a single binary value or bit of data at a time. As shown in Fig. 2.1, a latch has two inputs and one output: a data input, an enable input, and a data output. The enable input determines the mode of operation of the latch, either transparent or latched. In the transparent mode, the data input is transferred to the output instantly. Changing the mode to the latched mode fixes the output data to the last input value while the latch was in the transparent mode. This data is now “latched” and all further inputs to the data input are ignored until the latch switches to the transparent mode.

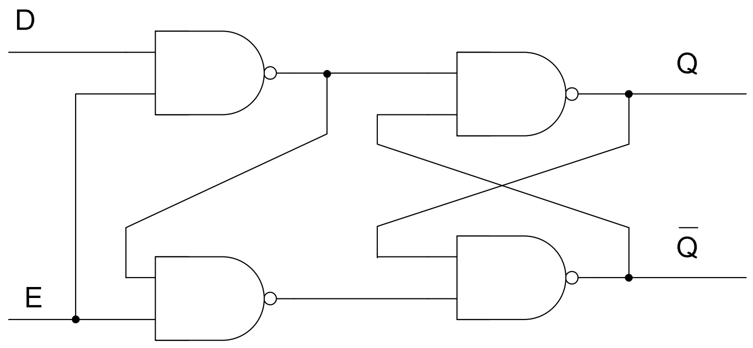

Fig. 2.1 - Latch with enable

2.2 Flip-Flops

Flip-flops are more complicated data storage circuits. The primary difference between flip-flops and latches is that while latches pass data from the input to the output whenever the enable signal is asserted, flip-flops only pass data at the instant that the clock changes [11]. Typical flip-flops trigger on a single edge of the clock, either the rising or the falling edge. Dual edge triggered flip-flops exist as well, but these are less common. A simple way to create a flip-flop, shown in Fig. 2.2, is to chain two latches in series and invert the enable signal on one of the latches. With flip-flops, finite state machines are easily designed with a storage register and a logic function.

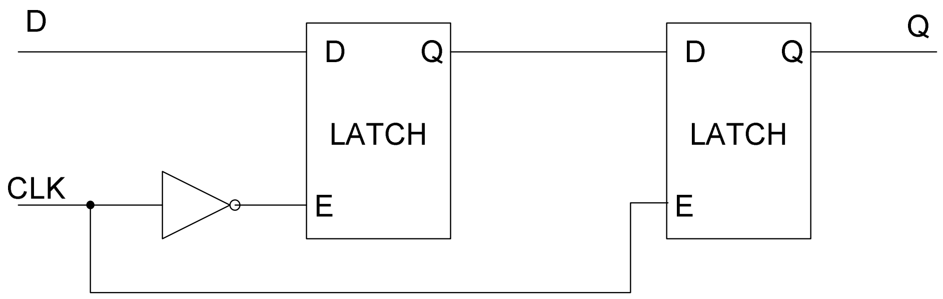

Fig. 2.2 - Master-slave flip-flop

2.3 Memory Arrays

For large quantities of data storage, arrays are often used to provide density savings. The tradeoff between a memory array and a single memory element is density versus accessibility. Memory arrays are designed to access only a subset of their elements at a time through a port. Multiple-port memories allow reading and writing simultaneously to different addresses. Small arrays may use flip-flops or latches as the storage elements due to easy design and signaling requirements, but specialized memory arrays provide the best performance and storage density. Discrete flip-flops and latches have an advantage in that placing the single-bit storage gates is significantly easier in a design.

2.3.1 Dynamic Random Access Memory

One form of memory array is the Dynamic Random Access Memory (DRAM). DRAM is dynamic due to the fact that there is no active restoring feedback. Typically, data is stored as a charge on a capacitor. A single transistor is used to regulate access to the capacitor [11]. This capacitor and transistor bit shown in Fig. 2.3 is arranged with other bits in a rectangular grid. Word lines run horizontally across the array, connecting every gate of each transistor in a single row. Vertical bit-lines connect the drains of every transistor in a column, and the source of each transistor is connected to a capacitor. Each bit-line is connected to a sense amplifier, pre-charge, and write circuitry. Each word-line is connected to a driver that is connected to an address decoder.

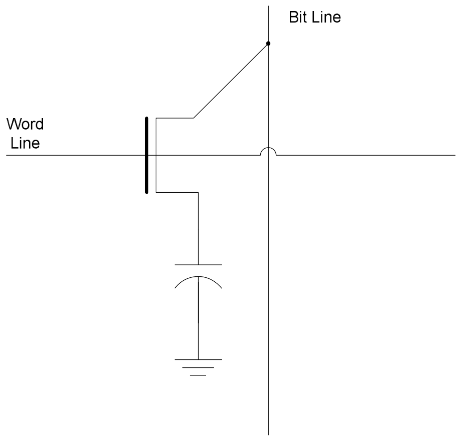

Fig. 2.3 - DRAM bit-cell

At the beginning of a DRAM memory cycle, each bit-line is pre-charged to ½ V_{DD}. The desired row is asserted and all bits in that row pull the bit-line towards either logic ‘1’, or logic ‘0’. Because the bit-line capacitance is usually much larger than the capacitance of a single bit, the actual voltage change is small. This small change is amplified by a sense amplifier, which decreases the time required to detect the stored voltage. The process of reading data from a DRAM bit destroys the data contents of the entire row. This requires each row to be temporarily stored and written back. If data is to be written to an address, then this overrides the data stored by the sense amplifiers. The final stage of a DRAM cycle is to write the row data back into the capacitors. This is accomplished by driving each bit-line to either logic ‘1’ or ‘0’. After the capacitors have been appropriately charged, the word-line is deactivated.

The charge on the capacitor degrades over time to a point when it is insufficient to register a large enough voltage difference on the bit-lines. This necessitates a periodic refreshing of each row in a DRAM array in addition to typical read and write operations. Due to this periodic refreshing, a DRAM module is constantly drawing power for the refresh read and write-back. In addition, if typical read and write operations do not service all addresses in a sufficiently timely manner; additional latency may be incurred when a scheduled refresh of one row conflicts with a read or write on a different row.

2.3.2 Static Random Access Memory

Static Random Access Memory (SRAM) differs conceptually from DRAM in one important feature: the data stored in a bit of SRAM will retain its value as long as power is applied to the circuit. This feature is obtained by a bi-stable circuit typically consisting of two cross-coupled inverters. Other topologies for bit-cells exist for special needs such as radiation hardness, separate read and write mechanisms, or low leakage in sub-90 nm processes. These inverters can be traditional CMOS inverters or even NMOS/PMOS with a resistive pull-up/down. Traditional SRAM bits are differential, storing a bit and its complement. Two additional access transistors are connected to each node to provide the ability to change the value of the bit-cell. SRAM arrays, as shown in Fig. 2.4, have two differential bit-lines that are connected to the drains of each access transistor, and a single word-line that connects to the gates of both access transistors. The peripheral circuitry is similar to DRAM in that there are address decoders driving the word-lines. Sense amplifiers, pre-charge circuits, and write drivers are also connected to both bit-lines.

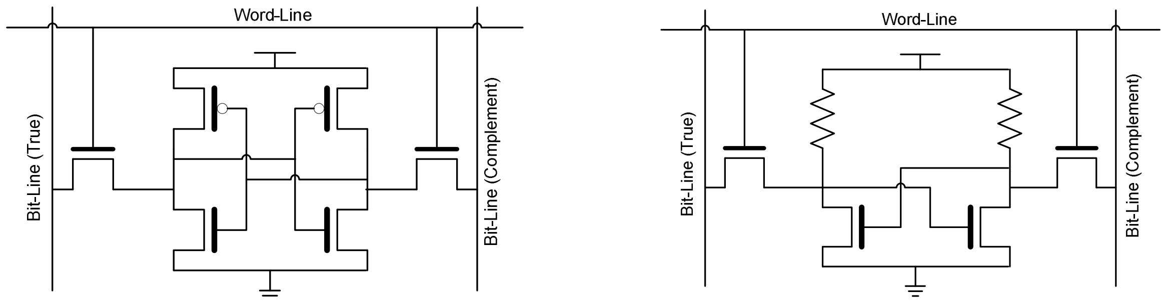

Fig. 2.4 - Six transistor CMOS SRAM cell (left) and four transistor resistive NMOS SRAM cell (right)

A write cycle begins with pre-charging the bit-lines and applying the data pattern to the bit columns that will be written. The word line for the row is asserted, and the bit-line voltages will overpower the internal state of the bit-cell. For the circuits in Fig. 2.4, writing a logic '1' to the memory cell would begin by pre-charging both bit-lines to V_{DD}, and then grounding the bit-line compliment. The word-line is then switched from logic '0' to logic '1', and the voltage on the complementary bit-line overpowers the weak PFET or resistor pull-up.

A read cycle begins with the pre-charging of the bit-lines. The word line for the row is asserted and the bit-cells begin to pull the bit-line voltages towards the internal voltages of the bit-cell. After sufficient time has elapsed to allow the bit-line voltages to change, the sense amplifier is activated, and it produces a logic-level output of the current bit-line state. The word-line is then de-asserted, and the memory array is ready for another cycle. The bit-cells in Fig. 2.4 could be read by pre-charging the bit-lines to VDD, and then switching the word-line from logic ‘0’to logic '1'. The relatively strong NFET on the node storing a logic '0' will pull the bit-line voltage towards ground, creating a voltage differential that can be read by a sense amplifier.

2.3.3 Read Only Memory

Read Only Memory (ROM) is often used in digital circuits for the storage of fixed data. This data is stored in the array outside normal operating conditions, and is not changeable during normal circuit conditions. ROM arrays have several usages in digital circuits, such as look-up tables for mathematical operations, machine code program storage for central processing units, finite state machines, and complex logic operations.

A common design for a ROM array starts with an array of NFET devices with all sources grounded. All the gates in a row are tied together into word lines, and all the drains in a column are connected to bit lines. At the beginning of a read cycle from the ROM array, all the bit-lines are pre-charged to V_{DD}. The desired word-line is then asserted, and the asserted transistors change the bit-line value from V_{DD} towards ground. This results in every bit registering as a ‘0’. A bit can be programmed with ‘1’s by disabling the transistor. This can be accomplished several ways and at many stages of the circuit’s design life. The programming of the transistors can be accomplished very early in the design stage by removing the transistor in the layout. After the design has been fabricated, the transistors in the array can be programmed by ion bombardment to raise the threshold voltage, effectively removing the transistor from the circuit. Additional technologies allow programming through tunneling effects such as EPROM, EEPROM, and Flash [11].

Chapter 3 – Memory Array Design

Design of an SRAM block is a sequential process. A design begins with the core element, the bit-cell, and the rest of the design is built around that single design. Each specification for subsequent blocks is dependent on earlier blocks. In order to preserve chronological design processes, each building block will be fully discussed before moving forward.

Table 3.1 - General System Specifications

Supply Voltage	3.0 V – 3.6 V
Operational Temperature	-180 ℃ to +125 ℃
Operational Frequency	25 MHz

The array topology was chosen to provide flexibility with the error correction logic, as well as reduce timing dependencies. Fig. 3.1 Block diagram for a 128 Byte memory array the selected topology for the 128 byte SRAM. The bit-cells are arranged in a rectangular grid to form the bit-cell array. The word-lines from the array are driven by a row decoder with an included driver. Bit-lines are connected to the bit-line access block, where the bit-lines are manipulated for read, write, and pre-charge operations. The bit-line access also provides a completion signal for use with read timing. The address and write data are fed into latches and then drive the row decoder and bit-line access blocks. The control block interprets the clocked control signals coming into the array, and translates those signals into precise signals for controlling the input latches, row decoder, and bit-line access blocks. While the diagram is specifically for the 128 byte SRAM, the topology requires only minor modifications to fit all memory blocks.

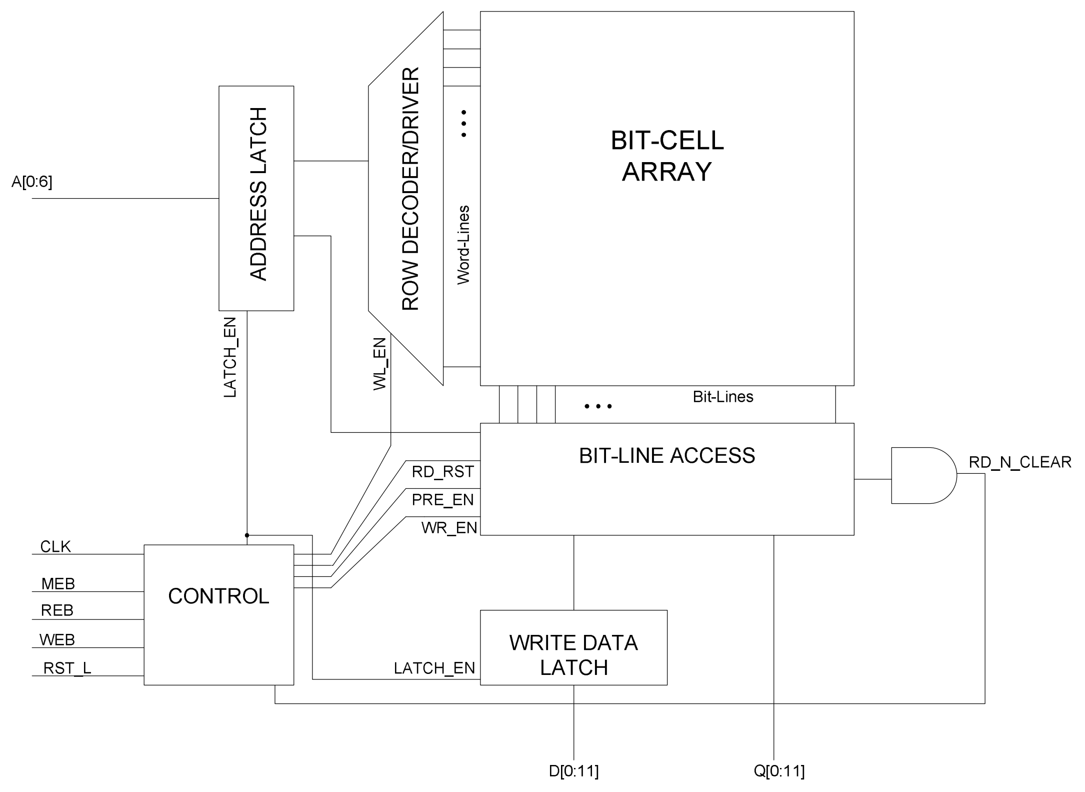

Fig. 3.1 Block diagram for a 128 Byte memory array

General system requirements for memories are outlined in Table 3.1. The supply voltage range and operational temperature are functions of the total REU requirements, and the operational frequency is based on the 21 MHz RDC system clock. The frequency is higher to allow for additional logic in the data path. The RDC (Fig. 3.2) required four sizes and types of memory, as illustrated in Table 3.2. The CAN bus interface requires 2 blocks of 64 bytes for send/receive buffers, 128 byte blocks are used for sample averaging and packetization of data for the CAN bus, and 2 kilobytes of program memory are needed for the 8031 core. A 1 kilobyte ROM containing start-up code for the 8031 is required as well. The 2 kilobyte block is implemented as two 1 kilobyte arrays to allow further design sharing with the 1 kilobyte SRAM.

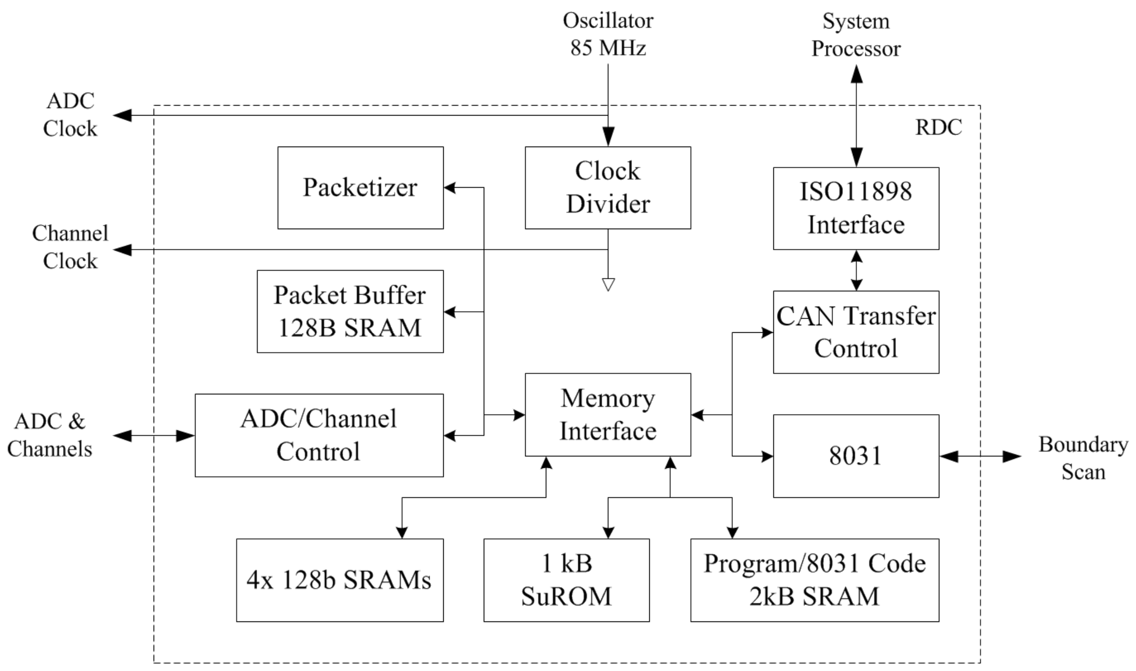

Fig. 3.2 - Remote Digital Controller (RDC) system diagram

Table 3.2 - Specific Feature Requirements For Each Memory Array

64 B SRAM 128 B SRAM	• Clocked inputs • Single Error Correction ECC • Single clock cycle operation
1 kB SRAM	• Clocked inputs • Single Error Correction, Double Error Detection ECC • Automatic correction of stored data on error detection
1 kB ROM	• Clocked inputs • Single clock cycle operation

3.1 General Design Techniques

Several techniques were used throughout the design to meet the system requirements. The wide temperature range specified for the REU module required care for both the high and low temperature extremes. At the high end, the primary concern is electromigration. This is the erosion of metal wires due to high current. This effect is exasperated at higher temperatures, and by constant DC currents [5]. In order to mitigate the risks of electromigration, high current nets should have wider traces. At cold temperatures, hot electrons are an additional risk for degraded functionality over time. As discussed previously, hot electron tunneling effects can be reduced at low temperature by decreasing the supply voltage or increasing the channel length of the NFET devices. Since the supply voltage is not a flexible constraint, all NFET transistors use a channel length of 1um or longer. Device characteristics vary greatly over the 305 ^{O}C temperature range. Both transistor threshold voltages as well as carrier mobility result in significant switching changes over the temperature ranges. As a result, great care was taken to understand the impact of temperature on circuit operation.

Radiation is also a major concern for the circuit, since the primary application is in space. There are two types of effects that were addressed at a system level. The first is Single-Event Latch-up (SEL), which is the activation of the parasitic bipolar transistors inherently present in a CMOS process. In order to minimize the SEL risk, all NFET and PFET transistors are enclosed in a guard ring [12]. This reduces the resistance in the parasitic bipolar latch to a point where the NPN and PNP transistors cannot turn on. The second type of radiation effect is Single-Event Upset, where a radiation event causes corruption in stored data. There are two storage elements that are considered for protection – the D flip-flops and the bit-cells. For the flip-flops, a dual interlocking cell topology prevents an ion strike on a single transistor from flipping the stored data value [13], [14]. To provide data protection for the SRAM arrays, Hamming error correction codes are applied to store redundant data. This redundant data provides the ability to correct a single bit of data. The algorithm and implementation will be addressed in greater detail in a following section.

With four similar arrays to design, a major focus is to minimize the amount of work that must be duplicated. Fig. 3.3 illustrates the extent of the design reuse throughout the project. When possible, blocks were designed to work in a broad range of operating configurations. A memory array is well suited to this task, as there is inherent modularity due to the array of bit-cells. In addition, a standard cell library is leveraged. The standard cell library, which is used for the entire RDC system design, meets all the base design requirements. All layouts are designed to easily tile for maximum compactness, and have been thoroughly simulated and characterized [14].

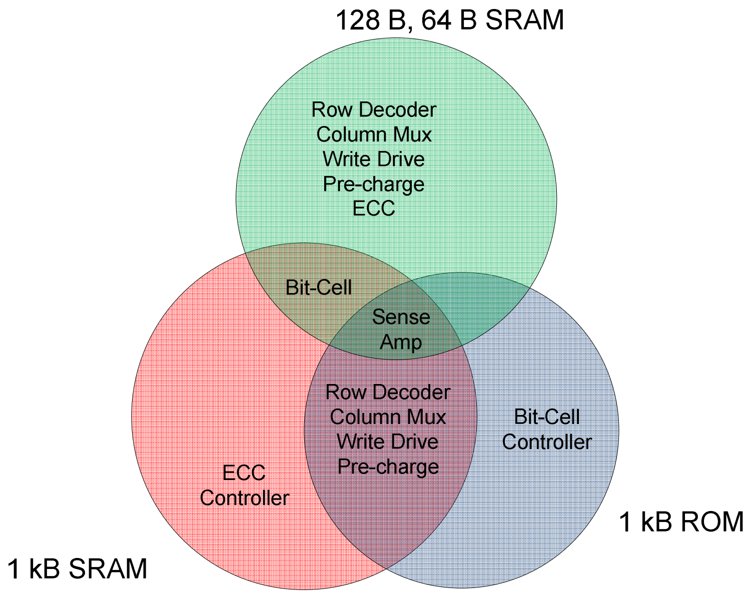

Fig. 3.3 - Venn diagram of design reuse across different SRAM models

3.2 Introduction to Simulations

Simulation of any circuit provides valuable feedback at every stage of the design process. Poor design topologies can be vetted out, device size choices can be made, circuit interactions can be monitored, and whole design functionality can be verified through simulations.

Two simulation paths are used for design analysis and verification. The primary simulator is the Spectre simulator through the Cadence Analog Environment [15]. This SPICE-type simulator provides detailed model support and high accuracy for many levels of simulation. The primary limitation of the Spectre simulator is poor scaling to large device counts. This is particularly evident on large array simulations with over 100,000 transistors. For large arrays or long simulations, Spectre simulations take too long to provide useful information. These simulations require a simulator capable of using algorithms optimized for digital signals, such as Synopsys' HSIM [16]. Such "Fast SPICE" simulations can be completed in a fraction of the time required for a full Spectre simulation. These simulators potentially sacrifice some accuracy for speed, although the tradeoffs are minimized for digital CMOS circuits.

Models used for the simulations are a combination of the standard IBM design kit models and custom models for use over an extended temperature range. With design temperatures ranging from -180 ℃ to +125 ℃, the importance of simulating the cir cuit over the entire temperature range cannot be overstated. The original IBM design kit models are valid over a temperature range of -55 ℃ to +125 ℃ , leaving the -180 ℃ to -55 ℃ range without models. Models were developed by ETDP team members for design validation at these lower temperatures. In addition to simulating over a wide temperature range, it is also important to simulate process variations. The models from the IBM design kit can be varied based on the measured statistical variation of properties such as gate oxide thickness, or doping concentrations. This can be used to verify functionality and performance of the circuit regardless of any manufacturing variability.

3.3 Bit-Cell Design

The bit-cell is the fundamental component of any memory array. Two basic topologies were considered for the bit-cell - a traditional NFET-based design using PFET pull-ups, and a PFET-based design using NFET pull-downs. The traditional NFET-based approach is usually preferred due to the increased frequency and drive strength of NFET devices. However, there are several disadvantages to the NFET devices when used over a wide temperature range and in high radiation environments. At extremely low temperatures, hot electron tunneling effects are more severe. For a temperature range extending to -180C, there are two primary techniques for mitigating these hot electron effects: reducing the supply voltage of the device, or increasing the channel length. In addition, the shallow trench isolation for the NFETs is susceptible to trapping charge, biasing the devices so that the off-state leakage increases as the total ionizing dose increases. These two effects significantly reduce the advantage NFET devices have over PFET devices for this particular application. For this reason, an alternative, PFET-based design shown in Fig. 3.4 was also considered.

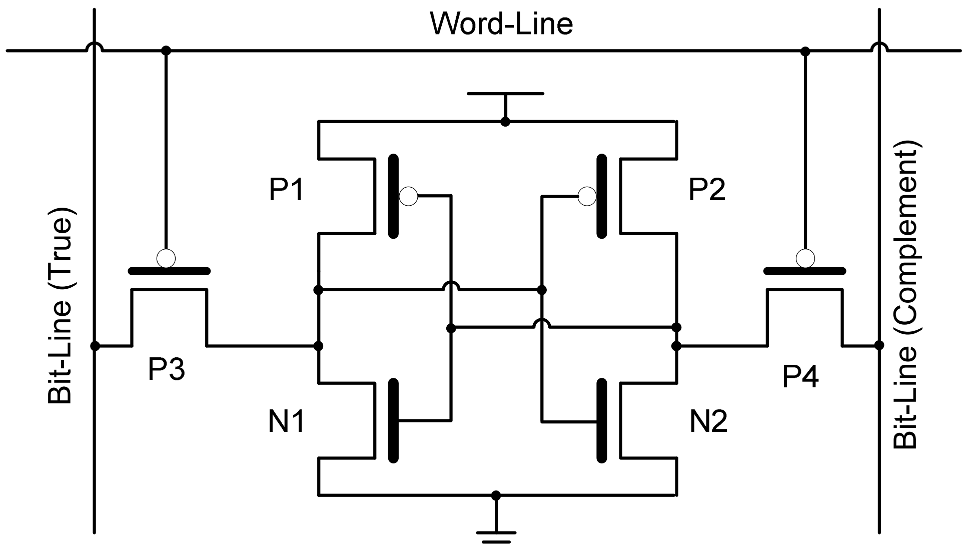

Fig. 3.4 - Bit-cell schematic

Three transistor sizes are required for each of the three functions performed in a CMOS SRAM bit-cell. The two transistors in each cross-coupled inverter consist of a dominant driver transistor (P1 and P2), and a weak load transistor (N1 and N2). The transistors that perform the gating function to the bit are the access transistors (P3 and P4), and those are the same type as the driver transistor. The driver transistor is the strongest, followed by the access transistor, and the smallest transistor is the load transistor. If the transistor W/L ratios between the three devices are not carefully considered, the bit-cell may not be writable, or it may have such a low noise threshold that performing a read operation may upset the bit-cell.

One way of characterizing how easy it is for a bit-cell to change values is through calculation of the Static Noise Margin (SNM). The SNM quantifies the voltage that must be added between the cross-coupled inverters to induce a change in the stored value. A static noise margin of greater than 10% of VDD, or > 0.33V was suggested through industry consultations with BAE Systems [17], [18]. Following the design formulas (3.1)-(3.6)shown below [19], a set of transistor ratios were calculated to meet the SNM target and minimize transistor area. This design process only calculates the relative strengths of the transistors. The transistors are indicated as the driver β_d (P1 and P2), access β_a (P3 and P4), and load/pull-down β_p (N1 and N2).

$$SNM_{6T} = V_T - \left(\frac{1}{k+1}\right)\left\{\frac{V_{DD} - \frac{2r+1}{r+1}V_T}{1 + \frac{r}{k(r+1)}} - \frac{V_{DD} - 2V_T}{1 + k\frac{r}{q} + \sqrt{\frac{r}{q}\left(1 + 2k + \frac{r}{q}k^2\right)}}\right\} \tag{3.1}$$

$$r = \frac{\beta_d}{\beta_a} \tag{3.2}$$

$$q = \frac{\beta_p}{\beta_a} \tag{3.3}$$

$$k = \left(\frac{r}{r+1}\right)\left\{\sqrt{\frac{r+1}{r+1-\frac{V_s^2}{V_r^2}}} - 1\right\} \tag{3.4}$$

$$V_s = V_{DD} - V_T \tag{3.5}$$

$$V_r = V_s - \left(\frac{r}{r+1}\right)V_T \tag{3.6}$$

Using (3.1)-(3.6) to solve for SNM, it is apparent that there are an infinite number of solutions possible. Transistor sizes were chosen to minimize area by setting the weakest transistor, the load transistor, to a W/L ratio of 1um/1um. The difference in carrier mobility between PFETs and NFETs inherent in the CMOS process is multiplied by the W/L ratio, to give a β_p equal to 2.2. With this value fixed, β_a and β_d were swept to find the combination that matched the SNM requirement and minimized the sum of β_a and β_d, shown below in Table 3.3.

Table 3.3 - Calculated Values For Static Noise Margin

Parameter	Value
$\frac{I_{DSAT_N}}{I_{DSAT_P}}$	2.2 [5]
V_{DD}	3.3 V
V_{T_P}	0.55 V
β_a	4
β_d	8
β_p	2.2
R	2
Q	0.275
K	0.227
V_s	2.75 V
V_r	2.38 V
SNM_{6T}	0.341 V

The resulting transistor sizes shown in Table 3.4 were derived from the ratios and were then simulated to verify that the signal-to-noise margin was sufficient. Minimum gate lengths were used for the PFETs, and a 1.0 µm gate length was used for the NFETs. The calculated ratios were converted into transistor dimensions using (3.7)-(3.9), shown below.

$$\beta_d = \frac{W_d}{0.5\mu m} \tag{3.7}$$

$$\beta_a = \frac{W_a}{0.5\mu m} \tag{3.8}$$

$$\beta_p = \frac{W_p}{1.0\mu m} \times \frac{I_{DSAT_N}}{I_{DSAT_P}} \tag{3.9}$$

Table 3.4 - SRAM Bit-cell Transistor Sizes

Transistor	W/L Ratio
P1, P2 (drive)	4.0 µm / 0.5 µm
P3, P4 (access)	2.0 µm / 0.5 µm
N1, N2 (load/pull-down)	1.0 µm / 1.0 µm

Two initial versions of each topology were designed and physically implemented. In preliminary layouts, the NFET access bit-cell layout was 11.2% smaller than the PFET access design. However, the PFET access design has lower word-line capacitance due to the shorter channel length. In addition, NFET devices are much more susceptible to TID induced leakage. This could potentially reduce the isolation of a bit-cell from the bit-lines while not in use, causing data corruption. However, in the PFET access design, if the NFET load transistors developed large leakage currents, the bit-cell would shift to a complementary passive pull-down topology. This would make the bit-cells tolerant of a larger TID. The circuit benefits of the PFET access bit-cell were selected over the area savings of the NFET access design.

3.3.1 ROM Cell Design

Traditional ROM designs use a single NFET transistor in an array, with a single bit-line for reading. This provides optimal layout density as well as a fast, low capacitance circuit. A primary design goal for the ROM was to maximize design reuse of the SRAM design at both the schematic and layout levels. This changes the design approach from minimizing area and transistor count to minimizing as many layout and interface changes as possible.

In order to make an SRAM bit-cell into a ROM bit-cell, modifications of the schematic were based on the assumptions of a fixed, pre-programmed value for each bit. This fixed the internal value of the cross-coupled inverter, with one node tied to ground, and one node tied to V_{DD}. Once these nodes are fixed, the driver and load transistors may be removed in favor of open circuits and shorts. This only leaves the two access transistors connected to both of the bit-lines. One of these access transistors will be connected to V_{DD}, and the other will be connected to ground. By analyzing the function of the access transistors during a read operation, the access transistor connected to ground does not affect the bit-line voltage, only the access transistor connected to V_{DD}. Thus, the grounded access transistor is not necessary and can be removed, resulting in the schematic shown in Fig. 3.5. The circuitry interfacing the array is only changed to disable writing.

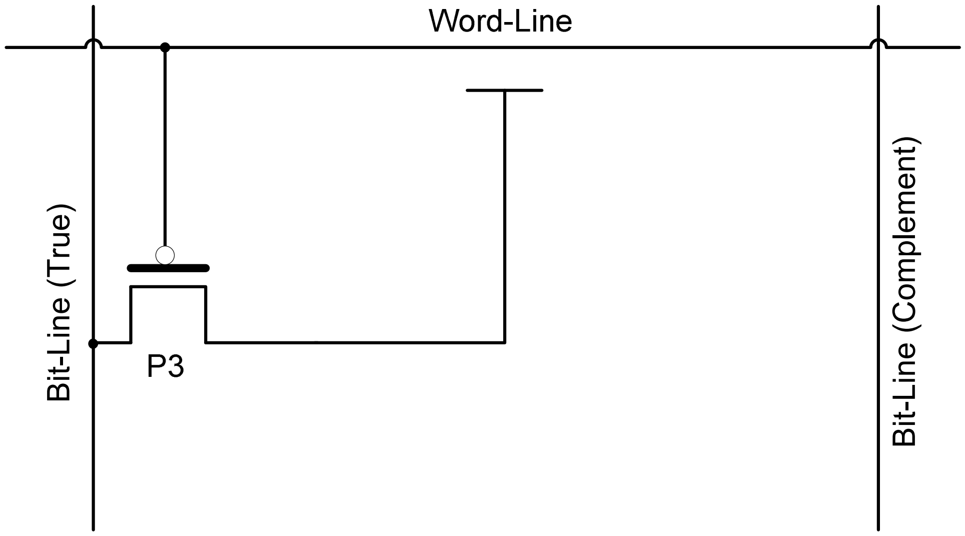

Fig. 3.5 - ROM bit-cell schematic (Storing '1')

3.4 Bit-Cell Simulations

The primary stand-alone simulation with the bit-cells tests the Static Noise Margin (SNM). This test verifies the minimum amount of noise or mismatch required to upset the value of a memory cell during a read operation. Fig. 3.6 shows the simulation schematic, which is identical to the bit-cell except for the stimulus voltage sources. Ideal voltage sources are placed between the gates of each cross-coupled inverter and the drain of the opposite inverter. These voltage sources act as a worst-case noise source. The bit-lines are fixed to the worst case read conditions, and then the word-line is pulsed to simulate a read. The bit-cell is then monitored for an upset. A typical simulation is shown in Fig. 3.7. Both the true and compliment bit-lines are shown. If the voltages cross, such as at the 35 ns mark in Fig. 3.7, then an upset has occurred, and the applied offset voltage exceeds the SNM of the bit-cell. The simulation is repeated for increasing offset voltages until the highest offset voltage that does not result in an upset is found. The noise margin simulations show the simulated SNM of the bit-cell over all expected operating conditions in Fig. 3.8, excluding random process variation.

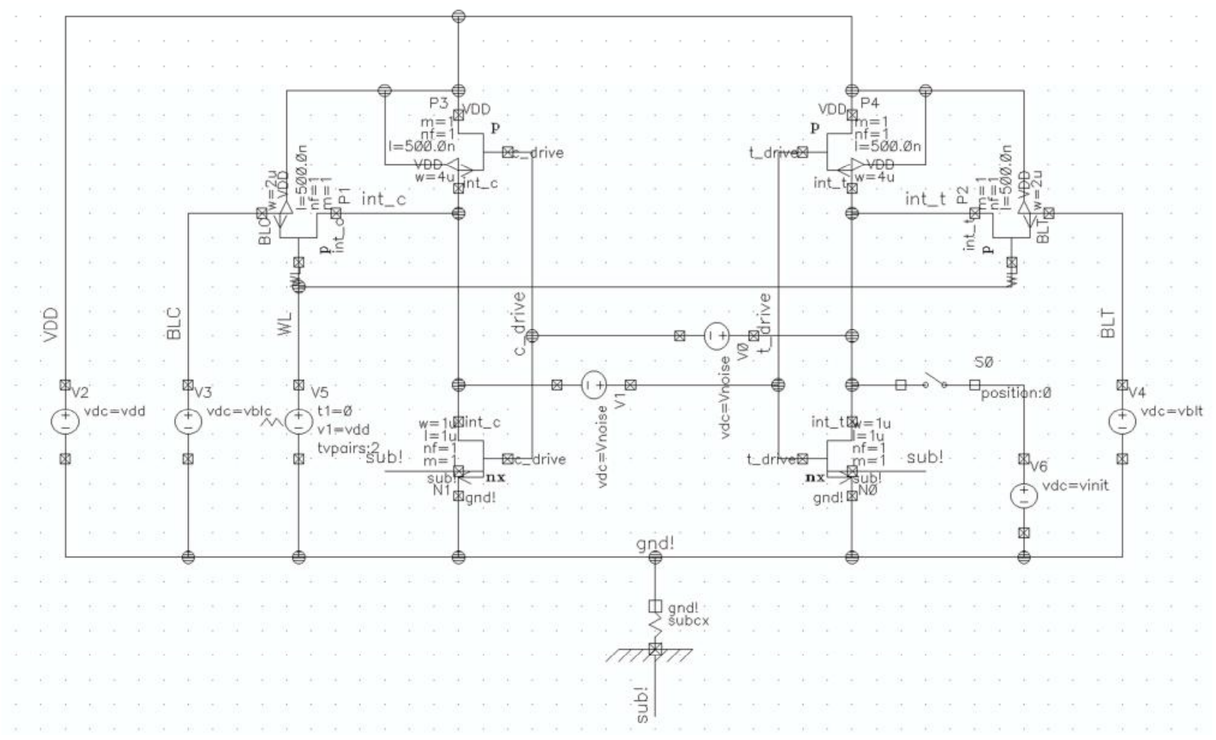

Fig. 3.6 - Bit-cell schematic modified for SNM measurement

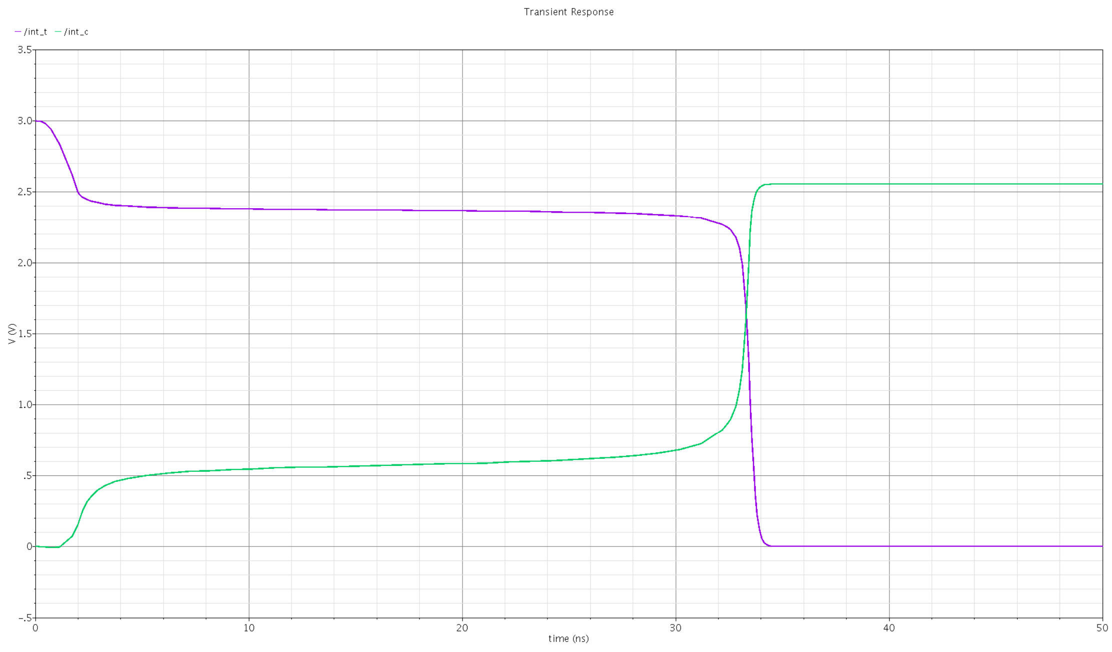

Fig. 3.7 - SNM simulation showing upset at 35 ns

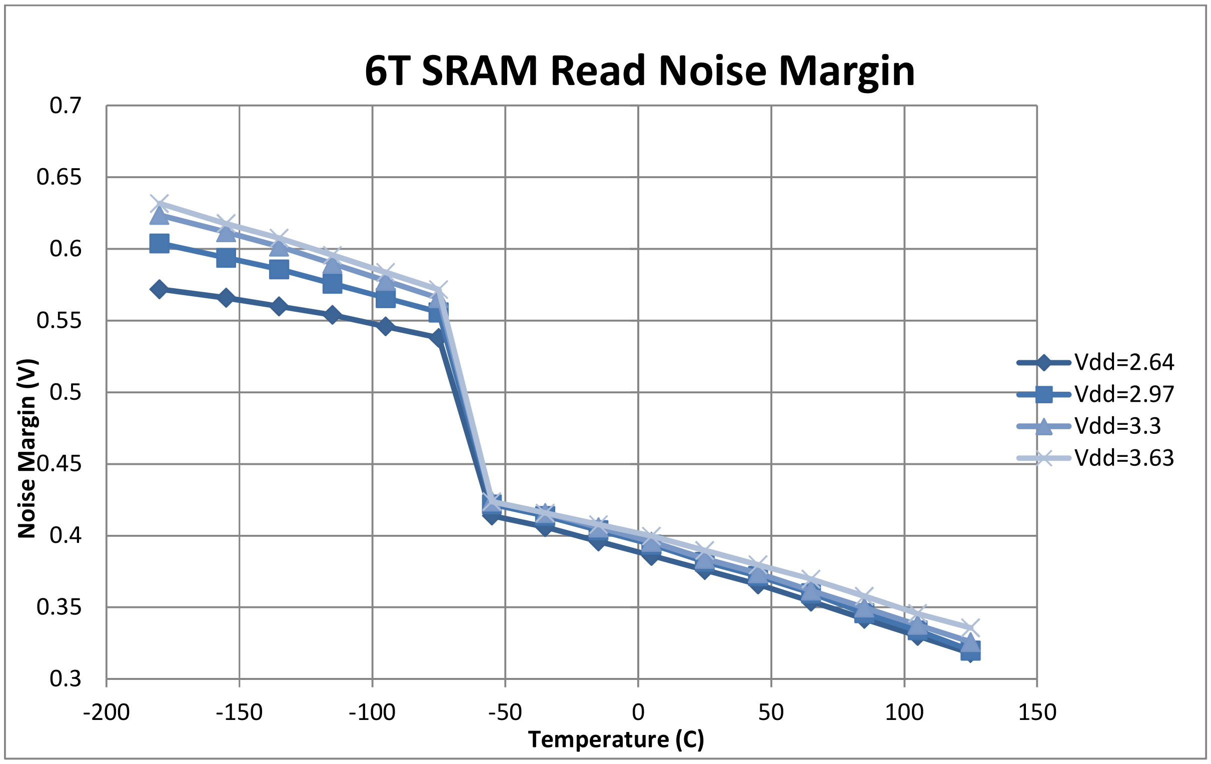

Fig. 3.8 - Simulated SNM over temperature and voltage

3.5 SRAM Bit Layout

Careful design of the bit-cell layout is a high priority for any memory design, because any redesign in the bit-cell layout will most likely result in major layout modifications of any adjacent circuitry. The bit-cell is thus the cornerstone of the memory layout, and must be completed first. Optimization of the bit-cell layout is important to obtain maximum memory density and performance. An image of the bit-cell layout is shown in Fig. 3.9. Conversely, many radiation hardening layout techniques add significant area to the design. The main layout feature for radiation hardening is the n-well guard ring. The guard ring adds several microns of length to the n-well / p-substrate junction and prevents the routing of signals on any layer below the second metal layer. The downside to this is that four metal layers are required to provide full power and ground connectivity in addition to the bit-line and word-line signals. Without guard rings, a fully connected design is possible using only the first two metal layers; leaving top metal layers available for global routing. In this design, signals are routed on the first three metal layers, and power and ground connections are present on all four metal layers. This effectively prevents routing of any signals over the array.

Power and ground routing in the bit-cell uses both vertical and horizontal metal straps. This ensures a low resistance across the array and increases the number of suitable places for supplying power to the entire circuit. In order to reduce the number of n-well / p-substrate junctions, every other row is flipped vertically. This allows each row to have only one set of guard rings. The guard rings are a single ring for an N-well diffusion.

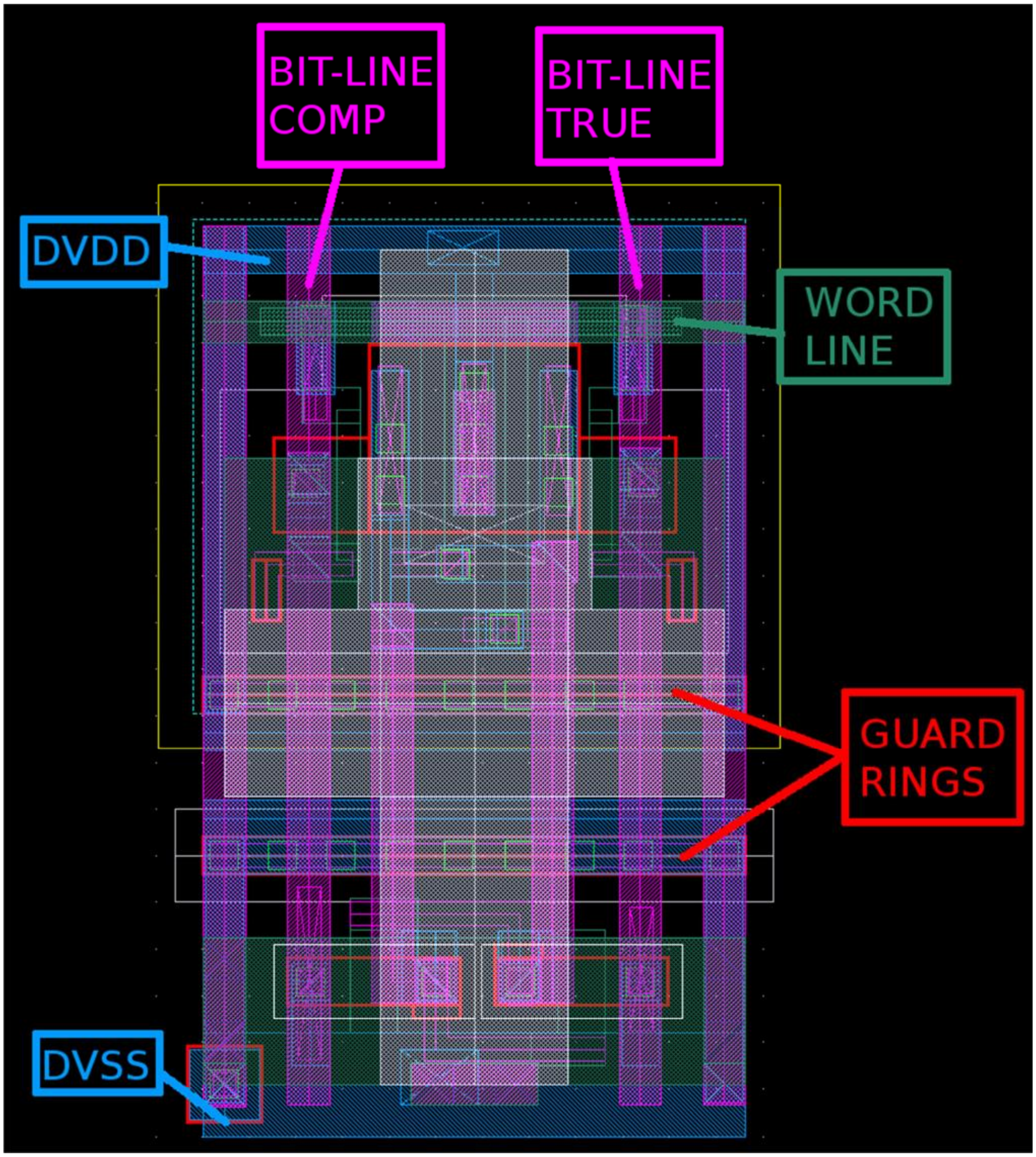

Fig. 3.9 - SRAM bit-cell layout

3.6 ROM Bit Layout

The ROM bit-cell was designed to act like an SRAM bit both electrically as well as physically. The bit-line and word-line wire locations were not moved in the ROM bit, creating a port layout similar to the RAM bit. The single transistor of the ROM bit is placed in the center of the n-well region. The drain of the PFET is connected up to the third layer of metal, through a metal bridge to either the complementary or true bit-line and down to one of the bit-lines on the second metal layer. This is done to reduce the shape changes to the bit-lines and word-lines and to facilitate efforts to change the ROM values. The third metal layer is the highest layer that could easily host the programming short. These shorting blocks are generated by a Python script from an Intel Hex file (Appendix A). This is shown below in Fig. 3.10 with the relevant areas highlighted.

Placing the programming layer as high on the metal stack as possible provides multiple benefits towards being able to reprogram the ROM array. Whole wafers can be preserved after processing all layers below the third metal layer, allowing a ‘reprogramming’ by changing only one processing mask. By placing the programming layer higher in the metal stack, the turn-around time of the redesign is shorter because fewer layers need to be added to finish the wafer. The other major advantage with a higher programming metal layer is that Focused Ion Beam (FIB) reprogramming is less costly due to the depth of the metal layer from the top of the wafer surface. A metal programming layer was chosen over using via programming for the relative ease of a FIB repair of a metal short programming over via programming.

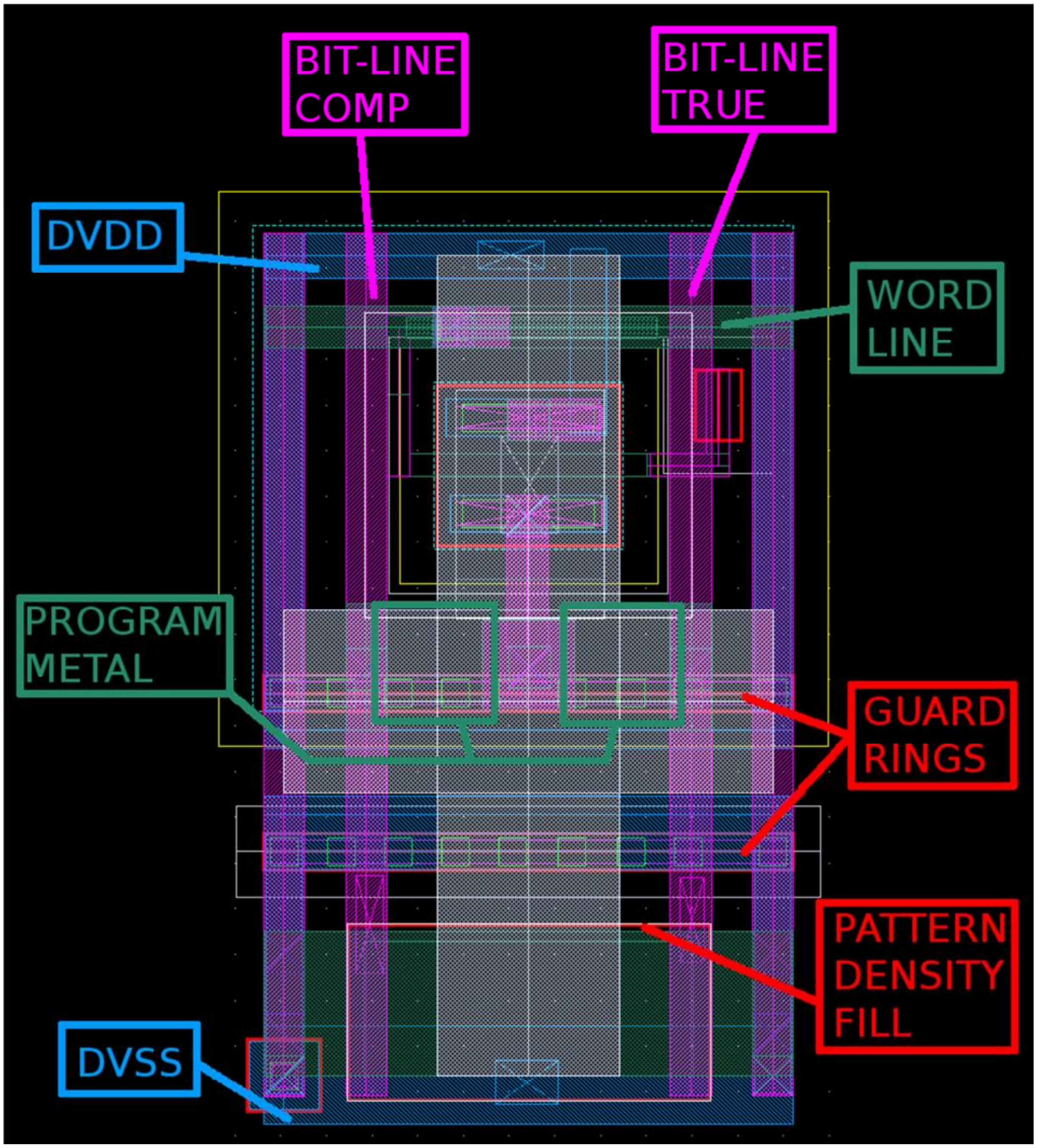

Fig. 3.10 - ROM bit-cell layout

One of the design rules that is often only considered once a chip is nearing completion is design layer pattern density. Many design layers require that the layer usage be within a range specified in the design manual. Since the ROM bit-cell is a slimmed down version of the SRAM bit-cell, the area of certain layers used for transistors is below the minimum percentage. This deficiency can be a major issue when an entire array is placed on a chip. Fill areas were added to supplement the active and polysilicon layers of the ROM bit-cell, thus avoiding a fill deficiency in the final memory array.

3.7 Array Layout

Once the bit-cell layout is ready for use, it must be combined into arrays to fit the size requirements for each block. Generally, the array aspect ratio determines the length of the bit-lines and word-lines. Square or wide rectangle arrays tend to balance the tradeoff between the word-line and bit-line capacitance, with a preference on longer word-lines. Table 3.5 below lists the array sizes that were chosen for the memory arrays.

Table 3.5 - Memory Array Dimensions

Array Name	Array Size	Array Grid	Physical Dimensions
SRAM 64	64 x 12 bits	16 x 48	286.4 x 604.8
SRAM 128	128 x 12 bits	32 x 48	572.8 x 604.8
SRAM 1024	1024 x 13 bits	64 x 208	1145.6 x 2620.8
ROM 1024	1024 x 8 bits	64 x 128	1145.6 x 1612.8

Once the array dimensions are selected, the array is constructed. Special layouts are used for the corner and edge bit-cells to properly contain the guard rings and power connections. After the layouts are finished, detailed parasitic extraction is performed to measure the estimated capacitance for the longest bit-lines and word-lines. These values are used in simplified test benches to ease the simulation load during the design of later blocks.

3.8 Row Select

The word-lines of a memory array determine if a row is connected to the bit-lines or not. During any single memory operation, only one word-line will be asserted. Many rows may be present in an array, requiring a method of ensuring that only one row is active at a time. In addition, reducing the number of control signals is a high priority for data busses inside a circuit. The optimal circuit for meeting both goals simultaneously is a decoder. To understand decoder operation, consider that each memory array row is assigned a unique sequential number. A binary data bus carries a value equal to the row number that is to be accessed. Logic blocks then “decode” the binary signal to a single output, providing the minimal number of control lines for a given number of rows.

In addition to the decoding of address bits, additional buffering and control is required for driving the word-lines. During memory operations, the desired word-line is not continuously asserted. This permits time for presetting the bit-lines to voltages appropriate for the operation and data values desired. A simple decoder circuit does not have a way to de-assert a word-line signal, so a simple scheme is devised to add this function: increase the number of input control signals by one, and only implement one half of the decoder physically. In this design, this is accomplished by a logical AND with each output bit and an enable signal from the controller. Word-lines also have large capacitive loads on the circuits, requiring multiple stage buffering.

For the 1 kB arrays, the design is implemented as a set of four 4-to-16 decoders with multiple enable inputs that allow several decoders to be placed in the system with minimal additional circuitry. A NAND gate drives a buffer, shown in Fig. 3.11, which provides enough drive strength to switch the word-line. The word-line enable signal is fed to one of the inputs of the NAND gate to reduce the propagation delay of the enable signal. The smaller arrays use one decoder to fully access all rows, but share the NAND gate and buffer output. Both designs leverage the standard cell library for all gates except the final drive gates. These cells have special layouts to match the pitch of the array. The drive strength of the final buffer is adjusted for both sizes.

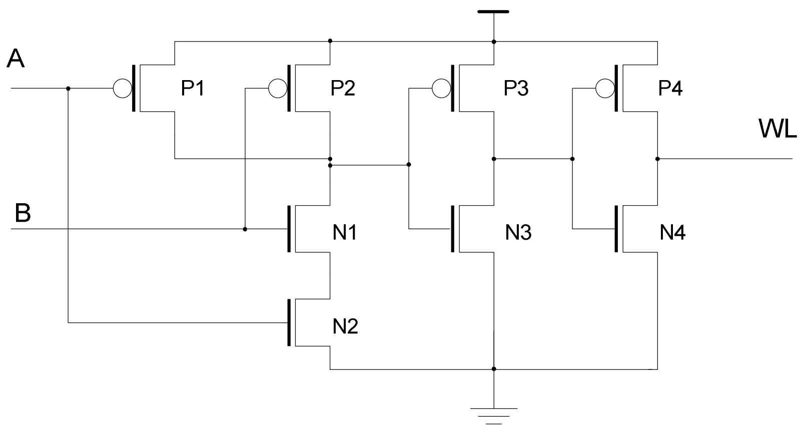

Fig. 3.11 - Word-line drive schematic

3.8.1 Row Access Simulation

The row decoder consists of a decoder and a driver for the word-lines. Since the decoding portion consists of standard cells, the primary interest in simulation is the propagation delay and the rise and fall times of the word-line. The word-line driver is connected to an ideal capacitor that emulates the capacitive load of the word-line in the bit-cell array. The optimization function of Analog Environment is then used to iteratively determine optimal transistor sizes of the word-line driver to minimize delay, rise time, fall time, and power consumption. The resulting transistor sizes are shown below in Table 3.6. The simulation test bench is shown below in Fig. 3.12. Two versions of the memory arrays were optimized: the 48 column wide version, and the 208 column wide version. The ROM width of 128 columns was not optimized, and uses the 208 column driver.

Table 3.6 - Word-line Drive Transistor Sizes

Device	**SRAM_12X64, SRAM_12X128**	**SRAM_13X1024, ROM_8X1024**
P1, P2	1.5 µm / 0.5 µm	6.675 µm / 0.5 µm
P3	1.5 µm / 0.5 µm	46.36 µm / 0.5 µm
P4	6.0 µm / 0.5 µm	82.4 µm / 0.5 µm
N1, N2	2.0 µm / 1.0 µm	15.875 µm / 1.0 µm
N3	1.0 µm / 1.0 µm	11.16 µm / 1.0 µm
N4	4.0 µm / 1.0 µm	25.55 µm / 1.0 µm

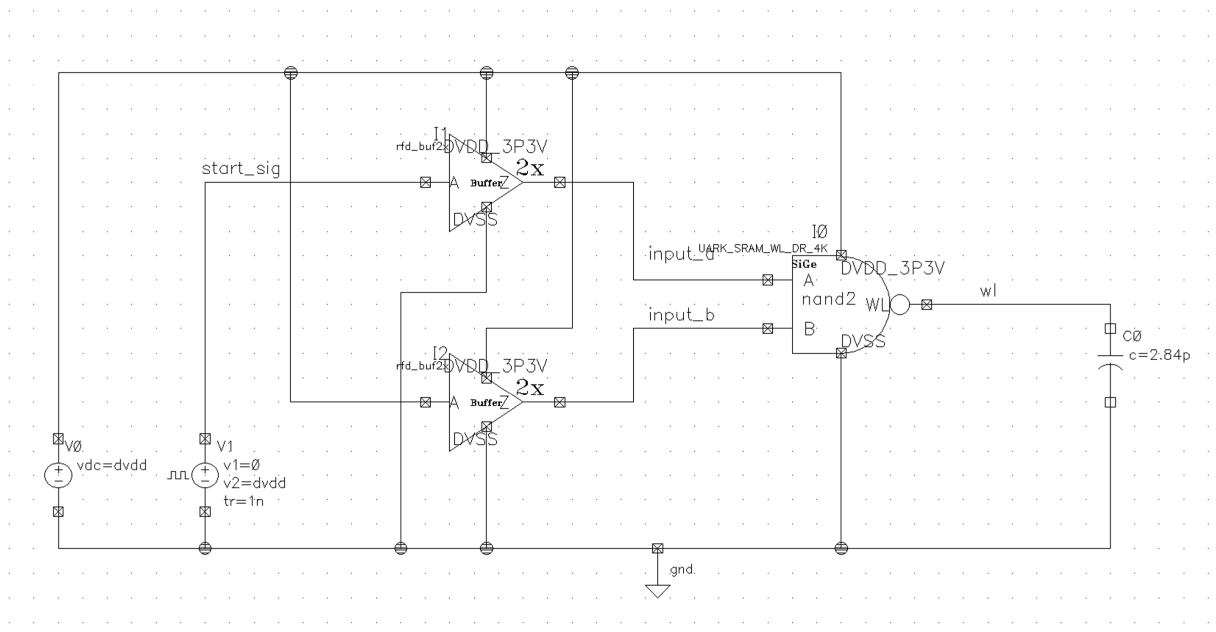

Fig. 3.12 - Word-line drive optimization schematic

3.8.2 Row Access Layout

The layout of the row decoder and drive circuit is heavily dependent on the height of the bit-cell. The final AND gate and buffer must exactly match the height of the bit-cell so that it can be flipped and abutted with it. The smallest module of the row decoder will be at least a multiple of the row height. Additional circuits with large fan-outs are placed below the row decoder in the rectangular space formed by the row decoder and the column circuitry. A sampling of all the final versions of the row decoders is shown in Fig. 3.13. The 16-row version is derived from the 32-row decoder.

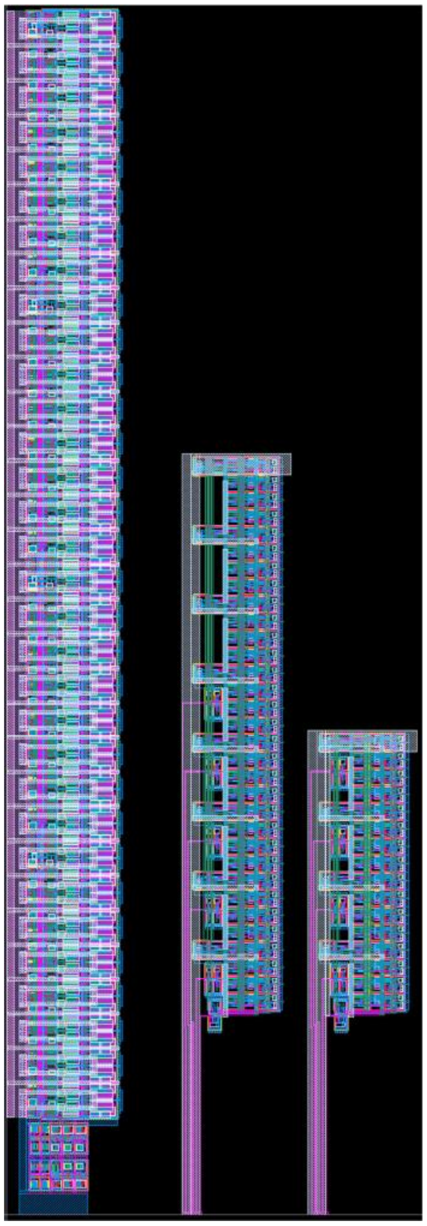

Fig. 3.13 - Row decoder layouts for 64, 32, and 16 row arrays

3.9 Column Access

The analog circuitry that connects directly to the bit-lines is called the column access block. Each array has eight to thirteen access blocks, depending on the array size. The sub-blocks all need to be in close physical proximity, and the design is simplified by grouping these four sub-blocks together. The configuration of these blocks is shown below in Fig. 3.14.

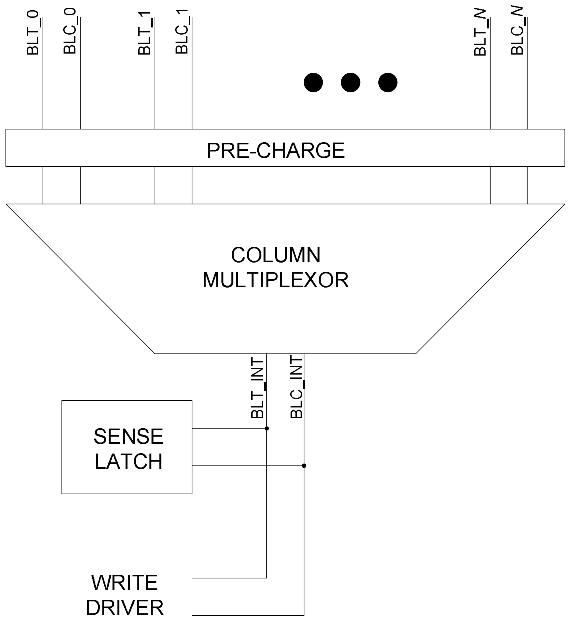

Fig. 3.14 - Column access block diagram

3.9.1 Column Multiplexing

To create large matrices that have more columns than the width of the data bus, some method of switching between the columns is required. Multiplexing the columns is a simple and effective method to interface multiple columns. Through multiplexing, a single set of columns in an array can be selected for read or write operations. Two choices exist on where to multiplex the columns. The first option is to include read and write circuitry for every column, and use digital multiplexers to select the desired column. This is a common approach for DRAM due to the need to read every bit in a row. One major downside to this method is the increased difficulty of laying out the read and write circuitry in the width of a bit-cell. The other approach is to use a transmission gate multiplexer to select a pair of bit-lines from all of the column bit-line pairs in a group and use only one set of read and write circuitry for every column associated with a certain bit, shown in Fig. 3.15. This allows more optimal layout design due to the significantly less restrictive width requirements.

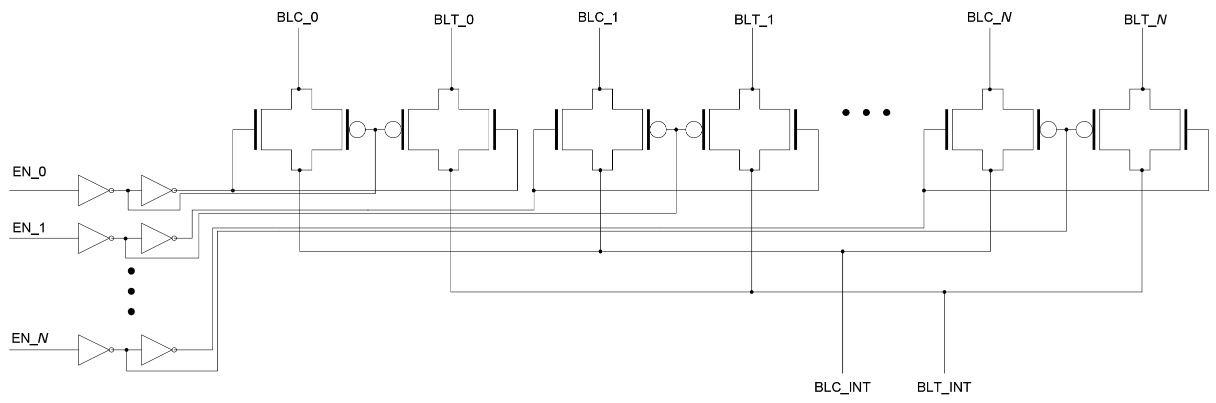

Fig. 3.15 - Column multiplexer schematic

A one-level multiplexer was used instead of a tree multiplexer to improve scalability as well as to minimize the number of transistors in series with the writing and pre-charging circuits. The original 128 byte SRAM uses four columns for every bit, requiring two 4-to-1 multiplexers to interface the array to the reading and writing circuitry. For the larger arrays, sixteen columns are used, so two 16-to-1 multiplexers are added. If the transistors are not large enough, there will not be enough drive strength to write to the array. Transistor sizing for the column multiplexers, shown in Table 3.7, was determined by analyzing simulations with the pre-charging and writing circuits to ensure that the drive strength is sufficient to function across all simulation corners.

Table 3.7 - Column Multiplexers Transistor Sizes

Device	SRAM_12X64, SRAM_12X128	SRAM_13X1024, ROM_8X1024
PFET	9.0 µm / 0.5 µm	24.0 µm / 0.5 µm
NFET	2.0 µm / 1.0 µm	8.0 µm / 1.0 µm

3.9.2 Pre-charge

At the beginning of a read or write operation, the bit-lines are at an unknown state. Either bit-line could be at a voltage between ground and V_{DD} due to previous array activity. A preexisting state on the bit-lines can pre-bias the sense amplifiers to produce the opposite result, or even hinder the writing of data into a bit-cell. In order to counter this effect, a pre-charge circuit is used to set the bit-lines to a known voltage before reading or writing. There are two functions of a pre-charge circuit; to bring the bit-line voltages towards a reference voltage and to reduce the voltage difference between the two complimentary bit-lines. The most convenient and stable reference available that allows the read operations to function properly is the negative supply rail, or ground. An alternative voltage reference used in other designs is ½ V_{DD}. By using ground as the voltage reference for the pre-charge circuit, high current, temperature stable voltage references are not required, which saves area, reduces design complexity, and simplifies wire routing. The pre-charge driver circuit uses two NFET devices to pull both bit-lines to ground. Every column has a pair of NFET devices in order to ensure that the bit-lines will not inadvertently write data during a read operation, as illustrated in Fig. 3.16. The transistor sizes were determined by simulating the pre-charge cycle of a memory array and adjusting the size of the transistors to take a bit-line at 3.3 V down to 0 V in approximately ¼ of the expected clock cycle, or 10 ns. The resulting transistor sizes are shown in Table 3.8.

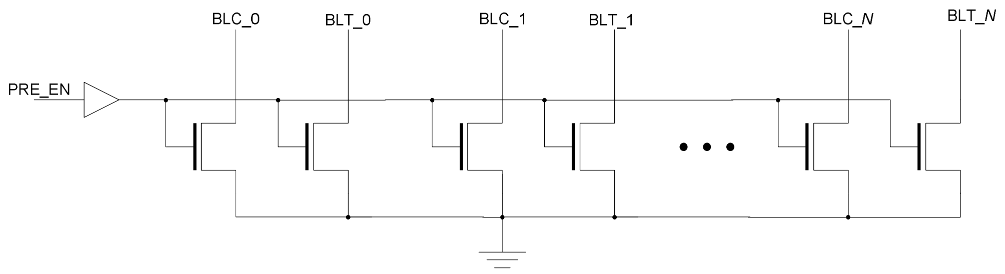

Fig. 3.16 - Pre-charge schematic

Table 3.8 - Pre-charge Transistor Sizes

Device	**SRAM_12X64, SRAM_12X128**	**SRAM_13X1024, ROM_8X1024**
NFET	4.0 µm / 1.0 µm	8.0 µm / 1.0 µm
Buffer	rfd_buf8x	rfd_buf18x

3.9.3 Write Drive

Writing to the bit-cells is accomplished after pre-charging the bit-lines to ground. Consider one of the bit-lines driven to V_{DD}. The appropriate word-line is then asserted, and the data value is written to the bit-cells in the row. Writing is then accomplished by two PFET devices controlled by a logic function of the write enable signal and the data to be written, shown in Fig. 3.17. When write enable is asserted, one of the PFET devices will activate, driving the connected bit-line to V_{DD}. This should be powerful enough to overcome the load transistor through the access transistor in the bit-cell, or else writing is not possible. Simulations across all corners were performed to ensure that the transistor sizes in Table 3.9 allowed reliable write performance. Asserting the write enable signal before asserting the appropriate word-line signal allows faster writing by allowing the bit-line voltage to rise to V_{DD} before the word-line is asserted. The result is the higher initial bit-line voltage switches the bit-cell faster than if the bit-line voltage is driven to V_{DD} after the word-line is asserted.

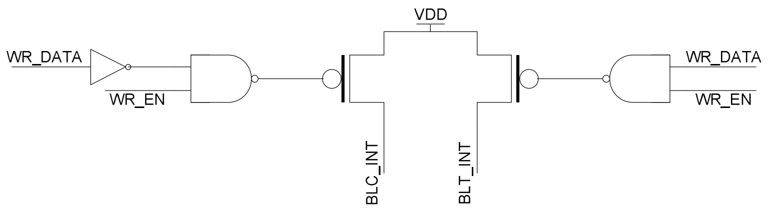

Fig. 3.17 - Write drive schematic

Table 3.9 - Write Drive Transistor Sizes

Device	SRAM_12X64, SRAM_12X128	SRAM_13X1024, ROM_8X1024
PFET	9.0 µm / 0.5 µm	32.0 µm / 0.5 µm
Inverter	rfd_inv2x	rfd_inv4x
NAND	rfd_nand2i2x	rfd_nand2i8x

3.9.4 Sense Amplifier

The sense amplifier is a circuit used during a read operation that accelerates the resolution of the data value on the bit-lines. It provides isolation from extra capacitive loads and noise present in downstream circuits. It also provides a driving capability to the data outputs, and a fast transition time. When the word line in a row is asserted, the bits charge the bit-line voltages through the access transistors. For this design, the bit-lines are pre-charged to 0 Volts, and the bit-cells pull either the true or complement bit-line towards the supply voltage of 3.3 Volts. The bit-line voltage's rate of change depends largely on the number of bit-cells in a column of an array. For the relatively small 64 and 128 byte arrays, there are 16 and 32 rows, respectively. The larger memory designs such as the 1 kB ROM and the 1 kB SRAM have 64 rows, resulting in much higher resistive and capacitive loads on the bit-cells. In the initial design with 32 rows, the bit-line voltage rises to the full value on the order of a several nanoseconds.

While high performance RAM uses a type of clocked latched comparator, generating timing signals used for such a latch is difficult over wide temperature and voltage variations. A conservative design shown in Fig. 3.18 was chosen to ensure that unexpected bit-line conditions could not cause a fault. The sensing input is provided by two NFETs (N1 and N4) with each gate tied to one of the bit-lines. After a pre-charge, both bit-lines will be at 0 Volts, and the drain voltages of the sensing NFETs are at 3.3 Volts. When the word-line is asserted, one of the bit-line voltages will begin to rise. This turns on one of the sensing NFETs, which pulls the drain voltage towards ground. Once one of the outputs reaches 0 Volts, both sensing transistors are disconnected by a series NFET (N2 or N5), and an inverter drives the latched output for both bit-lines. Additional logic shown in (3.10)-(3.14) controls the unlatching (CP, CN, AP, BLT_INT) and ensures that the four transistors on each leg cannot shunt current directly from the power supply to ground.

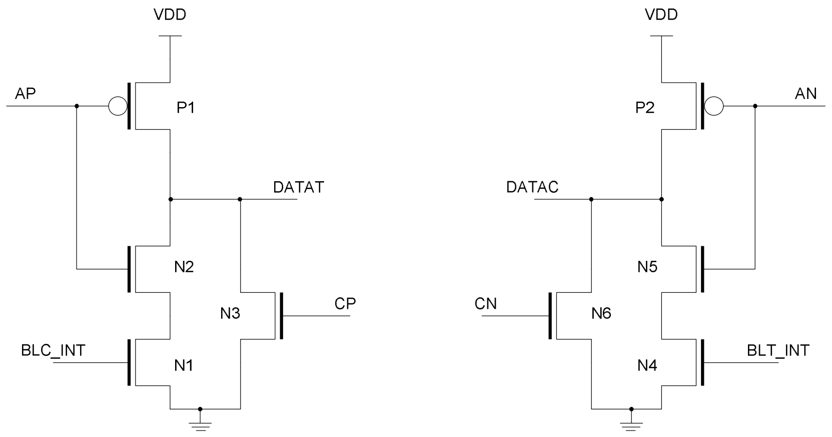

Fig. 3.18 - Sense amplifier schematic

$$CP = \overline{RD\_RST + \overline{DATAC} + DATAT} \tag{3.10}$$

$$CN = \overline{RD\_RST + \overline{DATAT} + DATAC} \tag{3.11}$$

$$AP = \overline{RD\_RST + \overline{\overline{DATAC}}} \tag{3.12}$$

$$AN = \overline{RD\_RST + \overline{\overline{DATAT}}} \tag{3.13}$$

$$RD\_COMPLETE = AP + AN \tag{3.14}$$

Table 3.10 - Sense Amplifier Transistor Sizes

Device	W/L
P1, P2	4.0 µm / 0.5 µm
N1, N4	32.0 µm / 1.0 µm
N2, N5	6.0 µm / 1.0 µm
N3, N6	4.0 µm / 1.0 µm

The result is a robust circuit that is well behaved regardless of any potential startup or transient anomalies. Transistor sizing shown in Table 3.10 was determined by minimizing the transition time on the output node. The holding transistors do not have a time critical function, so the W/L ratio is relatively small. The sensing transistor and the series-pass transistor were optimized to result in minimal fall time during sensing. The effects of transistor sizing on the fall time is shown below in Fig. 3.19. The design performs well with both fast rise times from the small arrays as well as with the longer rise times from the 1 kB arrays. As a result, the sense amp was used on all arrays to reduce the total number of designs required.

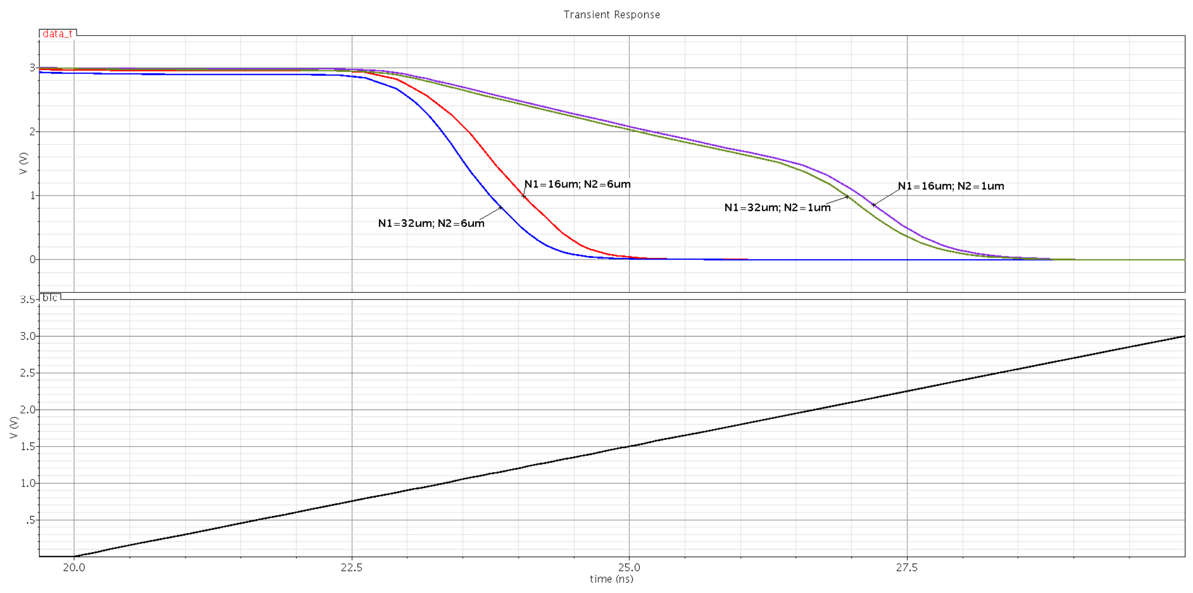

Fig. 3.19 – Simulation results of transistor sizing on sense amplifier fall time

3.9.5 Layout of BL Access

There are two layout variants of the bit-line access circuitry, but the topology is essentially the same. At the top of the layout, nearest to the bit-cell array, are the pre-charge drivers. Positioning them here allows for an efficient layout of the drivers and minimizes the signals passing through to just the enable signal. Immediately below the pre-charge is the column multiplexer. The write drive transistors and the sense amplifier transistors are clustered together for a balanced layout. Balance is more important in this layout than transistor matching. Balance affects different wire resistances and capacitances, which can cause differences in performance from the intended design. At the bottom of the cell is all the standard cell logic used to drive the four blocks. The logic leverages the standard cell library for all gates. The final layouts of the 4 column (Fig. 3.20) and 16 column (Fig. 3.21) column access layouts are shown below.

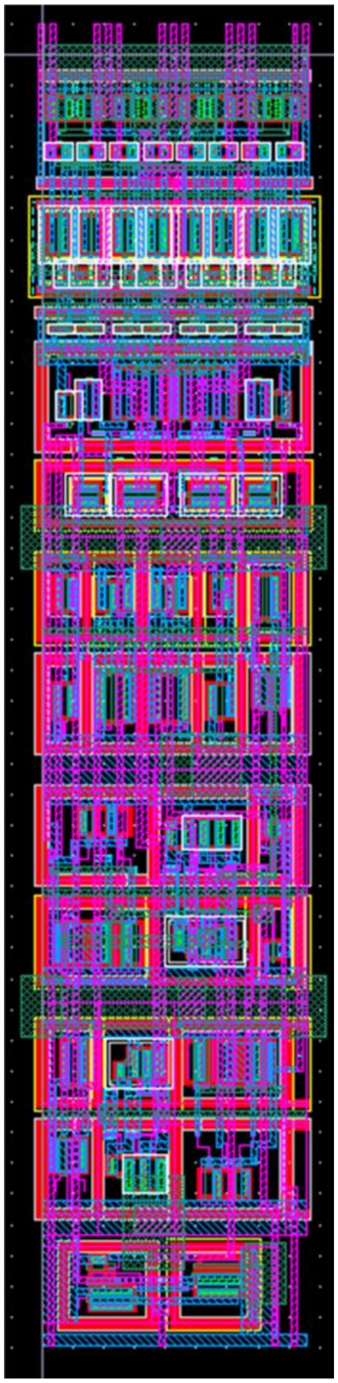

Fig. 3.20 - Column access layout - 4 columns

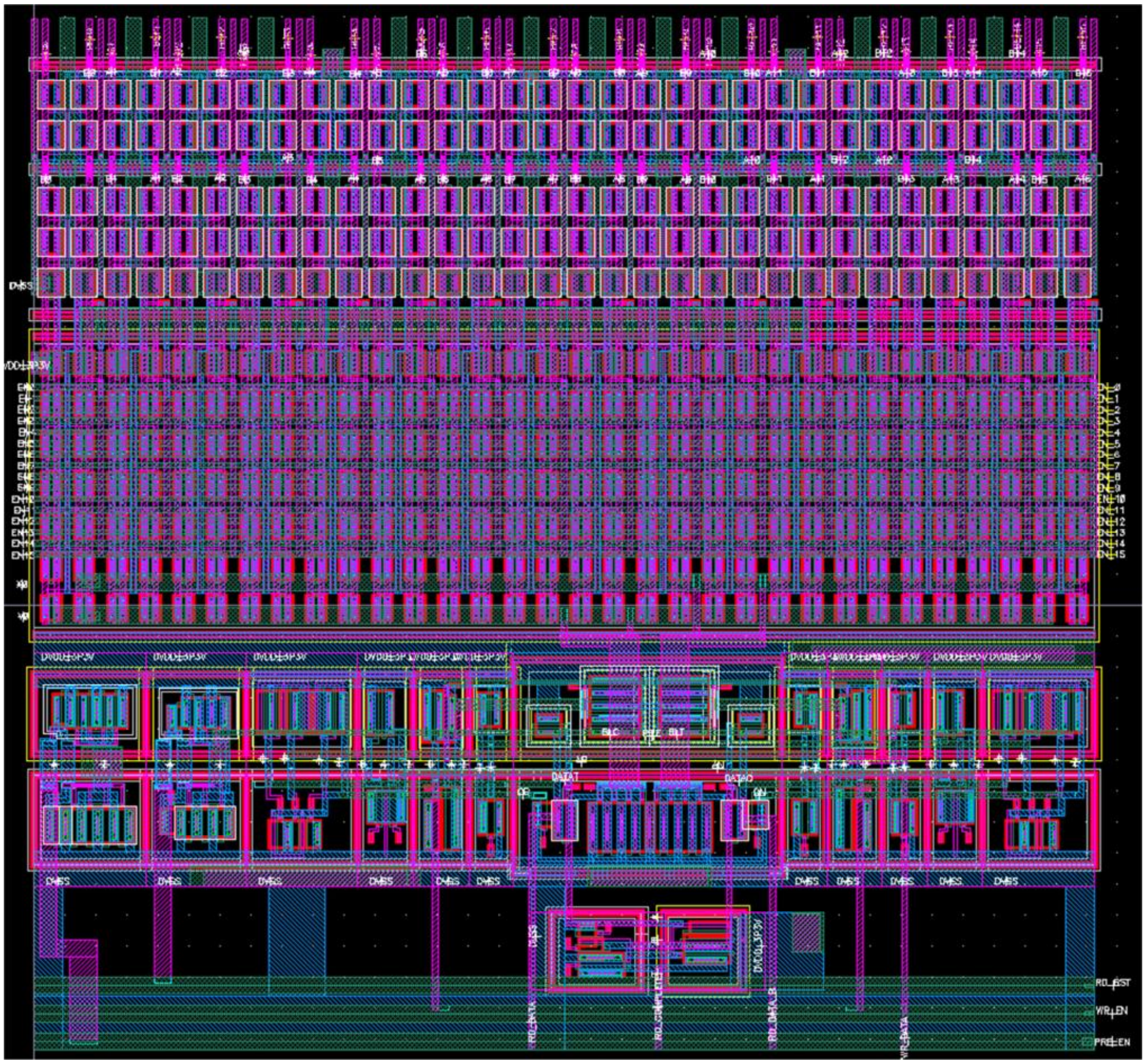

Fig. 3.21 - Column access layout - 16 columns

3.10 Control Logic

Once the array has been built, additional control logic is necessary to convert the raw control signals needed to activate the array into inputs that are compatible with a synchronous system. The target system is a synchronous state machine, and the memory integrates into the state register portion of the state machine diagram. This translates into a system requirement that the memory array processes inputs based on a rising clock edge, and the output must be ready for the following stage by the next rising clock edge.

The control signals on the input of the memory array are active-low, and consist of a reset, block enable, read enable, and write enable. The reset signal is an asynchronous reset that places the array control logic in a known state. The reset will not affect any contents inside the memory array, but asserting the reset signal during a write may cause data corruption. The enable control signals are synchronous inputs that are latched on a rising clock edge. These inputs, along with the clock, must be converted into carefully timed signals for the array control signals. Any mistiming of the control signals has the potential of corrupting data or preventing the data from being read back. Table 3.11 lists the input and output signals of the control block, and Table 3.12 shows the expected functionality for a given set of control inputs.

Table 3.11 - List of Control Inputs and Outputs

Pin Name	Type	Description
CLK	INPUT	Input clock – maximum frequency 25 MHz
RST_L	INPUT	Asynchronous reset for all control logic
MEB	INPUT	Array enable (“Chip Select”)
REB	INPUT	Read Enable
WEB	INPUT	Write Enable (SRAM only)
ADR	INPUT BUS	Active Address
D	INPUT BUS	Write Data
WR_SRC	OUTPUT	Control signal for external input multiplexer (1 kB SRAM only)
A_INT	OUTPUT BUS	Latched address fed to row decoder and column mux
D_INT	OUTPUT BUS	Latched write data fed to write drivers
WL_EN	OUTPUT	Control signal for asserting the word-line driver
PRE_EN	OUTPUT	Control signal for asserting the pre-charge transistors
WR_EN	OUTPUT	Control signal for driving the write data drivers (SRAM only)
RD_RST	OUTPUT	Control signal for resetting the sense amplifier to a blank state

Table 3.12 - Memory Array Operation Truth Table

CLK	RST_L	MEB	REB	WEB	Action
X	0	X	X	X	Reset all control logic
R	1	1	X	X	No operation
R	1	0	0	1	Read from address
R	1	0	X	0	Write to address

Due to the wide temperature, voltage, and process corners considered, the circuit behavior is difficult to control. Specifically, propagation delays and rise/fall times are highly dependent on current circuit conditions. To ensure that the control signals are reliably activated, the signals are generated from the clock signal and outputs of clocked flip-flops. Feedback is also used for controlling the sense amp reset. For signals where sensing completion is difficult, such as writing and pre-charging, plenty of time is allocated to guarantee that the bit-cell and bit-line voltages reach the intended values.

For ease of discussion, only the 1 kB SRAM will be presented as it is the most complicated. The control logic for the other arrays is presented in Appendix B, but the basic structure is the same. There are four basic parts of the control block: state registers, delayed clock pulses, read completion detection, and combinational control signal generation.

The state registers consist of three flip-flops that control the current state of the memory array, and ensure that all conflicting input commands are handled correctly. For the 1 kB SRAM, the read operation consists of a read on the first clock cycle, and a write-back of ECC checked data on the following clock cycle. The *RD_EN* register is active during the first clock cycle of a read, and the write-back register *WB_EN* is activated on the falling edge of the first clock pulse. Writing takes priority over reading, except during a write-back. The formulas for the inputs to the state registers are given below in (3.15)-(3.17).

$$RD\_EN\_D = \overline{REB + MEB + \overline{(WEB \cdot \overline{RD\_EN\_INT})}} \tag{3.15}$$

$$WR\_EN\_D = \overline{\overline{RD\_EN\_INT} \cdot (WEB + MEB)} \tag{3.16}$$

$$WB\_EN\_D = RD\_EN\_INT \tag{3.17}$$

With the stated intent to derive control signals from the clock and the control registers, an issue arises when the clock signal changes before the control registers hold the current data. A delay is needed to ensure that control signals are not generated before the current read or write operation is latched in the state registers. This delay needs to be greater than or equal to the clock-to-Q delay of the control register. By using two flip-flops and XOR gates (Fig. 3.22), a delayed signal is generated that meets the timing requirements.

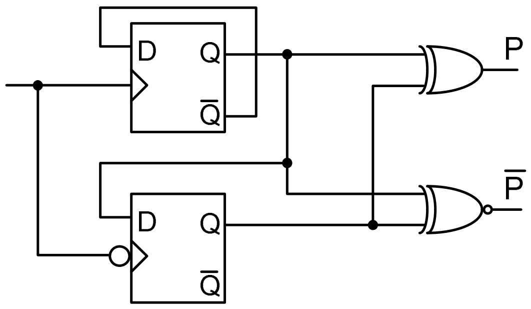

Fig. 3.22 - Control logic delayed clock generator

The combinational logic begins with controlling the input address and data bus latches. These latches are transparent D-type latches. The address latch captures the data starting at the beginning of every operation. If the operation is a write, then the address is only latched for the first half of the clock cycle. However, if the array is issued a read command, then the address must be latched for not only the initial read, but the write-back on the following positive clock cycle. The write-data latch is only latched for the positive clock half to simplify the logic for generating the data latch enable signal.

$$ADDR\_LATCH\_EN = \overline{RD\_EN\_INT + WB\_EN\_INT + \overline{(\overline{CLK} + \overline{P})}} \tag{3.18}$$

$$DATA\_LATCH\_EN = \overline{CLK} + \overline{P} \tag{3.19}$$

The basic array controls for the word-line, pre-charging, and writing are controlled by combinational logic from the state registers and the delayed clock signal. The start time for these signals is much more important than when the signal is de-asserted. As such, these signals are closely tied to the delayed clock signal *P* to ensure that the signals are asserted long enough. Word-line enable and write enable only occur during an active read or write, but pre-charge needs to occur before the rising clock edge. The end result of these timing signals is shown in Fig. 3.23.

$$WL\_EN = \overline{\bar{P} + \overline{(RD\_EN\_INT + WR\_EN\_INT + WB\_EN\_INT)}} \tag{3.20}$$

$$WR\_EN = \overline{\bar{P} + \overline{WR\_EN\_INT} + \overline{(WB\_EN\_INT + WR\_EN\_INT}} \tag{3.21}$$

$$PRE\_EN = \bar{P} \tag{3.22}$$

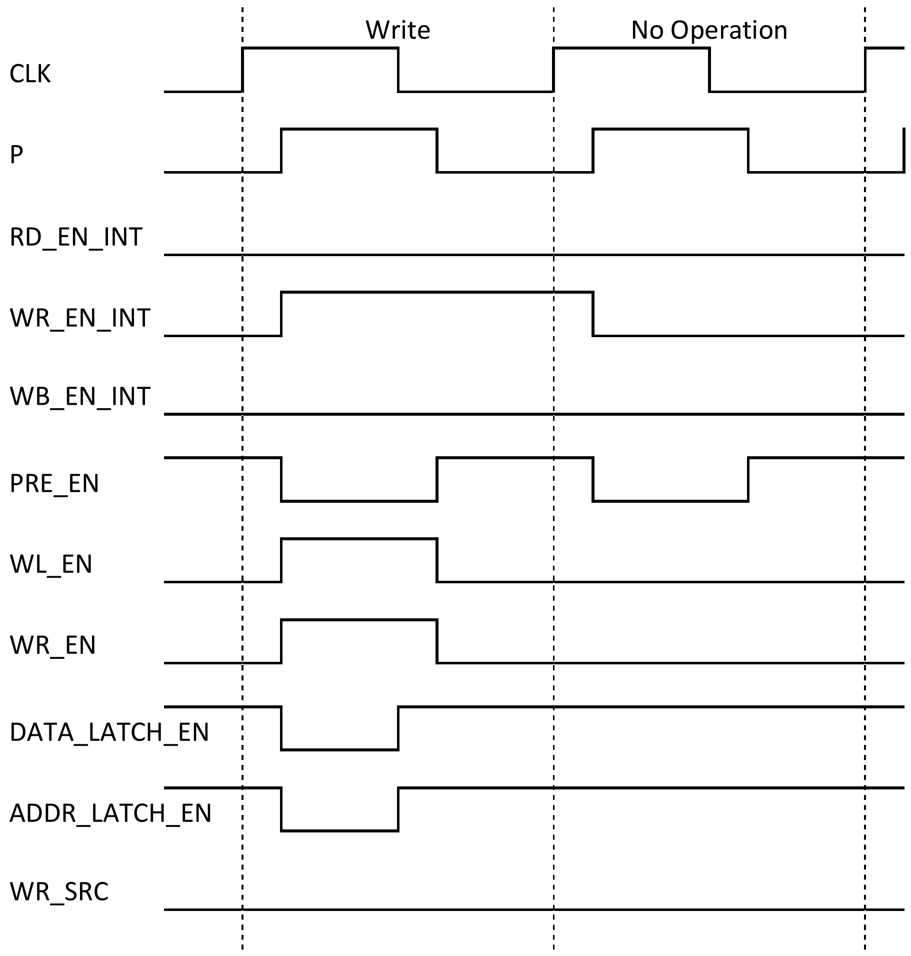

Fig. 3.23 - 1 kB SRAM control signals - write

The sense amplifier latches the data after each read operation; as such, it needs to be cleared before each read. The sense latch should be held in reset at the beginning of a read operation until the controller verifies that the sense latches are ready for a new value. The *RD_N_CLEAR* signal is the result of taking all the *READ_COMPLETE* signals from all the sense latches and OR-ing them together. If one latch still contains data, the *RD_N_CLEAR* signal will be asserted. As a result, *RD_N_CLEAR* is always asserted except during the resetting of the sense latch during a read operation. At the start of a read cycle, S will be '0', and $\bar{S}$ will be '1'. This results in the *RD_RST* signal going high as soon as the rising clock edge arrives. The reset signal will be held high until the *RD_N_CLEAR* signal goes low, which causes S to stay high for the remainder of the positive half of the clock pulse. In turn, this releases the *RD_RST* signal and the sense latches are left to sense and latch the bit-lines.

$$RD\_RST = \bar{S} \cdot CLK \cdot RD\_EN\_D \tag{3.23}$$

$$S = \overline{RD\_N\_CLEAR + \bar{P}} + P \cdot S \tag{3.24}$$

One feature of the 1 kB SRAM is the automatic write-back of ECC corrected data. In order to do this, a multiplexer is placed on the input of the write-data bus that chooses between the input data and the ECC verified output data. While the multiplexer is external to the SRAM macro, the control signal *WR_SRC* is generated internally. A complete diagram of the timing signals during a read operation is given in Fig. 3.24.

$$WR\_SRC = RD\_EN\_INT + WB\_EN\_INT \tag{3.25}$$

With the control logic finalized, the final layout containing the control logic and the array need to be combined into a final layout. Since the control logic consists entirely of gates from the standard cell library, it is possible to use automated place and route software. Each Cadence schematic was exported using the Verilog netlister, and the control logic cells were combined together into a single source code file. The array, row access, and column access circuits were combined together into a square layout, and then an abstract of the layout was created with the LEF Generator. These files and the gate library were imported into the place-and-route tool, Astro [20], and the control logic was placed and routed. The layout was then exported as a GDS2 from Astro into Cadence, and checked to ensure that the layout matched the original Cadence schematic. At this point, the power and ground networks were augmented to reduce parasitic resistance and any possible electro-migration risk. The final layouts for the 64 Byte SRAM (Fig. 3.25), 128 Byte SRAM (Fig. 3.26), 1024 Byte SRAM (Fig. 3.27), and 1024 Byte ROM (Fig. 3.28) are below.

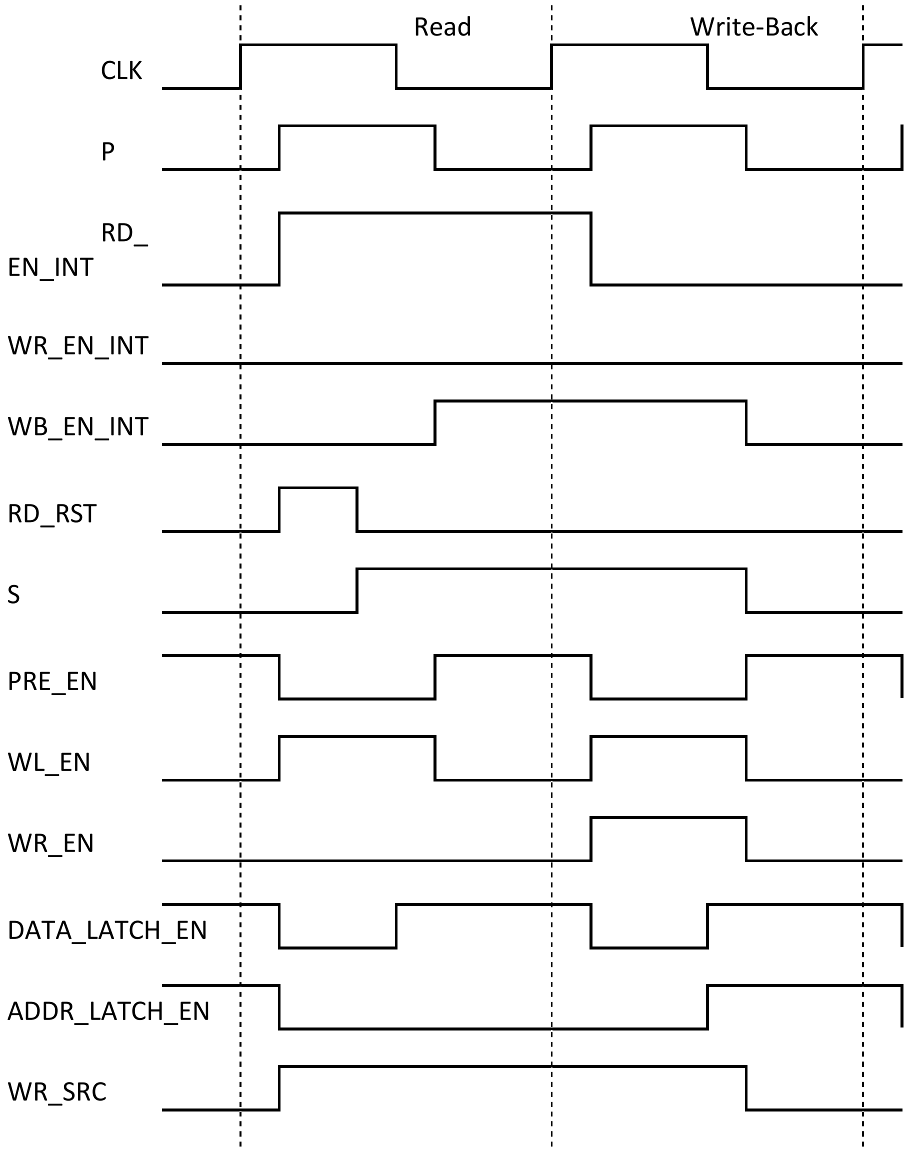

Fig. 3.24 - 1 kB SRAM control signals - read with write-back

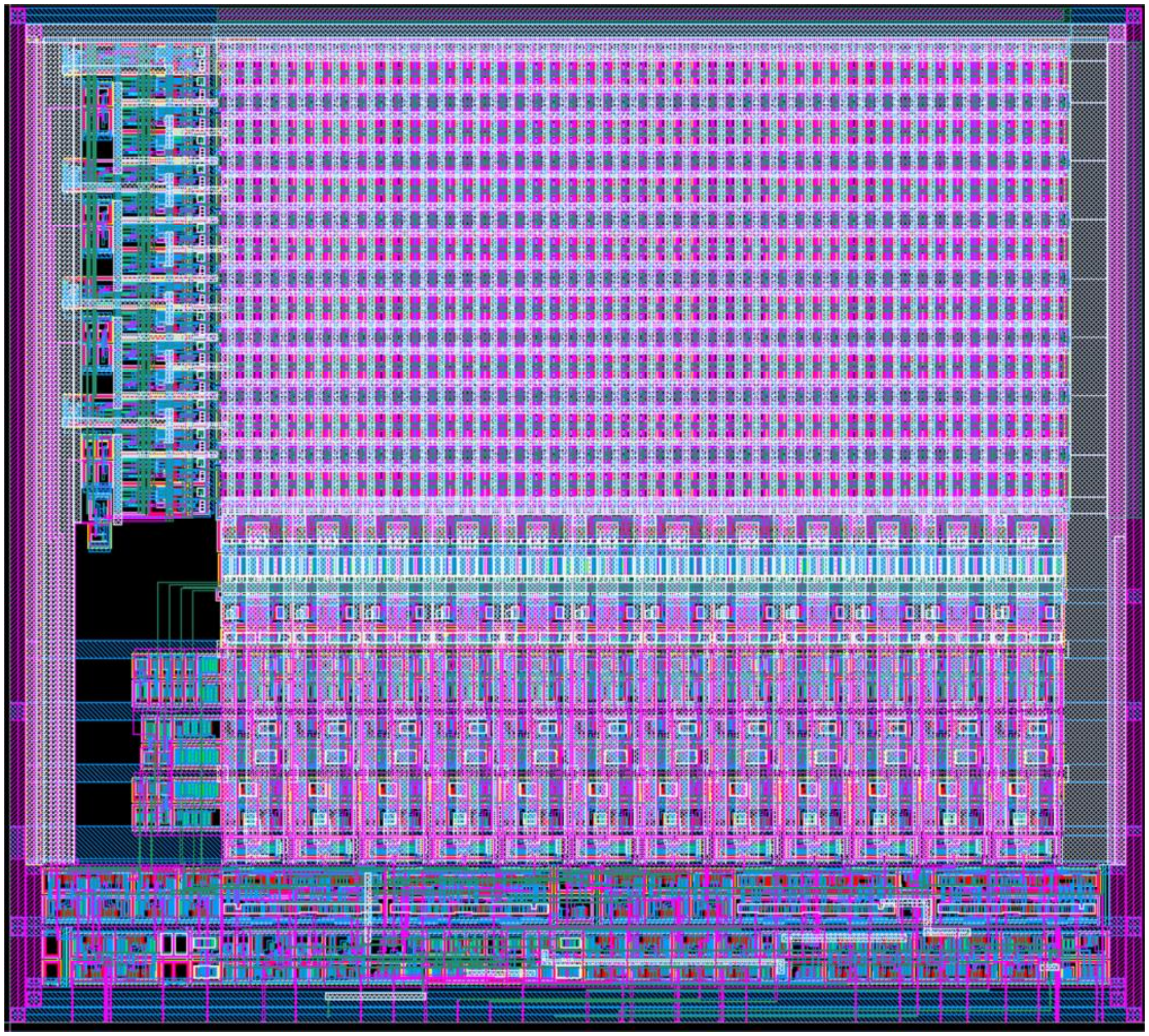
Fig. 3.25 - UARK_SRAM_12X64 layout

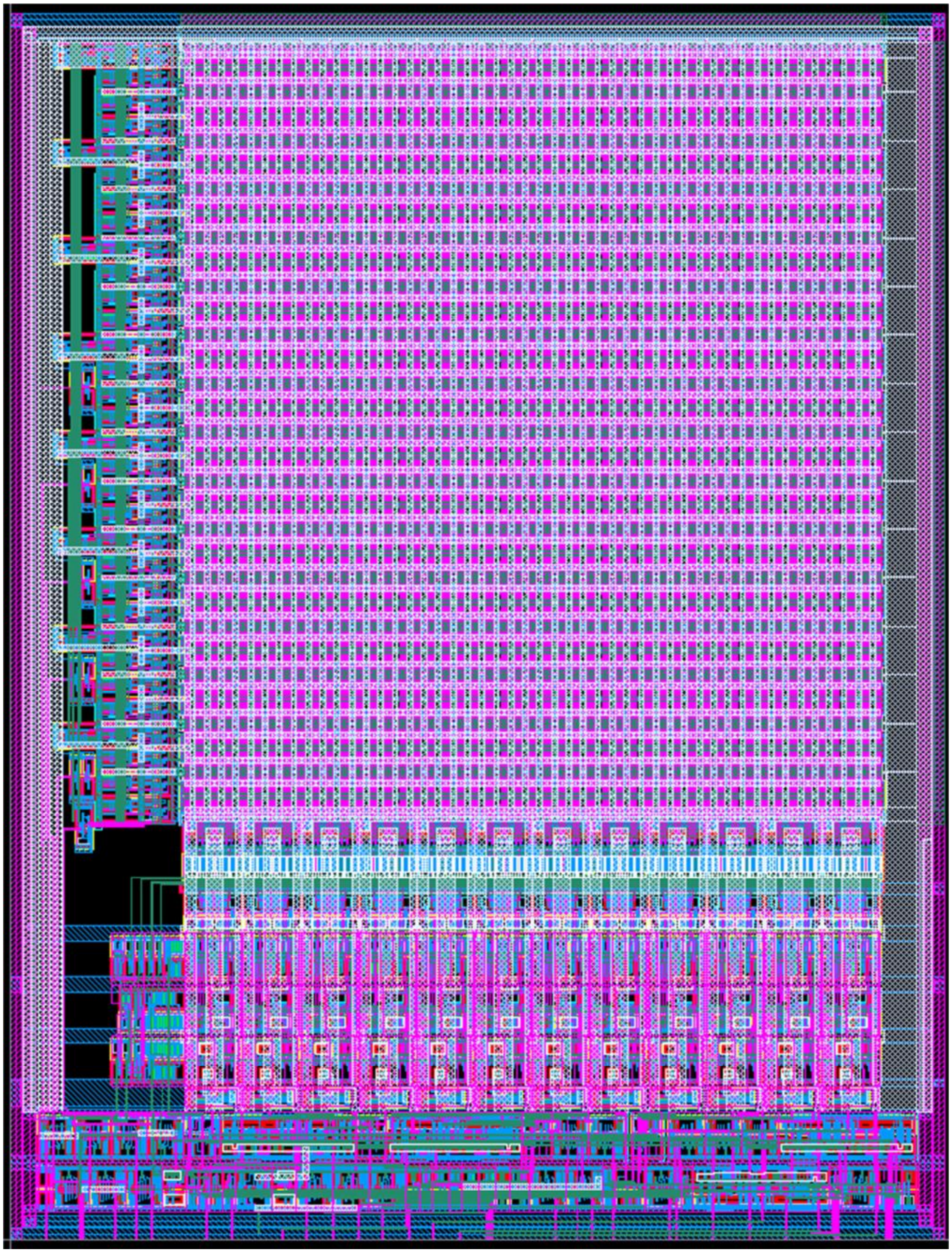

Fig. 3.26 - UARK_SRAM_12X128 layout

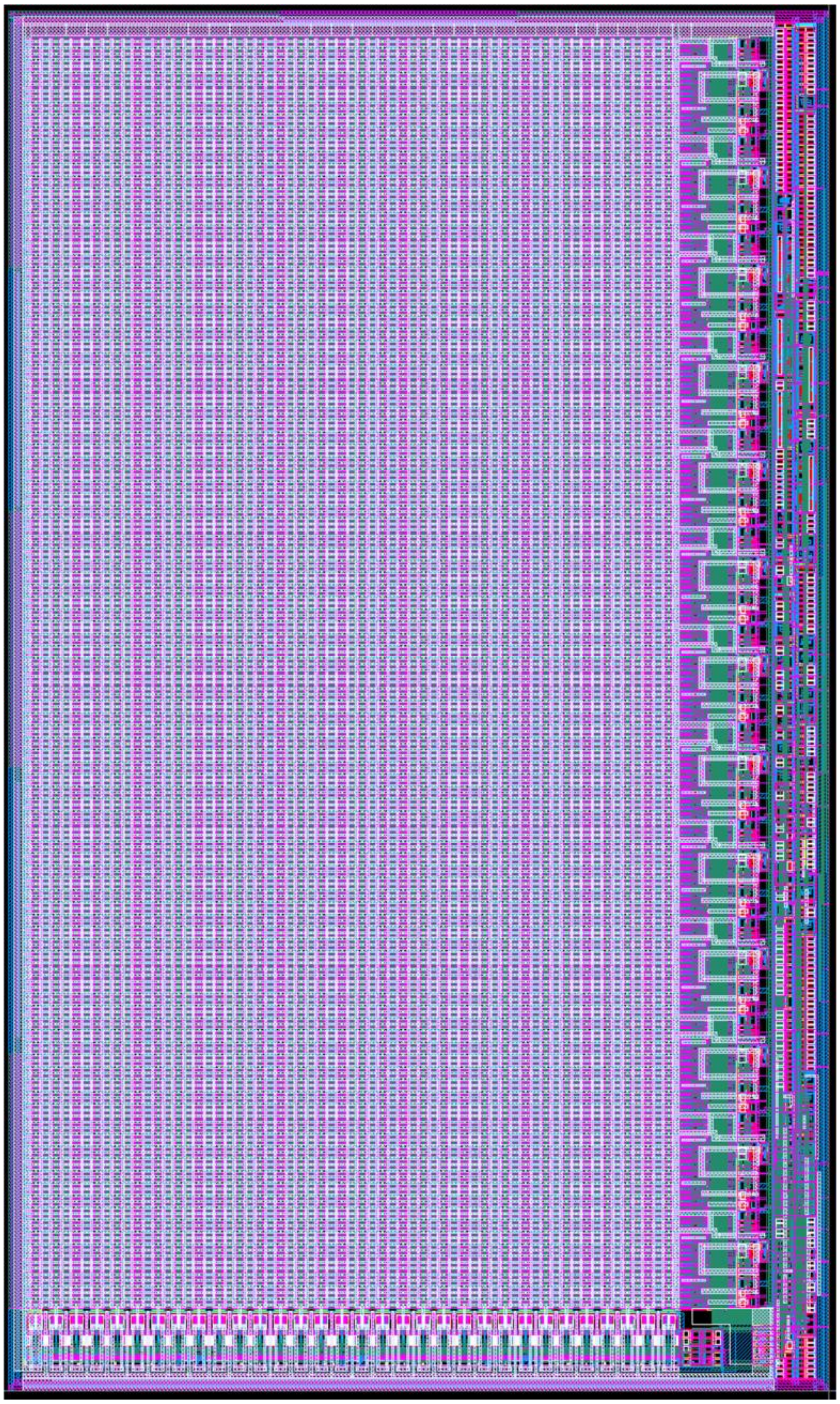

Fig. 3.27 - UARK_SRAM_13X1024 layout

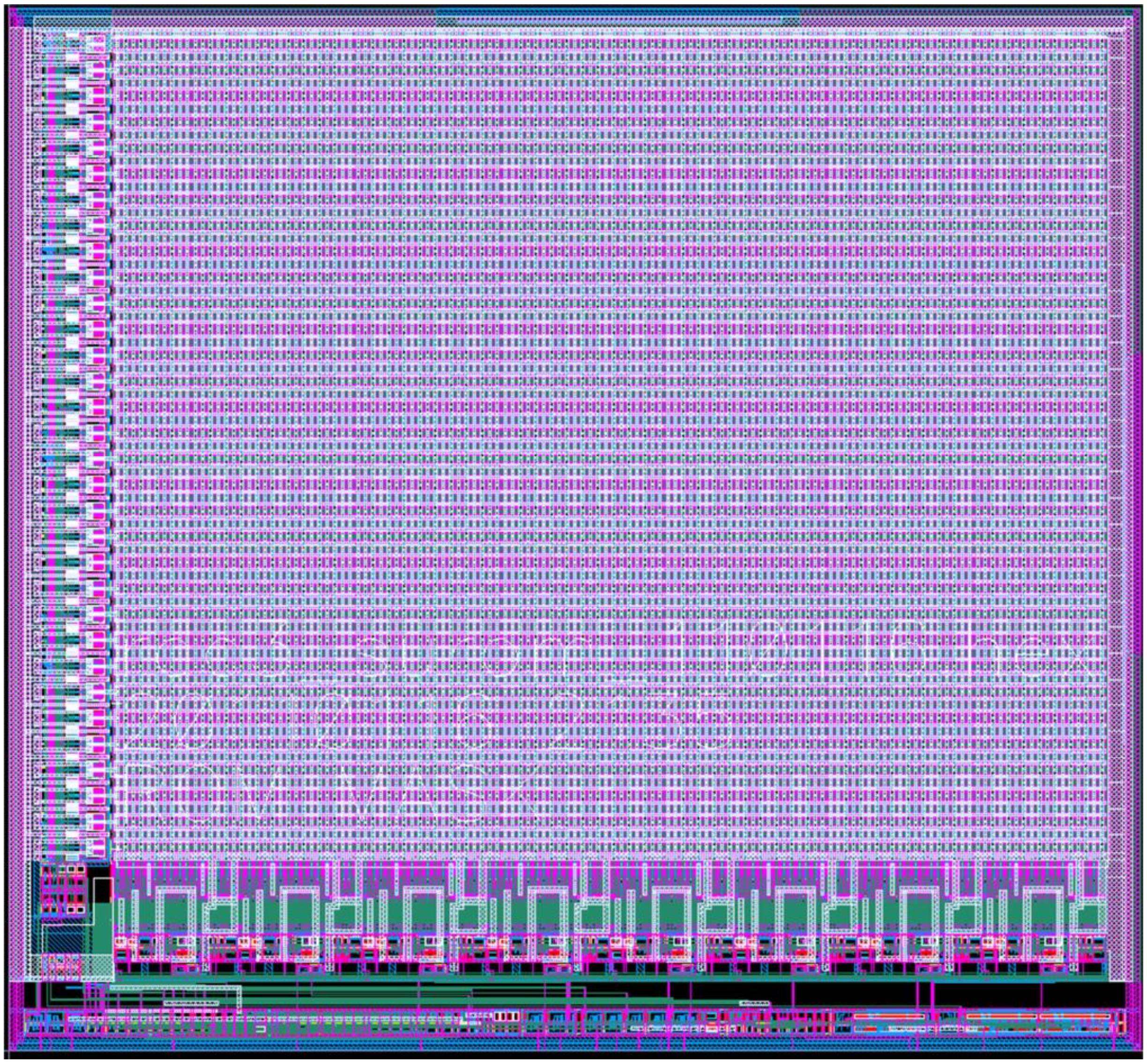

Fig. 3.28 - UARK_ROM_8X1024 layout

3.10.1 Hamming ECC

Error correction in the form of Hamming Error Correction Codes (ECC) is used to provide error detection and correction in the presence of data corruption. Data corruption primarily occurs from manufacturing defects or radiation induced upset. The specification for the SRAM modules is to be able to correct a single bit of data corruption; and in some cases detect a second bit of data corruption.

The Hamming method of Error Detection and Correction (EDAC) [21] calculates a redundant check code from the input data. For an eight bit input, four bits of additional redundant data will be generated and stored alongside the data word. Each check bit is calculated by an XOR operation on different groups of bits. After the data is stored, the check code is generated again from the retrieved data. The new check code is compared to the stored check code by an XOR operation to generate the syndrome. If the syndrome is not equal to zero, then an error has been detected. If only one bit has been corrupted, the value of the syndrome indicates the bit position that needs to be repaired. Double bit errors cannot be distinguished from single bit errors, and three or more errors can even result in a syndrome that indicates no error.

Table 3.13 - Hamming ECC Encoding

■	1	2	3	4	5	6	7	8	9	10	11	12
Original Number			0		1	1	1		0	0	1	1
ECC1	1	■	0	■	1	■	1	■	0	■	1	■
ECC2	■	0	0	■	■	1	1	■	■	0	0	■
ECC4	■	■	■	0	1	1	1	■	■	■	■	1
ECC8	■	■	■	■	■	■	■	0	0	0	1	1

For example, consider the generation of the check code for the byte "01110011". The bits are arranged in Table 3.13 above to facilitate the visualization of the check bits. Each check bit column is highlighted in yellow. The value for each ECC bit is the XOR of all bits shown on the row. For example,

$$ECC4 = 1(5) \oplus 1(6) \oplus 1(7) \oplus 1(12) = 0 \tag{3.26}$$

Now suppose bit 10 becomes switched. When the ECC code is recomputed, the different values are highlighted in red. ECC2 and ECC8 both have changed. The effects of the corrupted bit are highlighted below in Table 3.14 using red.

Table 3.14 - Hamming ECC Single Error Decoding

■	1	2	3	4	5	6	7	8	9	10	11	12
Corrupted Number			0		1	1	1		0	1	1	1
ECC1	1	■	0	■	1	■	1	■	0	■	1	■
ECC2	■	1	0	■	■	1	1	■	■	1	0	■
ECC4	■	■	■	0	1	1	1	■	■	■	■	1
ECC8	■	■	■	■	■	■	■	1	0	1	1	1

Table 3.15 - Hamming ECC Single Error Syndrome Generation

	ECC8	ECC4	ECC2	ECC1
Original	0	0	0	1
Corrupted	1	0	1	1
Syndrome	1	0	1	0

The resulting syndrome, as shown in Table 3.15, is 1010, or the binary value for bit 10. As illustrated, the ECC successfully points towards the switched bit. A simple decoder and XOR array is all that is required to correct the error at this point. Part of the specification for the program memory is the ability to detect when two bits have been corrupted. A basic parity scheme is used to detect when an odd number of bits have been altered. A parity bit is generated by an XOR operation on all bits of the data word and the check code. The additional parity bit is stored alongside the original data word and the check code. When the data is retrieved, the entire stored data word and check code are used to generate a parity bit. If this parity bit matches the parity bit that was stored, then there is either no error present, or an even number of errors. If the parity bits are different, then there are an odd number of errors. Because the Hamming ECC correctly identifies that an error exists from a single or double bit error, this parity scheme can be used to determine how many errors are present.

With the basic structure in place, the ECC design was implemented as behavioral VHDL (Appendix C). The top-level module contains a module for generating the error correction bits, a module for decoding and correcting the output, and the array itself. Additionally, the 1 kB SRAM has a 2-to-1 multiplexer for selecting the correct source for the automatic write-back. The top modules are instanced in the RDC code. At this point, the ECC is integrated into the RDC rather than a part of an individual SRAM array.

Chapter 4 – System Integration and Simulation

With the goal of integrating the memory arrays into the larger RDC design, extensive testing and characterization of the behavior is required. The embedded nature of the arrays leads to limited hardware troubleshooting ability, and faults are catastrophic for the full chip operation.

4.1 Memory Array Characterization

The digital design flow used in the RDC begins with behavioral simulations in ModelSim [22]. After the design passes all test cases, the HDL code is synthesized using Design Compiler [23]. The synthesized design is then placed and routed in Astro [20]. The parasitic capacitances and resistances from the layout are then extracted and combined inside PrimeTime [24] to run Static Timing Analysis (STA) on the design to confirm that setup and hold requirements are met. The results from STA can then be used to generate a list of delays for the circuit for accurate post-layout digital simulations in ModelSim.

Static Timing Analysis is a method for determining propagation delay through a complex digital circuit without running transistor-level simulations for each input and state combination. Before STA is run, all individual blocks and logic gates must be characterized extensively to determine their logical function and propagation characteristics. This detailed characterization is stored in the Liberty format. For combinational gates, a set of formulas or look-up tables characterize the output transition time and propagation delay for a given capacitive output load and input transition time. Sequential gates, such as flip-flops, have additional data characterizing the setup and hold requirements for proper data latching. The input capacitive load of every gate is also included. With this information, the capacitive load of every net can be calculated. All permutations of input changes are switched at a specified transition time. The output transition time and propagation delay for each subsequent gate can be calculated since the net capacitance and input transition time is known. This allows the propagation time for a block of logic to be calculated based on using look-up tables and addition, which is significantly faster and more scalable than the comparable transistor-level simulation. With STA, the entire RDC chip can be simulated in just a few seconds.

In order for STA to produce accurate results, every block and gate must have characterization data available for the tools to use. For combinational and sequential gates, there are tools that exist that can take a set of gate netlists and command files and generate a full characterization for a gate library. With a library of memory arrays, the difficulty of setting up a tool for automatic characterization can often outweigh the effort required to manually simulate and measure the parameters and fill in the library file. Different characterizations are also required for each process/voltage/temperature point that is desired, as shown in table below. As a result, over 100 individual simulations are required to characterize all four arrays completely.

Table 4.1 - List of Liberty Model Characterization Points

Library Name	**Temperature**	**VDD**	**Process Skew**	**Description**
PNN_T25_3P3V	25 ℃	3.3 V	None	Nominal, room temperature
PFF_TM180_3P6V	-180 ℃	3.6 V	+1.5 σ	Fastest corner
PSS_T125_3P0V	125 ℃	3.0 V	-1.5 σ	Slowest corner

Each characterization of each array consists of several simulation runs. The testbench for measuring input capacitance applies a pulse to each input through a resistor, and the capacitance is calculated by taking the measured rise and fall time constant and dividing by the resistance of the source. For determining the output propagation time and output transition time, alternating ones and zeros were read out of each array. This process was repeated for each array a total of nine times to get different values for different input transition times and output capacitive loads. Setup and hold times were measured by delaying the clock signals until the circuit no longer behaved as intended.

The error correction logic was not included in the characterization of the memory. Characterizing the SRAM arrays with ECC would be more difficult for multiple reasons. First, the setup and hold requirements for each input would be drastically different according to the topology of the ECC logic. Outputs would all have different propagation delays as well as rise and fall times for the same reason. Additionally, the propagation delay through the ECC logic is highly dependent on the pattern applied. This means that identifying corner cases for each input and output pin would be required, and there are too many possible combinations of inputs to completely satisfy the problem space. The characterization would take longer due to the additional states required in the simulation. The alternative of characterizing the ECC with the SRAM is to code the ECC logic in VHDL and model it using logic gates and static timing analysis. This allows easy characterization of the memory array, and modeling the ECC with static timing analysis is easy and seamless. The downside is that the ECC logic gates will be mixed in with other logic gates inside the RDC. The arrangement of gates is controlled by the place and route tool, and the ECC delays are different for identical blocks because the ECC layouts are not identical.

4.2 Simulating SRAM Using Vectors from ModelSim

Full-chip simulations for the RDC3 were performed using ModelSim. These simulations used the final netlist from the placed and routed design, as well as timing delays based off the gate libraries and parasitic capacitances extracted from the layout. This results in a fast simulation with good accuracy. Full chip operations are possible, and all higher level functions of the chip can be simulated. However, this simulation method depends on the accuracy of the models. Without a simple way to duplicate the entire simulation at a transistor level, a method was devised to verify the simulation results using the models.

In order to validate the ModelSim simulations, the most complicated and most commonly used patterns were selected to validate the behavior. Five cases were selected that exercised all four types of memory arrays. Each case was located in a ModelSim simulation. The inputs and outputs of the array were exported into a Value Change Dump (VCD) file. The VCD was then adjusted so that the beginning of the VCD began at 200 ns. The VCD file was then converted into the Spectre vector format, or VEC. At this point, additional vectors were added to either store expected data into the array, or read data out of the array so that the vectors could be validated fully. These vectors were then used in transistor-level extracted simulations. The simulations were checked to make sure that the data read and written matched the expected results, and the read operations were checked to make sure the delay from the rising clock edge to the read data output changing matched. Table 4.2 below shows the simulations that were run and the results.

Table 4.2 - Memory Array Simulations Using ModelSim Vectors

Vector File Name	**Memory Block Used**	**Result**
can_data_fifo_writes_in	SRAM_12X64	PASS
sample_accum_2writes_in	SRAM_12X128	PASS
sample_accum_3reads_in	SRAM_12X128	PASS
surom_read_in	ROM_8X1024	PASS
program_base_read_in	SRAM_13X1024	PASS

This simulation method does contain some limitations. In the process of exporting the delay data using a format that ModelSim can use, signal transition times are lost. While this does not significantly impact the simulations inside ModelSim, it reduces the accuracy possible at the transistor level. The load capacitance for the output nets of the memory arrays is not available, resulting in the testbench not being configured to completely emulate the in-circuit conditions. Instead of matching these conditions, the conditions were set to match one of the Liberty model characterization points of 1.0 ns rise/fall times, and 200 fF load capacitance. The delay and rise/fall times of the transistor-level simulation matched the model characterization, and the simulation gave delays that were on target for the ModelSim simulation.

Chapter 5 – Hardware Testing

The SRAM block has gone through many revisions throughout a total of five tape-outs. As more blocks were integrated, the testability of individual SRAM blocks decreased dramatically. In the final release, there is very little testing that can be done directly on the individual SRAM arrays, aside from qualitative testing. While the latest arrays have significant differences from the first design, there is a core of testing results that are valuable and help increase confidence in the final design.

5.1 CRYO3 Setup

The CRYO3 chip consisted of several circuit blocks on a 3 mm by 5 mm tile. These blocks had a combined total of 80 pins, and were assembled and bonded into an 84 pin PLCC package by Arkansas Power Electronics. A die photo is shown below in Fig. 3.15 with the SRAM block highlighted in red. The corresponding pads for the SRAM block are highlighted in yellow. In total, there are thirteen outputs, eleven inputs, and two pairs of power and ground connections.

Due to the high pin count and form factor of the package, none of the printed circuit boards for testing from older projects were suitable for this design. A custom two board design was required to test the circuit at room temperature and over a wide temperature range in a Dewar. The motherboard contained power supply connections, power supply capacitors for each block, jumpers for disconnecting the power to any block, resistors for measuring ground current from each block, and headers for connecting the daughterboard. The daughterboard consisted of an 84-pin PLCC socket with every pin wired to two 44 pin headers. The daughterboard had 4 holes in the corner of the board to mount to the sample plate of the Dewar. The holes for the sample plate are on a 5.5 inch diameter circle, spaced at 45 degree angles. In order to center the chip package on the sample plate and have a stable mounting for the daughterboard, the board dimensions were widened significantly. The headers on the daughterboard were placed with the same spacing and pin order as the motherboard so that the daughterboard could be placed directly above the motherboard for minimal wire length. This configuration is shown connected to the pattern generator and logic analyzer in Fig. 5.2. Headers were placed on the motherboard for logic analyzer and pattern generator connections. Input and output signals were sorted into five input groups and three output groups of eight signals each.

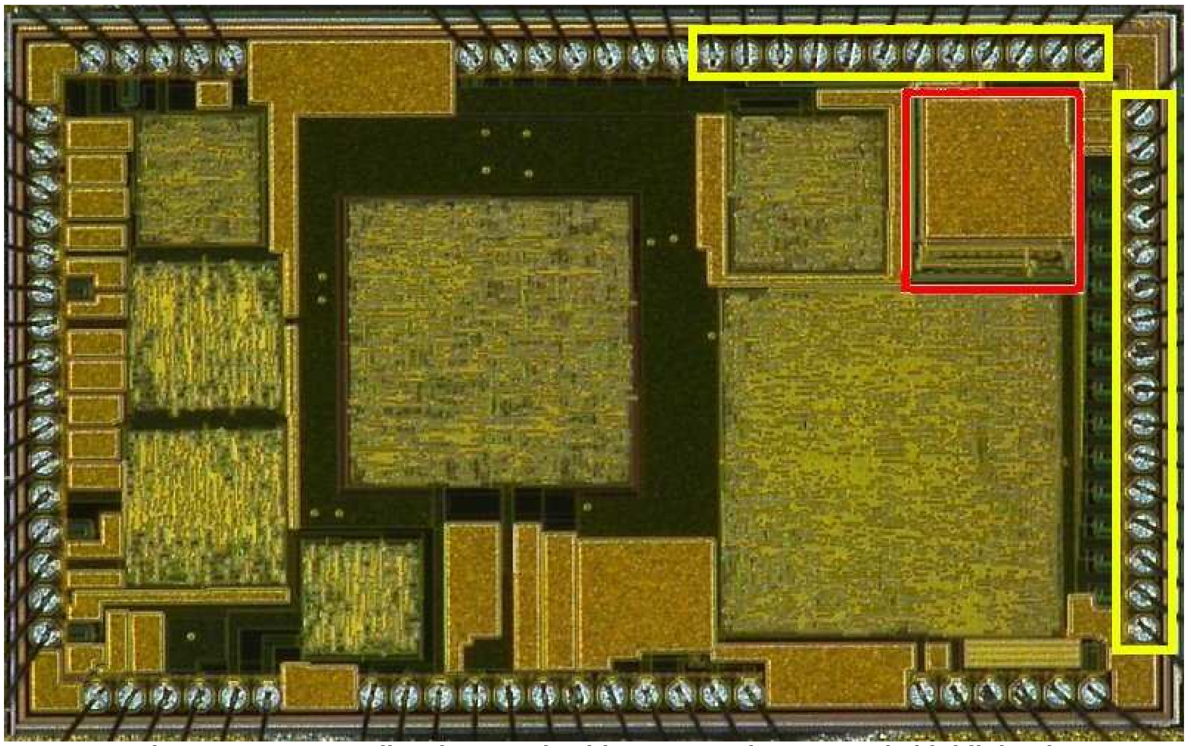
Fig. 5.1 - CRYO 3A die micrograph with SRAM and SRAM pads highlighted

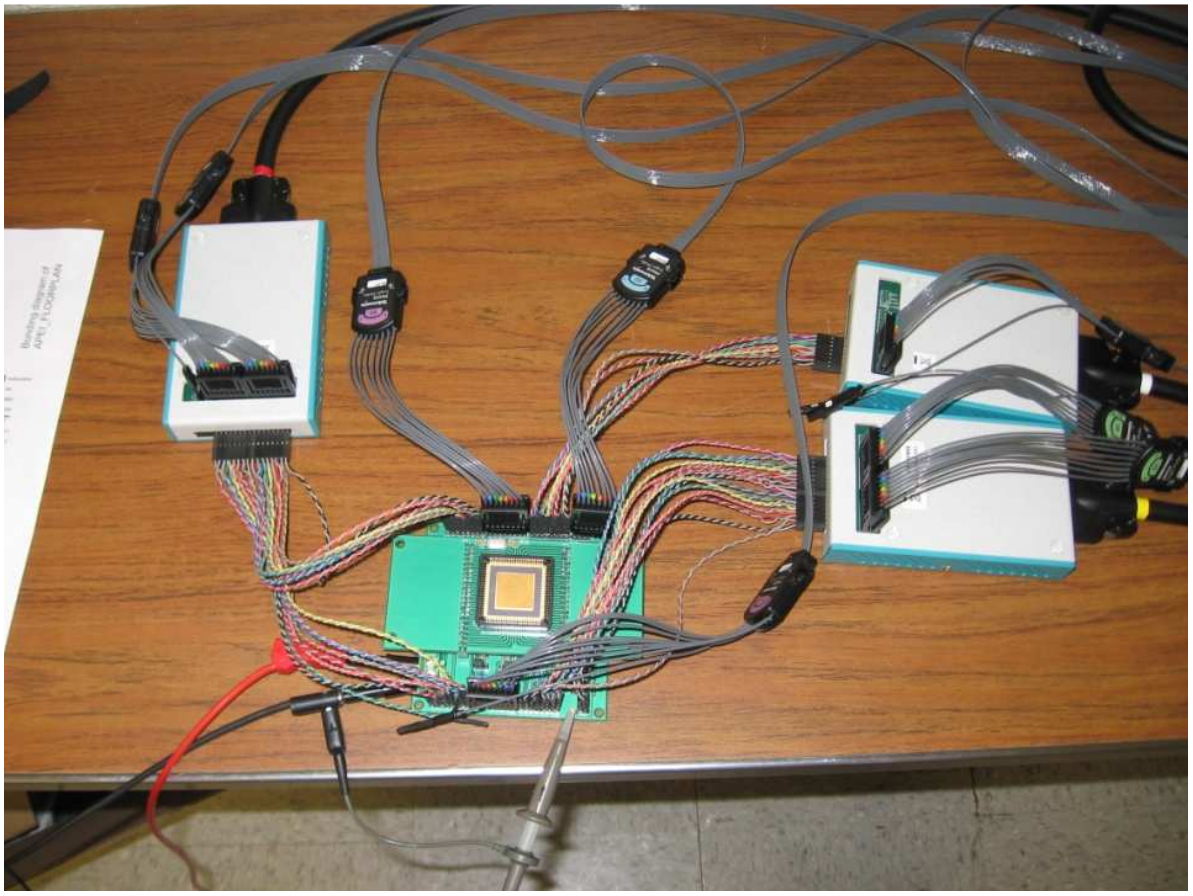

Fig. 5.2 - CRYO 3A testing boards

The initial version of the memory used external signals for every function. No internal timing or automation was included. As such, the timing signals for a read or write operation consist of the multiple phases required to switch internal blocks at the correct time. In addition to complicated control signals, the data and address input are fed through an on-chip shift register. In order to shift in the address and data, 30 different states are required. Reading or writing takes an additional 12 steps.

A pattern generator and logic analyzer combination were used to apply and verify test vectors. Custom Python-language scripts played an essential role in the generation of test vectors and the verification of the resulting logic analyzer outputs. A test regimen was devised to exercise functionality and test for fault conditions. A script then summarized the test plan in terms of the three nominal operations for this version: read, write, and shift in serial data. After a test plan was translated into a ‘script’, another python program converted this script into a vector form readable by the pattern generator software, along with another file containing the expected result. Exact timing of signals can be tweaked for optimal performance at this step without altering the test plan. After importing the pattern vectors into the pattern generator software, this test plan ran an arbitrary number of times. The logic analyzer was connected to all pattern generator outputs at the source, as well as the SRAM data outputs. During a test run, the logic analyzer is programmed to record one sample for every vector the pattern generator outputs. The result is a capture of both the inputs and the outputs of the circuit. The captured data is then exported into a text file and compared against the expected result by another program.

5.1.1 Room Temperature Setup

The room temperature configuration of the motherboard and daughterboard involves connecting the two boards by headers. The resulting wire length is as short as possible with this test setup, reducing the likelihood of transmission line effects or crosstalk. This configuration was ideal for adjusting the phases of the control signals. An eight vector control sequence for the SRAM was finally adapted to provide the shortest pattern possible for a read or write operation. The test setup is shown below in Fig. 5.3.

The primary issue with the test configuration was the inability of the pattern generator to execute a read or write pattern fast enough to test the maximum speed of the circuit. At a maximum clock rate of 200 MHz, the pattern generator outputs a single vector every 5 nanoseconds. This is too coarse a value for many phases of operation, requiring extra time for functionality. As a result, a read or write operation controlled by the pattern generator takes 40 nanoseconds. This is about half of the speed projected by simulation.

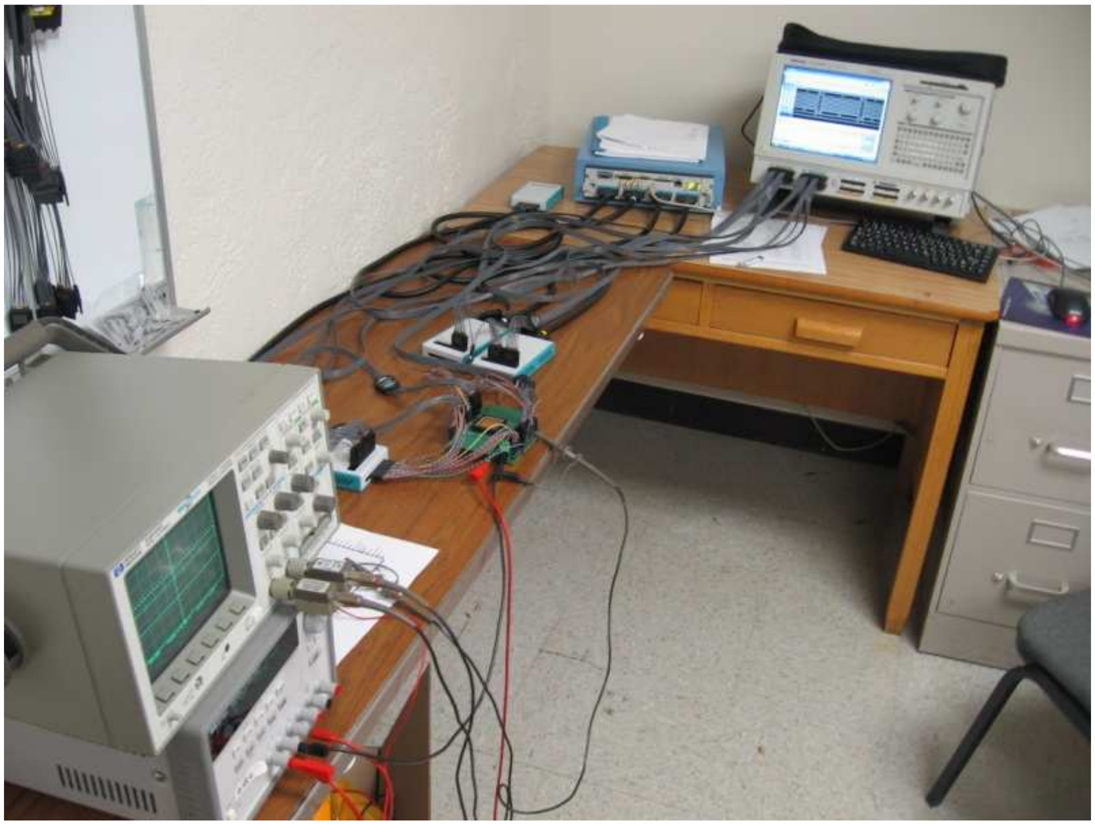

Fig. 5.3 - CRYO 3A room temperature testing setup

5.1.2 Cryogenic Setup

For cryogenic testing of the SRAM, a custom Janus liquid helium Dewar was used. This Dewar, shown below in Fig. 5.4, allows the cooling of a circuit down to about 4 Kelvin at one atmosphere, and even lower by reducing the internal pressure of the Dewar.

The daughterboard was mounted to the sample plate at the bottom of the Dewar, and male headers were installed on the board. Two 44-wire ribbon cables ran from the bottom of the Dewar to the top, where they were soldered to three multiple pin connector sockets. Two ribbon cables from the motherboard were also soldered to the matching multiple pin connectors. This linked the daughterboard to the motherboard in a manner identical to the room temperature setup, except for the length of the wire. This extended length of wire, approximately six feet long, resulted in significant signal integrity issues. The complex control signals required to use the SRAM did not have enough signal integrity at the maximum pattern generator frequency to allow proper memory operation. In fact, the maximum pattern generator frequency in this test setup was only 50 MHz, down from the maximum of 200 MHz used in room temperature testing. Because the testing frequency was not representative of the limits of the circuit, the testing was reduced to a check for functionality at a given voltage and frequency, as well as the power consumption of the circuit. Fig. 5.5 shows all functional operating points tested. As temperature decreases, the ability of the circuit to operate at lower voltages decreases due to the increasing threshold voltage in the bit-cell, so the lower supply voltage cases have a higher minimum temperature.

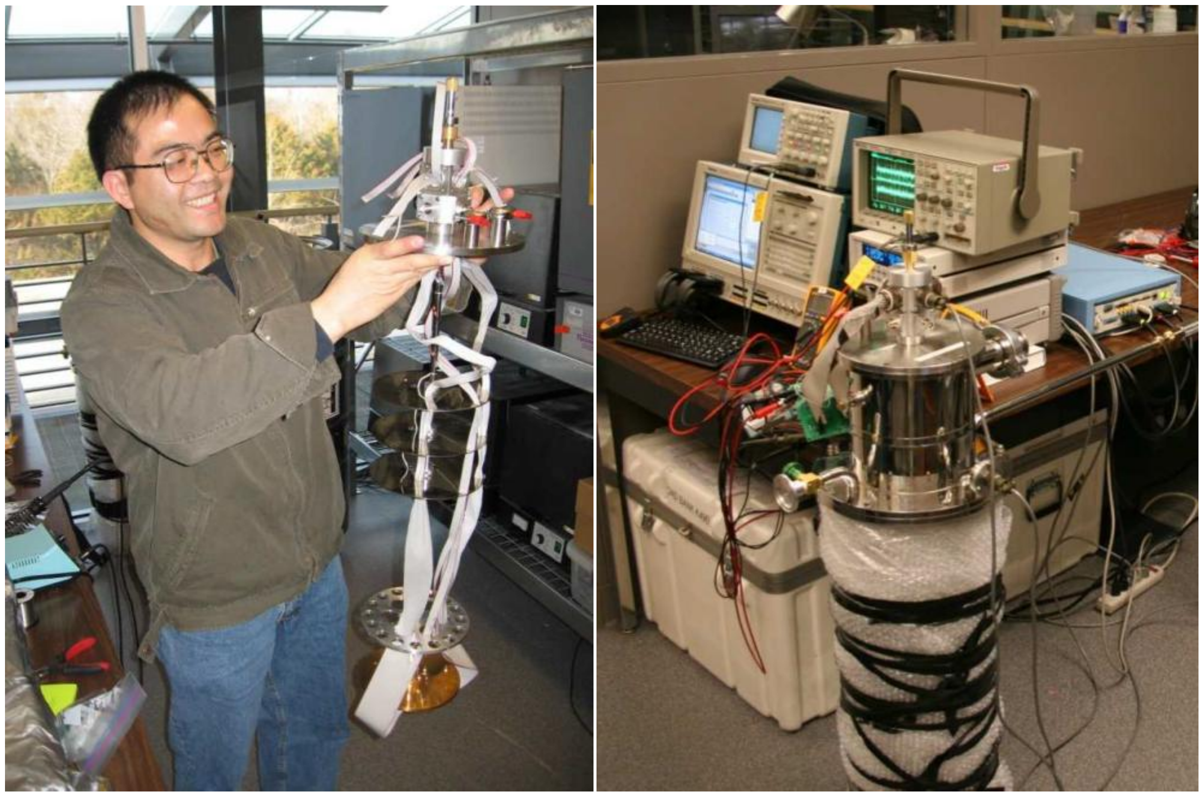

Fig. 5.4 - CRYO 3A cold temperature setup

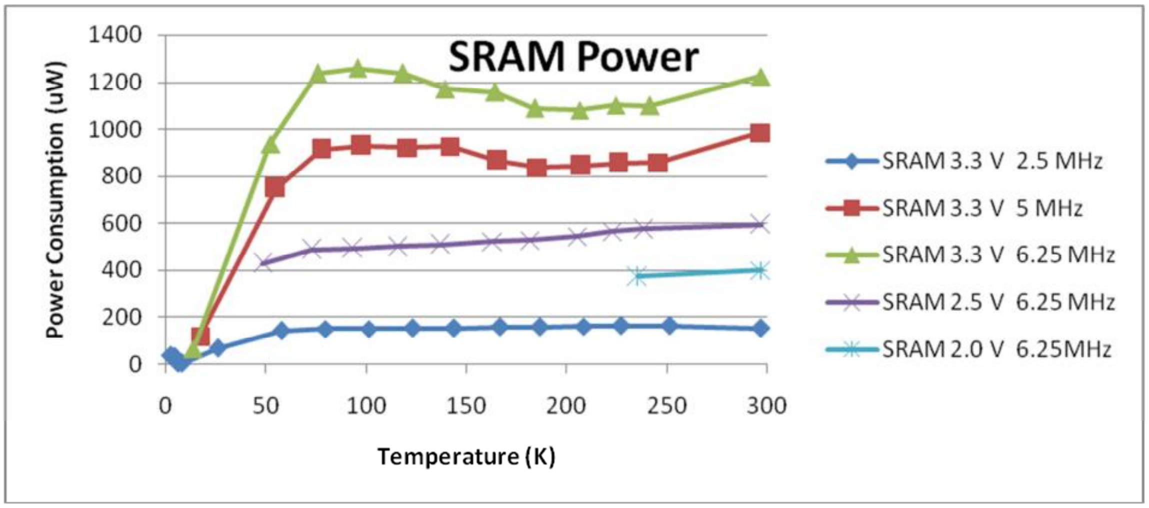

Fig. 5.5 - Measured power consumption over temperature with varying V_{DD} and execution rate

5.2 CRYO4 Testing

The second version of the asynchronous SRAM was fabricated in June 2008. Two major design aspects were changed from the previous design. This version featured an internal control block to generate the most sensitive timing signals inside the design. This changed the number of timing and phase critical signals from nine to two, which would improve high speed testing or testing inside a Dewar or environmental chamber. Data loading was also simplified over the previous design by a parallel data input that was shared between the address and data input. A parallel output could be switched between normal data out and additional control signals.

Unfortunately, an internal inverter in the control block was switched to a non-inverting buffer, as highlighted below in Fig. 5.6. This signal controlled the data source of the ECC block through a multiplexer during a read or a write. Due to the design flaw, the ECC block would take the data to be written from the sense amplifiers instead of the data input bus, and would source the read data from the data input bus instead of the sense amplifiers. For very simple test patterns, the circuit could appear to be working if the data input was set to the value expected from memory. Once large patterns were tested, it became obvious that there was an issue.

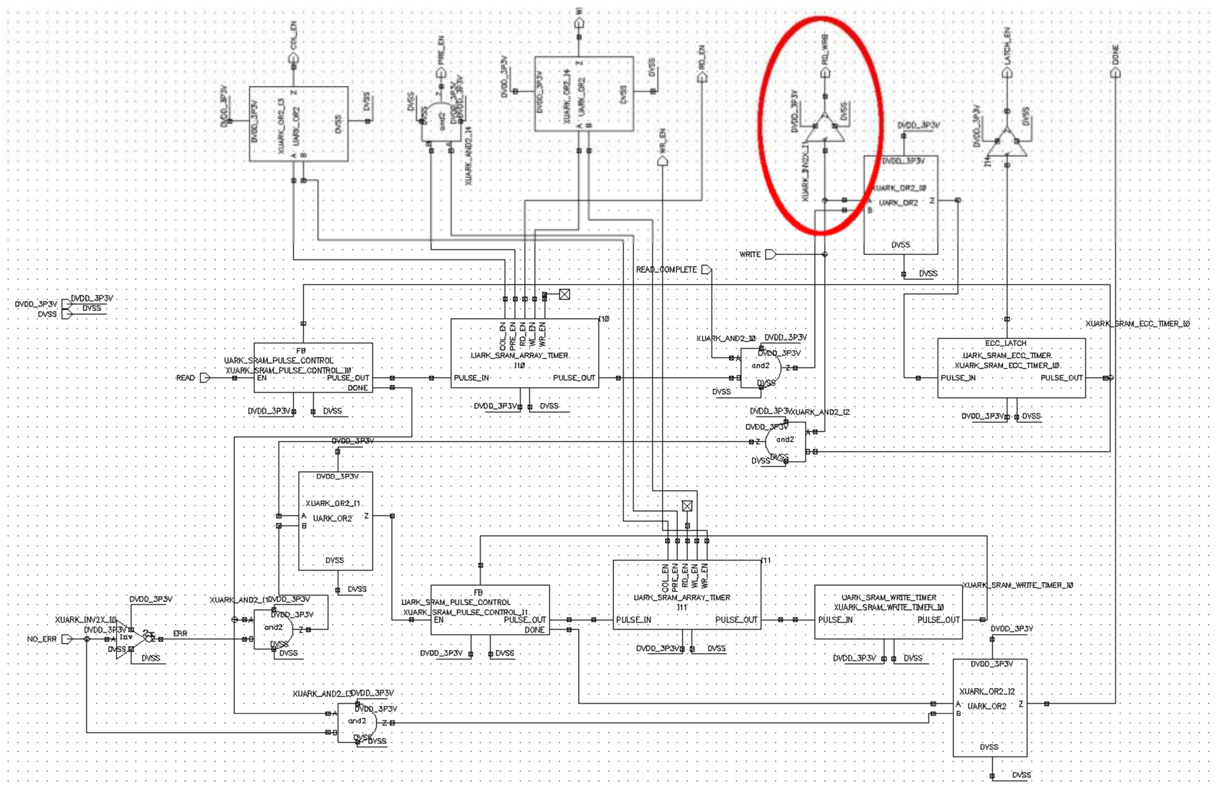
Fig. 5.6 - CRYO 4 logic error. Highlighted buffer should be an inverter.

In order to observe some functionality from the SRAM, an investigation was done to determine if using the control signals in unintended ways could demonstrate a read or write. All the internal control signals switched at the right time, so during a read, the correct value is present on the sense amplifier outputs. The correct data input is also on the data input bus during a write. The internal signal controlling the multiplexer was connected directly to the WR_EN pin, opening the possibility of switching the WR_EN signal at just the right moment to change data sources in time to latch on the output.

To perform a write, it is possible to start a read until the data from the input bus is stored in the latch. Once the data is stored in the latch, READ_EN is set to ‘0’, and WR_EN is set to ‘1’, and the write operation occurs using the correct data source. Reading from the array is harder, because immediately after the sense amplifiers resolve the data, WR_EN needs to be asserted in order to properly select the sense amplifiers as the source for the ECC decoding. Both of these techniques can be used to store data to independent addresses, but a high reliability or repeatability was never achieved. The timing of the

READ_EN and WR_EN signals required significant experimentation to achieve this level of functionality, but the 5 nanosecond minimum timing of the pattern generator made things difficult.

Many issues arose from the modified timings. The state of the memory after a read cycle might not be the same due to the write cycle initiated in the middle of the read. Power data could not be collected because neither of the modified read or write cycles accurately represented the real power of a read or write. Because the internal timings change over temperature, the same read and write patterns that work at room temperature may not work at high temperatures or cryogenic temperatures.

5.2.1 CRYO4 Logic Alteration through Focused Ion Beam Micromachining

In order to achieve meaningful test results verifying the design, the only way was to correct the logic error in the control block. A Focused Ion Beam (FIB) repair was investigated and several points made a FIB repair more feasible. A Focused Ion Beam is a tool used for modification of fabricated circuits. It is capable of removing material or depositing material with features smaller than one micron. The incorrect circuit, a buffer shown in Fig. 5.8, could be modified to an inverter without adding any transistors to the design. The circuit had no upper metals covering the layout hiding essential features, and all operations could be done on a single layer as shown in Fig. 5.7. The downsides to the operation were that the required changes needed to be done on the lowest metal layer, making any repair slow and expensive. The other problem was that the actual repair was complicated – two traces needed to be severed, and two metal areas needed to be bridged.

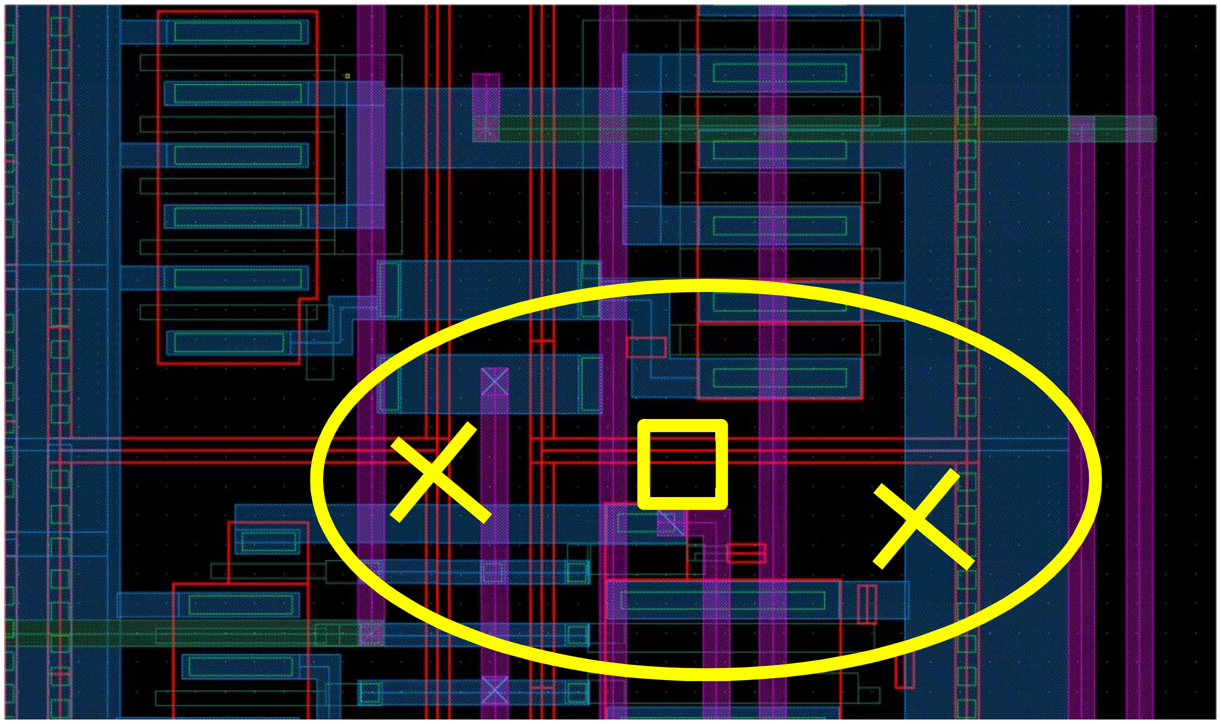
Fig. 5.7 - CRYO 4 FIB layout: window (oval), bridging (square), and cuts (X)

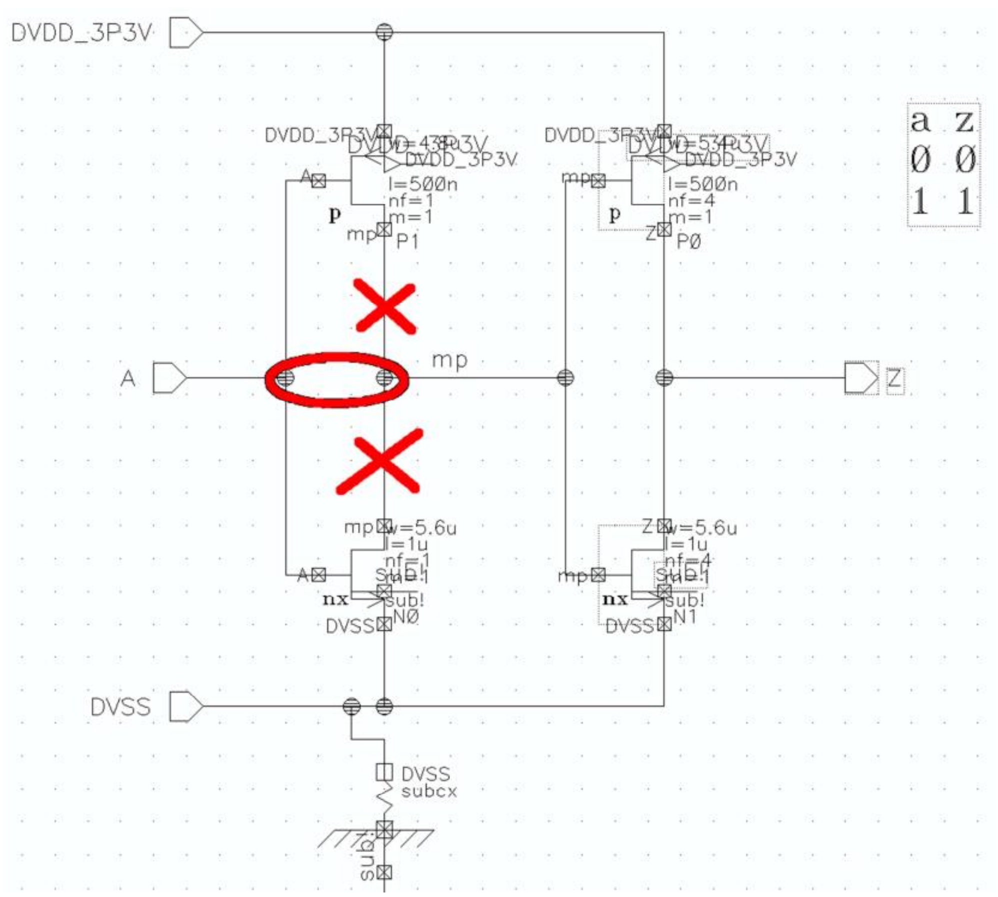

Fig. 5.8 - CRYO 4 FIB schematic: Bridging (oval) and cuts (X)

After receiving the modified part, a simple test vector was created that would illustrate proper functionality. The vector was kept short to reduce the chance of destroying the wire bridge added in the FIB repair. The test plan is to read the uninitialized values from two adjacent memory addresses. Two data values that are inverse (0xAA and 0x55) are written to the two addresses. The data input is then changed to a third value (0xFF) to ensure that the data input was not being passed to the output, and then the two addresses are read again. An original, inoperative part was placed in the test socket, and the pattern was executed. The resulting waveform is shown in Fig. 5.9. The data read from the memory addresses is indicated as 0xFF, which demonstrates that the source of the output data is the data input, not the memory array. After that, the part with the FIB repair was placed in the socket, and the same pattern was applied to the chip. The logic analyzer waveform is shown in Fig. 5.10. As expected, the data read from the initialized array is the same data written to the array.

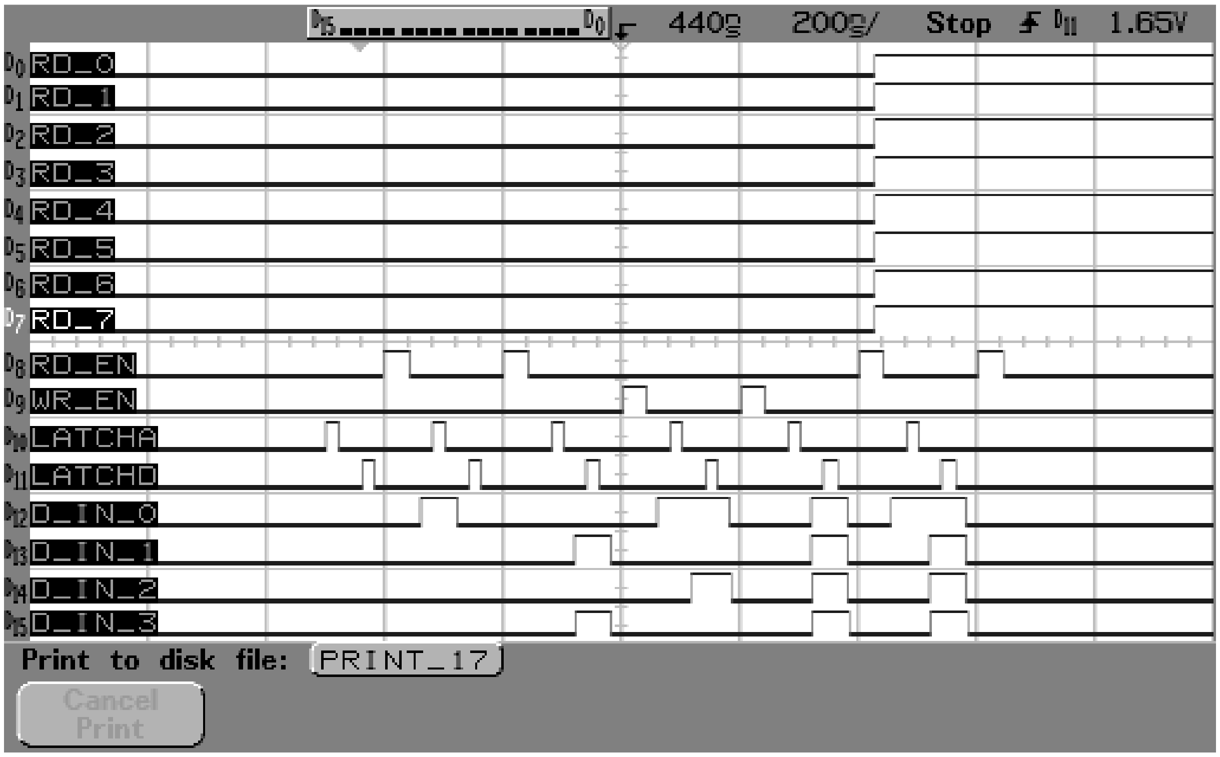

Fig. 5.9 - Logic analyzer trace of CRYO 4 chip without FIB repair.

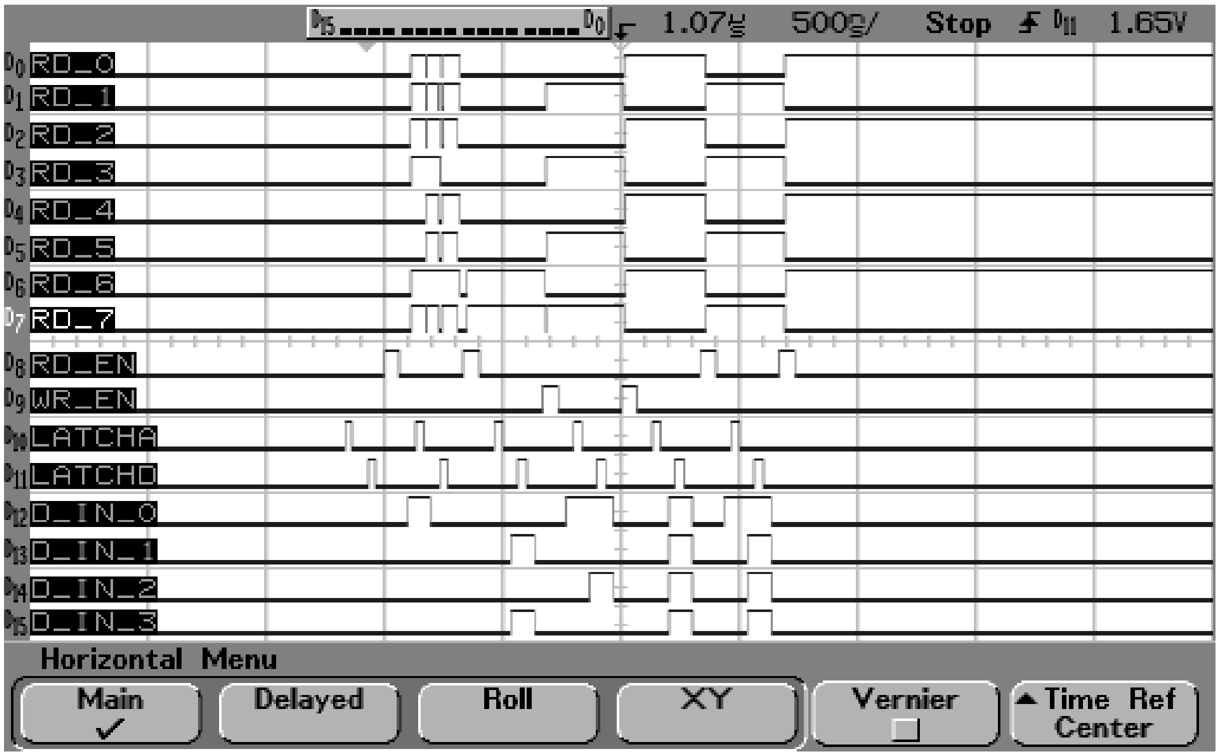

Fig. 5.10 - Logic analyzer of CRYO 4 after FIB. Read data is the top eight rows.

5.2.2 Brookhaven Radiation Latch-up Testing

At high temperatures, two effects increase the likelihood of single event latch-up: increased parasitic resistance and lowered minimum V_{BE} for triggering. A latch-up in any of the digital circuitry could be fatal to the proper operation of the entire REU, so experimental verification of the radiation-hardened features was necessary. The primary technique used to mitigate the chance of latch-up in the digital circuits was the addition of guard rings to every N-well boundary. A normal CRYO4 part was used for the latch-up testing.

Originally, the CRYO3 parts were to be used. However, guard rings were only used in the bit-cells, making the remainder of the circuitry more vulnerable to latch-up. The CRYO4 parts arrived one week before the radiation testing was scheduled. After discovering the logic error in the CRYO4 design, the decision to test the CRYO4 design for latch-up was made. The newer design, while flawed, contained the guard ring structure that would be used on all future digital circuitry and memory arrays. Ideally, single event upsets would be measured, but the test still provided more useful data with the focus on latch-up. For the latch-up testing, the circuit was powered on with a supply voltage of 3.6 Volts. The target temperature of the test was 125 Celsius. The current of the power supply was monitored for any increase in current that would indicate a latch-up condition. During the test, no increase of current was observed through various ion types and angles. A full list of the test conditions is given below in Table 5.1.

The latch-up effect is most likely to occur at high temperatures, when the internal substrate resistance is highest, and the cut-in voltage of the parasitic transistors is the lowest. While high temperatures are the worst case for latch-up, they are not the worst case for Single Event Upsets.

Table 5.1 - CRYO 4 Latch-up Test List

V_{DD}	Sensor Temp	Tilt	Roll	Ion	Effective LET	Time	SEL Observed
3.6 V	~ 25 °C	0°	0°	Ti	20.00	279 s	NONE
3.6 V	> 115 °C	0°	0°	Ti	20.00	173 s	NONE
3.6 V	> 125 °C	0°	0°	Ti	20.00	173 s	NONE
3.6 V	> 125 °C	60°	0°	Ti	40.00	354 s	NONE
3.6 V	> 125 °C	45°	0°	Ti	28.28	246 s	NONE
3.6 V	> 125 °C	45°	0°	Cl	16.55	115 s	NONE
3.6 V	> 125 °C	60°	0°	Cl	23.40	149 s	NONE
3.6 V	> 125 °C	0°	0°	Cl	11.70	70 s	NONE
3.6 V	> 125 °C	0°	0°	I	59.70	87 s	NONE
3.6 V	> 125 °C	60°	90°	Ti	40.00	147 s	NONE
3.6 V	> 125 °C	45°	90°	Ti	28.28	95 s	NONE
3.6 V	> 125 °C	45°	90°	Cl	16.55	119 s	NONE
3.6 V	>125 °C	60°	90°	Cl	23.40	175 s	NONE
3.6 V	>125 °C	45°	90°	O	3.54	988 s	NONE

5.3 CRYO5 and CRYO5A Testing

The CRYO5 release was the first fully integrated version of the RDC. Unfortunately, there were errors in the place and route process that resulted in a non-functional chip. After identifying the issues in the place and route process, other latent issues remained in the design, including several issues with the memory arrays. Since the RDC design was broken, no conclusion about the SRAM performance or functionality could be drawn from the tests. This resulted in several serious issues in the memory arrays passing on to the second revision of the RDC, CRYO5A.

With the CRYO5A release, several issues were identified with the memory arrays that prevented the CRYO5A RDC from functioning properly. The most significant flaw was that the sense amplifiers in the large arrays did not work as intended. It was determined that timing race conditions caused the large arrays to return a constant value of ‘0’. The sense amplifier in the arrays had a very narrow timing window to operate correctly. Unfortunately, this window was easy to miss. As such, the memory arrays returned a constant output value that was produced regardless of the value stored in the arrays. Another issue with the sense amplifiers for the small arrays caused the amplifier to draw excessive current when unexpected bit-line conditions existed. This flaw caused the RDC to draw significantly more DC current than expected. The final issue identified with the memory arrays was that the ROM output bus had been wired incorrectly. Even if the sense amplifier had worked correctly, the output from the ROM would be incorrect. The discovery of these flaws led to a significant overhaul of the design for the sense amplifier and the control logic for the memory arrays.

5.4 RDC3 Testing

The RDC3 design is the final version of the RDC. With this design, there is no direct testability for any of the memory arrays. As a result, quantitative measurements of the memory array timing and power performance are practically impossible. Memory array functionality can be verified through a functional RDC. After the RDC3 design was tested, it was determined that the address and data buses of all the memory arrays were transposed. As a result, the SuROM code was scrambled and unusable. The ROM was readable, and the contents could be manually decoded and verified, but the 8031 core could not access the code. This issue was traced back to improper bus declaration statements in the memory Liberty models. The least invasive fix to the issue would be to reprogram the ROM and fabricate a new chip. The fix would require the changing of only one metal mask (Metal 3), so the run costs would not be as large as a fresh start. All new designs should use a corrected Liberty file as a solution to this issue.

In order to continue testing the design, the 8031 was manipulated using the boundary scan interface to load program data into the program SRAM. This allowed further testing of the RDC3 to the point where CAN messages could be sent and received. The memories performed correctly as tested at 32 MHz, but full speed and full temperature range testing was not performed due to other lingering issues with the RDC3. Fortunately, the tests validate the functionality of all the memory arrays.

Chapter 6 – Summary and Conclusion

Four separate memory arrays were presented in this paper. A general design methodology is presented and discussed for designing a robust and versatile memory array. Design techniques were implemented in order to provide resilience towards SEL, SEU, and TID radiation effects. General wide-temperature strategies were adopted, as well as careful analysis of the system over the entire range of operating conditions. Each sub-block was validated individually and as a part of each memory array. Array simulations demonstrate functionality over the specified ranges for temperature, power supply voltages, and process variation. Error correction logic was implemented in VHDL to allow easy integration into a digital design flow. Characterization simulations for each memory array were run to derive timing and capacitance values. These values were compiled into Liberty model files, and were used throughout the synchronous digital design flow of a digital ASIC, the RDC. The models were then used to generate calculated delays that allowed full chip digital simulations with timing information. The digital simulations were then checked against transistor-level simulations of the memory arrays to validate the accuracy of the chip-level digital simulations. Testing shows that the designs work over the intended wide temperature range and voltage variation range. Latch-up testing indicates that the design will not latch-up under intense radiation. Simulation of the single event upset energy also indicates a sufficiently high threshold for data corruption. The memory arrays have been successfully integrated into an integrated circuit design with a total of eleven instances of four basic designs in the RDC3 release.

Chapter 7 – Future Work

7.1 Layout Integration of ECC into Layout Macro

One potential optimization to the digital flow would include the error correction logic inside the array layout macro, while leaving the ECC logic as a structural Verilog netlist. Placing the ECC logic in the same macro as the control logic compacts the design and reduces parasitic capacitance in the circuit. The characterization of the memory array would require fewer output points since the exact output capacitance for each pin can be measured. The ECC logic delay would also be predictable between macro instances since the layout would not change each time the layout is used. The difficulty lies with implementing the different boundaries in the schematic and layout.

7.2 Explore Different Bit-Cell Transistor Sizes

While the relative transistor drive strength ratios for the bit-cell have served the project well, there is still some latitude allocated with the actual transistor sizes. For example, doubling the length of the NFETs in the bit-cell to 2µm would allow the reduction in width of the PFETs by a factor of two. This could reduce the size and power consumption of the bit-cell. Alternatively, if the widths and lengths of the load and drive transistors were increased to maintain the W/L ratios, the internal gate capacitance could be increased. This would provide additional resistance to SEU in the bit-cell.

7.3 Automation of Array Characterization

Due to unfamiliarity with the waveform viewer used to view HSIM simulations, very little automation of measurements was performed. This increases the risk of human error in transcribing measurements and potentially decreases the accuracy. Rise and fall, delays, input capacitance, and setup/hold measurements can all be automated, and the collected data could then be used to automatically synthesize a new Liberty model. In addition to the accuracy and reliability improvements, more characterization points would be easy to add.

7.4 Improve Speed of Designs

In general, the design decisions tended to favor reliability and flexibility with the design over speed. There are opportunities to decrease the delays in several parts of the circuit. The primary area that could benefit from a speed increase would be the length of time it takes for the output bus to reach a correct value during a read operation. Decreasing this delay would give additional system flexibility to add more logic between the outputs of the SRAM to the next registers.

Bibliography

[1] “MCS-51 8-bit Control-Oriented Microcomputers. 8051 datasheet.” Intel, 1988.

[2] D. J. Pack and S. F. Barrett, *68HC12 Microcontroller*. Prentice Hall, 2001.

[3] R. Garbos-Sanders, L. Melvin, B. Childers, and B. Jambor, “System health management/vehicle health management for future manned space systems,” in *, AIAA/IEEE Digital Avionics Systems Conference, 1997. 16th DASC*, 1997, vol. 2, pp. 8.5-8-8.5-17 vol.2.

[4] Jun Bongim et al., “The Application of RHBD to n-MOSFETs Intended for Use in Cryogenic-Temperature Radiation Environments,” *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, pp. 2100-2105, Dec. 2007.

[5] IBM Microelectronics Division, *SiGeHP (BiCMOS 5HP) Design Manual*. IBM, 2004.

[6] J. D. Cressler, “Silicon-Germanium as an Enabling IC Technology for Extreme Environment Electronics,” in *2008 IEEE Aerospace Conference*, 2008, pp. 1-7.

[7] P. Heremans, G. Van den Bosch, R. Bellens, G. Groeseneken, and H. E. Maes, “Temperature dependence of the channel hot-carrier degradation of n-channel MOSFET’s,” *IEEE Transactions on Electron Devices*, vol. 37, no. 4, pp. 980-993, Apr. 1990.

[8] Tianbing Chen et al., “CMOS Device Reliability for Emerging Cryogenic Space Electronics Applications,” in *Semiconductor Device Research Symposium, 2005 International*, 2005, pp. 328-329.

[9] P. E. Dodd and L. W. Massengill, “Basic mechanisms and modeling of single-event upset in digital microelectronics,” *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 583- 602, Jun. 2003.

[10] S. E. Kerns et al., “The design of radiation-hardened ICs for space: a compendium ofapproaches,” *Proceedings of the IEEE*, vol. 76, no. 11, pp. 1470-1509, Nov. 1988.

[11] D. A. ; P., David Patterson, *Computer Organization and Design: The Hardware/Software Interface Second Edition*. Morgan Kaufmann Pub, 2003.

[12] J. D. Black et al., “HBD layout isolation techniques for multiple node charge collection mitigation,” *IEEE Transactions on Nuclear Science*, vol. 52, no. 6, pp. 2536- 2541, Dec. 2005.

[13] T. Calin, M. Nicolaidis, and R. Velazco, “Upset hardened memory design for submicron CMOS technology,” *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 2874-2878, Dec. 1996.

[14] C. Lee, “A SiGe Standard Cell Library and Digital Design Flow For Extreme Environments,” University of Arkansas, 2011.

[15] *Virtuoso Spectre Circuit Simulator User Guide (5.1.41)*. Cadence Design Systems, 2004, version 5.1.41.

[16] *HSIM Simulation Reference Manual*. Synopsys, Inc, 2008, version A-2008.03.

[17] A. Bumgarner and R. Berger, “Phone Conversations on Memory Design,” 20-Jun-2007.

[18] J. Ross and R. Berger, “Phone Conversation on Memory Design,” 23-Jul-2008.

[19] E. Seevinck, F. J. List, and J. Lohstroh, “Static-noise margin analysis of MOS SRAM cells,” *IEEE Journal of Solid-State Circuits*, vol. 22, no. 5, pp. 748- 754, Oct. 1987.

[20] *Astro User Guide*. Synopsys, Inc, 2007, version Z-2007.03.

[21] R. W. Hamming, “Error detecting and error correcting codes,” *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[22] *ModelSim SE User’s Manual*. Mentor Graphics Corporation, 2008, version 6.4a.

[23] *Design Compiler User Guide*. Synopsys, Inc, 2008, version B-2008.09.

[24] *PrimeTime User Guide: Fundamentals*. Synopsys, 2008, version B-2008.12.

[25] *Liberty User Guide, Volume 1*. Synopsys, Inc, 2003, version 2003.12.

Appendix A – ROM Mask Generation Code

ROM_MASK_GEN.py

```
#####################################################
#  rom_mask_gen.py
#
# This program takes a standard intel hex file and generates a GDS2 mask for
# the UARK_ROM_8X1024 cell. It also creates a verilog netlist of the array
# that can be imported and used for LVS comparisons involving the ROM array.
#
# Supporting libraries used:
# IntelHex
# GDS_Key
#
# Theory of Operation:
# The unprogrammed ROM consists of an array of PFETs that need to be connected
# to either the data-true or data-complement bit-line. This connection is
# made on the third metal layer (MT). A GDS2 file is generated with patches of
# metal shorting the PFETs to either BLC or BLT, along with some information
# useful for version tracking. As the GDS2 file is written, a netlist for the
# array is written as well.
#
# To update the ROM:
# 1. Change the hexfilename declaration below to the latest HEX file
# 2. Run this program
# 3. Import the gds_out.gds2 file into Cadence. For the 8031 SuROM, the
#        cell name was UARK_ROM_MASK_SUROM_110116. After importing the gds2,
#        the PIPO.log should show 8192 rectangles imported on layer 'mt', and
#        4 objects on the text layer.
# 4. Open the layout for the cell 'UARK_ROM_1K_PR_ARRAY' and select the
#        existing ROM mask. Edit the properties, and change the cell name to
#        the new ROM mask layout cell. This ensures that the ROM mask doesn't
#        need to be rotated, flipped, or re-aligned. The HEX file name, date,
#        time, and the words "ROM MASK" should be visible in the layout. Save and
#        close this cell.
# 5. The next step is to import the verilog into a temporary working library. Add
#        "RDC3_UARK_SRAM" to the Reference Libraries list, and enter the path to
#        the "verilog_out.v" file generated by this program. For Verilog Cell
#        Modules, select "Import". Click "OK". The log file should use the cells
#        "UARK_ROM_BIT_0/1".
# 6. Delete the schematic view for the cell "UARK_ROM_1K_FILLED_ARRAY"
# 7. Copy the schematic view of the imported verilog "UARK_ROM_FILLED" to the
#        cell "UARK_ROM_1K_FILLED_ARRAY". Check and save this schematic.
# 8. Open the ROM top cell, "UARK_ROM_8X1024". You should see the updated HEX
#        file name on the layout. The layout should pass all DRC and LVS checks.
#
#





#Input Hex File Name
#hexfilename = 'cryo5a_surom_100111.hex'
hexfilename = 'rdc3_surom_110116.hex'

#Output GDS2 File Name
gdsfilename = 'gds_out.gds2'

#Output verilog file name
```

```
verilogfilename = 'verilog_out.v'

byte_offset = 428
bit_offset = 6848
row_offset = 716
bit_true_offset = 164

#Top Cell Name
topcellname = 'rom_gen_mask'

# GDS_KEY outputs all data through stdout. This will reroute stdout to a gds2 file
import sys, time
print time.localtime()
sys.stdout = open( gdsfilename, "w" )


sys.path.append('intelhex/')
sys.path.append('gds_key/')
from gds_key import *
from intelhex import IntelHex

#Prepare input hex file
ih = IntelHex()
ih.loadhex(hexfilename)

#Generate Timestamp
ts = time.localtime()
timestamp = '%d%02d%02d %02d%02d' % (ts[0], ts[1], ts[2], ts[3], ts[4])

#setup GDS2 stream
set_file_format(GDS_File_Format) #Write to GDSII
set_grid(25E-9) # all dimensions are snapped to the grid (5nm)
set_unit(25E-9) # all dimensions are expressed in this unit (1um)
gds_write(header())  #first thing that should be written
str_el  = el_rect  (layer=63, center=(27392 ,22912),box_size=(54784, 45824)) #Outline
set_text_style (Text_Style_Labels)
str_el += el_text (layer = 63, text = "ROM MASK", alignment = (Key_Align_Left, Key_Align_Top), font = 0, height
= 3000, coordinate = (0, 44824))
str_el += el_text (layer = 63, text = timestamp, alignment = (Key_Align_Left, Key_Align_Top), font = 0, height
= 3000, coordinate = (0, 40824))
str_el += el_text (layer = 63, text = hexfilename, alignment = (Key_Align_Left, Key_Align_Top), font = 0,
height = 3000, coordinate = (0, 36824))

#Setup Verilog Netlist
v_out = open(verilogfilename, "w")

v_out.write('module UARK_ROM_FILLED ( DVDD_3P3V,')
for x in range(8):
  v_out.write('\n\t')
  for y in range(8):
    v_out.write( 'WL%d, ' % ( (x*8 + y), ) )

for x in range(16):
  v_out.write('\n\t')
  for y in range(8):
    v_out.write( 'BLC%d, BLT%d, ' % ( (x*8 + y), (x*8 + y)) )

v_out.write('\n\tDVSS );\n\n')

v_out.write('input\tDVDD_3P3V, DVSS;\n')
for x in range(8):
  v_out.write('input\t')
  for y in range(7):
    v_out.write( 'WL%d, ' % ( (x*8 + y), ) )
  v_out.write( 'WL%d;\n' % ( (x*8 + 7), ) )

for x in range(16):
  v_out.write('inout\t')
  for y in range(7):
    v_out.write( 'BLT%d, ' % ( (x*8 + y), ) )
  v_out.write( 'BLT%d;\n' % ( (x*8 + 7), ) )

for x in range(16):
  v_out.write('inout\t')
  for y in range(7):
    v_out.write( 'BLC%d, ' % ( (x*8 + y), ) )
  v_out.write( 'BLC%d;\n' % ( (x*8 + 7), ) )



#Main Generation Loop
```

```
for x in range(64): #Row
    string_out = ''
    y_offset = x * row_offset
    for y in range(16): #column
        addr = y + 16*x
        string_out += "%4d:%02x " % (addr, ih[addr])
        #Break byte into bit
        current_byte = ih[addr]
        for z in range(8): #bit
            bit_mask = 1 << z
            if (bit_mask & current_byte) > 0:
                bit_value = 1
            else:
                bit_value = 0
            x_offset = y * byte_offset + z * bit_offset + bit_value * bit_true_offset

            str_el  += el_rect  (layer=33, center=(x_offset ,y_offset),box_size=(84, 80)) #rectangle
            v_out.write( '\nUARK_ROM_BIT_%d ROM_ADDR_%d_BIT_%d ( .DVDD_3P3V(DVDD_3P3V), .DVSS(DVSS), .WL(WL%d), 
.BLC(BLC%d), .BLT(BLT%d));' % ( bit_value, x*16+y, z, x, y+16*z, y+16*z))

    sys.stderr.write(string_out+'\n')



#Clean-up
v_out.write('\n\nendmodule')
gds_write(comment("Top layout structure"))
gds_write(make_str(topcellname, str_el))
gds_write(footer())   #close the file
```

Appendix B – Memory Array Control Logic Equations

Control Logic Equations for the 64 Byte and 128 Byte SRAM:

$$WL\_EN = \overline{\overline{P} + \overline{RD\_EN\_INT + WR\_EN\_INT}} \qquad \textbf{(A.1)}$$

$$RD\_RST = \bar{S} \cdot CLK \cdot RD\_EN\_D \qquad \textbf{(A.2)}$$

$$WR\_EN = \overline{\overline{P} + \overline{WR\_EN\_INT}} \qquad \textbf{(A.3)}$$

$$LATCH\_EN = \overline{\overline{P} + CLK} \qquad \textbf{(A.4)}$$

$$PRE\_EN = \overline{\overline{P} + CLK} \qquad \textbf{(A.5)}$$

$$S = \overline{RD\_N\_CLEAR + \overline{CLK}} + CLK \cdot S \qquad \textbf{(A.6)}$$

$$RD\_EN\_D = \overline{REB + MEB + \overline{WEB}} \qquad \textbf{(A.7)}$$

$$WR\_EN\_D = \overline{WEB + MEB} \qquad \textbf{(A.8)}$$

Control Logic Equations for the 1024 Byte ROM:

$$WL\_EN = \overline{\overline{P} + \overline{RD\_EN\_INT}} \qquad \textbf{(A.9)}$$

$$RD\_RST = \bar{S} \cdot CLK \cdot RD\_EN\_D \qquad \textbf{(A.10)}$$

$$LATCH\_EN = \overline{P + CLK} \qquad \textbf{(A.11)}$$

$$PRE\_EN = \overline{P + CLK} \qquad \textbf{(A.12)}$$

$$S = \overline{RD\_N\_CLEAR + \overline{CLK}} + CLK \cdot S \qquad \textbf{(A.13)}$$

$$RD\_EN\_D = \overline{REB + MEB} \qquad \textbf{(A.14)}$$

Appendix C – Error Correction Logic VHDL

ECC_SECDED.VHD

```
-----------------------------------------------
-- Error Correction Code Interface
-- Single Error Correction, Double Error Detection
--
-- This block is intended to be a wrapper for a memory array that generates
-- error correction data for an 8-bit wide data input. The codes are then
-- decoded after a read, and if an error is present, it is corrected.
--
-- The target application is a 1k x 13 SRAM array, with the capability of
-- performing a write on a falling clock edge after a read. This will allow
-- the automatic and transparent correction of single bit errors during a
-- single clock cycle.
--
-- First Edit: August 24, 2010
-- Latest Edit: August 24, 2010
-- Author: Matthew Barlow
--------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ecc_demux is
  port (
    A   : in  std_logic_vector (3 downto 0);
    Y   : out std_logic_vector (11 downto 0);
    NE  : out std_logic;
    SE  : out std_logic
    );
end ecc_demux;
architecture behav of ecc_demux is
begin
  process(A)
  begin

    case A is
    when "0000" => -- No error condition
      Y <= "000000000000";
      NE <= '1';
      SE <= '0';
    when "0001" => -- One error ...
      Y <= "000000000001";
      NE <= '0';
      SE <= '1';
     when "0010" =>
      Y <= "000000000010";
      NE <= '0';
      SE <= '1';
     when "0011" =>
      Y <= "000000000100";
      NE <= '0';
      SE <= '1';
      when "0100" =>
      Y <= "000000001000";
      NE <= '0';
      SE <= '1';
     when "0101" =>
      Y <= "000000010000";
      NE <= '0';
      SE <= '1';
     when "0110" =>
      Y <= "000000100000";
      NE <= '0';
      SE <= '1';
    when "0111" =>
      Y <= "000001000000";
      NE <= '0';
      SE <= '1';
    when "1000" =>
      Y <= "000010000000";
      NE <= '0';
      SE <= '1';
    when "1001" =>
      Y <= "000100000000";
      NE <= '0';
      SE <= '1';
    when "1010" =>
      Y <= "001000000000";
      NE <= '0';
      SE <= '1';
    when "1011" =>
      Y <= "010000000000";
      NE <= '0';
      SE <= '1';
    when "1100" =>
      Y <= "100000000000";
      NE <= '0';
      SE <= '1';
    when others => -- Multiple Error
      -- Detect a non-recoverable error condition. The ECC syndrome indicates
      -- that a bit that does not exist is corrupted. This should be interpreted
      -- as a multiple bit error. Since only a single error can be detected and
      -- corrected, we will just indicate a single error and not change any bits.
      Y <= "000000000000";
      NE <= '0';
      SE <= '1';
    end case;
  end process;
end behav;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity XOR4 is
  port (
    A    : in  std_logic_vector(3 downto 0);
    B    : in  std_logic_vector(11 downto 0);
    Z    : out std_logic_vector(3 downto 0)
    );
end XOR4;
architecture behav of XOR4 is
begin
  process(A,B)
    begin
      Z(0) <= A(0) xor B(0);
      Z(1) <= A(1) xor B(1);
      Z(2) <= A(2) xor B(3);
      Z(3) <= A(3) xor B(7);
  end process;
end behav;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity XOR12 is
  port (
    A, B    : in  std_logic_vector(11 downto 0);
    DATA    : out std_logic_vector(7 downto 0);
    Z       : out std_logic_vector(11 downto 0)
    );
end XOR12;
architecture behav of XOR12 is
begin
```

```
  process(A,B)
    variable Z_int : std_logic_vector(11 downto 0);
    begin


      for i in 11 downto 0 loop
        Z_int(i) := A(i) xor B(i);
      end loop;
      Z <= Z_int;
      DATA <= Z_int(11) & Z_int(10) & Z_int(9) & Z_int(8) & Z_int(6) & Z_int(5) & Z_int(4) & Z_int(2);


  end process;
end behav;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity ECC_CHECK_GEN is
  port (
    READ_IN   : in  std_logic_vector(11 downto 0);
    CHECK     : out std_logic_vector(3 downto 0)
    );
end ECC_CHECK_GEN;
architecture behav of ECC_CHECK_GEN is
begin
  process(READ_IN)
    begin
    CHECK(0) <= READ_IN(2) XOR READ_IN(4) XOR READ_IN(6) XOR READ_IN(8) XOR READ_IN(10);
    CHECK(1) <= READ_IN(2) XOR READ_IN(5) XOR READ_IN(6) XOR READ_IN(9) XOR READ_IN(10);
    CHECK(2) <= READ_IN(4) XOR READ_IN(5) XOR READ_IN(6) XOR READ_IN(11);
    CHECK(3) <= READ_IN(8) XOR READ_IN(9) XOR READ_IN(10) XOR READ_IN(11);
  end process;
end behav;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity ecc_sec_decode is
  port (
    RD_IN       : in  std_logic_vector(11 downto 0);
    RD_OUT      : out std_logic_vector(11 downto 0);
    DATA_OUT    : out std_logic_vector(7 downto 0);
    NO_ERR      : out std_logic;
    ONE_ERR     : out std_logic
    );
end ecc_sec_decode;

architecture behavioral of ecc_sec_decode is
  component ECC_DEMUX
    port (
      A   : in  std_logic_vector (3 downto 0);
      Y   : out std_logic_vector (11 downto 0);
      NE  : out std_logic;
      SE  : out std_logic
      );
  end component;

  component XOR4
    port (
      A     : in  std_logic_vector(3 downto 0);
      B     : in  std_logic_vector(11 downto 0);
      Z     : out std_logic_vector(3 downto 0)
      );
  end component;

  component XOR12
    port (
      A, B  : in  std_logic_vector(11 downto 0);
      DATA  : out std_logic_vector(7 downto 0);
      Z     : out std_logic_vector(11 downto 0)
      );
  end component;

  component ECC_CHECK_GEN
    port (
      READ_IN      : in  std_logic_vector(11 downto 0);
      CHECK        : out std_logic_vector(3 downto 0)
    );
  end component;

  signal INVERT       : std_logic_vector(11 downto 0);
  signal RD_ECC       : std_logic_vector(3 downto 0);
  signal CHECK        : std_logic_vector(3 downto 0);
  signal SYNDROME     : std_logic_vector(3 downto 0);

begin
  G_DEMUX:   ECC_DEMUX port map (SYNDROME, INVERT, NO_ERR, ONE_ERR);-- DEMUX (SYNDROME in, INVERT out)
  G_CHK_GEN: ECC_CHECK_GEN port map (RD_IN, CHECK);
  G_XOR4:    XOR4 port map (CHECK, RD_IN, SYNDROME);
  G_XOR12:   XOR12 port map (RD_IN, INVERT, DATA_OUT, RD_OUT);




end behavioral;


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ecc_sec_encode is
  port(
    WR_IN  : in  std_logic_vector(7 downto 0);
    WR_OUT : out std_logic_vector(11 downto 0)
    );
end ecc_sec_encode;
architecture behavioral of ecc_sec_encode is

begin
  process(WR_IN)
      variable CHECK : std_logic_vector(4 downto 0);
    begin
      CHECK(0) := WR_IN(0) xor WR_IN(1) xor WR_IN(3) xor WR_IN(4) xor WR_IN(6);
      CHECK(1) := WR_IN(0) xor WR_IN(2) xor WR_IN(3) xor WR_IN(5) xor WR_IN(6);
      CHECK(2) := WR_IN(1) xor WR_IN(2) xor WR_IN(3) xor WR_IN(7);
      CHECK(3) := WR_IN(4) xor WR_IN(5) xor WR_IN(6) xor WR_IN(7);

      -- Pack the bits correctly
      WR_OUT  <=  WR_IN(7)  &  WR_IN(6)  &  WR_IN(5)  &  WR_IN(4)  &  CHECK(3)  &  WR_IN(3)  &  WR_IN(2)  &  WR_IN(1)  &
CHECK(2) & WR_IN(0) & CHECK(1) & CHECK(0);
  end process;
end behavioral;



library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity ECC_SEC is
  port (
    WR          : in  std_logic_vector(7 downto 0);
    RD_RAW      : in  std_logic_vector(11 downto 0);
    RD_WRB      : in  std_logic;
    RD          : out std_logic_vector(7 downto 0);
    WR_RAW      : out std_logic_vector(11 downto 0);
    NE, SE, DE  : out std_logic
    );
end ECC_SEC;
architecture df of ECC_SEC is

  component ecc_sec_encode
    port (
    WR_IN  : in  std_logic_vector(7 downto 0);
    WR_OUT : out std_logic_vector(11 downto 0)
    );
  end component;

  component ecc_sec_decode
    port (
    RD_IN       : in  std_logic_vector(11 downto 0);
    RD_OUT      : out std_logic_vector(11 downto 0);
    DATA_OUT    : out std_logic_vector(7 downto 0);
    NO_ERR      : out std_logic;
    ONE_ERR     : out std_logic
    );
  end component;
  signal WR_ECC, RD_ECC : std_logic_vector(11 downto 0);

  begin

    G_ENCODE: ecc_sec_encode port map (WR, WR_ECC);
    G_DECODE: ecc_sec_decode port map (RD_RAW, RD_ECC, RD, NE, SE);

end df;
```

ECC_SEC.VHD

```
-----------------------------------------------
-- Error Correction Code Interface
-- Single Error Correction, Double Error Detection
--
-- This block is intended to be a wrapper for a memory array that generates
-- error correction data for an 8-bit wide data input. The codes are then
-- decoded after a read, and if an error is present, it is corrected.
--
-- The target application is a 1k x 13 SRAM array, with the capability of
-- performing a write on a falling clock edge after a read. This will allow
-- the automatic and transparent correction of single bit errors during a
-- single clock cycle.
--
-- First Edit: August 24, 2010
-- Latest Edit: August 24, 2010
-- Author: Matthew Barlow
--------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ecc_demux is
  port (
    A   : in  std_logic_vector (4 downto 0);
    Y   : out std_logic_vector (12 downto 0);
    NE  : out std_logic;
    SE  : out std_logic;
    DE  : out std_logic
    );
end ecc_demux;
architecture behav of ecc_demux is
begin
  process(A)
  begin

    case A is
    when "00000" => -- No error condition
      Y <= "0000000000000";
      NE <= '1';
      SE <= '0';
      DE <= '0';
    when "10001" => -- One error ...
      Y <= "0000000000001";
      NE <= '0';
      SE <= '1';
      DE <= '0';
     when "10010" =>
      Y <= "0000000000010";
      NE <= '0';
      SE <= '1';
      DE <= '0';
     when "10011" =>
      Y <= "0000000000100";
      NE <= '0';
      SE <= '1';
      DE <= '0';
      when "10100" =>
      Y <= "0000000001000";
      NE <= '0';
      SE <= '1';
      DE <= '0';
     when "10101" =>
      Y <= "0000000010000";
      NE <= '0';
      SE <= '1';
      DE <= '0';
     when "10110" =>
      Y <= "0000000100000";
      NE <= '0';
      SE <= '1';
      DE <= '0';
    when "10111" =>
      Y <= "0000001000000";
      NE <= '0';
      SE <= '1';
      DE <= '0';
    when "11000" =>
      Y <= "0000010000000";
      NE <= '0';
      SE <= '1';
      DE <= '0';
    when "11001" =>
      Y <= "0000100000000";
      NE <= '0';
      SE <= '1';
      DE <= '0';
    when "11010" =>
      Y <= "0001000000000";
      NE <= '0';
      SE <= '1';
      DE <= '0';
    when "11011" =>
      Y <= "0010000000000";
      NE <= '0';
      SE <= '1';
      DE <= '0';
    when "11100" =>
      Y <= "0100000000000";
      NE <= '0';
      SE <= '1';
      DE <= '0';
    when "10000" =>
      Y <= "1000000000000";
      NE <= '0';
      SE <= '1';
      DE <= '0';
    when others => -- Multiple Error
      -- Detect a non-recoverable error condition. This is categorized as a
      -- double error (A = "1XXXX") or an invalid code for A (such as "01110").
      -- There isn't enough information to recover the data, so no attempt
      -- should be made to change the stored value.
      Y <= "0000000000000";
      NE <= '0';
      SE <= '0';
      DE <= '1';
    end case;
  end process;
end behav;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity XOR5 is
  port (
    A    : in  std_logic_vector(4 downto 0);
    B    : in  std_logic_vector(12 downto 0);
    Z    : out std_logic_vector(4 downto 0)
    );
end XOR5;
architecture behav of XOR5 is
begin
  process(A,B)
    begin
      Z(0) <= A(0) xor B(0);
      Z(1) <= A(1) xor B(1);
      Z(2) <= A(2) xor B(3);
      Z(3) <= A(3) xor B(7);
      Z(4) <= A(4) xor B(12);
  end process;

--  process(A,B)
--    variable B_temp : std_logic_vector(4 downto 0);
--    variable Z_temp : std_logic_vector(4 downto 0);
--    begin
--      B_temp := B(12) & B(7) & B(3) & B(1) & B(0);
--      for i in 4 downto 0 loop
--        Z_temp(i) := A(i) xor B(i);
--      end loop;
--      Z <= Z_temp;
--  end process;
end behav;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity XOR13 is
  port (
    A, B    : in  std_logic_vector(12 downto 0);
    DATA    : out std_logic_vector(7 downto 0);
    Z       : out std_logic_vector(12 downto 0)
    );
end XOR13;
architecture behav of XOR13 is
begin
  process(A,B)
    variable Z_int : std_logic_vector(12 downto 0);
    begin


      for i in 12 downto 0 loop
        Z_int(i) := A(i) xor B(i);
      end loop;
      Z <= Z_int;
      DATA <= Z_int(11) & Z_int(10) & Z_int(9) & Z_int(8) & Z_int(6) & Z_int(5) & Z_int(4) & Z_int(2);


  end process;
end behav;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity ECC_CHECK_GEN is
  port (
    READ_IN   : in  std_logic_vector(11 downto 0);
    CHECK     : out std_logic_vector(4 downto 0)
    );
end ECC_CHECK_GEN;
architecture behav of ECC_CHECK_GEN is
begin
  process(READ_IN)
    begin
    CHECK(0) <= READ_IN(2) XOR READ_IN(4) XOR READ_IN(6) XOR READ_IN(8) XOR READ_IN(10);
    CHECK(1) <= READ_IN(2) XOR READ_IN(5) XOR READ_IN(6) XOR READ_IN(9) XOR READ_IN(10);
    CHECK(2) <= READ_IN(4) XOR READ_IN(5) XOR READ_IN(6) XOR READ_IN(11);
    CHECK(3) <= READ_IN(8) XOR READ_IN(9) XOR READ_IN(10) XOR READ_IN(11);
    CHECK(4)  <=  READ_IN(0)  XOR  READ_IN(1)  XOR  READ_IN(3)  XOR  READ_IN(7)  XOR  READ_IN(2)  XOR  READ_IN(4)  XOR
READ_IN(5) XOR READ_IN(6) XOR READ_IN(8) XOR READ_IN(9) XOR READ_IN(10) XOR READ_IN(11);
  end process;
end behav;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity ecc_secded_decode is
  port (
    RD_IN       : in  std_logic_vector(12 downto 0);
    RD_OUT      : out std_logic_vector(12 downto 0);
    DATA_OUT    : out std_logic_vector(7 downto 0);
    NO_ERR      : out std_logic;
    ONE_ERR     : out std_logic;
    TWO_ERR     : out std_logic
    );
end ecc_secded_decode;

architecture behavioral of ecc_secded_decode is
  component ECC_DEMUX
    port (
      A   : in  std_logic_vector (4 downto 0);
      Y   : out std_logic_vector (12 downto 0);
      NE  : out std_logic;
      SE  : out std_logic;
      DE  : out std_logic
      );
  end component;

  component XOR5
    port (
      A     : in  std_logic_vector(4 downto 0);
      B     : in  std_logic_vector(12 downto 0);
      Z     : out std_logic_vector(4 downto 0)
      );
  end component;
  component XOR13
    port (
      A, B  : in  std_logic_vector(12 downto 0);
      DATA  : out std_logic_vector(7 downto 0);
      Z     : out std_logic_vector(12 downto 0)
      );
  end component;

  component ECC_CHECK_GEN
    port (
      READ_IN      : in  std_logic_vector(11 downto 0);
      CHECK        : out std_logic_vector(4 downto 0)
    );
  end component;

  signal INVERT       : std_logic_vector(12 downto 0);
  signal RD_ECC       : std_logic_vector(4 downto 0);
  signal CHECK        : std_logic_vector(4 downto 0);
  signal SYNDROME     : std_logic_vector(4 downto 0);

begin
  G_DEMUX:   ECC_DEMUX port map (SYNDROME, INVERT, NO_ERR, ONE_ERR, TWO_ERR);-- DEMUX (SYNDROME in, INVERT out)
  G_CHK_GEN: ECC_CHECK_GEN port map (RD_IN(11 downto 0), CHECK);
  G_XOR5:    XOR5 port map (CHECK, RD_IN, SYNDROME);
  G_XOR13:   XOR13 port map (RD_IN, INVERT, DATA_OUT, RD_OUT);


end behavioral;


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ecc_secded_encode is
  port(
    WR_IN  : in  std_logic_vector(7 downto 0);
    WR_OUT : out std_logic_vector(12 downto 0)
    );
end ecc_secded_encode;
architecture behavioral of ecc_secded_encode is

begin
  process(WR_IN)
      variable CHECK : std_logic_vector(4 downto 0);
    begin
      CHECK(0) := WR_IN(0) xor WR_IN(1) xor WR_IN(3) xor WR_IN(4) xor WR_IN(6);
      CHECK(1) := WR_IN(0) xor WR_IN(2) xor WR_IN(3) xor WR_IN(5) xor WR_IN(6);
      CHECK(2) := WR_IN(1) xor WR_IN(2) xor WR_IN(3) xor WR_IN(7);
      CHECK(3) := WR_IN(4) xor WR_IN(5) xor WR_IN(6) xor WR_IN(7);
      -- CHECK(4) is all WR_IN bits xor'd with CHECK(3..0).
      -- Simplification has been done to result in a simplified equation
      -- for CHECK(4). Bits 3 and 6 occur an even number of times. The following
      -- logic identities are used: A xor A = 0; A xor 0 = A;
      CHECK(4) := WR_IN(0) xor WR_IN(1) xor WR_IN(2) xor WR_IN(4) xor WR_IN(5) xor WR_IN(7);

      -- Pack the bits correctly
      WR_OUT <= CHECK(4) & WR_IN(7) & WR_IN(6) & WR_IN(5) & WR_IN(4) & CHECK(3) & WR_IN(3) & WR_IN(2) &
WR_IN(1) & CHECK(2) & WR_IN(0) & CHECK(1) & CHECK(0);
  end process;
end behavioral;


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity MUX2x13 is
  port (
    I1    : in  std_logic_vector(12 downto 0);
    I0    : in  std_logic_vector(12 downto 0);
    SEL  : in  std_logic;
    Z    : out std_logic_vector(12 downto 0)
    );
end MUX2X13;
architecture behav of MUX2X13 is
begin
  process(I1,I0,SEL)
    begin
```

```
      if SEL = '1' then
        Z <= I1;
      else
        Z <= I0;
      end if;
  end process;
end behav;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity ECC_SECDED is
  port (
    WR          : in  std_logic_vector(7 downto 0);
    RD_RAW      : in  std_logic_vector(12 downto 0);
    RD_WRB      : in  std_logic;
    RD          : out std_logic_vector(7 downto 0);
    WR_RAW      : out std_logic_vector(12 downto 0);
    NE, SE, DE  : out std_logic
    );
end ECC_SECDED;
architecture df of ECC_SECDED is

  component ecc_secded_encode
    port (
    WR_IN  : in  std_logic_vector(7 downto 0);
    WR_OUT : out std_logic_vector(12 downto 0)
    );
  end component;

  component ecc_secded_decode
    port (
    RD_IN       : in  std_logic_vector(12 downto 0);
    RD_OUT      : out std_logic_vector(12 downto 0);
    DATA_OUT    : out std_logic_vector(7 downto 0);
    NO_ERR      : out std_logic;
    ONE_ERR     : out std_logic;
    TWO_ERR     : out std_logic
    );
  end component;

  component MUX2X13
    port (
      I0    : in  std_logic_vector(12 downto 0);
      I1    : in  std_logic_vector(12 downto 0);
      SEL  : in  std_logic;
      Z    : out std_logic_vector(12 downto 0)
      );
  end component;

  signal WR_ECC, RD_ECC : std_logic_vector(12 downto 0);

  begin

    G_ENCODE: ecc_secded_encode port map (WR, WR_ECC);
    G_DECODE: ecc_secded_decode port map (RD_RAW, RD_ECC, RD, NE, SE, DE);
    G_MUX: MUX2X13 port map ( WR_ECC, RD_ECC, RD_WRB, WR_RAW);

end df;
```

Appendix D – Liberty Models

The Liberty models contain vast amounts of timing data from simulations. These simulations are analyzed and assembled into look-up tables (LUT) and values that can be used during static timing analysis. Pre-existing Liberty models as well as the Liberty User Guide [25] were used to create a working base memory model. Additional libraries for different operating conditions then used the original working model as a template. Below is the nominal case model for all four memory arrays. This model represents the expected behavior and characteristics of the memory arrays at 25 ℃, VDD = 3.3 V, and nominal process skew. The libraries are then compiled through Design Compiler into a Synopsys database, and tested in circuit to verify functionality.

After testing, it was found that the bus declarations in the Liberty model are incorrect. This causes the bit order of the address and data lines to be reversed. While this will not adversely affect any SRAM, ROM data will be unreadable. The original code used in the RDC 3 release is presented here. Any future use must address the bus statements before use.

RDC3_UARK_SRAM_PNN_V330_T25.lib

```
library (RDC3_UARK_SRAM) {
  delay_model : table_lookup ;
  date : "January 13, 2011" ;
  revision : "1.00" ;
  time_unit : 1ns ;
  voltage_unit : 1V ;
  current_unit : 1mA ;
  capacitive_load_unit(1, pf);
  pulling_resistance_unit : 1kohm ;
  leakage_power_unit : 1uW ;
  input_threshold_pct_fall : 50.0 ;
  input_threshold_pct_rise : 50.0 ;
  output_threshold_pct_fall : 50.0 ;
  output_threshold_pct_rise : 50.0 ;
  slew_derate_from_library : 1.0 ;
  slew_lower_threshold_pct_fall : 10.0 ;
```

```
slew_lower_threshold_pct_rise : 10.0 ;
slew_upper_threshold_pct_fall : 90.0 ;
slew_upper_threshold_pct_rise : 90.0 ;
nom_process : 1.0 ;
nom_temperature : 25 ;
nom_voltage : 3.3 ;
default_cell_leakage_power : 0.0000 ;
default_fanout_load : 1.0 ;
default_inout_pin_cap : 0.0500 ;
default_input_pin_cap : 0.0500 ;
default_leakage_power_density : 0.0 ;
default_output_pin_cap : 0.0000 ;


operating_conditions(BEST) {
  tree_type : best_case_tree ;
  process : 1.500 ;
  temperature : -180.000 ;
  voltage : 3.600 ;
}

operating_conditions(TYPICAL) {
  tree_type : balanced_tree ;
  process : 1.000 ;
  temperature : 25.000 ;
  voltage : 3.300 ;
}

operating_conditions(WORST) {
  tree_type : worst_case_tree ;
  process : 0.500 ;
  temperature : 125.000 ;
  voltage : 3.000 ;
}

default_operating_conditions : TYPICAL ;

type(bus6) {
  base_type : array;
  data_type : bit;
  bit_width : 6;
  bit_from  : 0;
  bit_to    : 5;
  downto    : false;
}

type(bus7) {
  base_type : array;
  data_type : bit;
  bit_width : 7;
  bit_from  : 0;
  bit_to    : 6;
  downto    : false;
}

type(bus8) {
  base_type : array;
  data_type : bit;
  bit_width : 8;
  bit_from  : 0;
  bit_to    : 7;
  downto    : false;
}

type(bus10) {
  base_type : array;
  data_type : bit;
  bit_width : 10;
  bit_from  : 0;
  bit_to    : 9;
  downto    : false;
}

type(bus12) {
  base_type : array;
  data_type : bit;
  bit_width : 12;
  bit_from  : 0;
  bit_to    : 11;
  downto    : false;
}
```

```
  type(bus13) {
    base_type : array;
    data_type : bit;
    bit_width : 13;
    bit_from  : 0;
    bit_to    : 12;
    downto    : false;
  }

  lu_table_template(SPORT_DLY) {
    variable_1 : input_net_transition ;
    variable_2 : total_output_net_capacitance ;
    index_1("1.0, 2.0, 3.0");
    index_2("1.0, 2.0, 3.0");
  }

  lu_table_template(SPORT_TCHK) {
    variable_1 : related_pin_transition ;
    variable_2 : constrained_pin_transition ;
    index_1("1.0, 1.5");
    index_2("1.0, 1.5");
  }




/* Modifications on the original cryo5a_rdc.lib file */
/* References used: template.lib (from BAE) and example */
/* 6-15 in the liberty file format user guide */
/* Retaining cell rise/fall is minimum observed - 5ns */
/* Setup time is copied from rfd_dffr_scan */
/* hold time is rfd_dffr_scan + 2ns */


    cell(UARK_ROM_8X1024) {
        area : 1575 ;
        memory() {
           type : rom ;
           address_width : 10 ;
           word_width : 8 ;
        }
        pin(RST_L) {
            direction : input ;
            capacitance : 0.030556 ;
        }
        pin(CLK) {
            direction : input ;
            clock : true ;
            capacitance : 0.050556 ;
        }
        pin(MEB) {
            direction : input ;
            clock : true;
            capacitance : 0.030556 ;
            timing() {
               related_pin : "CLK" ;
               timing_type : setup_rising ;
               rise_constraint(SPORT_TCHK) {
                 index_1("1.0, 1.5");
                 index_2("1.0, 1.5");
                 values("3.21, 3.21",\
                        "3.21, 3.21");
               }
               fall_constraint(SPORT_TCHK) {
                 index_1("1.0, 1.5");
                 index_2("1.0, 1.5");
                 values("3.22, 3.22",\
                        "3.22, 3.22");
               }
            }
            timing() {
               related_pin : "CLK" ;
               timing_type : hold_rising ;
               rise_constraint(SPORT_TCHK) {
                 index_1("1.0, 1.5");
                 index_2("1.0, 1.5");
                 values("3.41, 3.41",\
                        "3.41, 3.41");
               }
               fall_constraint(SPORT_TCHK) {
```

```
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.42, 3.42",\
                "3.42, 3.42");
       }
   }
}
pin(REB) {
    direction : input ;
    clock : true;
    capacitance : 0.030556 ;
    timing() {
       related_pin : "CLK" ;
       timing_type : setup_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.21, 3.21",\
                "3.21, 3.21");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.22, 3.22",\
                "3.22, 3.22");
       }
    }
    timing() {
       related_pin : "CLK" ;
       timing_type : hold_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.41, 3.41",\
                "3.41, 3.41");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.42, 3.42",\
                "3.42, 3.42");
       }
   }
}
bus (A) {
    bus_type : "bus10"
    direction : input ;
    capacitance : 0.030556 ;
    timing() {
       related_pin : "CLK" ;
       timing_type : setup_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.21, 3.21",\
                "3.21, 3.21");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.22, 3.22",\
                "3.22, 3.22");
       }
    }
    timing() {
       related_pin : "CLK" ;
       timing_type : hold_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.41, 3.41",\
                "3.41, 3.41");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.42, 3.42",\
                "3.42, 3.42");
       }
   }
}
```

```
    bus (Q) {
        bus_type : "bus8" ;
        direction : output ;
        memory_read() {
           address : A ;
           }
        timing () {
           related_pin : "CLK" ;
           timing_sense : non_unate;
           timing_type : rising_edge ;
           cell_rise(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("3.00, 3.00, 4.00",\
                    "3.00, 4.00, 4.00",\
                    "4.00, 4.00, 5.00");
             }
           rise_transition(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("1.21, 1.96, 3.56",\
                    "1.26, 1.97, 3.58",\
                    "1.27, 1.97, 3.59");
             }
           cell_fall(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("11.00, 12.00, 12.00",\
                    "12.00, 12.00, 12.00",\
                    "12.00, 12.00, 13.00");
             }
           fall_transition(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("0.95, 1.24, 1.75",\
                    "0.95, 1.24, 1.75",\
                    "0.95, 1.24, 1.76");
             }
           retaining_rise(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("3.00, 3.00, 4.00",\
                    "3.00, 4.00, 4.00",\
                    "4.00, 4.00, 5.00");
             }
           retain_rise_slew(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("1.21, 1.96, 3.56",\
                    "1.26, 1.97, 3.58",\
                    "1.27, 1.97, 3.59");
             }
           retaining_fall(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("3.00, 3.00, 4.00",\
                    "3.00, 4.00, 4.00",\
                    "4.00, 4.00, 5.00");
             }
           retain_fall_slew(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("0.95, 1.24, 1.75",\
                    "0.95, 1.24, 1.75",\
                    "0.95, 1.24, 1.76");
             }
           }
    }
}

cell(UARK_SRAM_12X128) {
    area : 439 ;
    memory() {
       type : ram ;
       address_width : 7 ;
       word_width : 12 ;
    }
    pin(RST_L) {
        direction : input ;
        capacitance : 0.030556 ;
    }
```

```
pin(CLK) {
    direction : input ;
    clock : true ;
    capacitance : 0.050556 ;
}
pin(MEB) {
    direction : input ;
    clock : true;
    capacitance : 0.030556 ;
    timing() {
       related_pin : "CLK" ;
       timing_type : setup_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.21, 3.21",\
                "3.21, 3.21");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.22, 3.22",\
                "3.22, 3.22");
       }
    }
    timing() {
       related_pin : "CLK" ;
       timing_type : hold_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.41, 3.41",\
                "3.41, 3.41");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.42, 3.42",\
                "3.42, 3.42");
       }
   }
}
pin(REB) {
    direction : input ;
    clock : true;
    capacitance : 0.030556 ;
    timing() {
       related_pin : "CLK" ;
       timing_type : setup_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.21, 3.21",\
                "3.21, 3.21");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.22, 3.22",\
                "3.22, 3.22");
       }
    }
    timing() {
       related_pin : "CLK" ;
       timing_type : hold_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.41, 3.41",\
                "3.41, 3.41");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.42, 3.42",\
                "3.42, 3.42");
       }
   }
}
pin(WEB) {
    direction : input ;
    clock : true;
    capacitance : 0.030556 ;
    timing() {
       related_pin : "CLK" ;
       timing_type : setup_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.21, 3.21",\
                "3.21, 3.21");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.22, 3.22",\
                "3.22, 3.22");
       }
    }
    timing() {
       related_pin : "CLK" ;
       timing_type : hold_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.41, 3.41",\
                "3.41, 3.41");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.42, 3.42",\
                "3.42, 3.42");
       }
   }
}
bus (A) {
    bus_type : "bus7"
    direction : input ;
    capacitance : 0.030556 ;
    timing() {
       related_pin : "CLK" ;
       timing_type : setup_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.21, 3.21",\
                "3.21, 3.21");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.22, 3.22",\
                "3.22, 3.22");
       }
    }
    timing() {
       related_pin : "CLK" ;
       timing_type : hold_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.41, 3.41",\
                "3.41, 3.41");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.42, 3.42",\
                "3.42, 3.42");
       }
   }
}
bus (D) {
    bus_type : "bus12"
    direction : input ;
    memory_write() {
       address : A ;
       clocked_on : "ENABLE_WRITE" ;
    }
    capacitance : 0.030556 ;
    timing() {
       related_pin : "CLK" ;
       timing_type : setup_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.21, 3.21",\
                "3.21, 3.21");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.22, 3.22",\
                "3.22, 3.22");
       }
    }
    timing() {
       related_pin : "CLK" ;
       timing_type : hold_rising ;
       rise_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.41, 3.41",\
                "3.41, 3.41");
       }
       fall_constraint(SPORT_TCHK) {
         index_1("1.0, 1.5");
         index_2("1.0, 1.5");
         values("3.42, 3.42",\
                "3.42, 3.42");
       }
   }
}
bus (Q) {
    bus_type : "bus12" ;
    direction : output ;
    memory_read() {
       address : A ;
       }
    timing () {
       related_pin : "CLK" ;
       timing_sense : non_unate;
       timing_type : rising_edge ;
       cell_rise(SPORT_DLY) {
         index_1("0.50, 1.00, 1.50");
         index_2("0.05, 0.20, 0.50");
         values("3.00, 3.00, 4.00",\
                "3.00, 3.00, 4.00",\
                "3.00, 4.00, 4.00");
         }
       rise_transition(SPORT_DLY) {
         index_1("0.50, 1.00, 1.50");
         index_2("0.05, 0.20, 0.50");
         values("1.46, 2.22, 3.18",\
                "1.46, 2.22, 3.18",\
                "1.46, 2.22, 3.18");
         }
       cell_fall(SPORT_DLY) {
         index_1("0.50, 1.00, 1.50");
         index_2("0.05, 0.20, 0.50");
         values("10.00, 10.00, 11.00",\
                "10.00, 11.00, 11.00",\
                "11.00, 11.00, 11.00");
         }
       fall_transition(SPORT_DLY) {
         index_1("0.50, 1.00, 1.50");
         index_2("0.05, 0.20, 0.50");
         values("0.98, 1.25, 1.74",\
                "0.99, 1.26, 1.75",\
                "0.99, 1.26, 1.76");
         }
       retaining_rise(SPORT_DLY) {
         index_1("0.50, 1.00, 1.50");
         index_2("0.05, 0.20, 0.50");
         values("3.00, 3.00, 4.00",\
                "3.00, 3.00, 4.00",\
                "3.00, 4.00, 4.00");
         }
       retain_rise_slew(SPORT_DLY) {
         index_1("0.50, 1.00, 1.50");
         index_2("0.05, 0.20, 0.50");
         values("1.46, 2.22, 3.18",\
                   "1.46, 2.22, 3.18",\
                   "1.46, 2.22, 3.18");
            }
          retaining_fall(SPORT_DLY) {
            index_1("0.50, 1.00, 1.50");
            index_2("0.05, 0.20, 0.50");
            values("3.00, 3.00, 4.00",\
                   "3.00, 3.00, 4.00",\
                   "3.00, 4.00, 4.00");
            }
          retain_fall_slew(SPORT_DLY) {
            index_1("0.50, 1.00, 1.50");
            index_2("0.05, 0.20, 0.50");
            values("0.98, 1.25, 1.74",\
                   "0.99, 1.26, 1.75",\
                   "0.99, 1.26, 1.76");
            }
          }
   }
   pin(ENABLE_WRITE) {
       direction : internal;
       clock : true
       state_function : "!WEB * !MEB * CLK";
   }
}
cell(UARK_SRAM_12X64) {
   area : 300 ;
   memory() {
      type : ram ;
      address_width : 6 ;
      word_width : 12 ;
   }
   pin(RST_L) {
       direction : input ;
       capacitance : 0.030556 ;
   }
   pin(CLK) {
       direction : input ;
       clock : true ;
       capacitance : 0.050556 ;
   }
   pin(MEB) {
       direction : input ;
       clock : true;
       capacitance : 0.030556 ;
       timing() {
          related_pin : "CLK" ;
          timing_type : setup_rising ;
          rise_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.21, 3.21",\
                   "3.21, 3.21");
          }
          fall_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.22, 3.22",\
                   "3.22, 3.22");
          }
       }
       timing() {
          related_pin : "CLK" ;
          timing_type : hold_rising ;
          rise_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.41, 3.41",\
                   "3.41, 3.41");
          }
          fall_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.42, 3.42",\
                   "3.42, 3.42");
          }
      }
   }
   pin(REB) {
       direction : input ;
       clock : true;
       capacitance : 0.030556 ;
       timing() {
          related_pin : "CLK" ;
          timing_type : setup_rising ;
          rise_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.21, 3.21",\
                   "3.21, 3.21");
          }
          fall_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.22, 3.22",\
                   "3.22, 3.22");
          }
       }
       timing() {
          related_pin : "CLK" ;
          timing_type : hold_rising ;
          rise_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.41, 3.41",\
                   "3.41, 3.41");
          }
          fall_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.42, 3.42",\
                   "3.42, 3.42");
          }
      }
   }
   pin(WEB) {
       direction : input ;
       clock : true;
       capacitance : 0.030556 ;
       timing() {
          related_pin : "CLK" ;
          timing_type : setup_rising ;
          rise_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.21, 3.21",\
                   "3.21, 3.21");
          }
          fall_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.22, 3.22",\
                   "3.22, 3.22");
          }
       }
       timing() {
          related_pin : "CLK" ;
          timing_type : hold_rising ;
          rise_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.41, 3.41",\
                   "3.41, 3.41");
          }
          fall_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.42, 3.42",\
                   "3.42, 3.42");
          }
      }
   }
   bus (A) {
       bus_type : "bus6"
       direction : input ;
       capacitance : 0.030556 ;
       timing() {
          related_pin : "CLK" ;
          timing_type : setup_rising ;
          rise_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.21, 3.21",\
                   "3.21, 3.21");
          }
          fall_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.22, 3.22",\
                   "3.22, 3.22");
          }
       }
       timing() {
          related_pin : "CLK" ;
          timing_type : hold_rising ;
          rise_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.41, 3.41",\
                   "3.41, 3.41");
          }
          fall_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.42, 3.42",\
                   "3.42, 3.42");
          }
      }
   }
   bus (D) {
       bus_type : "bus12"
       direction : input ;
       memory_write() {
          address : A ;
          clocked_on : "ENABLE_WRITE" ;
       }
       capacitance : 0.030556 ;
       timing() {
          related_pin : "CLK" ;
          timing_type : setup_rising ;
          rise_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.21, 3.21",\
                   "3.21, 3.21");
          }
          fall_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.22, 3.22",\
                   "3.22, 3.22");
          }
       }
       timing() {
          related_pin : "CLK" ;
          timing_type : hold_rising ;
          rise_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.41, 3.41",\
                   "3.41, 3.41");
          }
          fall_constraint(SPORT_TCHK) {
            index_1("1.0, 1.5");
            index_2("1.0, 1.5");
            values("3.42, 3.42",\
                   "3.42, 3.42");
          }
      }
   }
   bus (Q) {
       bus_type : "bus12" ;
       direction : output ;
       memory_read() {
          address : A ;
          }
       timing () {
          related_pin : "CLK" ;
          timing_sense : non_unate;
          timing_type : rising_edge ;
          cell_rise(SPORT_DLY) {
            index_1("0.50, 1.00, 1.50");
            index_2("0.05, 0.20, 0.50");
```

```
              values("3.00, 3.00, 4.00",\
                     "3.00, 3.00, 4.00",\
                     "3.00, 4.00, 5.00");
              }
            rise_transition(SPORT_DLY) {
              index_1("0.50, 1.00, 1.50");
              index_2("0.05, 0.20, 0.50");
              values("1.40, 2.15, 3.21",\
                     "1.42, 2.18, 3.21",\
                     "1.42, 2.18, 3.22");
              }
            cell_fall(SPORT_DLY) {
              index_1("0.50, 1.00, 1.50");
              index_2("0.05, 0.20, 0.50");
              values("8.00, 9.00, 9.00",\
                     "9.00, 9.00, 9.00",\
                     "9.00, 9.00, 10.00");
              }
            fall_transition(SPORT_DLY) {
              index_1("0.50, 1.00, 1.50");
              index_2("0.05, 0.20, 0.50");
              values("0.89, 1.17, 1.68",\
                     "0.89, 1.17, 1.68",\
                     "0.90, 1.17, 1.68");
              }
            retaining_rise(SPORT_DLY) {
              index_1("0.50, 1.00, 1.50");
              index_2("0.05, 0.20, 0.50");
              values("3.00, 3.00, 4.00",\
                     "3.00, 3.00, 4.00",\
                     "3.00, 4.00, 5.00");
              }
            retain_rise_slew(SPORT_DLY) {
              index_1("0.50, 1.00, 1.50");
              index_2("0.05, 0.20, 0.50");
              values("1.40, 2.15, 3.21",\
                     "1.42, 2.18, 3.21",\
                     "1.42, 2.18, 3.22");
              }
            retaining_fall(SPORT_DLY) {
              index_1("0.50, 1.00, 1.50");
              index_2("0.05, 0.20, 0.50");
              values("3.00, 3.00, 4.00",\
                     "3.00, 3.00, 4.00",\
                     "3.00, 4.00, 5.00");
              }
            retain_fall_slew(SPORT_DLY) {
              index_1("0.50, 1.00, 1.50");
              index_2("0.05, 0.20, 0.50");
              values("0.89, 1.17, 1.68",\
                     "0.89, 1.17, 1.68",\
                     "0.90, 1.17, 1.68");
              }
            }
    }
    pin(ENABLE_WRITE) {
        direction : internal;
        clock : true
        state_function : "!WEB * !MEB * CLK";
    }
}
cell(UARK_SRAM_13X1024) {
    area : 2434 ;
    memory() {
       type : ram ;
       address_width : 10 ;
       word_width : 13 ;
    }
    pin(RST_L) {
        direction : input ;
        capacitance : 0.030556 ;
    }
    pin(CLK) {
        direction : input ;
        clock : true ;
        capacitance : 0.050556 ;
    }
    pin(MEB) {
        direction : input ;
        clock : true;
        capacitance : 0.030556 ;
        timing() {
           related_pin : "CLK" ;
           timing_type : setup_rising ;
           rise_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.21, 3.21",\
                    "3.21, 3.21");
           }
           fall_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.22, 3.22",\
                    "3.22, 3.22");
           }
        }
        timing() {
           related_pin : "CLK" ;
           timing_type : hold_rising ;
           rise_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.41, 3.41",\
                    "3.41, 3.41");
           }
           fall_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.42, 3.42",\
                    "3.42, 3.42");
           }
       }
    }
    pin(REB) {
        direction : input ;
        clock : true;
        capacitance : 0.030556 ;
        timing() {
           related_pin : "CLK" ;
           timing_type : setup_rising ;
           rise_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.21, 3.21",\
                    "3.21, 3.21");
           }
           fall_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.22, 3.22",\
                    "3.22, 3.22");
           }
        }
        timing() {
           related_pin : "CLK" ;
           timing_type : hold_rising ;
           rise_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.41, 3.41",\
                    "3.41, 3.41");
           }
           fall_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.42, 3.42",\
                    "3.42, 3.42");
           }
       }
    }
    pin(WEB) {
        direction : input ;
        clock : true;
        capacitance : 0.030556 ;
        timing() {
           related_pin : "CLK" ;
           timing_type : setup_rising ;
           rise_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.21, 3.21",\
                    "3.21, 3.21");
           }
           fall_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.22, 3.22",\
                    "3.22, 3.22");
           }
        }
        timing() {
           related_pin : "CLK" ;
           timing_type : hold_rising ;
           rise_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.41, 3.41",\
                    "3.41, 3.41");
           }
           fall_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.42, 3.42",\
                    "3.42, 3.42");
           }
       }
    }
    bus (A) {
        bus_type : "bus10"
        direction : input ;
        capacitance : 0.030556 ;
        timing() {
           related_pin : "CLK" ;
           timing_type : setup_rising ;
           rise_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.21, 3.21",\
                    "3.21, 3.21");
           }
           fall_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.22, 3.22",\
                    "3.22, 3.22");
           }
        }
        timing() {
           related_pin : "CLK" ;
           timing_type : hold_rising ;
           rise_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.41, 3.41",\
                    "3.41, 3.41");
           }
           fall_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.42, 3.42",\
                    "3.42, 3.42");
           }
       }
    }
    bus (D) {
        bus_type : "bus13"
        direction : input ;
        memory_write() {
           address : A ;
           clocked_on : "ENABLE_WRITE" ;
        }
        capacitance : 0.030556 ;
        timing() {
           related_pin : "CLK" ;
           timing_type : setup_rising ;
           rise_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.21, 3.21",\
                    "3.21, 3.21");
           }
           fall_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.22, 3.22",\
                    "3.22, 3.22");
           }
        }
        timing() {
           related_pin : "CLK" ;
           timing_type : hold_rising ;
           rise_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.41, 3.41",\
                    "3.41, 3.41");
           }
           fall_constraint(SPORT_TCHK) {
             index_1("1.0, 1.5");
             index_2("1.0, 1.5");
             values("3.42, 3.42",\
                    "3.42, 3.42");
           }
       }
    }
    bus (Q) {
        bus_type : "bus13" ;
        direction : output ;
        memory_read() {
           address : A ;
           }
        timing () {
           related_pin : "CLK" ;
           timing_sense : non_unate;
           timing_type : rising_edge ;
           cell_rise(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("6.00, 6.00, 7.00",\
                    "6.00, 7.00, 7.00",\
                    "7.00, 7.00, 8.00");
             }
           rise_transition(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("1.12, 1.95, 3.44",\
                    "1.12, 1.98, 3.45",\
                    "1.12, 1.98, 3.45");
             }
           cell_fall(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("16.00, 16.00, 17.00",\
                    "16.00, 17.00, 18.00",\
                    "17.00, 17.00, 18.00");
             }
           fall_transition(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("2.00, 2.60, 2.97",\
                    "2.00, 2.60, 2.97",\
                    "2.09, 2.65, 2.96");
             }
           retaining_rise(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("6.00, 6.00, 7.00",\
                    "6.00, 7.00, 7.00",\
                    "7.00, 7.00, 8.00");
             }
           retain_rise_slew(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("1.12, 1.95, 3.44",\
                    "1.12, 1.98, 3.45",\
                    "1.12, 1.98, 3.45");
             }
           retaining_fall(SPORT_DLY) {
             index_1("0.50, 1.00, 1.50");
             index_2("0.05, 0.20, 0.50");
             values("6.00, 6.00, 7.00",\
                    "6.00, 7.00, 7.00",\
                    "7.00, 7.00, 8.00");
```

```
                }
              retain_fall_slew(SPORT_DLY) {
                index_1("0.50, 1.00, 1.50");
                index_2("0.05, 0.20, 0.50");
                values("2.00, 2.60, 2.97",\
                       "2.00, 2.60, 2.97",\
                       "2.09, 2.65, 2.96");
                }
              }
        }
        pin(WR_SRC) {
            direction : output ;
            timing () {
              related_pin : "CLK" ;
              timing_sense : non_unate;
              timing_type : rising_edge ;
              cell_rise(SPORT_DLY) {
                index_1("0.50, 1.00, 1.50");
                index_2("0.05, 0.20, 0.50");
                values("6.00, 6.00, 7.00",\
                       "6.00, 7.00, 7.00",\
                       "7.00, 7.00, 8.00");
                }
              rise_transition(SPORT_DLY) {
                index_1("0.50, 1.00, 1.50");
                index_2("0.05, 0.20, 0.50");
                values("1.12, 1.95, 3.44",\
                       "1.12, 1.95, 3.44",\
                       "1.12, 1.95, 3.44");
                }
              cell_fall(SPORT_DLY) {
                index_1("0.50, 1.00, 1.50");
                index_2("0.05, 0.20, 0.50");
                values("6.00, 6.00, 7.00",\
                       "6.00, 7.00, 7.00",\
                       "7.00, 7.00, 8.00");
                }
              fall_transition(SPORT_DLY) {
                index_1("0.50, 1.00, 1.50");
                index_2("0.05, 0.20, 0.50");
                values("1.12, 1.95, 3.44",\
                       "1.12, 1.95, 3.44",\
                       "1.12, 1.95, 3.44");
                }
              }
        }
        pin(ENABLE_WRITE) {
            direction : internal;
            clock : true
            state_function : "!WEB * !MEB * CLK";
        }
    }
}
```