A DRAM-Based Parallel Processor for Real Time Video

by

Robert N. McKenzie

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfilment of the requirements
for the degree of
Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering,
Department of Electronics,
Carleton University,
Ottawa, Ontario, Canada

December 18, 1997

Abstract

The computer industry is currently dominated by the single processor architecture. The fixed bandwidth between processor and memory is now such a limiting factor that machines are best described by their bus speed. Also, single processor machines have a fixed performance limit that cannot be surpassed.

The Computational RAM (C•RAM) project attempts to solve these problems by placing thousands of Processing Elements inside an existing computer part with lots of internal bandwidth: the main memory. A prototype C•RAM was designed at IBM Microelectronics in Burlington Vermont by adding 1024 SIMD Processing Elements (PEs) to a platform 16 Mbit Dynamic RAM. They were installed on a pitch of 8 sense amplifiers outside the predesigned array blocks using static CMOS logic and a 0.8 µm process. This change increased the chip's area by 14% and simulations showed a power consumption increase of 11%.

C•RAM is a JEDEC-compatible Dynamic RAM chip with an additional parallel processing mode. In addition to regular memory I/O operations, it can be used to perform vector computations, database operations, and real-time image processing. A single chip can perform 462 million 32-bit additions per second, and sort M data elements with an O(M) algorithm. Discrete Cosine Transforms and motion estimation, the two heaviest components of MPEG-2 video compression, can be performed for more than 30 frames per second. The DRAM's fast page mode is used to increase memory bandwidth to the PEs, and makes computations three times faster.

Acknowledgements

I greatly appreciate all the help given by Howard Kalter and John Barth of IBM Microelectronics in Burlington Vermont. Their time and expertise have been invaluable to this project.

Thanks to Duncan Elliott, Peter Nyasulu, Thinh Le, and Phil Lauzon for all their interesting and helpful discussions.

Thanks also to Alana Wirta for administrative help, to James Cherry for helping me with the typesetting software, and to Nick Bilaniuk for a useful editing iteration.

Thanks most of all to my supervisor, Martin Snelgrove, for his patience, motivation, and refreshing ideas. I enjoyed our work evenings at Café WIM, and all the engineering, artistic, and architectural diversions he provided during my time at Carleton.

Table of Contents

List of Figures

CHAPTER 1 Introduction

"Memory is the main bottleneck of an automatic very high speed computing device."

-John von Neumann [Asp90]

The Computational RAM project is an investigation into the viability of massively parallel processing in memory. Random Access Memory (RAM) chips are storage components which abound in modern computers, and have a very large and underexploited internal bandwidth. Placing processors inside the chips to intercept some or all of this huge data flow could prove very useful.

This thesis deals with the possibility of installing several low-cost processors inside a Dynamic RAM chip. DRAM has a low per-bit storage cost, and is the most popular of all types currently available. At the time of writing, most computers have lots of DRAM on board: 32 MB is a typical amount which comprises sixteen 16 Mb chips.

Compared to the amount of work required to make a processor-in-memory chip from scratch, changing a platform DRAM into a C•RAM requires much less effort. Also, the risk factor is decreased by using a proven memory design. As partially illustrated in Figure 1.1, the work required to change a DRAM into a C•RAM is the design of individual Processing Elements (PEs), relocation and reworking of column decoders, and installation

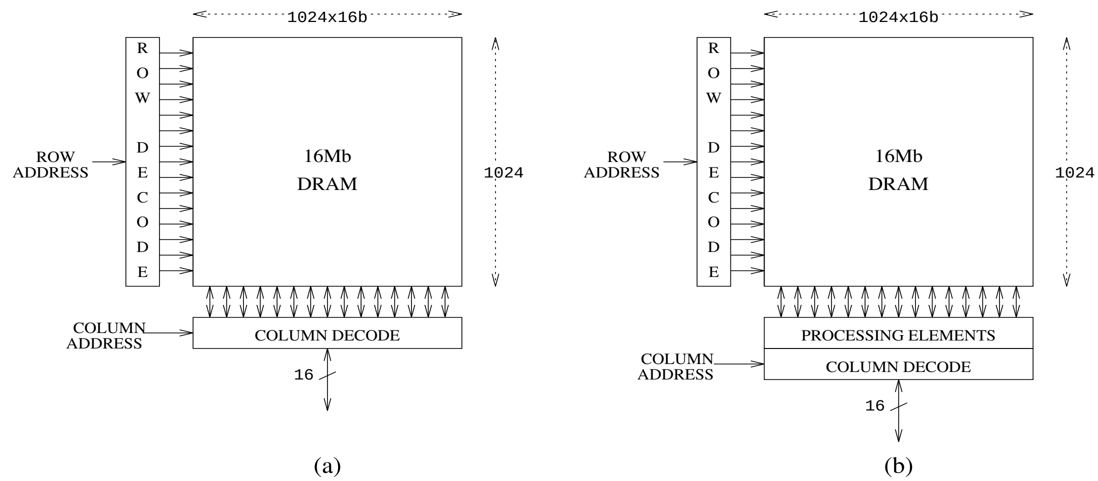


FIGURE 1.1 Conceptual change from (a) DRAM to (b) C•RAM

of associated timing circuitry. The finished product is a Dynamic RAM and massively parallel processor with 1024 PEs.

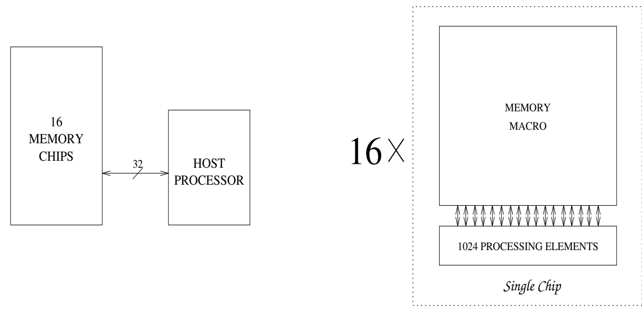
Parallel processing was first proposed in the 1940s by Hungarian mathematician and scientist John von Neumann. He was working on the model of a computing machine with working storage for executable code, and suggested that it include several Central Processing Units (CPUs), a program control unit, and a main memory. Since the logic circuitry was made with large vacuum tubes and relays, he conceded that a multiple-processor machine would be too difficult to build, and settled with a single processor architecture. Computational speeds of his group's mechanical computers were so dazzling in comparison to what was available at the time (mostly pen and paper arithmetic) that his colleague noted: "the electronic technique is potentially so fast, one can abandon all parallelism and still obtain enough speed" [Gol72].

Although advancements in electronic technology since then have made single processor machines much faster, processors have become too fast for their memories. To make matters worse, there is a widening gap between processor and memory technologies: at the beginning of this decade, processor performance improved 50% each year, and memory performance only improved 7% [Hen90]. This memory bandwidth problem is called “the von Neumann bottleneck” [Sto87] and has become so severe that the processor-to-memory transfer rate now determines the price and performance of a machine. A 200MHz Pentium machine has a 200MHz data link to its L2 cache, and costs twice as much as a 120MHz Pentium.

Realizing that main memory is the bottleneck in a computer, we want to find a location that obtains its largest possible data rate. The wide datapath inside memory chips at the sense amplifier level is an attractive location for processing not only because so many bits are simultaneously available: processing the data at a location physically close to the memory cells ensures us that we are obtaining close to the fastest possible memory bandwidth.

The chart in Figure 1.2 compares the processor to memory bandwidth differences between a Pentium and C•RAM. The Pentium processor operates at 200MHz while retrieving data from its cache, and can at best receive 0.8 GBits per second. Processing Elements outside a memory array can receive data bits every 50ns, worst case. Processing in memory yields a much higher memory bandwidth, and saves energy that would have otherwise been spent relaying bits across printed circuit boards and backplanes.

One challenge associated with designing a component that radically changes how computers operate is figuring out its introduction into a market that resists change. No-one can expect a product which gives a modest performance boost for a huge upgrade cost to be a widespread success. Similarly, a product which radically alters current machines might not succeed since the costs of retraining computer users and upgrading would outweigh the benefits.



200 MHz PENTIUM **16 C•RAM chips**

32b	DATAPATH WIDTH	16 x 1024b
5 ns	CYCLE TIME	50 ns
0.8 GBytes/s	**DATA BANDWIDTH**	**41 GBytes/s**
10cm	TRANSMISSION DISTANCE	5mm
20pF	CAPACITANCE PER DATA BIT	1pF
3.3V	V_{DD}	3.3V
110pJ	**ENERGY CONSUMPTION PER DATA BIT TRANSFER**	**5.5pJ**

Metal capacitance: 0.2pF/mm
Energy per cycle: $E=0.5CV_{DD}^2$

FIGURE 1.2 **200MHz pentium vs C•RAM**

The C•RAM presented in this thesis started as a IBM 16 Megabit Dynamic RAM, to which we added a 1024 Processing Element array. It should be attractive to system designers, since it will have the same packaging and pinout as the memory chip, and will function exactly the same as a DRAM while not in parallel processing mode. Consumers should like it too. Instead of buying regular memory chips, they would buy the slightly more expensive processor-enhanced memory and receive a large performance increase in exchange.

Parallel processors like C•RAM are intended for a specific type of application. Since the Processing Elements are driven by a single controller and work in unison, the tasks they perform must be identical. It follows that jobs suited for C•RAM execution must contain many identical and independent tasks. Certain applications run so efficiently on parallel processors due to the independence of their tasks that they are called “embarrassingly parallel” [Fox94].

Applications such as real-time image processing, database manipulating, graphics rendering, fluid flow simulation, portfolio risk analysis, and weather prediction can all benefit from C•RAM’s parallel processing capabilities. While not all embarrassingly parallel, their computational structures are uniformly divisible and well suited to parallel processing. We are most interested in real-time image processing due to its challenging frame rate requirements and potentially large market for multimedia, games, and television broadcasts.

Most researchers know the value of vector processing to these applications, and a significant amount of research and design work has already been done. We introduce in Chapter 2 some of the work preceding C•RAM by describing the architectures of vector processors designed over the past two decades, and looking at the basic design and operation of memory chips.

Since research on processing in memory has been done in theses by Duncan Elliott [Ell97] and Christian Cojocaru [Coj95] and their small scale prototypes have been fabricated, the work presented in this document attempts to make the architecture industry acceptable. A C•RAM has been designed by integrating custom pitch-matched Processing Elements into the IBM DRAM chip, reworking the displaced circuitry, and installing new timing circuitry. The new chip functions exactly like its memory chip predecessor while not in parallel processing mode, so once fabricated it can be immediately installed into a desktop system. In addition, DRAM’s fast page mode feature can be used to “cache” data on the bitlines during computation and increase performance. Chapter 3 describes the existing DRAM architecture, and outlines the C•RAM design steps.

This chip was designed as a prototype, and some precautionary steps were taken in this first iteration to maximize the probability that the chip will work. Dynamic logic circuitry, although area efficient in its implementation, has risk of error from charge sharing and timing problems [Sne95]. We therefore designed the processing elements with area-inefficient but lower risk static logic circuitry, realizing that the layout could be further optimized in the next iteration. Using a proven memory design reduces both risk and design time.

In addition to the design of a C•RAM, we also look at how it performs certain applications. In Chapter 4, we start with the formation of a small library of arithmetic microroutines, with which we generate procedures and performance estimates for larger applications. We look at motion estimation and Discrete Cosine Transforms (the heaviest components of MPEG-2 compression), as well as parallelized data searches and sorts.

Improvements to the C•RAM design which could boost performance are discussed in Chapter 5. Cojocaru introduced a bit-parallel Processing Element which uses full-adder circuitry to perform multi-bit additions across several PEs. A bit-parallel extension to the PEs is proposed as an extension to this C•RAM architecture, and the change in performance is discussed. Also, a synchronous front-end DRAM interface for the C•RAM, which makes more time-efficient memory accesses, is examined.

CHAPTER 2 Parallel Processing and Memory

"In essence, we are trying to determine whether it is better to plow a field with one ox, four horses, or 1024 chickens."

Harold S. Stone [Sto87]

The focus of this thesis is directed towards parallel processing inside conventional memory chips, and this chapter takes a look at the "prior art" of both parallel processing and memory technologies. After a discussion of what is motivating the push towards parallel processing, we are given an introduction to some of the high-performance machines built over the past twenty years. A general introduction to memory technology is given, showing the basic architecture and operation of standard RAM chips.

2.1 Parallel Processing and SIMD Computing

The computer market is currently dominated by the single processor machine: at the top of the price curve are 200+ MHz Pentiums with 32Mb RAM. For the average user in 1997, this type of computer is usually good enough since word processors and web browsers are limited by the speed at which a person types, or the rate at which a modem can provide data. However, applications which require significantly more computing power begin to expose the architecture's weaknesses.

Memory bandwidth limitations are a problem, since the processor is forced to wait for data transfers to occur. Caches are fast memories used to store working data, and have alleviated the difficulty somewhat by providing better bandwidth. At the moment, data rates between the processor and memory are so critical to performance that cache speed is the most important feature of a system.

Leading-edge technologies like games, multimedia, and image processing have very demanding performance requirements, even after software designers have scaled back the quality of their products to match the performance ceiling of desktop machines. This upper bound is a fixed value caused by performance limitations of the single processor, and can only be increased with improvements to processor and memory technologies. If significantly better performance is required from desktop machines in the near future, then more processors need to be added to them.

Parallel processing is starting to emerge as a widely accessible form of computing. The Internet allows simultaneous access to many machines, making it easy to run several jobs at once on different processors. Apple computers have a dual processor Macintosh available which can run graphics intense software applications faster than the single processor version. The MMX feature on a Pentium machine uses existing hardware and an extended instruction set to implement parallelism. By dividing the Pentium’s 64-bit bus into parallel groups of 32, 16 or 8 bits, the processor can perform parallel computations on the groups of data, giving slightly better multimedia and video performance [Dia97].

There are two types of parallel processing, the first and simplest architecture being Single Instruction stream, Multiple Data stream (SIMD). It has a group of identical Processing Elements (PEs) all connected to a common controller. Once per cycle, the controller broadcasts a single instruction, and each PE executes this instruction on data residing in its local memory. SIMD’s synchronous and parallel execution of tasks requires that the data structures be uniform among PEs, and that the tasks they perform be identical. PE architectures can vary in size and computational abilities between machines, but usually just contain basic computing components: an ALU, registers, and some inter-PE communication.

The second parallel processing architecture is Multiple Instruction stream, Multiple Data stream (MIMD). Like SIMD, it has a group of processors and a controller, but the processors may have access to either a local or shared memory, and their execution of tasks is not necessarily synchronous. MIMD processors are usually bigger than SIMD PEs, since in addition to a larger ALU and set of registers, they contain instruction fetch and extra control circuitry.

The two architectures' computational styles are very different: SIMD machines can be used to achieve very high performance for numerical problems, and MIMD machines generally comprise separate computers which exploit any possible parallelism in a large job [Sto87]. SIMD PEs require instruction once per cycle, but MIMD processors can independently perform millions of operations from a single instruction.

Our interest is in SIMD computing due to its design simplicity, high performance potential, and ability to perform many of the large market applications. Other researchers have realized SIMD's potential too, since several machines have been designed in the last 15 years as research projects or application specific products. Many of the earlier machines were built with large budgets for universities or companies, but none ever emerged as products for the mass market. The recent high demand for real-time image processing has accelerated research in parallel processing, and has resulted in the design of some lower cost application-specific image processors.

2.1.1 Massively Parallel Processor [Asp90]

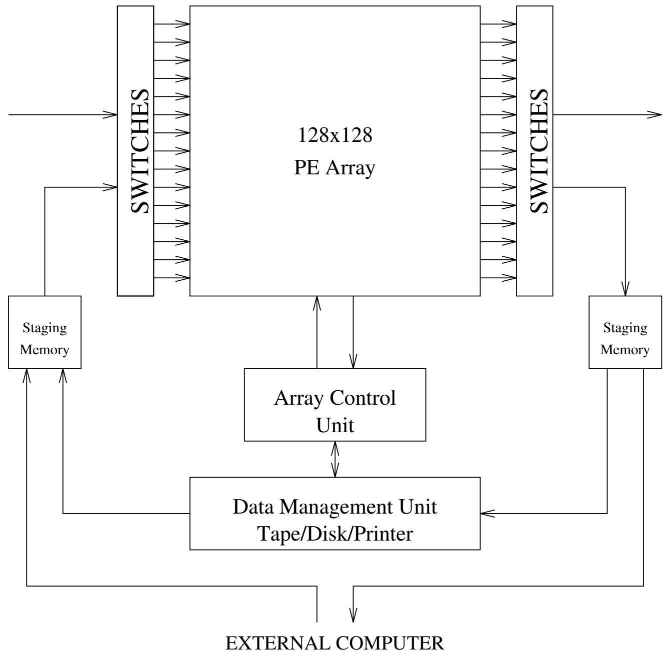


FIGURE 2.1 MPP Block Diagram

One of the earliest parallel processing machines was Goodyear's Massively Parallel Processor (or MPP), built for NASA and completed in 1980. It was designed to handle the increasing workload of processing satellite transmitted data, but had enough flexibility to handle more general parallel processing tasks.

Figure 2.1 shows a block diagram of the MPP. It contains a 128x128 array of PEs, each with 1024 bits storage. Intercommunication between PEs was 2-dimensional, with data links to the north, south, east and west neighbours. The PEs themselves were fairly substantial, each containing a full adder, 30-bit shift register, and several single-bit registers.

It could perform 6.5 billion 8-bit integer adds per second, and 430 million 32-bit floating point adds per second.

An entire system was 88 printed circuit boards housed in a large cabinet, with each board holding 24 chips, and each chip holding 8 PEs. The system had more than 128^2 PEs since a redundant column of 128 was installed. It is interesting to note that the total storage in this system is 16Mb, which is now available on a single chip.

2.1.2 VASTOR [Lou82]

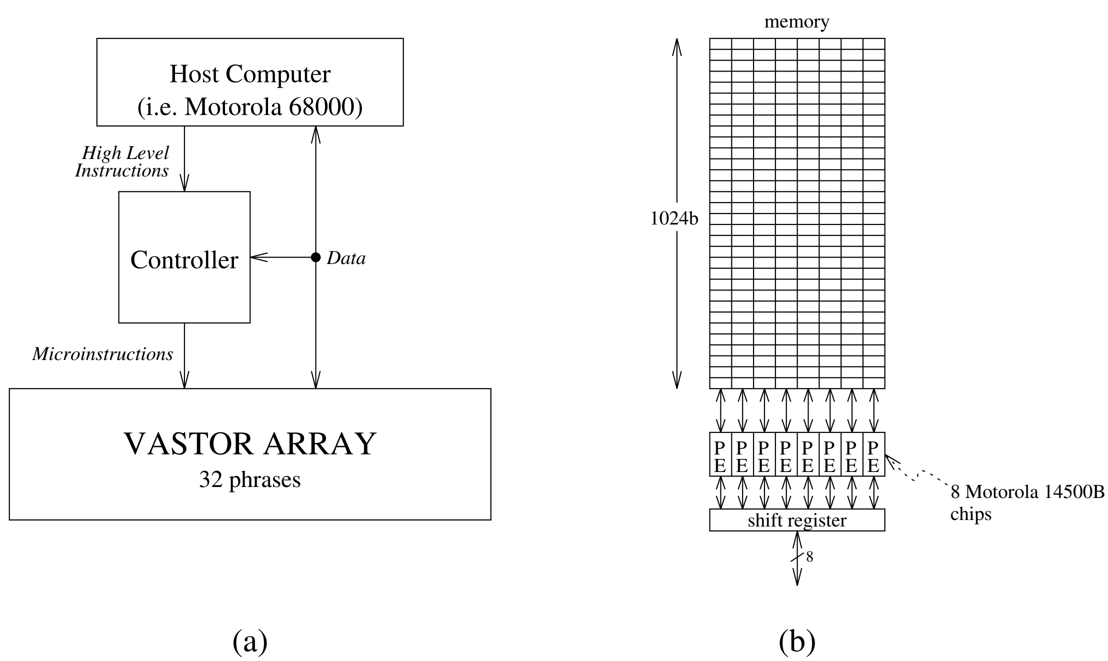


FIGURE 2.2 VASTOR architecture (a) entire system (b) single phrase

The VASTOR (Vector Associative Store TORonto) machine was created by a group at the University of Toronto in 1982. Their aim was to create a low-cost, modular parallel processing machine that could be used with APL, a programming language used for vector manipulation. The Motorola 14500B was a 16-pin, single bit processor with a small logic unit, 2 storage registers, and a write-enable register [Bay96]. It was originally designed as an industrial controller (and mainly used for washing machine relay replacement) but the

VASTOR designers used it as their Processing Element. Generic RAM chips were used for the memory block. Figure 2.2 illustrates the system's architecture. The full array consists of 32 "phrases", each of which comprises 8 PEs.

Performance testing against the PDP-11 for vector addition and other operations showed VASTOR to be 5-10 times faster. The machine's cost was estimated at $2000.

2.1.3 GAPP II [Hor90]

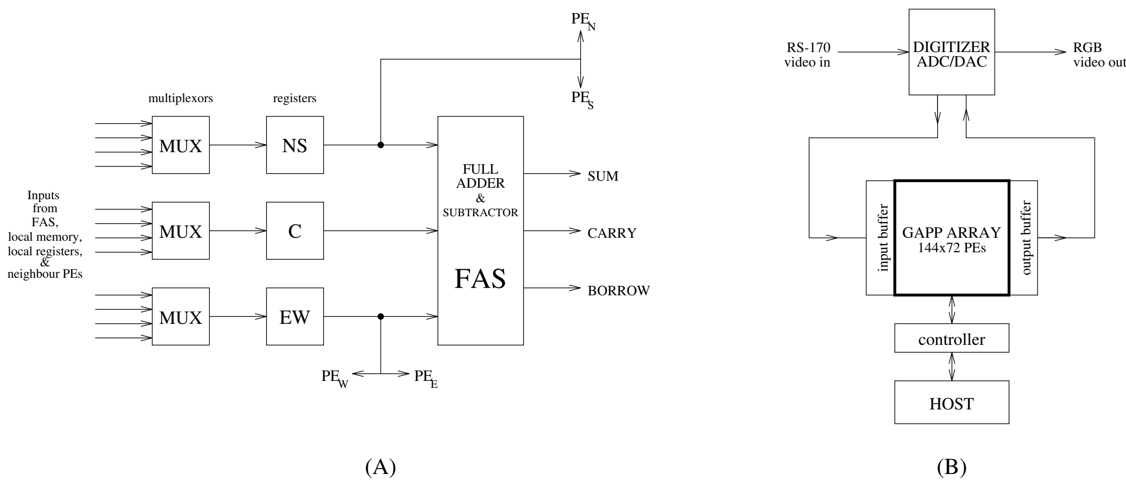


FIGURE 2.3 GAPP II (a) PE architecture (b) system

The first Geometric Arithmetic Parallel Processor (GAPP I) was developed by Martin Marietta Electronics and Missiles group of Orlando in 1983. An improved version, the GAPP II, was designed with a higher cell density and completed later the following year. As shown in Figure 2.3, the system architecture included a GAPP array of Processing Elements connected in 2-dimensions, and each PE had 128b Static RAM, and 3 registers connected to a full adder and subtractor. The system was used for real-time conversion of

Forward Looking Infrared (FLIR), video camera, or recorded analog image data into an RGB signal fed to a video monitor. The GAPP array consisted of 144 chips, each containing a 6x12 PE array.

2.1.4 Connection Machine [Tan90]

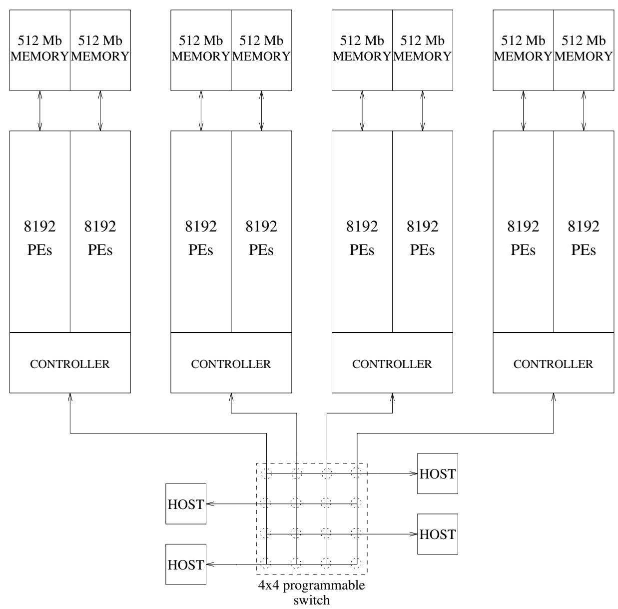


FIGURE 2.4 **Connection Machine**

Originally part of a research project at the Massachuetts Institute of Technology in the mid-1980s, the Connection Machine became a marketed product in 1986. It had 65536 PEs, each with an ALU, several flag registers, and 64 kb local memory. It had a performance rating of 2500 MIPS (millions of 32-bit integer additions per second) and 4000 Mflops (millions of 32-bit floating point additions per second). Four host stations could be attached to its 4x4 crossbar switch (Figure 2.4), thereby allowing any host to access any or all quadrants. It was a very big machine, comprising 25,000 chips, measuring over 1.5 meters high, dissipating 28kW, and weighing 1200 kilograms.

2.1.5 AIS-5000 Parallel Processor [Sch88]

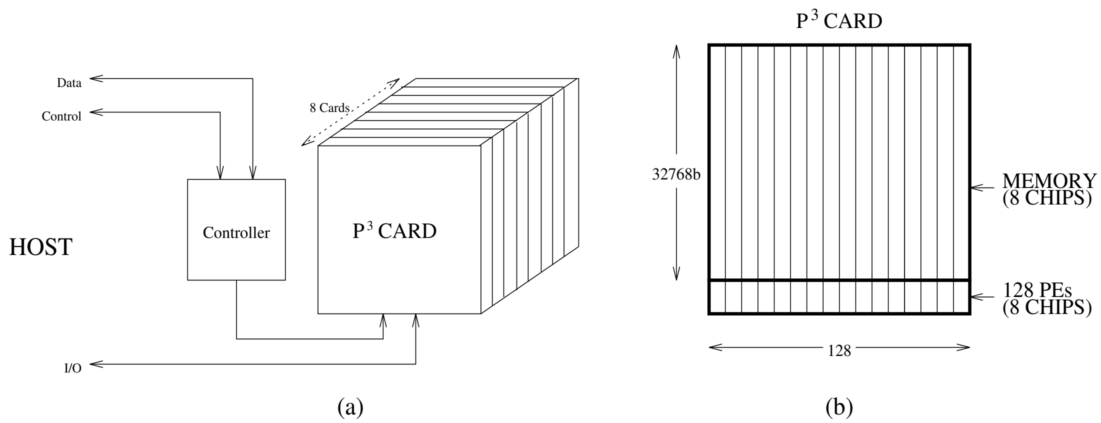


FIGURE 2.5 AIS-5000 architecture (a) parallel processor (b) P^3 card

A group at Applied Intelligent Systems Inc. in Ann Arbor, MI developed the AIS-5000 machine as a general purpose parallel processor that could be connected as a peripheral to a workstation. Figure 2.5 shows the architecture of the parallel processor and a single P^3 card. The whole system could have as many as 1024 PEs, depending upon the number of P^3 cards installed. The primary application for which it was designed was image processing, although it could also be used for general-purpose parallel processing. Each PE contained 4 storage registers, a 16-to-1 mux, and 1-dimensional interprocessor communication.

2.1.6 Elliott C•RAM [Ell97]

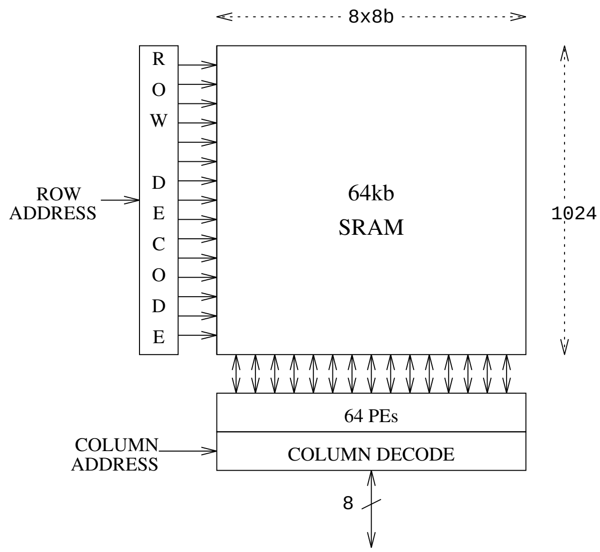


FIGURE 2.6 Elliott C•RAM architecture

Duncan Elliott began the Toronto/Carleton research on SIMD processing in memory. He showed that the wide bus and fast data rate available on a memory chip made for an excellent location to install SIMD Processing Elements. He designed and fabricated 6 prototype chips with 2 different PE structures: the first had a single storage register and a 4-to-1 mux ALU, and the larger one had 2 storage registers and an 8-to-1 multiplexor ALU.

2.1.7 Cojocaru C•RAM [Coj95]

Cojocaru continued the research of Elliott, producing three additional C•RAM designs. The first two designs had 64PEs in 64kb and 32kb Static RAM respectively. He used a Bell Northern Research modular Static RAM design as the memory platform, and designed dynamic logic Processing Elements. The basic structure of his first two PEs was similar to Elliott's second design, consisting of an 8-to-1 multiplexor ALU, and 2 single-bit storage registers.

The Processing Elements in his third design were enhanced with full adder logic so that operands could be added in both bit-serial and bit-parallel manners. Bit-parallel computation requires fewer memory accesses per addition, which lowers the PEs' performance dependency on memory bandwidth. The bit-parallel architecture is discussed further in Section 5.1.

2.1.8 MIT Pixel Parallel Image Processor [Gea97]

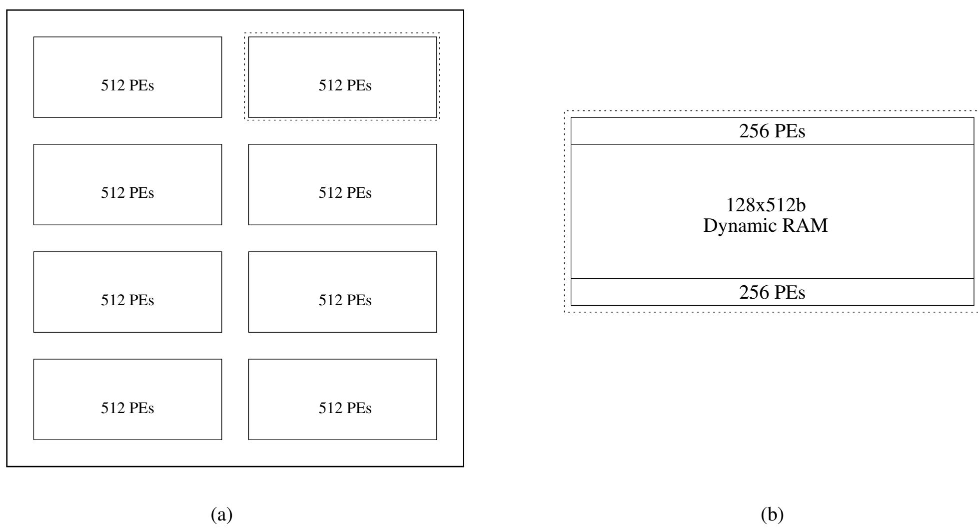


FIGURE 2.7 Pixel Parallel Image Processor architecture (a) chip (b) chip octant

The Pixel-Parallel Image Processor was designed at the Massachusetts Institute of Technology as a video acceleration card for desktop computers. Each chip has 4096 Processing Elements interconnected in 2 dimensions, yielding a 64x64 PE grid. Every PE has 128 bits local memory, 3 storage registers, a 256-function ALU, and data links to and from its four neighbours.

It was designed as a pixel-per-PE processor, and can be scaled to handle arbitrarily large images. For example, a 512x512 image could be processed by 64 chips.

The documented working model had 4 chips processing 128x128 images. A camera provided real-time image data through format converters to the PE array, and the processed data was passed through more format converters to a Sparcstation. Per-pixel optical flow calculations were performed at a rate of 32 frames per second.

2.1.9 Retrospective

The first five large-budget machines had impressive performance figures, and were successfully manufactured for their image processing, database, and mathematical applications. None of them had any hope of being used by mainstream computer users mainly due to their high cost induced by low production volume, but also because each was designed for low-level use on specific parallel processing applications. Few computer users will ever use vector processors without a significant amount of hardware and software abstraction.

Elliott and Cojocaru's C•RAM prototypes were initial design iterations of processing in memory research. Different Processing Element architectures were examined, and chips were designed using both custom and industry-designed memory arrays. A small amount of testing was done after fabrication to validate the chips' functionality and compare their performance against standard uniprocessors.

MIT's Pixel-Parallel Image Processor group directed its research away from C•RAM's by designing an application-specific parallel processor with a similar Processing Element, but with a 2-dimensional PE array architecture. Unlike the general-purpose C•RAM memory chip, this machine was designed to be a peripheral pixel-per-PE image processor with an entirely custom design.

2.2 *Random Access Memory (RAM)*

The Random Access Memory (or RAM) chip is the prime storage unit used by a working processor. It features a fast data retrieval time, much faster than that of a mechanical hard disk, and is therefore used as working storage during processor jobs. The size and speed of a computer's memory is critical to its overall performance since they both determine how much working storage can be made available, and how long the processor must idle during data transfers.

All memories are designed with the same basic structure shown in Figure 1.1a, which is an array of storage cells arranged in rows and columns. A group of cells is accessed with a row and column address, which properly routes data on or off the chip. The structure of each memory cell defines the memory's type, and the two most widely used are Static RAM (SRAM) and Dynamic RAM (DRAM).

The most frequently used SRAM cell is the 6 transistor version shown in Figure 2.8a, which is a pair of end-to-end inverters with two access N channel MOSFETS. Its static CMOS structure continuously holds the storage nodes close to the power rails, thereby retaining data indefinitely and dissipating low amounts of power during periods of inactivity. The strong logic levels maintained in each cell make it particularly noise resistant.

A DRAM cell, shown in Figure 2.8b, is considerably smaller than its static counterpart, comprising a single access transistor and storage capacitor. The smaller number of transistors per cell allows DRAM arrays to be as dense as possible, which results in a low cost per storage bit. For example, a 1Mb DRAM chip occupies approximately the same silicon area, and has almost the same cost as a 256kb SRAM chip.

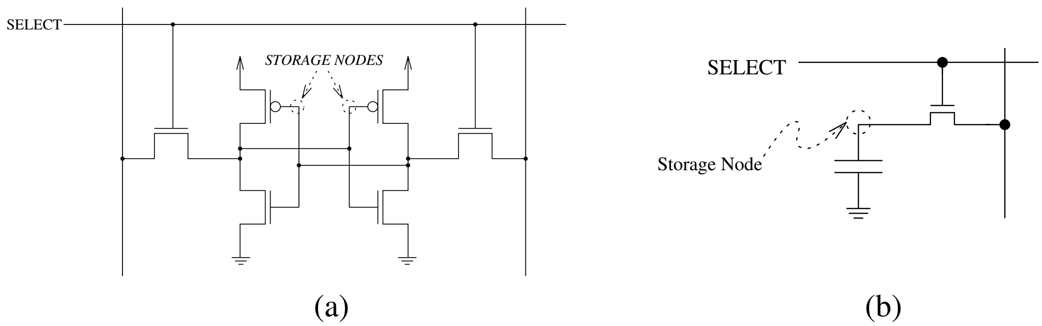


FIGURE 2.8 (a)Static RAM (SRAM) cell (b) Dynamic RAM (DRAM) cell

SRAM is used for its speed, noise immunity, and low inactivity power consumption, and is typically employed as a cache. DRAM is used as inexpensive high volume storage, and currently holds the largest portion of a huge memory market. With its cost steadily decreasing, and software applications' memory requirements steadily increasing, DRAM is being purchased in high volumes. Since our aim is to make SIMD processing appealing to the mass market, we have chosen to implement C•RAM in DRAM.

The DRAM cell has a small capacitance, typically about 100*f*F, and stores bit data as a charge on the capacitor. This small charge must be sensed and amplified, rather than read directly, due to charge sharing problems with the vertical bitline to which it is attached. A sense amplifier is used for cell content amplification, and comprises two end-to-end inverters. A typical DRAM column has one sense amplifier and 512 memory cells: 256 attached to each of the two bitlines. Figure 2.9 shows the structure of a memory column, and the timing sequence required for a cell access. The two bitlines are equalized to logic $\frac{1}{2}/\frac{1}{2}$ and the cell's access transistor is switched on by asserting the "ROW 0" wordline. Power is applied to the sense amplifiers, and the bitlines are swung to either the 0/1 or 1/0 state depending upon the bit data stored in the cell.

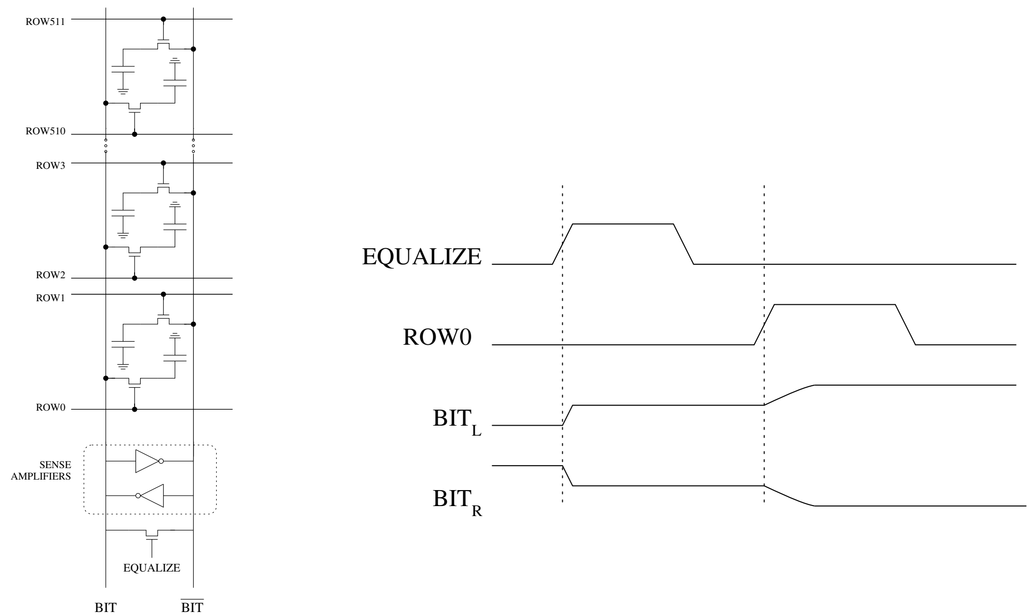


FIGURE 2.9 (a) DRAM column (b) Timing diagram: reading a logic 1 from row 0

A memory array is created by placing memory columns side-by-side, as illustrated in Figure 2.10. Wordlines now select entire rows of memory cells, and data amplified on bitlines is column decoded so that a subset of the selected data is buffered offchip.

2.2.1 DRAM Reads and Writes

Using the IBM 16Mb DRAM as an example, we now look at the external timing sequences required for reads and writes. It is shown in Figure 1.1a that the chip has 1024 rows, 1024x16 columns, and 16 bits I/O. A 10-bit row address selects one row of 1024, and a 10-bit column address selects 16 columns of 16,384.

Figure 2.11 shows the timing sequence for an asynchronous DRAM read cycle. The Row Address Strobe ($\overline{RAS}$) signal is asserted and the row address is strobed from the address

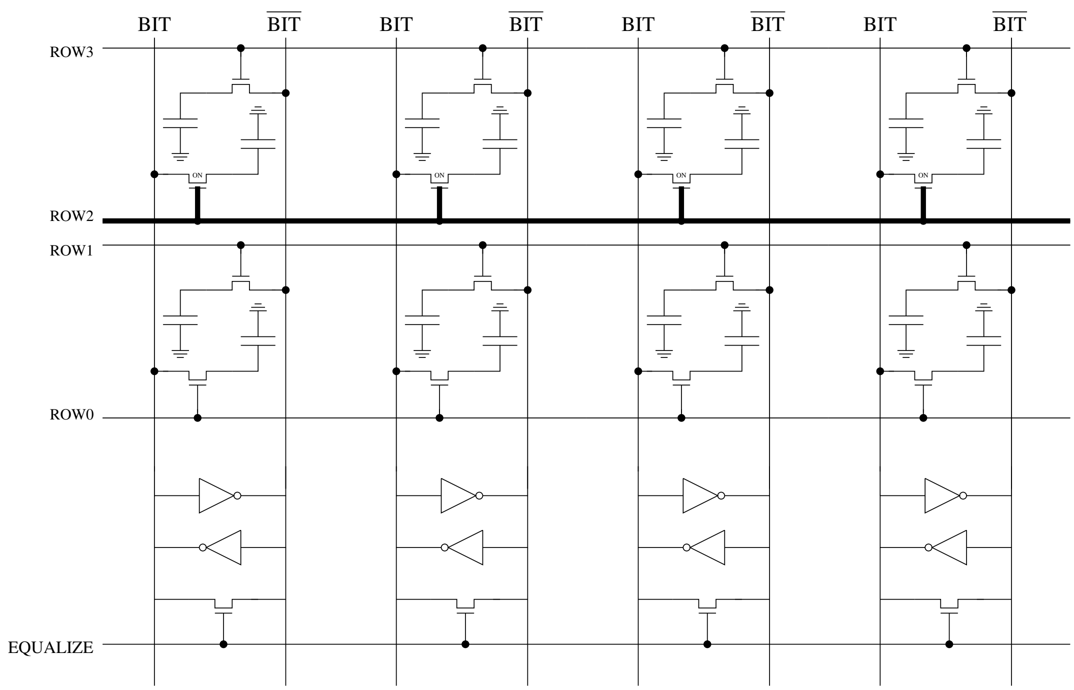


FIGURE 2.10 **DRAM array**

lines. Internally, a wordline is asserted in the array and a row of 16,384 bits are sensed and amplified. The Column Address Strobe ($\overline{\text{CAS}}$) signal strobes the column address next, and sixteen of the amplified bits are buffered through to the data pins. The entire cycle is 100ns.

A write cycle, shown in Figure 2.12, is similar to a read as a row address and column address are strobed. The Write Enable ($\overline{\text{WE}}$) signal is asserted in this case, and the data provided from offchip is buffered through to the selected bitlines. New data is written to both the bitlines and the selected memory cells.

Reads and writes can be combined in a single cycle so that data at a memory location can be read and subsequently replaced. The timing sequence for a read-modify-write cycle is shown in Figure 2.13. The row and column address are strobed, and data is buffered

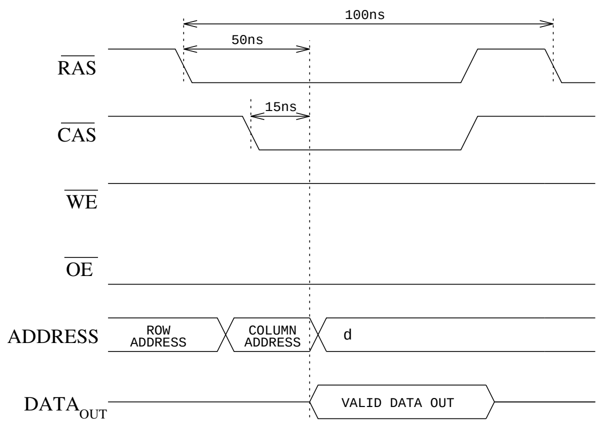


FIGURE 2.11 DRAM Read

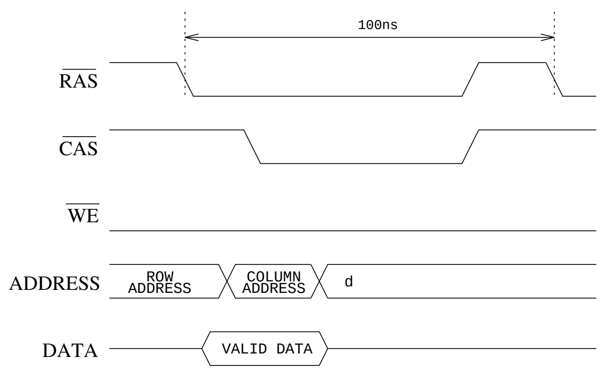


FIGURE 2.12 DRAM Write

through to the data pins. If the Write Enable signal is asserted within 30ns of the CAS assertion, then the data at the *same* memory location can be replaced.

2.2.2 Page Mode Access

A row access in a DRAM activates a large number of cells: in the IBM Dynamic RAM 16,384 are activated by asserting the $\overline{RAS}$ signal. The cells' data bits are amplified on the bitlines, but only sixteen are accessed from offchip when the column address is strobed.

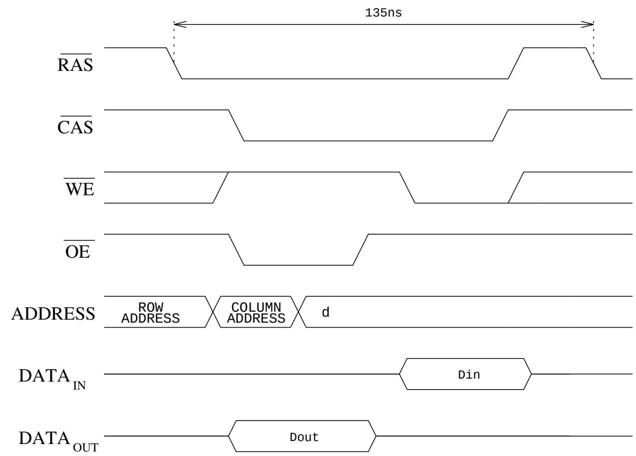


FIGURE 2.13 DRAM Read-Modify-Write

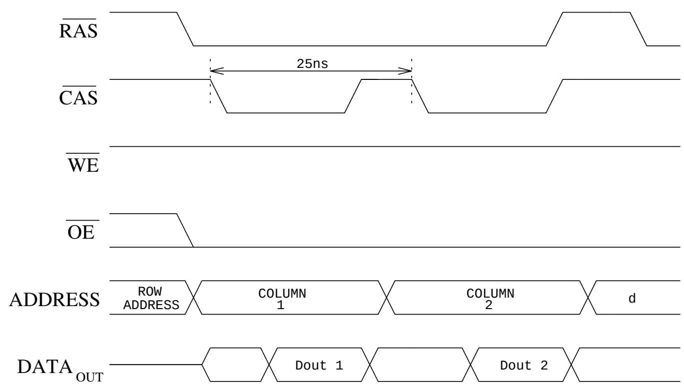


FIGURE 2.14 DRAM Page Mode Access

DRAM's fast page mode allows the amplified data bits to be accessed at a better rate. As shown in the page mode timing sequence of Figure 2.14, a cycle time of 25ns can be attained by changing only the column address between accesses. Since the sense-and-amplify is both a time and power consuming procedure, using these "cached" bits provides a free boost in the DRAM's data rate and reduces overall power consumption.

Using page mode to its fullest extent involves reading or writing every bit in a row before strobing a new row address. Since our chip has 16 data pins and there are 16384 bits per row, we can make 1023 unique fast page mode accesses for every row access.

2.3 Summary

This chapter has provided some of the general background information required for Chapter 3, which discusses in detail the modification of a Dynamic RAM into a SIMD processor-enhanced memory. Becoming acquainted with the vector processors built between 1980 and 1997 will help us to understand where this new Computational RAM fits in the progression of research, and will later (in Section 4.5) allow us to quantitatively compare new technology versus the old.

In the second section, an introduction to Random Access Memory was given. We looked at the basic architecture and operation of Dynamic RAM and were introduced to components and terminology that will be used in future chapters.

CHAPTER 3

C•RAM Design

To this point we have gained a basic understanding of SIMD processing and Dynamic RAM. The C•RAM group is interested in SIMD processing in memory, but would rather avoid the design of memory since it is time consuming, difficult to do correctly, and not within the focus of our research. It was decided that using a predesigned memory chip to which SIMD processors would be added was the most efficient and prudent design approach.

This chapter looks at the design steps taken during a five month session at IBM Microelectronics in Burlington Vermont, where we designed the SIMD processor-enhanced memory chip. We had to choose a DRAM chip suitable for the overhaul, analyze its features and architecture, and sketch a preliminary layout plan with dimension estimates for the Processing Elements. After we decided on a basic model for the PE, it was designed and laid out to match the required physical dimensions. With the Processing Element array complete, we had to devise a timing scheme for operating the chip in its new parallel processing mode.

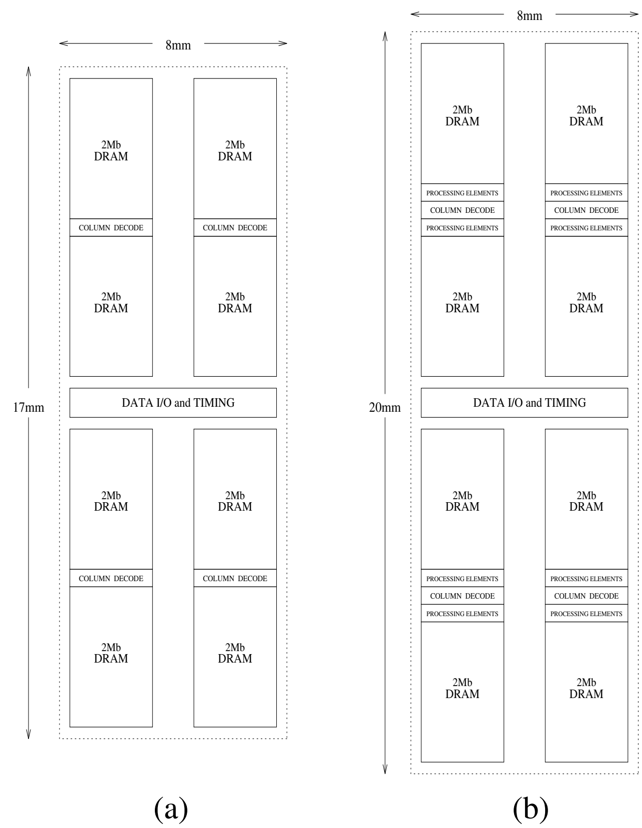


FIGURE 3.1 Floorplans of (a) DRAM and (b) C•RAM

3.1 Design Outline

The objective of this design is to install an array of SIMD Processing Elements inside a 16Mb Dynamic RAM. We were not intending to increase the chip area much in doing so, not more than about 15% to 20% since additional area has a detrimental effect on manufacturing yield [Pri90]. We chose a platform memory chip with an architecture that could accomodate enough PEs without requiring changes to predesigned components or increasing the area more than our target percentage.

Floorplans of the Dynamic RAM and the proposed C•RAM are shown in Figure 3.1. One can see conceptually in this figure that the design involves removal of the existing column

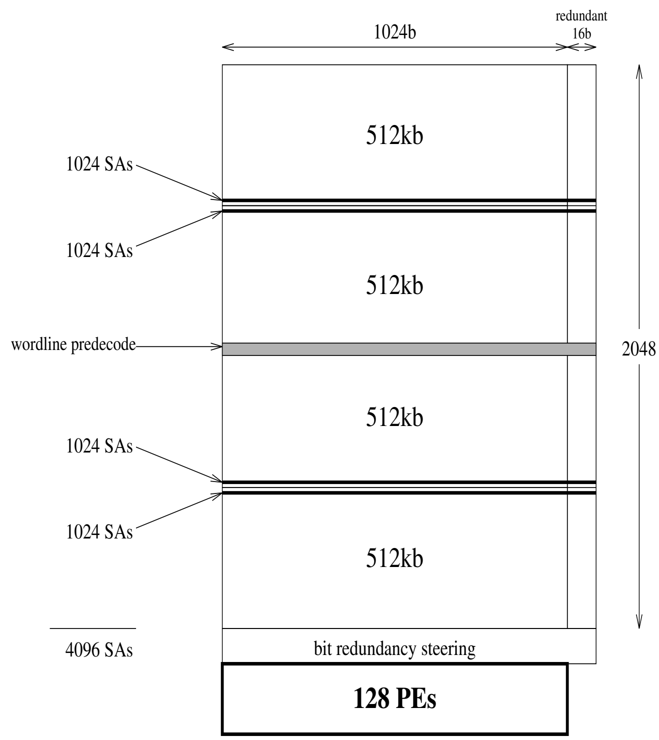


FIGURE 3.2 **DRAM octant with proposed PE location**

decoders between octants, and installation of Processing Elements and new column decoders. Other required tasks are rewiring of data lines from the new column decoders to the data I/O pins at the center of the chip, wiring instruction signals for the Processing Elements, designing new timing circuitry to accomodate a parallel processing mode, and replacing or reworking any displaced or conflicting circuitry.

The octant architecture, which shows the proposed location for the Processing Elements, is illlustrated in Figure 3.2. From the first two figures, we can collect some general facts about the chip's architecture:

- The 16Mb DRAM comprises 2Mb octants (Figure 3.1)
- Each octant contains four 512kb blocks (Figure 3.2)

- Each 512kb block contains 1024 DRAM columns and 1024 Sense Amplifiers (SAs)

The entire chip has 32,768 sense amplifiers by these calculations, but we see in Figure 3.1 that a single row has only 16,384 data bits. For conservation of power during accesses, the chip has two separate memory banks: half the sense amplifiers are active and the other half sit idle during a read or write operation.

Since manufacturing faults in dense circuitry are common, each octant has 16 extra columns which can be used to replace those with physical defects. A bit redundancy steering block is an optional feature used to route data around the defective columns, and is located above the PE location. When defective columns are detected during post-fabrication testing, their column addresses are blown on a bank of fuses which run alongside the array. When the fuses are blown, a redundant column is activated and the data is properly routed through the steering block. Since the PEs receive only good data bits, a small number of column defects will not affect the functionality of C•RAM. Word redundancy is implemented in each 512kb block as three additional rows which, similar to redundant columns, can be activated by blowing row addresses of faulty word lines into a bank of fuses.

Bit-error correction is done onchip by storing extra parity bits with large groups of data. A group of 1024 stored bits has an additional six parity bits, which are used to correct single-bit errors and detect multi-bit errors. Error Correcting Circuitry (ECC) is a differential cascode voltage switch logic tree that inputs data and parity bits, and outputs syndrome bits that point to faulty data bits.

A DRAM design with only Error Correcting Circuitry will have 50% yield if the manufacturing process generates an average of 500 failing cells per chip. With both ECC and bit redundancy, 50% yield is obtained with an average of 2700 failing cells per chip [Kal90a]. ECC is not used in our design, but both column and word redundancy are.

Figure 3.2 shows that 128 PEs will be installed across the pitch of 1024 sense amplifiers, which means each PE will be 8 sense amplifier pitches wide. The cells and sense amplifiers on this chip are 2.8 µm wide [Kal90a], so the PE's width is 8 times that, or 22.4 µm. Each PE will have a local memory of 16kb (= $\frac{2\text{Mb}}{128}$), and access to 32 Sense Amplifiers.

A detailed floorplan of the new architecture is shown in Figure 3.3. Eight blocks of 16 PEs and memory interfaces were installed at the base of each array, and data output drivers were inbetween. Data input and data output lines were already installed in the "spine" (the chip's center vertical column), and the new PE instruction wiring was run from the I/O pads to the PEs up the "centerholes" (the two vertical spaces running through each array). The DRAM designers used only two layers of metal, but to meet time and chip area constraints, the C•RAM components were implemented with three. These conflicting strategies posed a problem for us, particularly since DRAM's metal-2 layer was run vertically up the spine and centerholes, and C•RAM's was installed horizontally across the PEs, spine, and centerholes. The problem was overcome by lifting all conflicting DRAM metal-2 up a layer to metal-3.

Due to the one-bit wide access to each local memory, a 16-bit DRAM-mode transfer routes data to and from the local memories of 16 adjacent PEs. Operands are therefore transferred to groups of 16 PEs bit-serially, which is certainly not ideal since the host processor usually transfers data in an operand-serial, bit-parallel manner. These orthogonal schemes require an intermediate "corner turning" stage, which can be done either in hardware with a 16x16 two-dimensional access memory, or with software that makes the host transfer data in a bit-serial manner.

With a basic implementation strategy in place, we can now look at the Processing Element design.

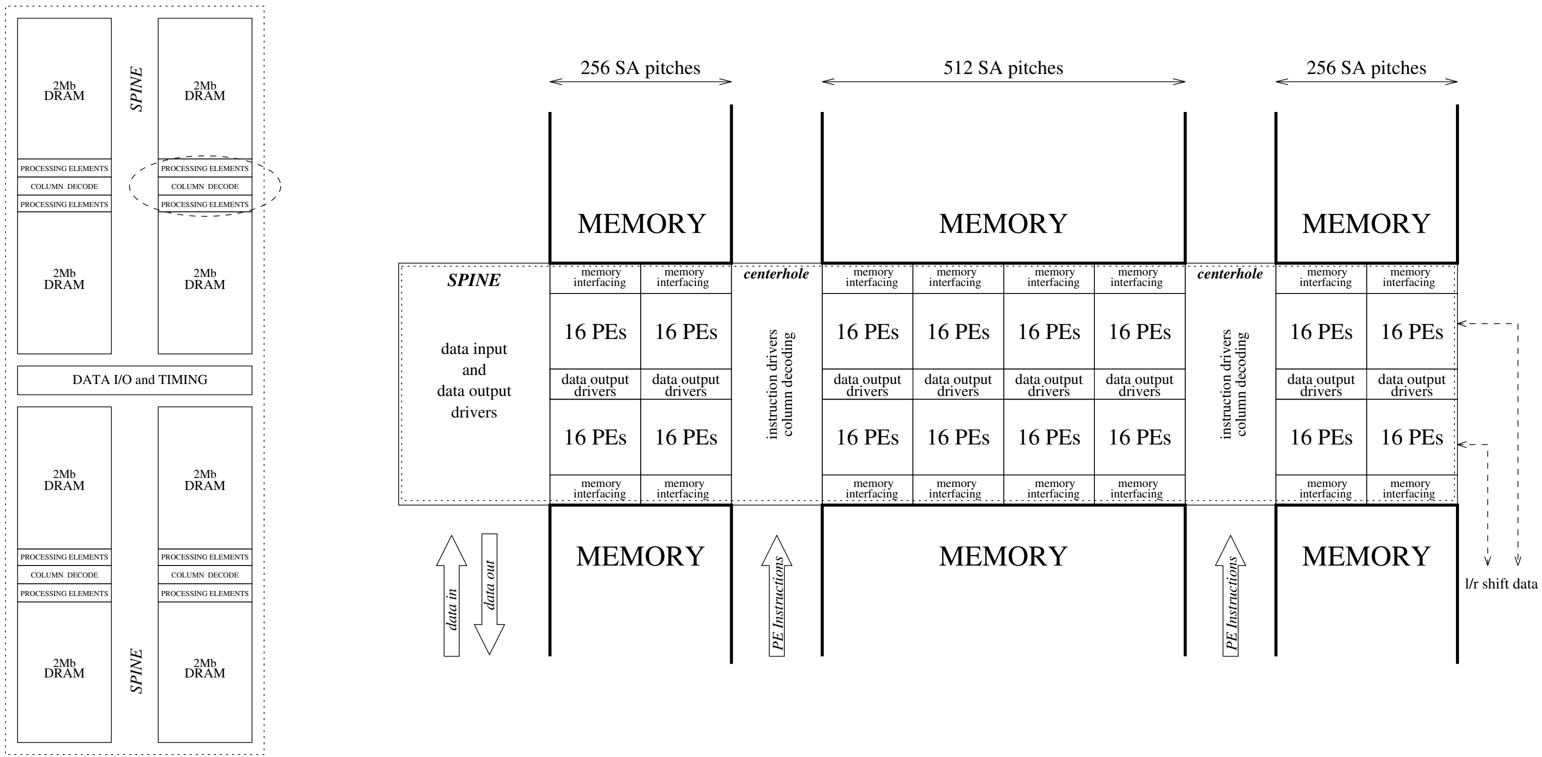


FIGURE 3.3 Layout of new circuitry between octants

3.2 The Processing Element (PE)

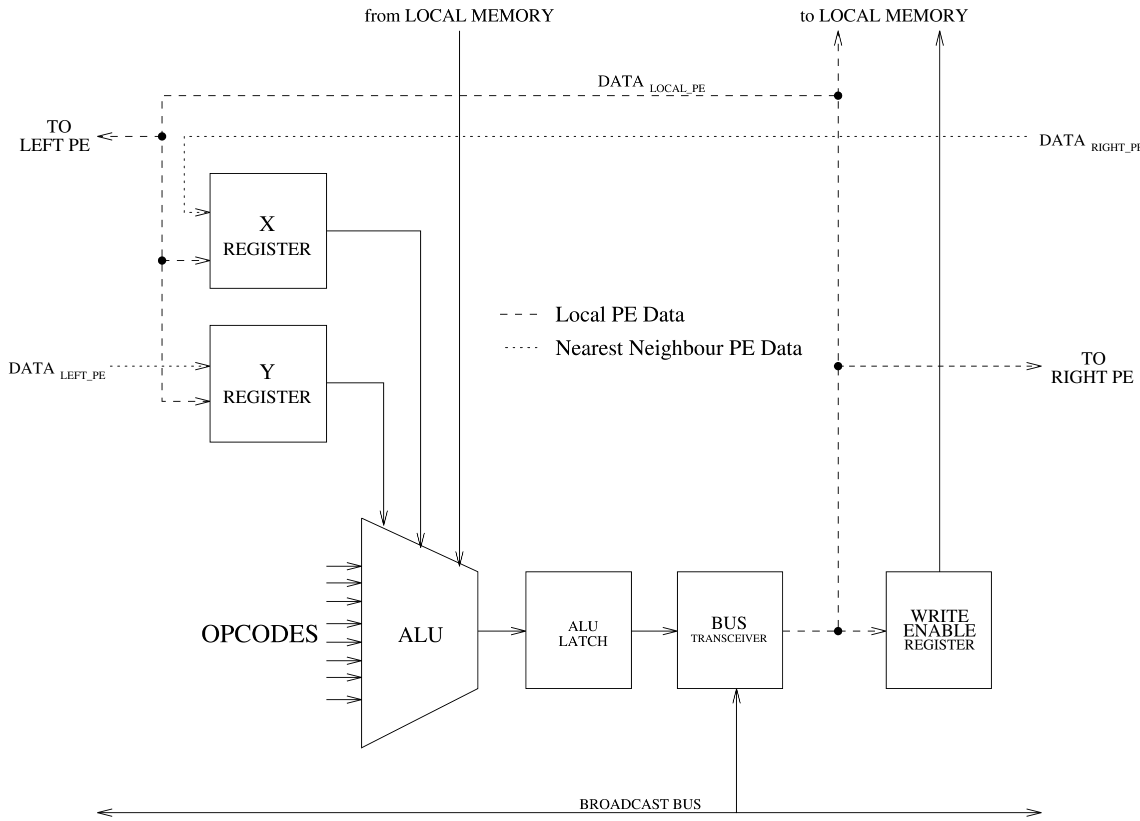
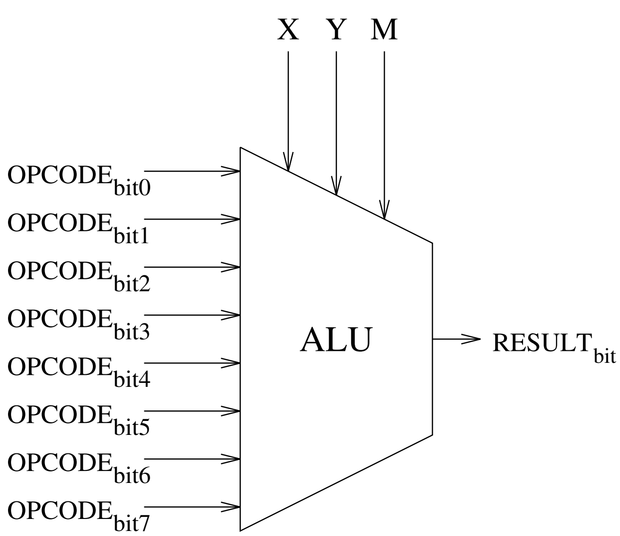


FIGURE 3.4 Processing Element

A Processing Element is a single-bit computer intended for use in large parallel processing machines. It is designed to be physically small, but is given as much computing ability as the silicon area will allow. Several Processing Element (PE) designs were proposed by Duncan Elliott [Ell97], where size increases caused by addition of extra features were compared against performance potential. His dual storage-register version was selected for implementation in this design, the structure of which illustrated in Figure 3.4. It has two storage registers (X and Y) an ALU, a Write Enable (WE) Register, and a broadcast bus transceiver. All calculations performed by the PE are one bit wide, which is different from conventional bit-parallel ALUs that load entire 8, 16, or 32-bit wide operands and use a ripple carry for addition. The PE architecture is useful when a large number of them is used to perform hundreds or thousands of operations in parallel.

Since the structure is SIMD, the processing array has a global instruction bus fed to all the PEs. The instruction set includes information about which function is to be calculated by the ALU, which register the result is to be written to, and whether the global bus transceiver should be enabled. Before looking further at the PE's operation, we need to examine each component in detail.

3.2.1 ALU



X	Y	M	RESULT$_{bit}$
0	0	0	OPCODE$_{bit0}$
0	0	1	OPCODE$_{bit1}$
0	1	0	OPCODE$_{bit2}$
0	1	1	OPCODE$_{bit3}$
1	0	0	OPCODE$_{bit4}$
1	0	1	OPCODE$_{bit5}$
1	1	0	OPCODE$_{bit6}$
1	1	1	OPCODE$_{bit7}$

FIGURE 3.5 ALU functionality

The ALU is implemented as an 8-to-1 multiplexor, as shown in Figure 3.5: values of the X-register, Y-register, and memory input (M) select one of the 8 opcode bits, and buffer it through to the output "result". This multiplexing structure can be used to calculate any function of the 3 input variables (X,Y,M) by defining its truth table with the opcodes. For example, if we wish to calculate the sum function $f(X, Y, M) = X \oplus Y \oplus M$, the opcode bits 7-0 would be "1001 0110", or 96_{16}.

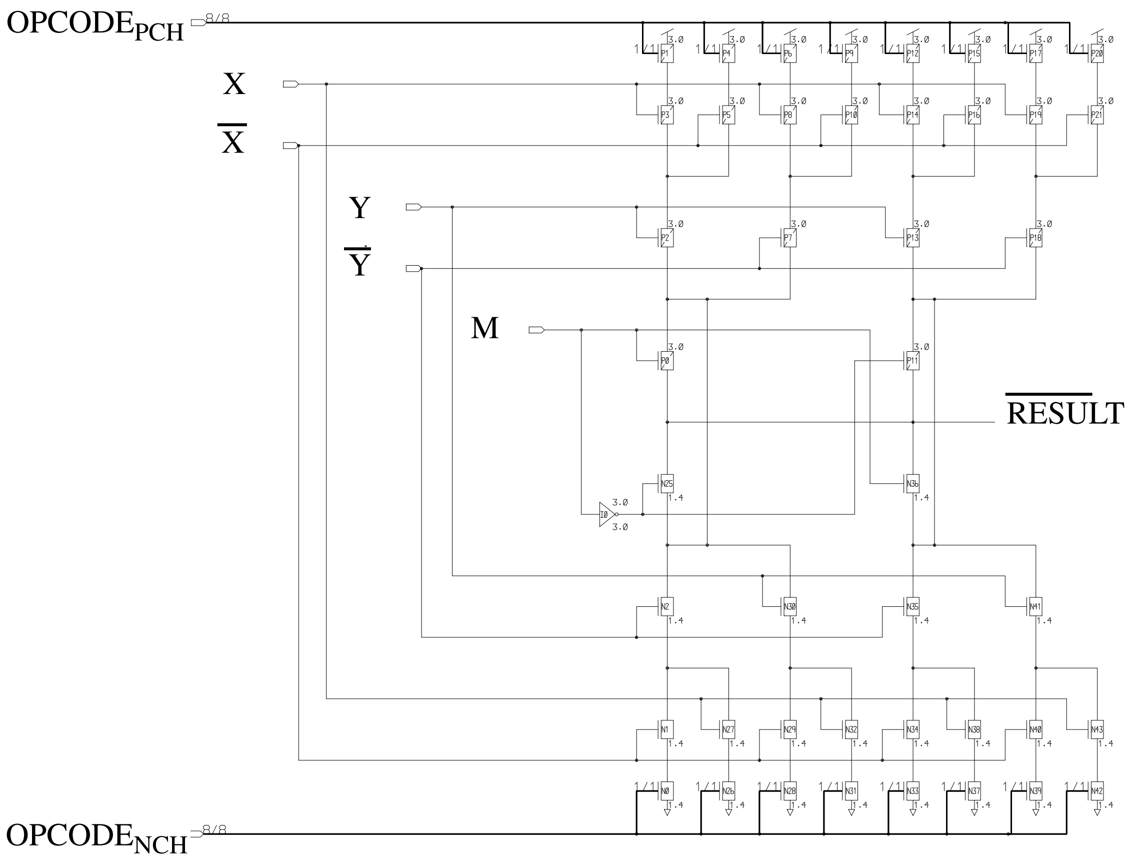


FIGURE 3.6 ALU schematic

Our implementation of the ALU is the static CMOS tree structure shown in Figure 3.6. Combinations of the X,Y,M and opcode inputs select a single path from either VDD or ground to the result node.

3.2.2 X and Y Registers

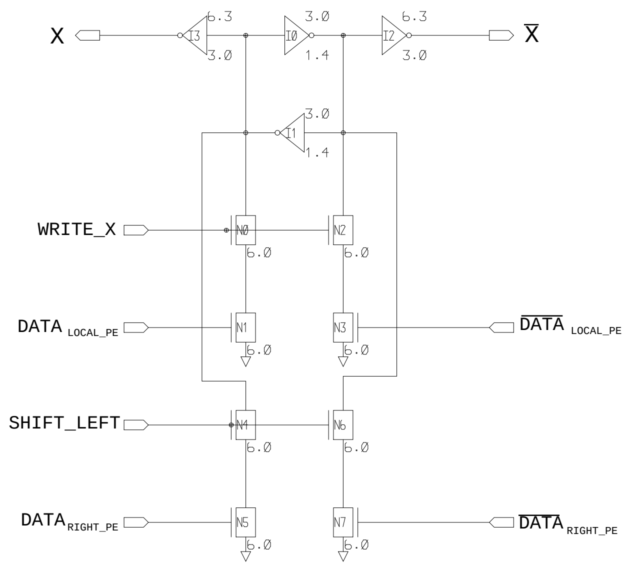


FIGURE 3.7 **X-register schematic**

The X and Y registers are single-bit storage elements capable of capturing either same-PE or nearest neighbour-PE data. The X-register can receive data from the right neighbour in a shift-left function, and the Y-register can receive data from the left neighbour in a shift-right function. Control inputs for the X-register are WRITE_X and SHIFT_LEFT, and the Y-register uses WRITE_Y and SHIFT_RIGHT. Data cannot be written and shifted into a register simultaneously since opposing values would generate unpredictible results and burn power needlessly.

3.2.3 ALU-latch

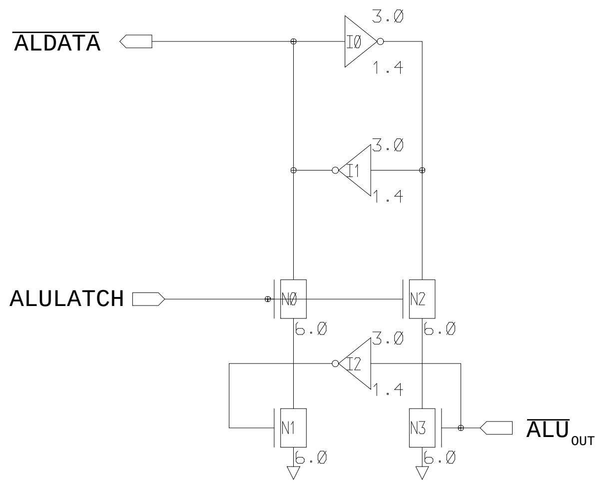
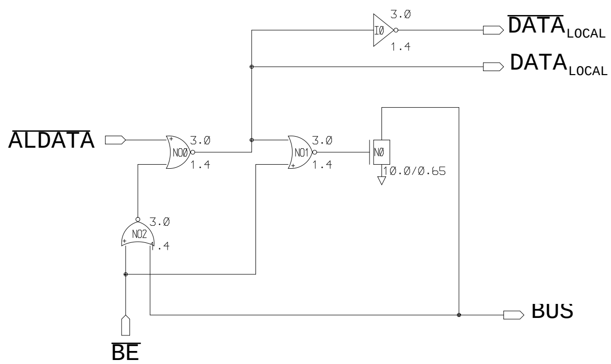


FIGURE 3.8 ALU-latch schematic

As is conventional for synchronous digital design, the Processing Element's static logic implementation requires that there be a break in the feedback loop from the ALU output to its input. For example, a ring oscillator is created when the ALU is calculating $\overline{X}$ and its output is connected to its X input. The ALU latch is a static register which, when activated, buffers through the ALU result bit to the bus transceiver - and its activation should not coincide with the activation of either the X or Y registers.

3.2.4 Bus Transceiver



(a)

BE	ALDATA	BUS	$DATA_{LOCAL}$	BUS
0	0	1	0	1
0	1	1	1	1
1	d	0	0	0
1	0	d	0	0
1	1	1	1	1

Bus Enabled

(b)

FIGURE 3.9 Bus Transceiver (a) schematic (b) truth table

The Bus Transceiver controls a full chip inter-PE bus. Once enabled, the Bus Tranceiver of any PE with a result 0 will drive the bus and results of every other PE low. The bus is precharged at the start of every cycle and weakly held at logic “1” by a separate controlling component.

By attaching this bus to all 1024 Processing Elements, we can quickly determine if any Processing Element computed a “0” in the most recent calculation. For example, a database search can be performed by assigning one data element to a PE, and having all the PEs compare their data elements to a key. The comparison is done such that those which match the key value calculate a “0”, and all others calculate a “1”. When the bus transceiver is enabled following this calculation, the bus will be driven low if and only if at least one PE has found the key.

3.2.5 Write-Enable Register

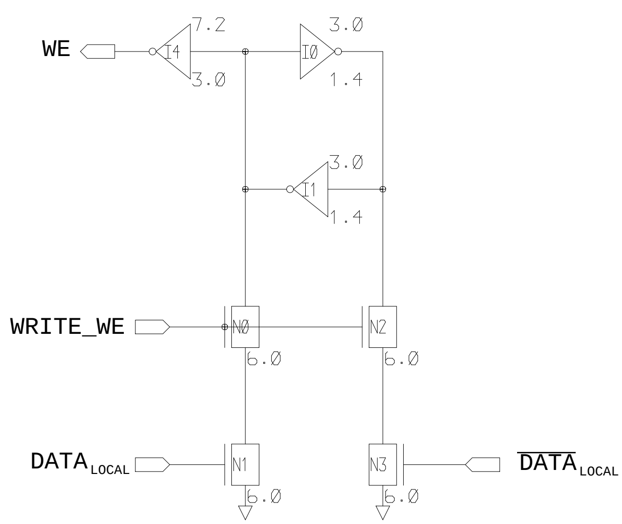


FIGURE 3.10 Write-Enable Register schematic

The Write-Enable Register governs the PE’s ability to alter its local memory. A PE with a WE-Register value of 0 cannot write to memory, whereas a PE with a WE-Register value of 1 can. Since the PEs are instructed by a common controller, they must all be led through every execution branch of data dependent “if-then-else” statements, and the Write-Enable register is used to switch off the writing capabilities of those PEs that do not belong to the branch of execution currently being performed. A multiplication algorithm, for example, loads this register with a multiplier bit at the start of each iteration to enable or disable PEs, making for a conditional add of the multiplicand.

3.2.6 Memory interfacing.

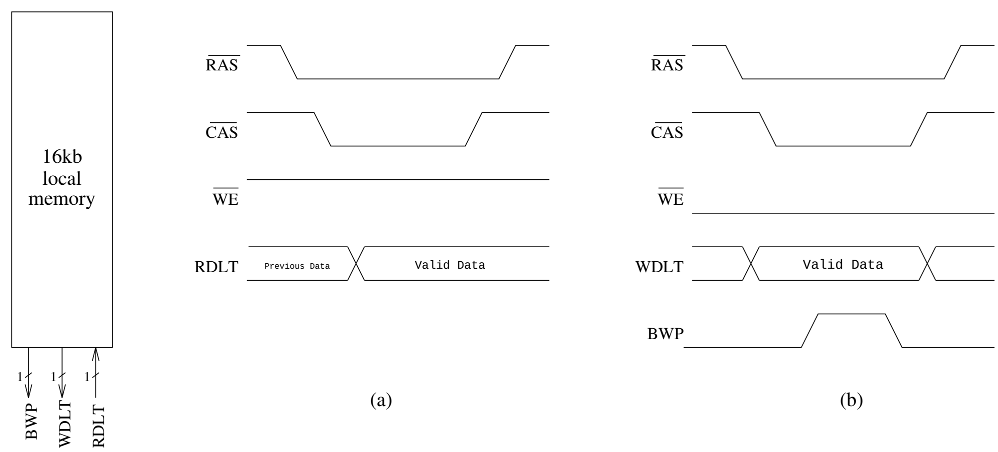


FIGURE 3.11 Interfacing with PE local memory: (a)read (b)write

One of the attractions of laying out the Processing Element on an 8 Sense Amplifier pitch is that the memory array control lines are available on the same interval. A PE has access to its local memory through 3 control signals: BWP (Bit Write Positive), RDLT (Read Data Line True) and WDLT (Write Data Line True). A memory interface circuit is installed between each PE and the array to control these lines in both regular memory (DRAM) and parallel processing (C•RAM) modes.

3.2.7 PE control.

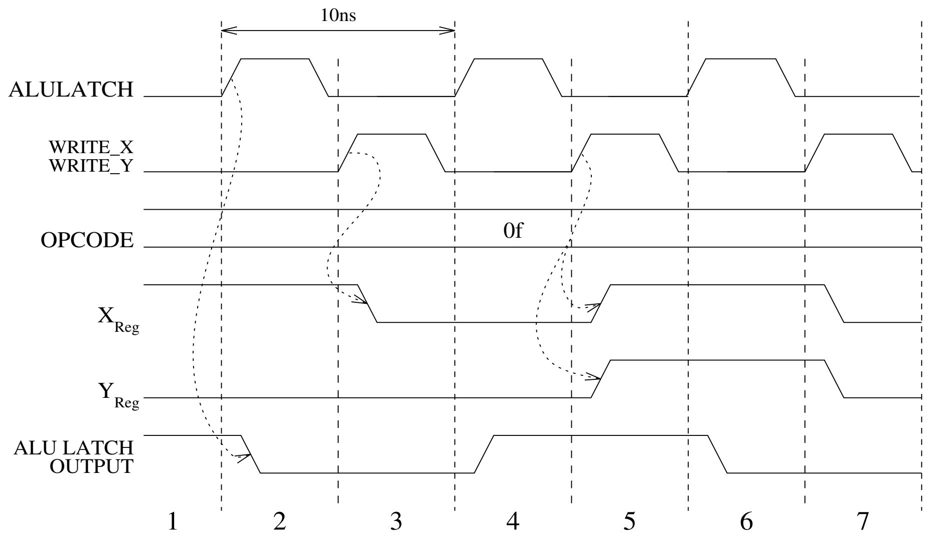


FIGURE 3.12 PE Timing Example: X-Register inversion

Direct control of the Processing Elements is achieved by providing opcodes to the ALU, and enabling latches of registers in a particular sequence. Figure 3.12 shows an example in which the bit content of the X register is repeatedly inverted. We analyse its first five steps:

1. The Opcode which calculates $\overline{X}$ is $0f_{16}$. Initial value of the X register is 1, Y register is 0, and RESULT (ALUlatch output) is 1. At this point, the ALU has already calculated the value of $\overline{X}$.
2. Assertion of the ALULATCH signal buffers through the bit calculated by the ALU (0) to the RESULT node.
3. Assertion of the WRITE_X and WRITE_Y signals writes the bit value of RESULT (0) to X and Y. The X input to the ALU is changed, so it computes a new value for $\overline{X}$.
4. ALULATCH buffers through the new value of $\overline{X}$ to RESULT.

5. WRITE_X and WRITE_Y change the values of the two registers to 1.

If the sequence were to continue on indefinitely, the values of the X and Y registers would alternate in unison.

3.3 Power and Area Increase Estimates

It is important to examine how much the Processing Element array contributes to the power consumption and silicon area of the processor enhanced memory. C•RAM is intended to be marketed as a memory chip with processing capabilities, and not just as a parallel processor. Power budgets of memory chips are tightly constrained since DRAM is used extensively in laptop computers – machines which are sold on the basis of their performance and battery life. If C•RAM is introduced as a new version of a memory chip which revolutionizes computing, an announcement that it consumes far more power than a regular memory chip will not please system designers. The area budget is important too, since the cost of high production volume memory chips can be approximated by the amount of silicon they occupy. Any area increases we make add to the premium price of C•RAM. Also, this PE array has no redundancy protection, so chip area increases will hit the manufacturing yield exponentially [Pri90].

Table 3.1 shows a breakdown of all the Processing Element components, which includes their energy requirements and transistor counts. The energy requirement figures were generated through simulation of the PE with every component activated, so values shown can be considered worst case. In practise, only the ALU, bus transceiver, and ALU latch are used during every cycle. The X and Y registers are used once every 3 to 4 cycles on average, and the WE register is used about once every 25.

COMPONENT	approximate energy requirements (measured)	# of transistors	% total energy requirement	% transistor count	activity factor
ALU	2.5pJ	46	39%	41%	+
X Register	1.0pJ	16	15%	14%	-
Y Register	1.0pJ	16	15%	14%	-
Bus Transceiver	1.0pJ	15	15%	13%	+
ALU Latch	0.5pJ	10	8%	9%	+
WE Register	0.5pJ	10	8%	9%	-
	6.5pJ	113			

TABLE 3.1 PE energy and transistor breakdown

The PE's biggest power consumer by far is the ALU: it has the highest energy requirement of all components, comprises the most transistors, and is used once every cycle. Further optimizations of the PE should therefore be focused on the ALU.

For an analysis of the PE array's approximate contribution to the chip's power consumption, we will ignore the various activity factors and assume that each PE consumes the worst-case 6.5pJ per cycle. With 1024 PEs, the full array consumes 6.7nJ.

Since so much of the memory chip's area is occupied by memory arrays, a reasonable estimate of its power consumption is the amount it consumes charging bitlines. The 16Mb DRAM has 4096 Sense Amplifiers per octant (Figure 3.2), and therefore has 32,768 in the whole chip. A row access does not activate all 32k since activating too many bits per access unnecessarily increases power consumption. On the other hand, enough memory cells need to be activated during a row access to keep the refresh overhead low. The IBM DRAM activates half its sense amplifiers during a cycle, while the other half sit equalized.

The formulae for power and energy consumption, given a capacitance (*C*), voltage (*V*), and cycle frequency (*f*) are:

$$\text{Power} = \frac{1}{2}CV^2f \tag{3.1}$$

$$\text{Energy} = \frac{\text{Power}}{f} = \frac{1}{2}CV^2 \tag{3.2}$$

At the beginning of a cycle, one bitline in a pair is at V_{DD}, and the other is at 0. Both are equalized to $\frac{V_{DD}}{2}$, and then one is swung to V_{DD}, and the other to 0. Two bitlines are swung $\frac{V_{DD}}{2}$, and then later swung another $\frac{V_{DD}}{2}$, so the power consumed during a cycle is the amount required to swing 2 bitline voltages V_{DD}.

$$\text{Power}_{\text{bitline}} = 2 \cdot \left[\frac{1}{2}C_{\text{bitline}}(V_{DD})^2 f\right] \tag{3.3}$$

Since the IBM DRAM activates half its sense amplifiers during a cycle, 2048 are active per octant, and 16,384 are active per chip. The power consumption of the entire chip's bitlines is the result of the previous equation multiplied by the number of active bitlines.

$$\text{Power}_{\text{all bitlines}} = 8\frac{\text{octants}}{\text{chip}} \cdot 2048\frac{\text{active bitline pairs}}{\text{octant}} \cdot \text{Power}_{\text{bitline}} \tag{3.4}$$

The capacitance of a single bitline in this DRAM is 322*f*F [Kal90a]. With V_{DD} at 3.3V and a cycle time of 100ns, the entire chip consumes 575mW charging bitlines, and the energy required to perform one cycle is 57.5nJ.

Since all 1024 PEs consumes 6.60nJ per cycle, and the DRAM consumes 57.5nJ, a C•RAM requires 11.7% more energy than a DRAM to run the Processing Elements. At a cycle time of 50ns, the power consumption of both array and PEs combined is 1282mW. With a ceramic package thermal conductivity of about 35°C/W, we can estimate the chip's temperature during operation.

$$
\begin{aligned}
\text{Temperature}_{\text{C•RAM}} &\cong \text{Temperature}_{\text{room}} + 35\frac{^{\circ}\text{C}}{\text{W}} \cdot 1282\,\text{mW} \\
&= 22.5^{\circ}\text{C} + 44.9^{\circ}\text{C} \\
&= 67.4^{\circ}\text{C}
\end{aligned}
\tag{3.5}
$$

3.4 C•RAM Timing

We have seen in Section 2.2 how a DRAM is operated to perform regular reads and writes. The chip's timing sequence set now needs to be expanded to accomodate operation of the Processing Elements.

It has been mentioned before that C•RAM is a memory chip with processing capabilities, and not just a parallel processor. Its appeal, we hope, will be due to its compatibility with current systems, and the fact that one can directly replace regular DRAM chips with C•RAM. But for C•RAM to be accepted as a memory chip, it must meet industry standards. The Joint Electron Device Engineering Council (JEDEC) is a committee that devises the standards with which industry standard memory must comply. JEDEC standards do not allow additional control pins on memory chips, so C•RAM's PEs must be controlled with only the RAS, CAS, and Write Enable signals. Fortunately, the chip has unused test modes which we can exploit for parallel processing operation.

Reads and writes in "DRAM mode" involve selecting a memory location with 10 row and 10 column address bits in a RAS/CAS sequence. Operation in "C•RAM mode" is similar except we want to select a different subset of bits. The row and column architecture of C•RAM is illustrated in Figure 3.13. For each PE to receive a bit from its local memory, a 16,384-to-1 decode is required. The row decode is 1024-to-1, which uses 10 row address bits. There is also a 16-to-1 column decode within each PE's memory, requiring 4 column address bits.

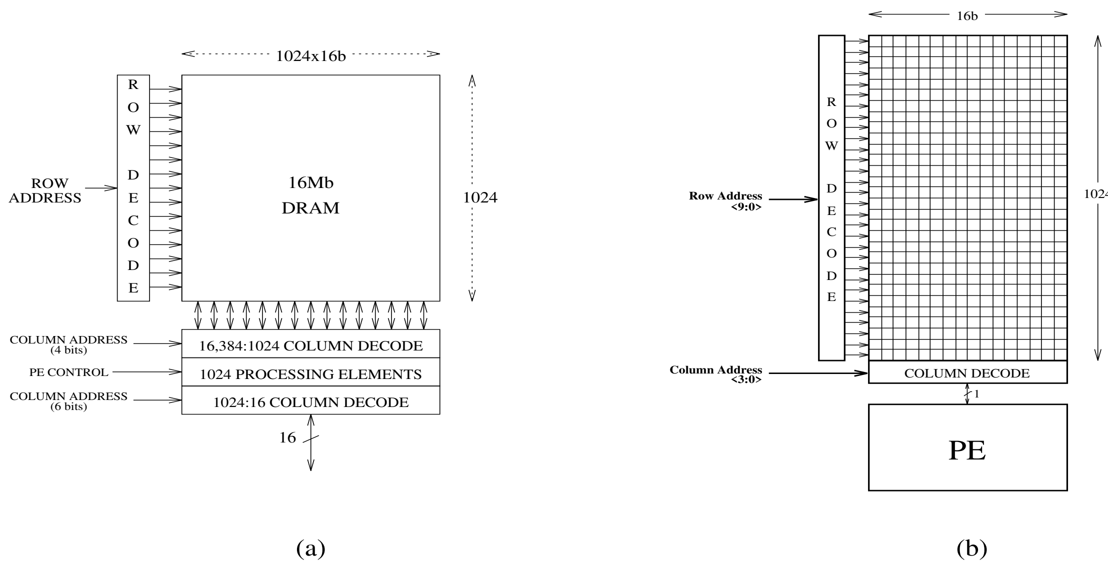


FIGURE 3.13 Architecture of (a) C•RAM (b) a single PE and its local memory

The basic structure of a C•RAM cycle is shown in Figure 3.14. A PE control code contains the 8-bit opcode, and information about which registers are to be latched, whether the bus transceiver should be enabled, and whether the PE should write back the cycle's result to memory. It is a 16-bit wide code strobed from the datalines upon assertion of $\overline{WE}$, and its structure is shown in Figure 3.15. For example, a C•RAM cycle with control code "00001111 10100011" would give the ALU an Opcode of $0f_{16}$, latch the ALU data, enable the bus tie, and write to the X and Y registers.

A single C•RAM cycle should be able to perform some or all of the following tasks:

- Read data from the local memory
- Perform an ALU operation with Opcodes provided in a control code
- Enable the bus transceiver
- Write to registers
- Write data back to the original memory location

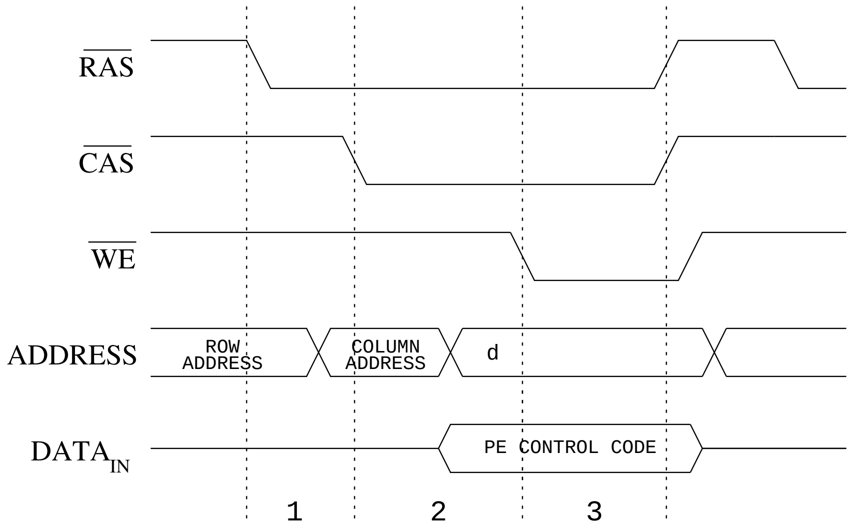


FIGURE 3.14 **Basic structure of a C•RAM cycle**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP7	OP6	OP5	OP4	OP3	OP2	OP1	OP0	ALU	PW	BT	SR	SL	WW	WY	WX

OP*x*- Opcode bit x
ALU- ALU_Latch
PW - PE_Write
BT - Bus Tie Enable
SR - Shift Right
SL - Shift Left
WW - Write to Write Enable Register
WY - Write to Y Register
WX - Write to X Register

FIGURE 3.15 **PE Control Code**

Rather than introduce a confusing list of different timing sequences and rules to operate the PEs, we shall generate only 3 or 4 useful ones. The first and most important one is the C•RAM mode entry sequence.

3.4.1 C•RAM mode entry

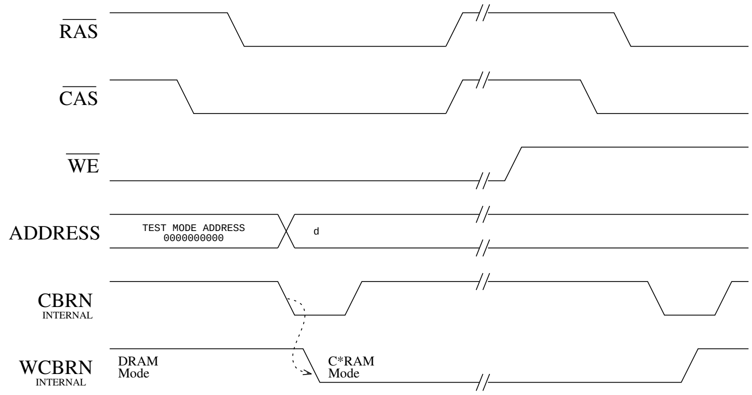


FIGURE 3.16 C•RAM Mode entry and exit

C•RAM mode was implemented as a vacant test mode on the DRAM, and is entered by asserting Write-Enable and CAS before asserting RAS (WCBR). Once the C•RAM mode is entered, the chip's regular I/O capabilities are disabled and only C•RAM-type cycles can be performed. The mode is exited by asserting CAS before RAS without Write Enable (which, incidentally, is a CAS before RAS refresh cycle).

3.4.2 Operate (Op)

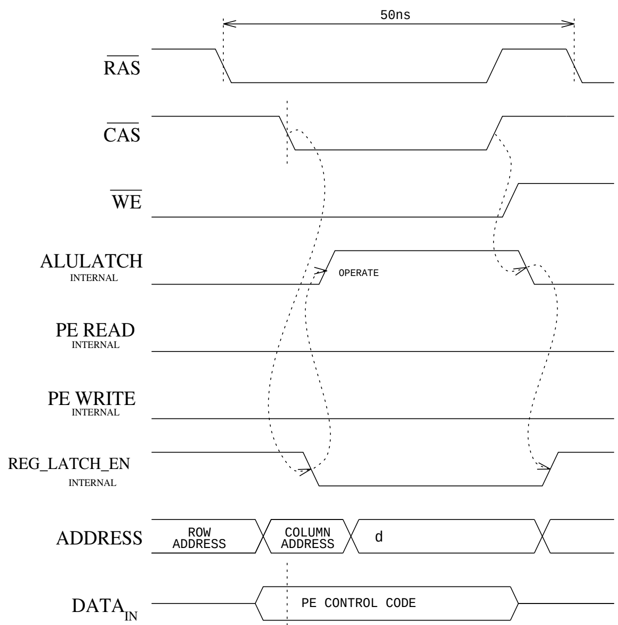


FIGURE 3.17 **C•RAM Operate (Op)**

An Operate performs no memory reads or writes during the cycle, so the ALU computes a function of the X and Y register values and the previously read memory bit. Figure 3.17 shows the timing constructs of an Operate, which is typically used when a specific value is required in one or more registers. For example, if a value of 1 is desired in the X, Y, and WE registers, one would perform an Operate with opcode ff_{16} and the appropriate register bits set in the PE control code.

3.4.3 Read Operate (ROp)

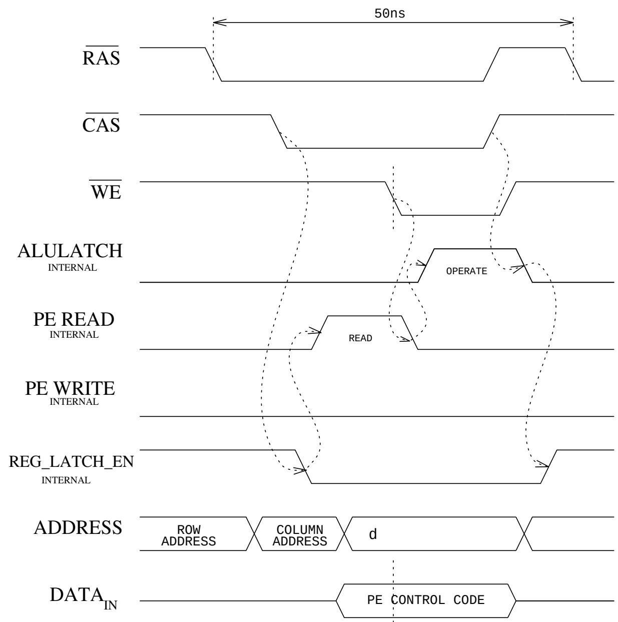


FIGURE 3.18 C•RAM Read Operate (ROp)

The Read Operate is a useful timing construct which reads data from memory, performs an ALU operation on the read data, and writes the result to any combination of internal registers. It should be noted that because cycling the Processing Elements involves the selection of a single memory location with a row and column address, a PE cannot read from a memory location, operate, and write back to a *different* memory location in a single cycle. Any Read-Operate-Writeback involving more than one memory location in each PE's local memory must be broken into a Read-Operate and a subsequent Write cycle.

One can see in the timing diagram shown in Figure 3.18 that the internal PE_Read signal is automatically generated by the timing circuitry if WE is high after CAS. Data from the selected memory cell is buffered through to the PE at this point. Once WE is driven low, ALULATCH is asserted, and the bit computed by the ALU is latched.

3.4.4 Read Operate Writeback (ROpW)

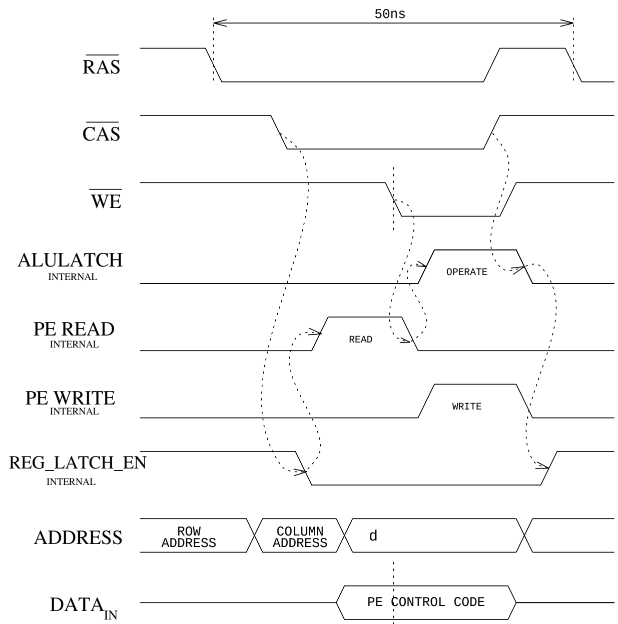


FIGURE 3.19 C•RAM Read Operate Writeback (ROpW)

Read Operate Writeback adds an extra step beyond the Read-Operate. Data is read from a memory location within the PE's local memory, the ALU computes the function defined by the opcodes, and the calculated bit is written back to the *same* memory location. Writeback is achieved by setting the "PE_Write" bit in the control code.

For example, the two-operand addition R+=A uses a Read Operate Writeback. During a bit-serial addition, each bit of R (r_x) is read as an operand, then immediately replaced by the new sum bit. The ROpW can be used in this instance to read the operand and write back the calculated bit to the same location in a single cycle.

3.4.5 Write (Wr)

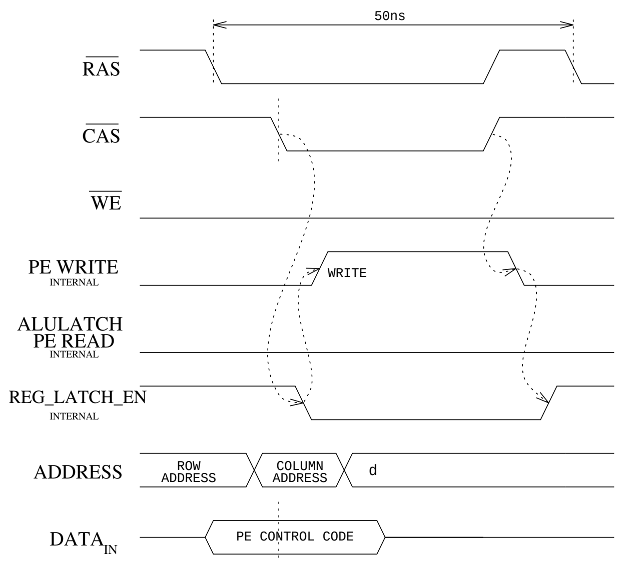


FIGURE 3.20 C•RAM Write (Wr)

The Write cycle selects a memory location within the PE's local memory, and writes to it the bit calculated in the most recent Read-Operate or Read-Operate-Writeback. It is the last step of a procedure which reads data from a local memory location and performs an ALU calculation (ROp), then writes the result back to a different local memory location (Wr). Write with no operate is achieved by setting just the PE_Write bit in the control code and leaving the ALU_latch bit unset.

3.5 Summary

This chapter has outlined the design of a processing in memory chip with a 16Mb IBM Dynamic RAM. We looked at the architectural and physical changes to the chip, the PE design, and the new timing sequence set for C•RAM mode.

The Processing Element was laid out with approximate dimensions of 320µm x 22.4µm, so the 1024 PE array alone adds about 8% to the chip length and area. When the layout was complete, it was noted that all additional circuitry required by C•RAM (PEs, memory interfacing, bit redundancy steering) increased the chip's length and area by 14%. The extra silicon required by C•RAM will make it a "premium priced" memory chip, but it is hoped that people will be willing to pay extra for a significant performance increase in multimedia and other such applications.

We showed in this chapter that the extra PE circuitry adds about 12% to the chip's power consumption. This percentage is not severe, particularly when we consider that the energy requirements increase substantially only when the PEs are active. Although there will be a modest power consumption change in DRAM mode caused by the additional capacitance of new circuitry and extra distances signals have to travel, the increase should be negligible.

The C•RAM timing sequences derived in the last section will be used in the next chapter for generating algorithms and performance estimations.

CHAPTER 4 C•RAM Performance

"Any sufficiently advanced technology is indistinguishable from magic."

Arthur C. Clarke

In the previous chapter we looked at the basic architecture of memories, and the design process taken to install PEs inside a DRAM. From there, we derived some approximations on how much the Processing Elements will add to the power consumption, area, and cost. A key feature of the C•RAM is its computational performance since people will be unlikely to pay extra for it unless they know how well (or if) it will perform target applications. The content of this chapter is an extensive analysis of the computational operation and performance of C•RAM.

We learned in the previous chapter that the Processing Elements are cycled with 4 timing constructs: Read-Operate (ROp), Operate (Op), Write (Wr), and Read-Operate-Writeback (ROpW). Large applications which require lots of computation perform integer adds, subtracts, multiplies and divides frequently, so it would be useful to generate a microroutine for each one. Each microroutine will comprise a list of PE cycles required to perform the arithmetic.

A C•RAM simulator was implemented to analyse and verify the microroutines. It performs as a 16-PE C•RAM with 16x512b memory, and has operation similar to the IBM part. It has 3 base functions available with the operation syntax shown below:

rop(READ$_{row}$,"OP$_{16}$","xyw")

ropw(RW$_{row}$,"OP$_{16}$","xyw")

wr(WRITE$_{row}$)

For example: rop(1,"cc","xy")

reads from row 1, performs ALU operation with opcode cc$_{16}$, and writes the result to the X and Y registers.

Once this library of microroutines is assembled, we can look to larger applications. It was mentioned earlier in the introductory chapter that C•RAM could be used for many different computing problems including real time video compression, database operations, graphics rendering, and large scientific calculations. We are most interested in real-time image processing due to its challenging frame rate requirements, suitability to SIMD processing, and potentially large market. The logistics of performing Discrete Cosine Transforms and motion estimation, the two computationally heavy components of MPEG-2 video compression, are discussed and performance estimates are derived.

C•RAM is very good at searching and sorting database elements, even though the inter-PE communication is one dimensional and a single bit wide. Performance estimations are generated and compared to those of standard uniprocessor algorithms.

4.1 DRAM Mode Performance

Computational RAM DRAM mode performance is important to us since data may have to be installed before, and results retrieved retrieved after a job is performed. The I/O rate for fine-grained parallelism can dominate a C•RAM's performance. For algorithms that perform several operations on the same data, the I/O rate is less critical.

Looking back at Figure 1.1, we see that the DRAM has 1024 rows and 1024x16 columns of bit storage. Each of the 1024 PEs works on 1 bit during any given cycle, so the entire array works on 1024 simultaneously. We are therefore interested in the time it takes to transfer on or offchip *n* bits per PE, or 1024*n* bits.

We recall from Section 2.2 that DRAM reads and writes can be performed every 100ns with a new row access taking place in between each one.

$$t_{\substack{\text{n-bit I/O} \\ \text{worst case}}} = \frac{16\ \text{b}}{100\text{ns}} = 160\frac{\text{Mb}}{\text{s}} \tag{4.1}$$

Also in that section, we saw that data can be transferred much more quickly if we choose not to change the row address between every transfer. Changing just the column address yields a 16 bit data transfer every 25ns, which is the fastest data I/O rate available. We cannot use fast page mode in every cycle, since there are only 16384 bits per row, but at 16b per transfer, 1023 fast page mode cycles can be done before a new row needs to be sensed and amplified.

$$t_{\substack{\text{n-bit I/O} \\ \text{bestcase}}} = \frac{16\ \text{b}}{\left(\frac{1}{1024}\right)100\text{ns} + \left(\frac{1023}{1024}\right)25\text{ns}} = 638\frac{\text{Mb}}{\text{s}} \tag{4.2}$$

Page mode can be used to its fullest extent during C•RAM data transfers. Looking back at our diagram of a PE and its local memory in Figure 3.13, we can see that installing a bit in each of the 1024 PEs' local memories requires sequencing all 2^6 combinations of the 6 unused column address bits (64 transfers of 16b each to 1024 PEs).

On a C•RAM with 1024 PEs, the time required to transfer *n* bits to/from each PE is 6.4*n* µs worst case, and 1.6*n* µs best case.

4.2 C•RAM Arithmetic

4.2.1 Addition

The simplest arithmetic operation, and the one from which many others are derived, is addition. A single addition can be performed on a C•RAM by arranging data in a PE's local memory as shown in Figure 4.1. The addition is performed bit-by-bit using the following algorithm:

```
procedure add(alsb,blsb,rlsb,nbit       :int)
    % Performs an nbit-bit addition

    rop(0,"00","y")             %Set the carry bit to 0
    for bit:0..nbit-1
        rop(alsb+bit,"aa","x")  %Load 1st bit
        rop(blsb+bit,"96","")   %Calculate sum
        wr(rlsb+bit)
        rop(blsb+bit,"e8","y")  %Calculate carry bit
    end for
%   ropw(rlsb+nbit,"cc","")     %Write final carry bit (optional)
end add
```

Multiple additions can be performed by broadcasting the same instruction sequence to multiple PEs with data arranged as shown in Figure 4.2.

We see from the algorithm that an n-bit three-operand addition with no carryout requires $4n + 1$ PE cycles. A two operand addition (R+=A) can be performed in $3n + 1$ cycles by

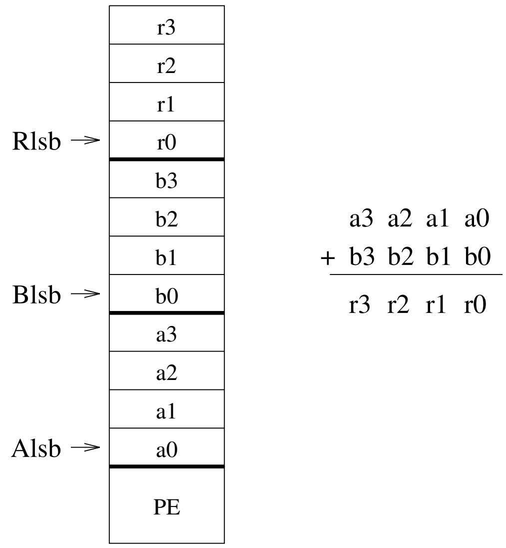


FIGURE 4.1 Data arrangement for a single addition

	r73	r63	r53	r43	r33	r23	r13	r03
	r72	r62	r52	r42	r32	r22	r12	r02
	r71	r61	r51	r41	r31	r21	r11	r01
Rlsb →	r70	r60	r50	r40	r30	r20	r10	r00
	b73	b63	b53	b43	b33	b23	b13	b03
	b72	b62	b52	b42	b32	b22	b12	b02
	b71	b61	b51	b41	b31	b21	b11	b01
Blsb →	b70	b60	b50	b40	b30	b20	b10	b00
	a73	a63	a53	a43	a33	a23	a13	a03
	a72	a62	a52	a42	a32	a22	a12	a02
	a71	a61	a51	a41	a31	a21	a11	a01
Alsb →	a70	a60	a50	a40	a30	a20	a10	a00
	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0

FIGURE 4.2 Data arrangement for multiple additions

using a Read Operate Writeback when calculating the sum bit. The longest a C•RAM would take to perform a three operand add would be:

$$t_{\substack{\text{n-bit addition}\\ \text{worst case}}} = (4n+1)50\text{ns} \tag{4.3}$$

where every cycle strobes both a row and column address (non page-mode). The C•RAM would perform best if every sixteenth cycle was a row access and all others were fast page mode accesses, in which case the add time would be:

$$\begin{aligned} t_{\substack{\text{n-bit addition}\\ \text{best case}}} &= \left(\frac{1}{16}\right)(4n+1)50\text{ns} + \left(\frac{15}{16}\right)(4n+1)15\text{ns} \\ &= (4n+1)\frac{275}{16}\text{ns} \end{aligned} \tag{4.4}$$

In calculating the number of n-bit additions completed per second on a single C•RAM, we must remember that 1024 are taking place simultaneously (1 addition per PE).

$$\begin{aligned} \#\text{additions}_{\text{worst case}} &= \frac{1024\frac{\text{additions}}{\text{chip}}}{(4n+1)50\text{ns}} \\ &= \frac{20.4}{(4n+1)}\frac{\text{billion additions}}{\text{second}} \text{ per chip} \end{aligned} \tag{4.5}$$

$$\begin{aligned} \#\text{additions}_{\text{best case}} &= \frac{1024\frac{\text{additions}}{\text{chip}}}{(4n+1)\frac{275}{16}\text{ns}} \\ &= \frac{59.6}{(4n+1)}\frac{\text{billion additions}}{\text{second}} \text{ per chip} \end{aligned} \tag{4.6}$$

A summary of addition performance is given in the table below.

TABLE 4.1 Single chip addition/subtraction performance

	Worst case	Best case
4-bit	1.2 billion/s	3.5 billion/s
8-bit	620 million/s	1.8 billion/s
16-bit	315 million/s	917 million/s
32-bit	158 million/s	462 million/s

4.2.2 Subtraction

Subtraction is a variation of addition with the same number of cycles. Twos complement notation, in which an integer is negated by inverting its bits and adding 1, is used. A subtraction (or negative addition) of R=A-B is therefore performed by adding the bits of A to the inverted bits of B after first installing a 1 in the carry register. Inverting the bits of B does not require an extra cycle: we just alter the opcodes to calculate difference and borrow, rather than sum and carry.

```
procedure subtr(alsb,blsb,rlsb,nbit       :int)
    %Performs an n-bit subtract R=A-B

    rop(0,"ff","y")                     %Set the carry bit to 1
    for bit:0..nbit-1
        rop(alsb+bit,"aa","x")          %Load the 1st bit
        rop(blsb+bit,"69","")           %Calculate sum
        wr(rlsb+bit)
        rop(blsb+bit,"d4","y")          %Calculate carry bit
    end for
%   ropw(rlsb+nbit,"cc","")             %Write final carry bit (optional)
end subtr
```

One can see from the algorithm that an n-bit subtraction requires the same number of cycles as addition, so their performance figures are identical.

4.2.3 Multiplication

Binary multiplication is a series of conditional adds that depends upon the value of individual bits of the multiplier, as illustrated in Figure 4.3. The Write-Enable register is used to enable and disable additions during each iteration, and the procedure "add2op" is two operand addition (R+=A) which uses a read-operate-writeback during the sum calculation.

```
procedure mult(alsb,blsb,rlsb,nbit          :int)
    %Performs an nbit multiply R=A*B
    rop(rlsb,"ff","w")

    for bit:0..(2*nbit)-1
        ropw(rlsb+bit,"00","")             %Set inital partial product to 0
    end for

    for bit:0..(nbit-1)
        rop(blsb+bit,"aa","w")             %If (blsb+bit)=1 then
            add2op(rlsb+bit,alsb,nbit)     % add to partial product
        %End if
    end for
    rop(rlsb,"ff","w")
end mult
```

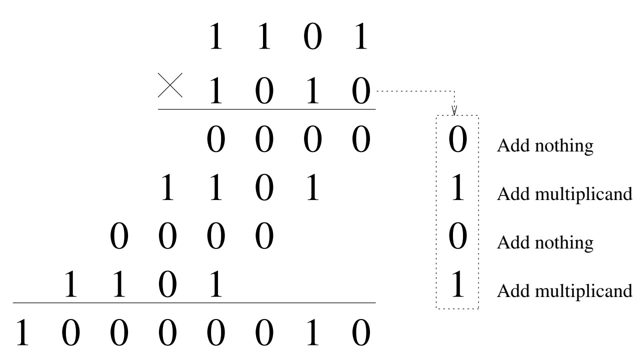


FIGURE 4.3 Binary multiplication example

The cycle tally for an n-bit multiplication is shown below:

$$\mathrm{add2op}_{\mathrm{n}} = 3n + 2 \tag{4.7}$$

$$\begin{aligned}\mathrm{multiplication}_{\mathrm{n}} &= 2 + 2n(1) + n(1 + \mathrm{add2op}_{\mathrm{n}}) \\ &= 3n^2 + 5n + 2\end{aligned} \tag{4.8}$$

We can calculate the multiplication performance of a single C•RAM by employing the same derivation procedure used for addition:

$$\#\mathrm{multiplications}_{\mathrm{worst\ case}} = \frac{20.4}{(3n^2 + 5n + 2)} \frac{\text{billion multiplies}}{\text{second}} \text{ per chip} \tag{4.9}$$

$$\#\mathrm{multiplications}_{\mathrm{best\ case}} = \frac{59.6}{(3n^2 + 5n + 2)} \frac{\text{billion multiplies}}{\text{second}} \text{ per chip} \tag{4.10}$$

A summary of C•RAM's multiplication performance is given in the table below.

TABLE 4.2 Single chip multiplication performance

	Worst case	Best case
4-bit	293 million/s	851 million/s
8-bit	88 million/s	254 million/s
16-bit	24 million/s	70 million/s
32-bit	6.3 million/s	18.4 million/s

4.2.4 Division

Binary integer division is a series of subtractions and conditional restores, as shown in Figure 4.4 [Ham90]. The procedure “sub2op” is a subtraction variant which calculates R-=A , and the procedure “blank” fills rows with zeroes.

```
procedure divide (alsb,blsb         :int,
                  rl                :int,
                  nbit              :int)

    %Performs an n-bit divide of A/B
    var rlsb            :int := rl
    var qlsb            :int := rlsb+nbit+1

    copy(alsb,rlsb-nbit,nbit)              %Copy A
    blank(rlsb,rlsb+nbit)                  %Clear remainder bits
    blank(blsb+nbit,blsb+nbit)             %Clear top bit in B

    for bit:0..nbit-1
        rlsb -= 1
        qlsb -= 1

        rop(rlsb,”ff”,”w”)
        sub2op(rlsb,blsb,nbit+1)
        rop(rlsb+nbit,”55”,”x”)
        wr(qlsb)                           %Write not(rlsb+nbit) to qlsb

        rop(rlsb+nbit,”aa”,”w”)
        add2op(rlsb,blsb,nbit)             %Conditional restore
    end for
    rop(rlsb+nbit,”ff”,”w”)
end divide
```

Integer division’s conditional restore could be done in one of two ways. The (n+1)-bit number could be copied to scratch memory, and conditionally copied back later. Alternatively, as implemented in the algorithm, it could be done by conditionally adding what was previously subtracted. In the first case, 2 (n+1)-bit copy routines must take place at a total cost of 4n+4 cycles. In the second case, a (n+1)-bit increment must take place, at a cost of 3n+5 cycles. Using the second restore method, the cycle tally for n-bit division is derived in the following equations:

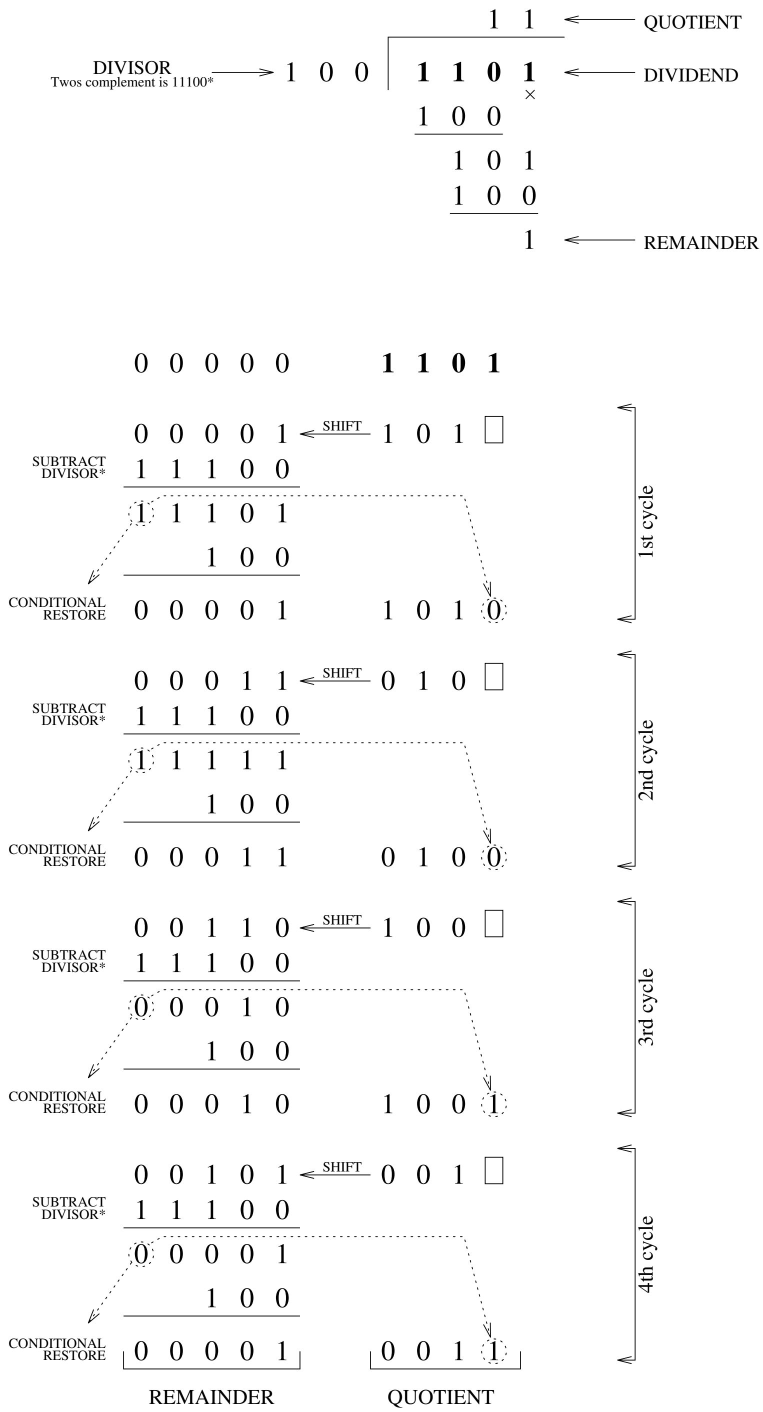


FIGURE 4.4 **Integer division example**

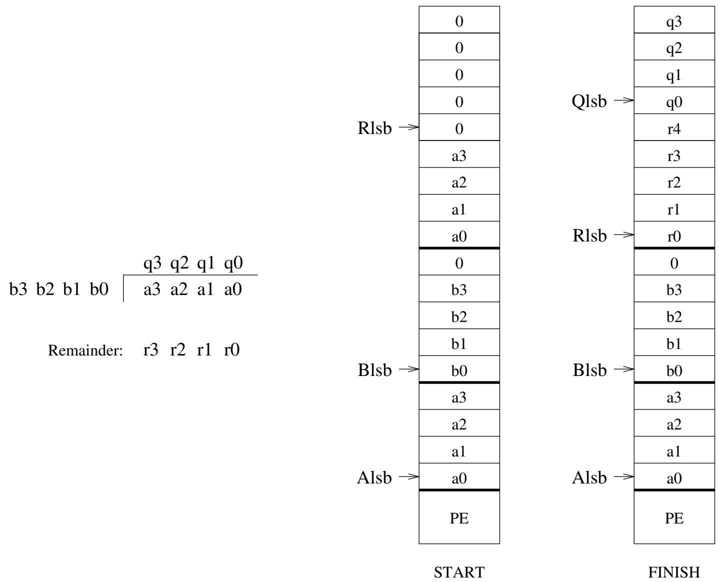


FIGURE 4.5 **Division implementation in C•RAM**

$$\text{copy}_n = 2n \tag{4.11}$$

$$\text{blank}_n = n \tag{4.12}$$

$$\text{sub2op}_n = 3n + 2 \tag{4.13}$$

$$\begin{aligned} \text{division}_n &= \text{copy}_n + \text{blank}_{n+1} + \text{blank}_1 + n(4 + \text{sub2op}_{n+1} + 1 + \text{add2op}_n) \\ &= 6n^2 + 15n + 2 \end{aligned} \tag{4.14}$$

From this equation, we derive a measure of performance:

$$\#\text{divisions}_{\text{worst case}} = \frac{20.4}{(6n^2 + 15n + 2)} \frac{\text{billion divides}}{\text{second}} \text{ per chip} \tag{4.15}$$

$$\#\text{divisions}_{\text{best case}} = \frac{59.6}{(6n^2 + 15n + 2)} \frac{\text{billion divides}}{\text{second}} \text{ per chip} \tag{4.16}$$

C•RAM's division performance is summarized in the table below.

TABLE 4.3 Single chip division performance

	Worst case	Best case
4-bit	129 million/s	377 million/s
8-bit	40 million/s	118 million/s
16-bit	11 million/s	34 million/s
32-bit	3.1 million/s	9.0 million/s

4.3 Database Application

C•RAM is well suited to database and data mining operations, where a number of tests have to be done against a set of data. Operations on elements in a large database are performed efficiently since we can employ the computing power of several thousand PEs simultaneously. SIMD processing power can be applied to data mining and financial analysis, among other things.

Data mining is a technique which analyzes historical performance data to show significant trends, patterns, or correlations. For example, sales data collected over a period of time can be analyzed to show distinct customer buying trends. Knowledge of these trends can be used to alter how products are promoted, priced, and purchased. Financial analysis includes real-time calculation of portfolio values, risks, and future value estimations.

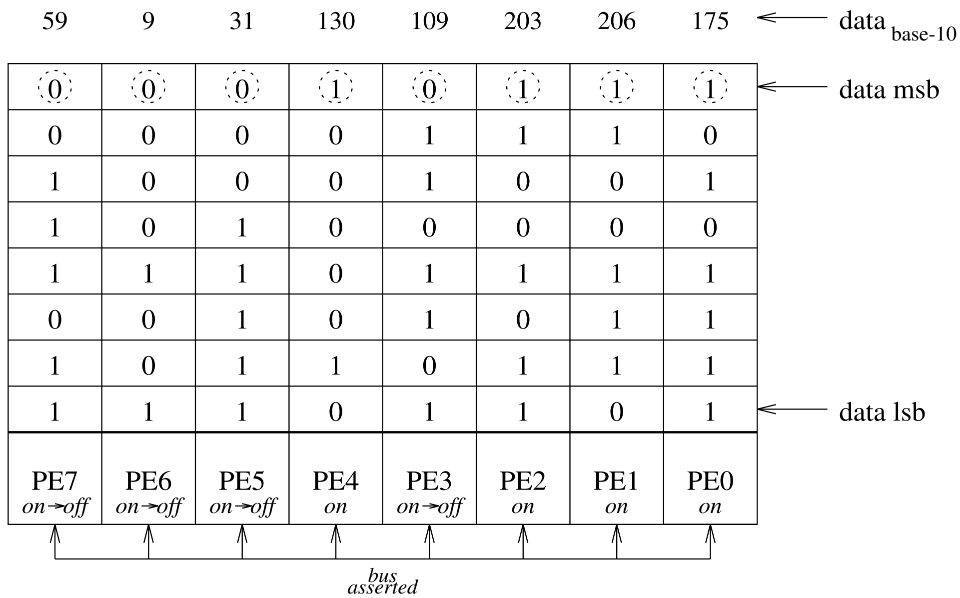
Cost functions are applied to data elements (or sets of data elements) on a PE-per-element basis. They generate information about the data set, which can then be compared to results of other data sets. For example, risks of several portfolios can be calculated in parallel by applying a predetermined risk function to the Processing Elements. The results are then manipulated by searching for the largest or smallest value, or sorting them from smallest to largest. Searches for largest and smallest elements are performed by using the broadcast bus for global data comparisons, and sorts are performed by having the PEs compare and exchange data with their neighbours.

4.3.1 Searching

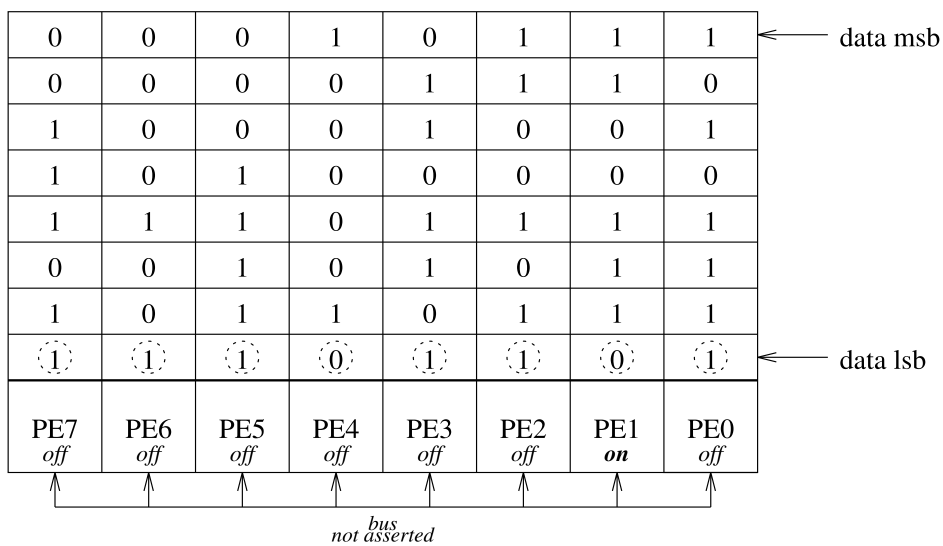
Several types of searches can be performed with SIMD processors: key searches, largest element searches, and smallest element searches. Key searches are performed by sequentially applying logical "and" operations to bits of the data elements.

The broadcast bus is a useful tool for searches. It can be used for global compares, where the largest (or the smallest) bit value in the PE array is broadcast globally, and compared by each PE to its own bit. For example, finding the largest element in an array is achieved by first looking at every number's most significant bit (msb), as illustrated in Figure 4.6a. If any elements have an msb of 1, their PEs will assert the bus. If the bus is asserted, the PEs with a 0 do not have the largest element, and exclude themselves from the search by logically turning themselves off. If all msb values are 0, the bus is not asserted and the comparison proceeds.

These broadcast & compare steps are performed for each bit from the msb to the lsb, during which time various PEs will logically turn themselves off. After the last iteration, the PE (or PEs) with the largest value will remain logically on, as illustrated in Figure 4.6b. Note that the bus has not been asserted during the final iteration because all PEs with a 1 have been logically turned off.



(a)



(b)

FIGURE 4.6 **Search for greatest element (a) first iteration (b) last iteration**

Simulator code for finding the largest element is shown below:

```
procedure findlargest (alsb,nbit           :int)
    %Finds the largest element in an array. It uses the bus transceiver
    %to compare bit-by-bit the PE’s local value versus the global value.
    %A “1” in the Y-register indicates that a PE is still “in the running”,
    %or “on”, otherwise the PE is “off”. The PE (or PEs) still “on” at the
    %end of execution has the greatest element.

    rop(alsb+nbit,”ff”,”y”)              %Write 1 to Y-Reg of all PEs
                                         % (i.e. turn them all “on”)
    for decreasing bit:(nbit-1)..0
        rop(alsb+bit,”77”,”tx”)          %Calculate a 0 if Abit is a 1, and PE is
                                         % still “on”. Broadcast and write to X-Reg.
        rop(alsb+bit,”48”,”y”)           %If PE is still “on”, compare broadcasted
                                         % bit (X) to Abit (M). Determine if PE
                                         % should switch “off”, and write to Y-Reg.

    end for
end findlargest
```

There is confusing logic in this algorithm that requires some explanation. We recall from Section 3.2.4 that the Bus Transceiver is asserted when *at least one PE* has an ALU result of 0. Since we want to know if *at least one PE* has encountered a 1 during each iteration, those which encounter a 1 broadcast a 0 on the bus. PEs which encounter a 0 (or are logically turned off) broadcast a 1.

Assuming we have one PE per element, the cycle tally for a largest element search in an array of M n-bit elements is:

$$\text{largest}_{M,n} = 2n + 1 \tag{4.17}$$

At the end of the search, the on/off bits are held in the PEs’ Y-registers (on=1, off =0). If we want to know which particular PE has the largest element, we could shift the Y-register values out sequentially at a cost of M shift-rights, or write the Y-register bits to scratch memory and look for the 1 in DRAM mode.

Searching for the largest of M n-bit unsorted elements with a single processor requires an O(M) search of n-bit numbers, which is equivalent to a single iteration of bubble sort. The parallel algorithm on C•RAM has an O(1) search followed by (at worst) M single-bit shifts.

A search for the smallest element in an array has an algorithm similar to this one, but uses different opcodes to favour PEs that find 0s instead of 1s.

4.3.2 Sorting

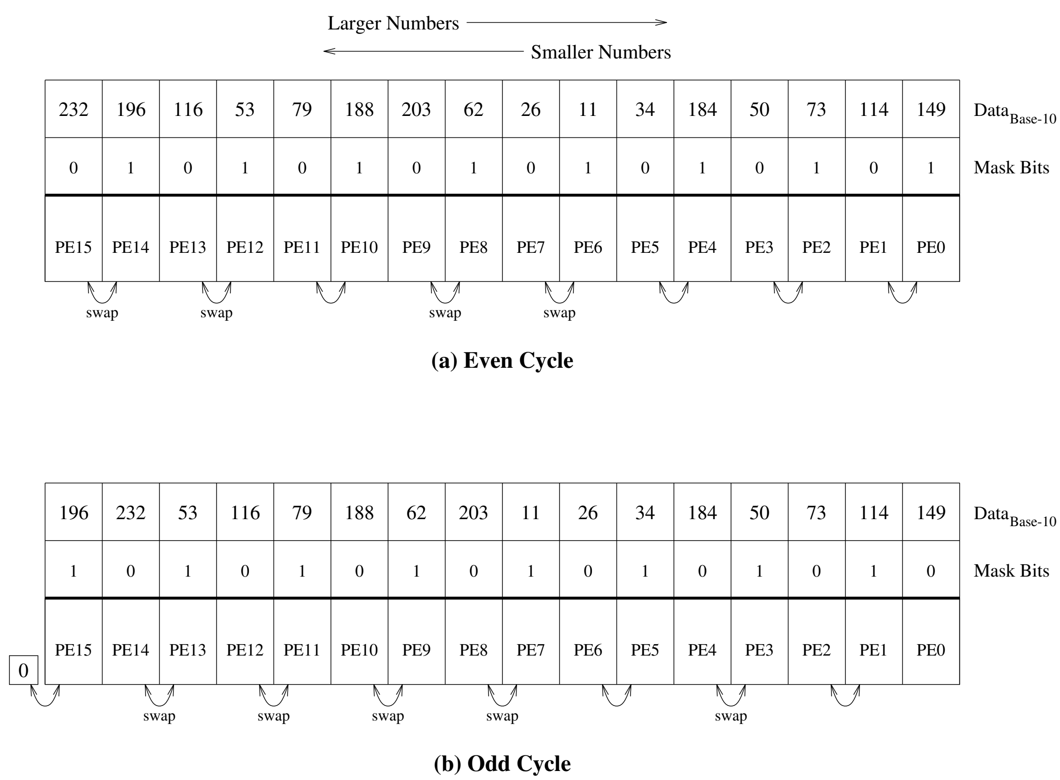


FIGURE 4.7 Parallel sort example

Parallel sorts are difficult to do with optimal efficiency due to the limited inter-PE communications. The sort algorithm presented in this section uses the inter-PE shuffle shown in Figure 4.7: PEs swap data with their left and right neighbours in alternate cycles until the array is sorted.

```
procedure sort(alsb,nbit        :int)

    %MEMORY ROWS:
    const lgn := 0                   %local greater than neighbour (lgn) row
    const mask:= 1                   %mask bits row

    for pe:0..npe
        ropw(lgn,"ff","w")

        %Determine if neighbours have to switch (calculate lgn)
        for bit:0..nbit-1
            rop(alsb+bit,"aa","xr")
            ropw(lgn,"b2","")
        end for

        rop(lgn,"55","x")            %Load /lgn into X-register
        rop(mask,"a0","yl")          %Calculate (/lgn)and(mask)
                                     % write to YReg and ShiftLeft
        rop(mask,"fc","w")           %Write to WE registers

        %Swap data between neighbours
        %(WE-register will stop unwanted swaps from taking place)
        for bit:0..nbit-1
            rop(alsb+bit,"aa","lr")  %Load bit ... shift left and right
            rop(mask,"d8","")        %If mask is "1" take Y-reg value
                                     %if mask is "0" take X-reg value
            wr(alsb+bit)
        end for

        rop(mask,"ff","w")
        ropw(mask,"55","")           %Invert mask bits
     end for
end sort
```

The number of cycles for a sort of M n-bit integers with an M-PE C•RAM is:

$$\mathrm{sort}_{\mathrm{M,n}} = (5n + 6)M \tag{4.18}$$

The complexity of the algorithm is O(M) to sort M elements, which means the sort time grows linearly with the number of elements. Quicksort, one of the most efficient uniprocessor sorting algorithms, has complexity O(M•$\log_2$M). At O(M), this parallelized bubble sort is the quickest technique available for a linear array of processors since a value

	PE15															PE0
START (reverse ordered)	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	15	13	14	11	12	9	10	7	8	5	6	3	4	1	2	0
	13	15	11	14	9	12	7	10	5	8	3	6	1	4	0	2
	13	11	15	9	14	7	12	5	10	3	8	1	6	0	4	2
	11	13	9	15	7	14	5	12	3	10	1	8	0	6	2	4
	11	9	13	7	15	5	14	3	12	1	10	0	8	2	6	4
	9	11	7	13	5	15	3	14	1	12	0	10	2	8	4	6
16 iterations	9	7	11	5	13	3	15	1	14	0	12	2	10	4	8	6
	7	9	5	11	3	13	1	15	0	14	2	12	4	10	6	8
	7	5	9	3	11	1	13	0	15	2	14	4	12	6	10	8
	5	7	3	9	1	11	0	13	2	15	4	14	6	12	8	10
	5	3	7	1	9	0	11	2	13	4	15	6	14	8	12	10
	3	5	1	7	0	9	2	11	4	13	6	15	8	14	10	12
	3	1	5	0	7	2	9	4	11	6	13	8	15	10	14	12
	1	3	0	5	2	7	4	9	6	11	8	13	10	15	12	14
	1	0	3	2	5	4	7	6	9	8	11	10	13	12	15	14
FINISH (forward ordered)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

FIGURE 4.8 **Dataflow of a 16 element parallel sort**

might have to single step end-to-end. The data flow of a worst case sort (where the list of elements is initially reverse ordered) is shown in Figure 4.8.

It is reasonable to assume that $\frac{15}{16}$ memory accesses made during a sort would be page mode, since each PE is accesses only a single integer and a row of mask bits throughout the operation. The time to complete the M element, n-bit integer sort is therefore:

$$t_{sort} = \left(\frac{1}{16}\right)(5n+6)M \cdot 50\text{ns} + \left(\frac{15}{16}\right)(5n+6)M \cdot 15\text{ns} \qquad \textbf{(4.19)}$$

A 1024-PE C•RAM can complete a sort of 1024 32-bit integers in 2.9ms. If one accounts for loading and unloading all 32,768 bits before and after the sort, the whole process takes

3.1ms. By comparison, a SPARCstation 5 running the quicksort algorithm takes an average 76.5ms to sort 1024 random 32-bit numbers.

Even with many more data elements than PEs, a large sort can be done with reasonable efficiency. If I integers are to be sorted, M-PEs are available and $I \gg M$, then one can have the available PEs perform $\frac{I}{M}$ sorts, and let the host perform a linear mergesort on the $\frac{I}{M}$ sorted subarrays.

4.4 Image Processing Application

Real-time image processing is a good application for C•RAM in many respects. SIMD processors work well with data sets containing large vectors, and video images are represented by large 2-dimensional pixel arrays. It is also best to have locality of reference in the data for cheap inter-PE communication overheads. Video is again appealing since most operations are performed on groups of neighbouring pixels. Most people would likely want C•RAM's real-time image processing for multimedia, games, and television - three areas with potentially large markets.

Compression is a frequently used image processing technique since the volume of image data generated by video sequences is often much too large for efficient storage or transmission.

Consider for example a 5 minute black-and-white video sequence on a screen with 512x512 pixels, and an 8 bit-per-pixel resolution. The display rate is 30 frames per second. With no compression, the sequence would occupy 2.4 Gigabytes storage space, and would require 32 minutes to transfer across an Ethernet connection with a 10 Megabit per second transfer rate.

A compression technique is required to keep video sequences at a reasonable size for storage and transmission. Fast compression and decompression techniques are desired so that the image can be displayed at the receiver in real time, which means the compressed version must be small enough to be received and decompressed in less time than its duration. For an audience to watch a program broadcast live through a computer network, real time image processing must be available.

Two video compression standards are currently in use. MPEG-1 is used for low quality, low bit-rate applications, such as videoconferencing and multimedia. MPEG-2 is used for higher quality and higher bit rate applications, like television broadcasts. Video sequences have a large amount of spatial and temporal redundancy and therefore are compressed as groups of pictures rather than as individual ones. In MPEG-2, three types of frames are generated: Intra-Coded (I) frames, Predictive-Coded (P) frames, and Bidirectional Predictive-Coded (B) frames.

Each image is first divided into 8x8 pixel blocks. MPEG-2 is typically used to compress images with 720x480 pixels, but for simplicity of explanation we use an image size of 512x512 pixels. As shown in Figure 4.9a, it has 4096 8x8 blocks, and can be processed by four C•RAMs on a block-per-PE basis.

An I-frame contains data for a full image. The 8x8 pixel blocks are coded with a Discrete Cosine Transform, and are ready for transmission or storage after further encoding techniques are performed. The DCT requires 256 multiplications and 416 additions, which can be performed by a single PE in 3ms. All PEs working in parallel can perform the 4096 DCTs in the same amount of time.

Sending just I-frames would give a good quality picture to the receiver, but would waste bandwidth. Instead, smaller P-frames and B-frames are generated by estimating optical

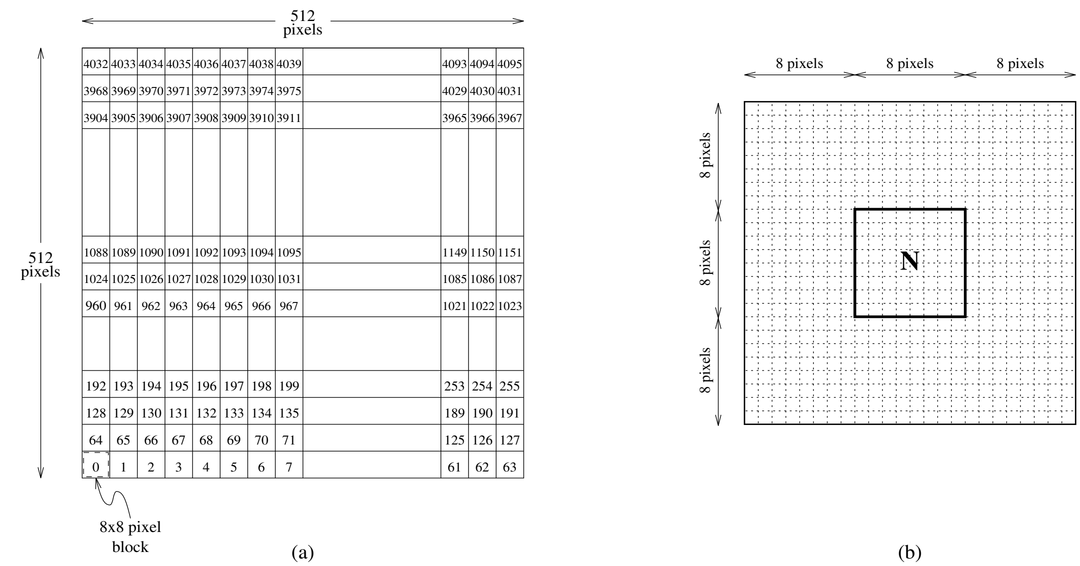


FIGURE 4.9 (a) 512x512 pixel^2 image divided into 8x8 pixel^2 blocks (b)block search space for motion estimation

flow of the 8x8 blocks between images This motion estimation is performed by searching the surrounding 24x24 pixel area of the next or previous image for a close pixel match.

As illustrated in Figure 4.10, P-frames are generated by estimating block motion between the previous I-frame and the current image. B-frames are generated by first calculating forward motion vectors between the previous I- or P-frame and the current image, then calculating backward motion vectors between the current image and next I or P frame, and interpolating the results.

The motion estimation technique used in MPEG-2 compression generates a motion vector for each block of 8x8 pixels. As illustrated in Figure 4.9, an image with 512x512 pixels is divided into 4096 blocks, and the motion of each one is estimated by calculating a vector that best describes the block's movement between frames. It doesn't matter how the vectors are generated. One can trade computation time for compression quality: poor motion estimation will yield larger residuals.

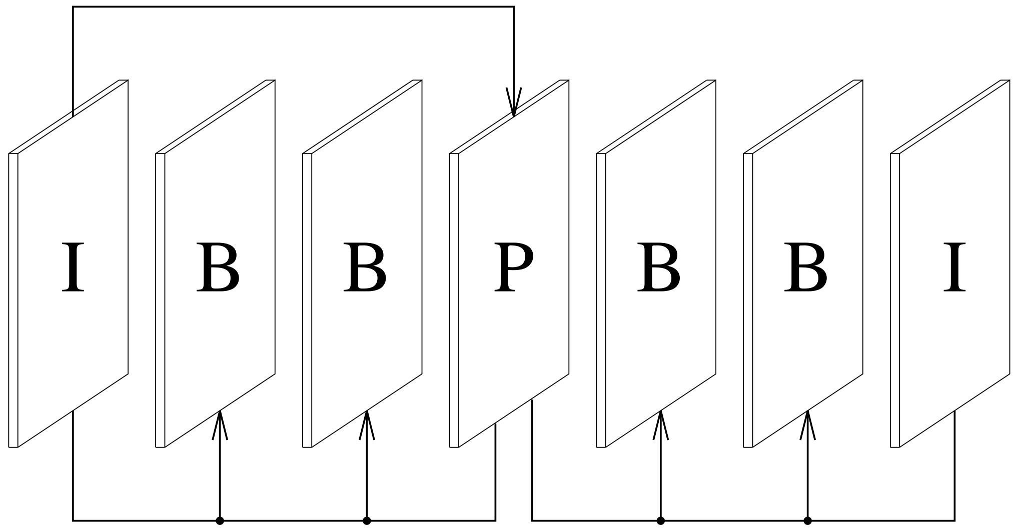


FIGURE 4.10 **MPEG-2 frame transmission sequence and generation dependencies**

4.4.1 Memory Mapping

The first step of motion estimation on C•RAM is the mapping of blocks to PEs. In the case of our 512x512 image, we assign one block to each of the 4096 PEs in four C•RAM chips, as shown in Figure 4.11. We need data from at least two consecutive images in the memory at any time to perform the operation. Since the PEs have 16kb local memories, and each 8x8 block occupies 512b storage space, each PE can hold 32 blocks with no colour information or working storage.

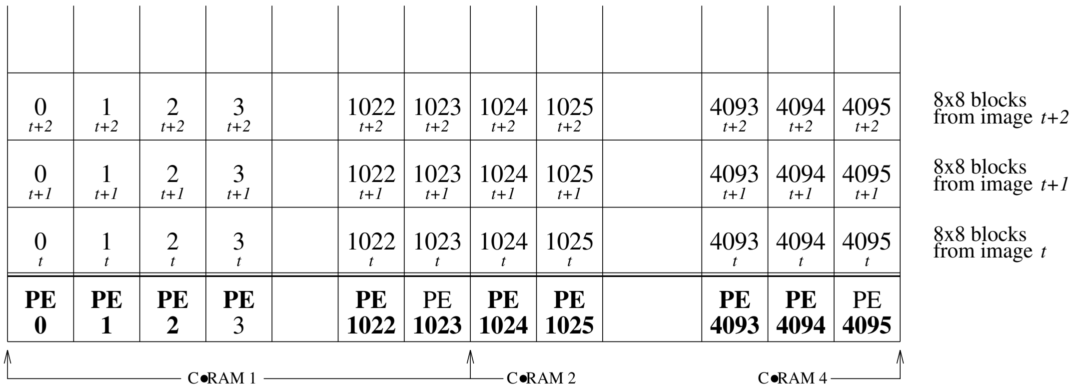


FIGURE 4.11 Memory mapping pixel blocks to PEs

The image is 64 blocks wide (Figure 4.9a), so blocks above and below one another are numbered 64 apart. With the selected memory mapping, vertically adjoining blocks are 64 PEs apart: the block above is 64 PEs to the right, and the block below is 64 PEs to the left.

For the block comparisons, we arrange data in the memory as shown in Figure 4.12. Blocks above and below the local one from image *t+1* are shifted from PEs 64 positions to the left and right. Once this operation is complete, comparison data are in the memory of the PE or its neighbour.

4.4.2 Decimation

One may wish to reduce the number of pixel comparisons required in motion estimation by decimating, or reducing the number of pixels in each image. Decimation by a factor of M replaces a group of M^2 pixels with a single pixel whose value is their average.

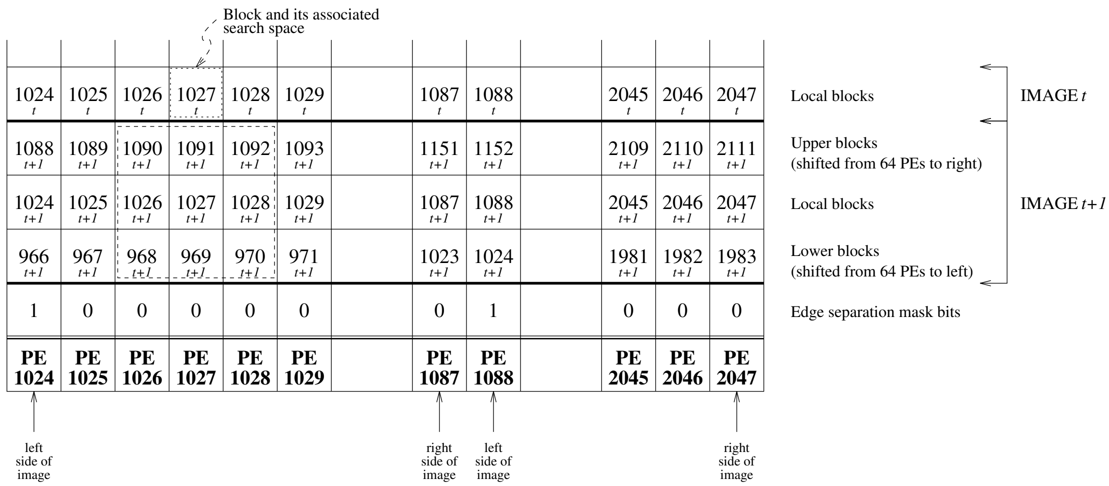


FIGURE 4.12 Data arrangement for block comparisons

The average value is obtained by first blanking a running sum, then adding the M^2 pixel values to it. The division is a shift, or omission of least significant bits in the running sum, if we assume M^2 is a power of 2. The bit length of the running sum is b:

$$b = 8 + 2 \cdot \log_2 M \tag{4.20}$$

The number of PE cycles for an M-decimation is therefore:

$$\begin{aligned} \text{decimation}_{\text{M}} &= \frac{64}{M^2}(\text{blank}_b + M^2 \text{add2op}_b) \\ &= 1600 + 384\log_2 M + 128\frac{\log_2 M}{M^2} + \frac{512}{M^2} \end{aligned} \tag{4.21}$$

Decimation is O(log M), and ranges from 2144 cycles (for M=2) to 2766 cycles (for M=8). It is a trivial procedure which can be performed essentially for free in comparison to error calculation.

4.4.3 Block Comparisons

Looking back at Figure 4.9b, we see that the 8x8 block can be moved 8 pixels left, right, up, and down to a total of $(2 \cdot 8 + 1)^2$, or 289 different locations.

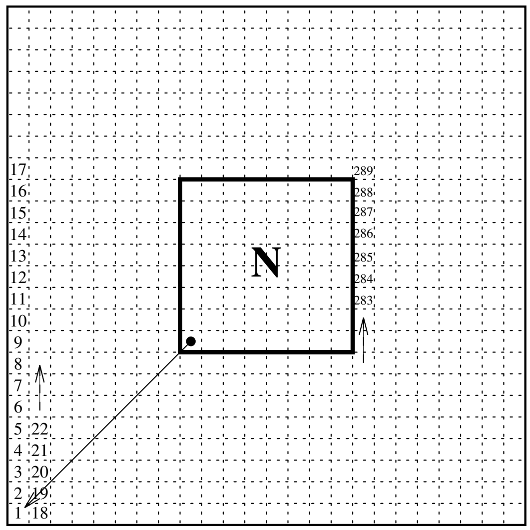


FIGURE 4.13 **Vector indexing format for block comparisons**

In the undecimated case, a block comparison is the accumulation of 8^2 quantitative pixel comparisons. Pixels are compared by first subtracting their 8-bit values, then conditionally negating the result to obtain a positive difference. This absolute difference is added to a 14-bit running sum of all 8^2 pixel differences.

Once all the 8^2 pixel comparisons are complete, the total difference is compared to the smallest block difference encountered so far in the search space. If the one just completed is smaller, then its value is copied as the new smallest value, and its vector index number (as shown in Figure 4.13) is recorded.

The C•RAM simulator code for a parallel motion estimation calculation is shown below:

```
procedure motion_estimation
    %Motion estimation is performed on blocks with size 8x8 in a
    %macroblock sized 24x24. Each pixel has 8-bit resolution.
    %When finished execution, each PE will have:
    % smallestlsb       Value of closest block match
    % smidxlsb          Closest block match vector index

    var smallestlsb         :int        %Closest block match
    var smidxlsb            :int        %Closest block match vector index
    var diffsumlsb          :int        %Accum. difference of 64 pixel values
    var difflsb             :int        %Difference of 2 pixels
    var vidx                :int:=0     %Vector index

    var ref_block           :array 0..63 of int     %Local block (64 lsbs)
    var comp_block          :array 0..63 of int     %Comparison block

    for xshift: -8..8                   %17 horizontal positions
        for yshift: -8..8               %17 vertical positions
            vidx += 1

            %14 bit sum <-> 64 8-bit adds
            %Clear the 14-bit running total
            blank(diffsumlsb,diffsumlsb+13)

            for pixel:0..63
                %Subtract 2 pixel values
                %and use subtr’s write-carry-out option
                subtr(ref_block(pixel),comp_block(pixel),difflsb,8)

                %Conditionally negate the difference (if diffmsb is “1”)
                rop(difflsb+8,”aa”,”w”)
                negate(difflsb,9)

                %Add the pixel difference to the running total
                rop(00,”ff”,”w”)
                add2op(diffsumlsb,difflsb,14)
            end for

            %Compare the block difference to smallest.
            %If it’s smaller, write it and its vector index.
            compare(smallestlsb,diffsumlsb,14)
            rop(00,”f0”,”w”)
            copy(diffsumlsb,smallest,14)
            writeval(vidx,smallestidx,9)
        end for
    end for
end motion_estimation
```

We can take a tally of the cycles performed in this motion estimation calculation, and estimate the amount of time C•RAM will require to complete it. Microroutines used in the algorithm are presented in the Appendix.

$$\text{negate}_n = 2n + 1 \tag{4.22}$$

$$\text{compare}_n = 2n \tag{4.23}$$

$$\text{writeval}_n = n \tag{4.24}$$

$$\begin{aligned} \underset{\text{estimation}_{8,8,8}}{\text{motion}} &= 17^2(\text{blank}_{14} + 8^2(\text{subtr}_8 + \text{negate}_9 + \text{add2op}_{14} + 2) + \text{compare}_{14} + \text{copy}_{14} + \text{writeval}_9 + 1) \\ &= 1,817,232 \text{ cycles} \end{aligned} \tag{4.25}$$

A single iteration of motion estimation requires 1.82 million cycles, during which C•RAM or group of C•RAMs with N-PEs estimate motion vectors for N 8x8 blocks. If every PE cycle is 50ns long, the iteration will take 90.1 milliseconds. If page mode is used to its fullest extent, the iteration will take 31.2 milliseconds. Depending upon how the block data are stored, 11.1 to 32.0 images per second can be motion estimated.

An 8x8 block size, 24x24 search space, and 8-bit pixel resolution may be much more computation and accuracy than is required. It might not be necessary to search with fine granularity over such a large area: perhaps we can use fewer pixels or fewer bits per pixel. We have generated the cycle tally for a very specific method for motion estimation, and we need a more generic formula to decide which corner might be best to cut.

As shown in Figure 4.14, our new generic example uses blocks sized *b* pixels by *b* pixels, each with an *n*-bit representation. The block can be moved in the search space *s* pixels left, right, up, and down from the center position..

When one adds *b*, *n*-bit pixel differences together, the number of bits in the total block difference is *m,* represented as:

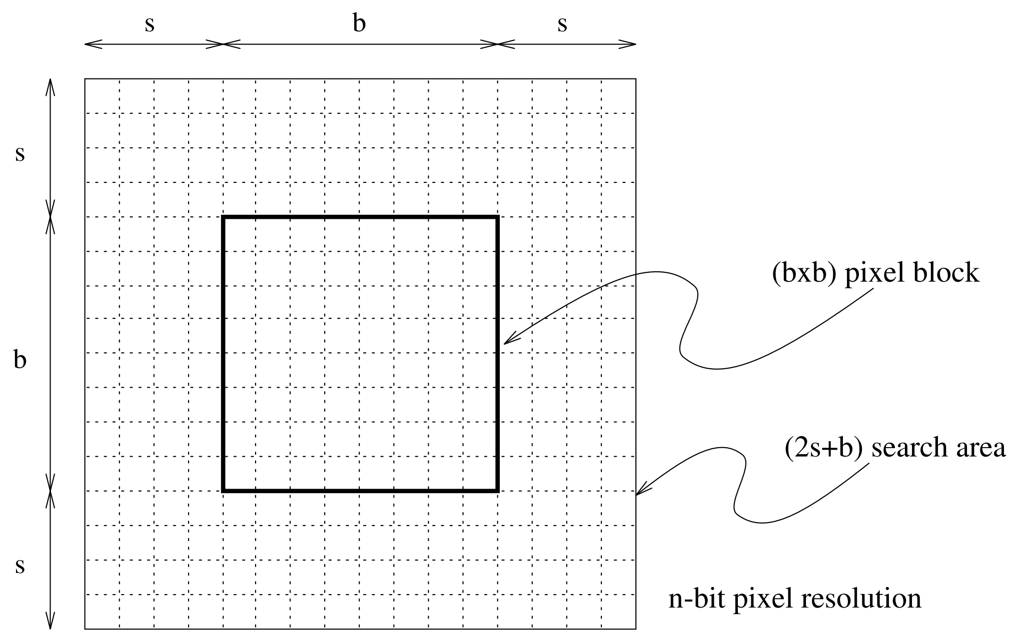


FIGURE 4.14 Generic motion estimation block and search space.

$$m = n + 2 \log_2 b \tag{4.26}$$

In our previous example, we added 64 8-bit pixel differences together, which yields a 14 -bit sum.

A block comparison index number (Figure 4.13) also has a variable number of bits. For example, in a search space of 6x6 and a block size of 2x2, there are 25 different positions for the block to be positioned. The binary representation of those 25 positions only needs to be 5 bits long, so we use the variable c to represent the bit-width of index numbers:

$$c = \lceil \log_2 (2s + 1)^2 \rceil \tag{4.27}$$

The motion estimation cycle tally for variables b, n, and s is:

$$\text{motion estimation}_{b,n,s} = (2s+1)^2(\text{blank}_m + b^2(\text{subtr}_n + \text{negate}_{n+1} + \text{add2op}_m + 2) + \text{compare}_m + \text{copy}_m + \text{writeval}_c + 1) \quad (4.28)$$

If we choose to make $s = b$ and work through the algebra, we get:

$$\text{motion estimation}_{b,n,b} = 24b^4n + 12b^4m + 28b^4 + 12b^3n + 6b^3m + 14b^3 + \text{lower order terms} \quad (4.29)$$

Terms with anything lower than b^4 can be dropped, and we substitute for *m*.

$$\begin{aligned} \text{motion estimation}_{b,n,b} &\cong b^4(24n + 12m + 28) \\ &= b^4(24n + 12 \cdot (n + 2\log_2 b) + 28) \\ &= b^4(36n + 24\log_2 b + 28) \end{aligned} \quad (4.30)$$

Motion estimation with $s = b$ relies heavily upon the block size *b*, and less heavily upon the pixel resolution *n*. For example, reducing the block size with decimation from 8x8 to 4x4 and maintaining an 8-bit pixel resolution would reduce the computation time by 94% to 4.7ms (worst case at 50ns per cycle). However, we would be only generating motion vectors in a 12x12 search space, and MPEG-2 requires that motion vectors be generated for a 24x24 search space.

Full motion estimation could be done in 2 stages:

1. Decimate images by 2, and perform motion estimation on 4x4 blocks with 12x12 search spaces. This step will generate a coarse-grained motion vector for each block.
2. After shifting pixel data, perform a fine grained motion estimation in the "area of interest", or the area pointed to by the motion vector from 1.

If the second stage is done with the original 8x8 blocks in a 10x10 area of interest and an 8-bit pixel resolution, it can be completed in 7.9ms (worst case). Assuming data shifts are negligible, the two steps combined take 12.6ms and can be performed at a rate of about 80 frames per second.

4.5 Summary

The page mode feature of DRAM is attractive for processing in memory since its fast data rate allows the PEs to be cycled faster. We saw in Section 4.2 that its is possible to increase performance 3 times by storing data column-wise in the local memories of PEs.

Some arithmetic microroutines were generated in this chapter with three timing constructs (Read Operate, Read Operate Writeback, and Write) to begin the compilation of a microroutine library. They were written to create benchmark performance estimates for integer arithmetic, and to simplify the coding and performance analysis of larger procedures which implement integer arithmetic.

To prove that C•RAM will have value to the average computer user, we showed its application to database operations and real-time image processing. The database operations we chose to analyze (searches and sorts) run at least $O(\log_2 M)$ more efficiently on C•RAM than with the most efficient single-processor algorithm. Discrete Cosine Transforms and motion estimation, the two most computationally intense components of MPEG-2 video compression, can be performed at greater than 30 frames per second.

The last two chapters have shown that an extension to a basic memory chip which increases its area and power consumption by about 15% can give significant and affordable computing power to desktop machines. In Table 4.4, we see that C•RAM has much less hardware than supercomputer SIMD machines of the 1980s, and its performance compares favourably.

SYSTEM	Year	# of PEs	local memory per PE	Physical Attributes	PERFORMANCE		
					8-bit addition	8-bit multiply	other
MPP **[Bat80]**	1980	16,384 (128x128)	1024b SRAM	Housed in large cabinet with 88 printed circuit boards and 2100 chips	6.5 billion per second	1.9 billion per second	Sorts 16,384 1024-bit numbers in 0.4s [Hor90].
GAPP II **[Hor90]**	1984	10,368 (144x72)	128b SRAM	Array consists of 144 chips	4.1 billion per second	0.4 billion per second	
Connection Machine **[Tan90]**	1986	65,536	65,536b	25,000 chips Weighs 1200 kg Dissipates 28kW 5 ft. x 5ft. x 5ft.	4 billion per second	3.3 billion per second	Sorts 65,536 32-bit numbers in 30ms
C•RAM **(16 chips)**	1997	16,384	16,384b DRAM	16 chips Chip temperature 64^{o}C Array dissipates 20.5W PEs integrated in memory	19 billion per second	4.7 billion per second	Sorts 16,384 1024-bit numbers in 1.4s Sorts 65,536 32-bit numbers in 187ms (64 C•RAMs)

TABLE 4.4 System comparisons

CHAPTER 5 Other Design Issues

In this chapter, we look at some further design issues relating to C•RAM that have not been discussed or implemented so far. The Processing Elements were designed for bit-serial computation, but the their reliance on memory bandwidth could be lowered and their performance increased with the addition of bit-parallel capabilities. A revised PE architecture is proposed, and its resulting performance is compared to the bit-serial performance figures generated in Chapter 4.

Again dealing with the memory bandwidth problem, we look at ways of improving the efficiency of our memory accesses through better management of memory banks. The Synchronous DRAM timing interface allows retrieval of data from one bank while data is simultaneously being prepared in another.

We have assumed to this point that the C•RAM will have a controller which provides an interface to the host, and generates timing sequences at proper intervals. The last section of this chapter gives a brief overview of C•RAM's system integration, describing the controller, compiler, and its location possibilities within a system.

5.1 Bit-Parallel Architecture

A bottleneck in our C•RAM design which affects performance is the relatively slow data rate provided by each PE's local memory. The PEs were designed for speed and can be cycled at 10ns, but their local memories can only provide data every 15ns at best (during a fast page mode cycle) and 50ns at worst (during a non page-mode cycle). Bit-parallel architecture can be used as a possible solution to this problem, since it performs multibit additions with fewer memory accesses.

The original PE performs calculations bit-serially: an addition is performed by loading two operand bits, then writing a result bit and computing a carry bit. The total number of memory accesses required in this operation depends upon the size of the largest operand. Bit-parallel Processing Elements in an n-bit Computing Unit (CU) load all bits of the first operand in one access, load all bits of the second operand in a second access, and write the entire result operand in the third access. As illustrated in Figure 5.1, full adder logic with a ripple carry is implemented in these "extended" PEs (ePEs) to facilitate the fast add.

A component with approximately 10 gates can be added to each PE in the IBM design to fulfil the extra full-bit adder hardware requirements. Figure 5.2 shows the location and functionality of this new addition block.

Logic required for the full add is shown below:

$$R = X \oplus M \text{ or } R{=}X \oplus Y \tag{5.1}$$

$$C_{out} = (X \cdot \bar{R}) + (C_{in} \cdot R) \tag{5.2}$$

$$SUM = (\overline{ADD} \cdot R) + (\overline{C_{in}} \cdot R) + (ADD \cdot C_{in} \cdot \bar{R}) \tag{5.3}$$

The ALU is used to compute the sum bit of X and Y (or X and M). The add block (when activated) calculates the sum of R and $Carry_{in}$. If we have the ALU compute the sum of X

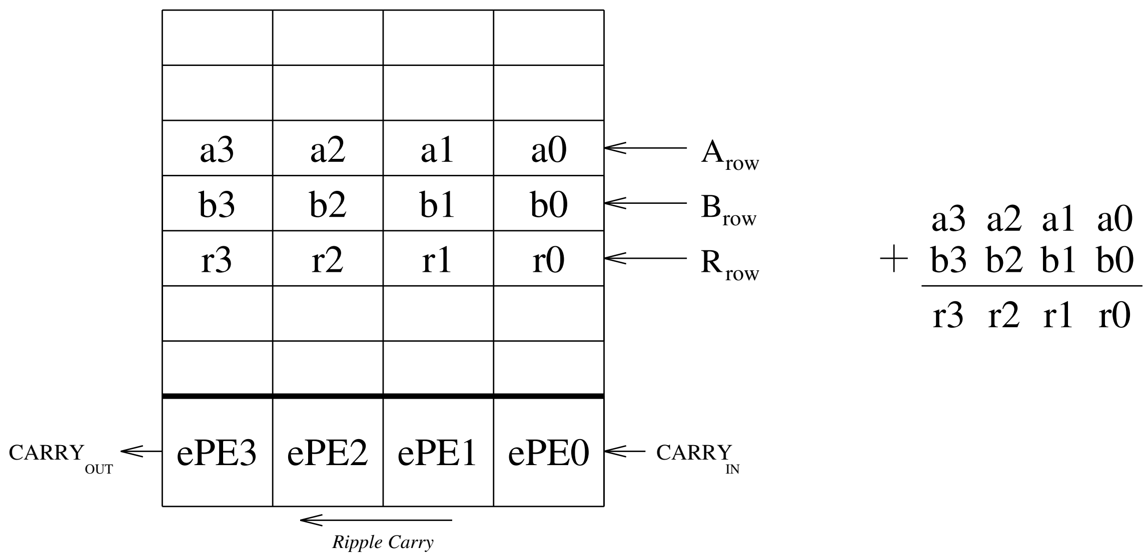
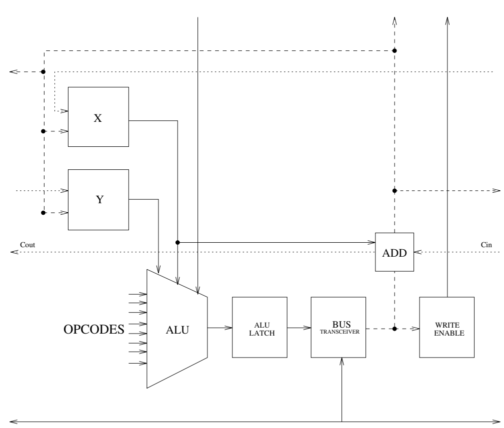
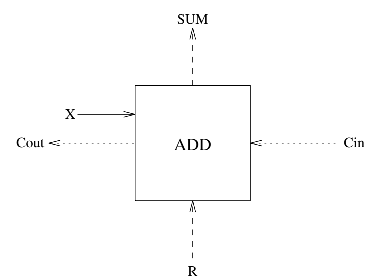


FIGURE 5.1 **Structure and functionality of bit-parallel computing**





ADD	Cin	X	R	Cout	SUM	
0	d	d	0	d	0	Regular Operation
0	d	d	1	d	1	
1	0	0	0	0	0	Addition
1	0	0	1	0	1	
1	0	1	0	1	0	
1	0	1	1	0	1	
1	1	0	0	0	1	
1	1	0	1	1	0	
1	1	1	0	1	1	
1	1	1	1	1	0	

FIGURE 5.2 **ADD block: location and functionality**

and Y (or X and M), then the add block (when activated) performs the rest. When not activated, the add block would pass the ALU result through unchanged.

5.1.1 Bit-Parallel Addition

Our simulator programming language is extended so that an “a” in the register set activates the ADD register. A bit-parallel add would contain the steps shown below:

```
procedure bp_add(arow,brow,rrow     :int)
    %Performs a bit parallel add R=A+B across PEs

    rop(arow,"aa","x")          %Read A, put in X-Reg
    rop(brow,"5a","ay")         %Read B, add, and put result in Y-Reg
    ropw(rrow,"f0","")          %Write result to R
end bp_add
```

Bit parallel addition with three operands requires only three cycles, but a single n-bit add requires the service of n-PEs. A 1024-PE chip can therefore do only $\frac{1024}{n}$ n-bit adds simultaneously at 3 cycles apiece. The cycle tally and computations times for this operation are shown below:

$$\mathrm{bpadd}_n = 3 \tag{5.4}$$

$$t_{\substack{\text{n-bit bpadd}\\ \text{worst case}}} = 3(50\text{ns}) = 150\text{ns} \tag{5.5}$$

$$t_{\substack{\text{n-bit bpadd}\\ \text{best case}}} = \left(\frac{1}{16}\right)(3)50\text{ns} + \left(\frac{15}{16}\right)(3)15\text{ns} = 51.6\text{ns} \tag{5.6}$$

Again, noting that only $\frac{1024}{n}$ additions can be performed simultaneously, we can derive the bit-parallel addition performance for C•RAM:

$$\#\mathrm{bpadds}_{\text{worst case}} = \frac{\left(\frac{1024}{n}\right)\frac{\text{bpadds}}{\text{chip}}}{150\text{ns}} = \left(\frac{6.8}{n}\right)\frac{\text{billion bpadds}}{\text{second}}\text{ per chip} \tag{5.7}$$

$$\#\mathrm{bpadds}_{\mathrm{best\ case}} = \frac{\left(\frac{1024}{n}\right)\frac{\mathrm{bpadds}}{\mathrm{chip}}}{51.6\mathrm{ns}} = \left(\frac{19.8}{n}\right)\frac{\mathrm{billion\ bpadds}}{\mathrm{second}}\ \text{per chip} \tag{5.8}$$

Specific bit-parallel add performance figures are shown in Table 5.1 (and the bit-serial addition performance values are in Table 4.1 on page 58). Bit-parallel has better performance than bit-serial since it makes fewer memory accesses per addition, and overlaps the sum/carry operations.

TABLE 5.1 Single chip bit-parallel addition performance

	Worst case	**Best case**
4-bit	1.7 billion/s	5.0 billion/s
8-bit	850 million/s	2.5 billion/s
16-bit	425 million/s	1.2 billion/s

5.1.2 Bit-Parallel Multiplication

We now look at a bit-parallel multiply, and its accompanying the hardware requirements to see how it compares overall to bit-serial multiplication.

As mentioned before in Section 4.2.3, a binary multiply involves a number of multiplicand shifts, and conditional adds that depend upon successive bits of the multiplier. For bit-parallel multiplication, the data in an 8-bit computing unit's memory is arranged as shown in Figure 5.3. The multiplicand (A) is installed in the X-register, and the multiplier (B) in the Y-register. In each iteration, we want to either enable or disable adds by installing a bit of the multiplier in all the WE-registers within the CU. Using mask bits to make the least-significant-bit-PE logically unique, we calculate the function:

$$\mathrm{ALU\_RESULT} = (\mathrm{YReg}_{\mathrm{bit}} \cdot \mathrm{mask}_{\mathrm{bit}}) + \overline{\mathrm{mask}_{\mathrm{bit}}} \tag{5.9}$$

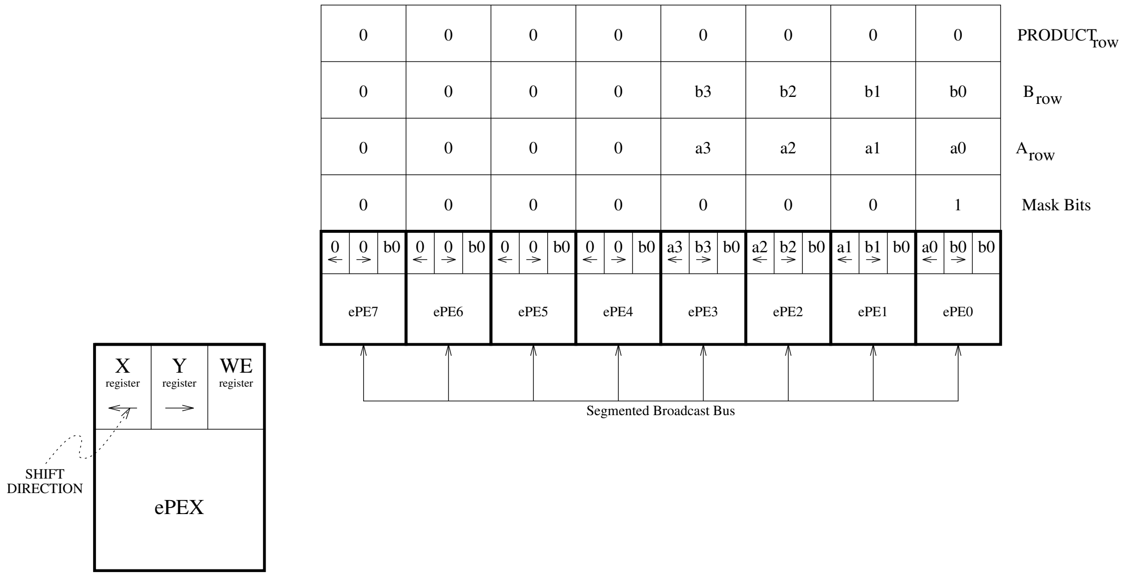


FIGURE 5.3 **Bit-parallel multiply setup**

which will cause the mask PE (ePE0) to calculate the B_{bit} residing in its Y-register, and all other PEs to calculate a 1. Enabling the bus transceiver "broadcasts" the masked B_{bit} to all PEs, and it can then be written to all WE-registers. Figure 5.3 shows the state of a CU after this step is performed in the first iteration. With the WE-registers set, we perform the two-operand addition of Product_{row} += XReg. We then shift the bits of A left, and the bits of B right. The mask bits will now be logically favouring bit b1, and the bits of A will be properly aligned beneath the partial product for the second iteration.

During shifts, the bits of A and B will cross borders into adjacent CUs. While shifting left, the only A bits that cross borders are zeroes, which is good since they will act as placeholders in the neighbouring multiplication. The nonzero bits of B will never reach the "1" mask bit of its right neighbour.

The algorithm to perform an n-bit bit-parallel multiplcation is shown below:

```
procedure bp_mult(arow,brow,mrow,prow,nbit  :int)
    %Performs an n-bit bit-parallel multiply R=A*B across PEs

    blank(prow,prow)            %Clear product
    rop(arow,"aa","x")          %Load A into X-register
    rop(brow,"aa","y")          %Load B into Y-register

    for bit:0..nbit-1
        rop(mrow,"dd","wt")     %Broadcast Bbit to all PEs' WE_Registers
        ropw(prow,"5a","a")     %Calculate partial product
        rop(mrow,"f0","l")      %Shift X-register (Abits) left
        rop(mrow,"cc","r")      %Shift Y-register (Bbits) right
    end for
end bp_mult
```

The cycle tally for a single multiplication is:

$$\text{bpmult}_{\text{n}} = (4n+3) \quad \textbf{(5.10)}$$

Bit parallel multiplication requires 2n-PEs to perform an n-bit multiplication, so a chip can perform $\frac{1024}{2n}$ simultaneously. The performance equations for bit parallel multiply are shown below:

$$\begin{aligned}\#\text{bpmult}_{\text{worst case}} &= \frac{\frac{1024}{2n}\frac{\text{bpmult}}{\text{chip}}}{(4n+3)50\text{ns}} \\ &= \left(\frac{10.2}{4n^2+3n}\right)\frac{\text{billion bpmult}}{\text{second}} \text{ per chip}\end{aligned} \quad \textbf{(5.11)}$$

$$\begin{aligned}\#\text{bpmult}_{\text{best case}} &= \frac{\frac{1024}{2n}\frac{\text{bpmult}}{\text{chip}}}{(4n+3)\frac{275}{16}\text{ns}} \\ &= \left(\frac{29.8}{4n^2+3n}\right)\frac{\text{billion bpmult}}{\text{second}} \text{ per chip}\end{aligned} \quad \textbf{(5.12)}$$

Bit-parallel multiplication performance figures are shown in Table 5.2 (bit-serial multiplication performance is listed in Table 4.2 on page 60).

TABLE 5.2 Single chip bit-parallel multiplication performance

	Worst case	Best case
4-bit	135 million/s	392 million/s
8-bit	36.6 million/s	106 million/s
16-bit	9.6 million/s	27.8 million/s

5.1.3 Hardware Choices

The bit-parallel C•RAM should be able to perform additions on many sizes of operands. An n-bit bit-parallel computation will require that the PEs be grouped into computing units so that the ripple carry will originate at the least significant end, and terminate at the most significant end. Borders must be defined between Computing Units, and two options exist: hardwired borders, or programmable borders.

Further choices need to be made when designing hardwired borders. Carry-in is useful for additions when tied low, but useless for subtraction unless tied high. The CU borders could be hardwired or programmable, but the carry-in should be programmable.

The broadcast bus needs to be segmented between CUs for multiplications to work. Again, it could be physically divided among CUs, but each bus segment would require its own control and voltage pullup.

The difficulty with hardwired borders is the lack of flexibility. A C•RAM with 8-bit hardwired CU borders can perform only 8-bit additions, so additions with fewer bits would still straddle 8 PEs. Additions with more bits would be forced to do awkward carry shifts across the borders. Multiplication presents a further difficulty since an n-bit multiply requires a 2n-bit CU, unless we are prepared to transport carry bits across borders by shifting, or calculate a sum of nxn products [Wak90]. The extra steps required to sidestep hardware limitations erode performance.

Making the borders programmable will ensure that every operation can be performed at near-optimal speeds. The disadvantage of adding extra registers to every PE is that it adds to chip size and complexity. Cojocaru [Coj95] designed both bit-serial and bit-parallel C•RAMs: his bit-parallel PE was double the size of the bit-serial PE, and the bit-parallel chip had twice the PE-to-memory area ratio.

5.1.4 Bit-Parallel Summary

We are designing C•RAM as a low-cost memory enhancement, and want the PEs to occupy a small percentage of the overall area. It was shown in Chapter 4 that a bit-serial architecture gives good performance with a small amount of hardware, but its performance figures rely heavily upon the memory's data rate. Bit parallel computation makes fewer memory accesses, but requires more hardware per PE, which increases the chip size and reduces manufacturing yield [Pri90]. Fast DRAM memories slow down bit-serial PEs 50% best case, but the architecture still offers maximum flexibility for minimum hardware.

The bit-parallel architecture has poor multiplication performance because half of its Processing Elements are performing placeholder additions during a multiply: a 2n-bit accumulator is being used to add n-bit values. Bit serial multiplications do not have this problem since they avoid performing any zero additions, and spend time only on significant computation. Bit-parallel processing gives superior performance to bit-serial processing when the memory accesses are slow relative to the fastest cycle time of the processor. Since the cycle times of the PE and its local memory are so closely matched in our Dynamic RAM C•RAM design, bit parallel computing is not really worth the extra silicon.

5.2 Synchronous DRAM Timing Interface

There is a fundamental problem with general Asynchronous DRAM interfaces like the one we have on the C•RAM. Every new row access requires a full sense and amplify of cell data, which is a time consuming process. In C•RAM or DRAM mode, the component requesting the row access must sit idle while the bits are prepared - even though half the sense amplifiers had been sitting idle during the previous access.

Memory "banks" are blocks of memory which are activated separately. The IBM memory chip has two banks, since half of its 32k sense amplifiers are activated during a single access. A higher data rate could be obtained with better management of the banks: while one is transferring data, the other could be sensing and amplifying a new row. In short, we would be preparing the PEs' cache ahead of time. Asynchronous DRAMs are not designed for this type of use, but could accomodate bank precharging with judicious alterations to the control circuitry [Kal96].

Synchronous DRAMs, on the other hand, allow interleaved bank accesses. While data is being received from one bank, another bank can be precharging a new row of data so that the data flow will be uninterrupted when a different row is accessed. Figure 5.4 shows the timing diagram of a read sequence on a generic Synchronous DRAM. Like the asynchronous version, there is a row address strobe and column address strobe signal, but the sequence is clocked. A "burst length" is preprogrammed on the chip to indicate how many groups of data from consecutive addresses should be provided with each access. The

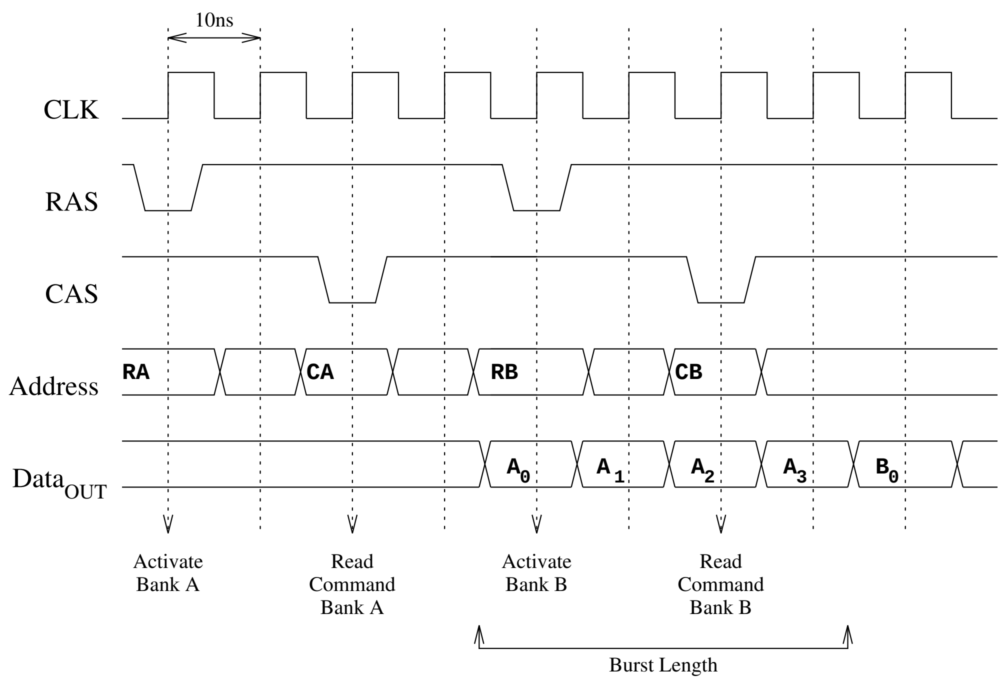


FIGURE 5.4 Synchronous DRAM timing interface (read)

“CAS latency”, also a programmed value, is the number of clock cycles after CAS that the data appears.

In the example shown, the burst length has been set to 4, and the CAS latency has been set to 2. Bank A is accessed first, and begins providing data two clock cycles after the column address is strobed. On the same clock pulse, bank B is activated and it starts precharging a row. When the burst of A data is complete, the burst of B data begins on the next clock cycle. Two different rows are accessed, and a constant data rate is achieved.

An interface similar to this one could be used to supply the PE with a consistently fast data rate from the memory. The PE’s fastest cycle time is 10ns, which means even at the page-mode cycle time of 15ns, the memory is 50% slower than the component it supplies data to. Any improvements to memory bandwidth will benefit performance.

5.3 C•RAM System Integration

We have not to this point discussed supporting hardware for the C•RAM, but have assumed that some component will be supplying it with instructions. Regular DRAM on a motherboard has a controller which queues memory access requests from the host and interfaces directly to the memory. As a main memory with SIMD processors, C•RAM would require an extended memory controller which functions like a regular DRAM controller but also accomodates parallel processing requests from the host.

The controller would have a writeable control store or microroutine memory for storing microroutine sequences similar to those presented in Chapter 4. Instructions from the host would therefore be microroutine calls with the address locations of one to three operands. The controller would translate these instructions into a series of timing sequences for the C•RAM. Peter Nyasulu at Carleton University has designed a controller for the 64 Processing Element bit-serial C•RAM designed by Christian Cojocaru. It is implemented on a XILINX field programmable gate array, and interfaces with the PCI local bus, as shown in Figure 5.5 [Nya97].

On a higher level, software abstraction is required so C•RAM can be accessed by a user with a standard programming interface like C, C++, or Java. Operands would be defined as parallel variables, and memory mapped automatically so that the user would not have to keep track of physical addresses. A compiler would translate code into controller instructions, complete with physical memory addresses and operand sizes. Implementation of a compiling technique which maps data to maximize the use of page mode during computations would be beneficial to system performance.

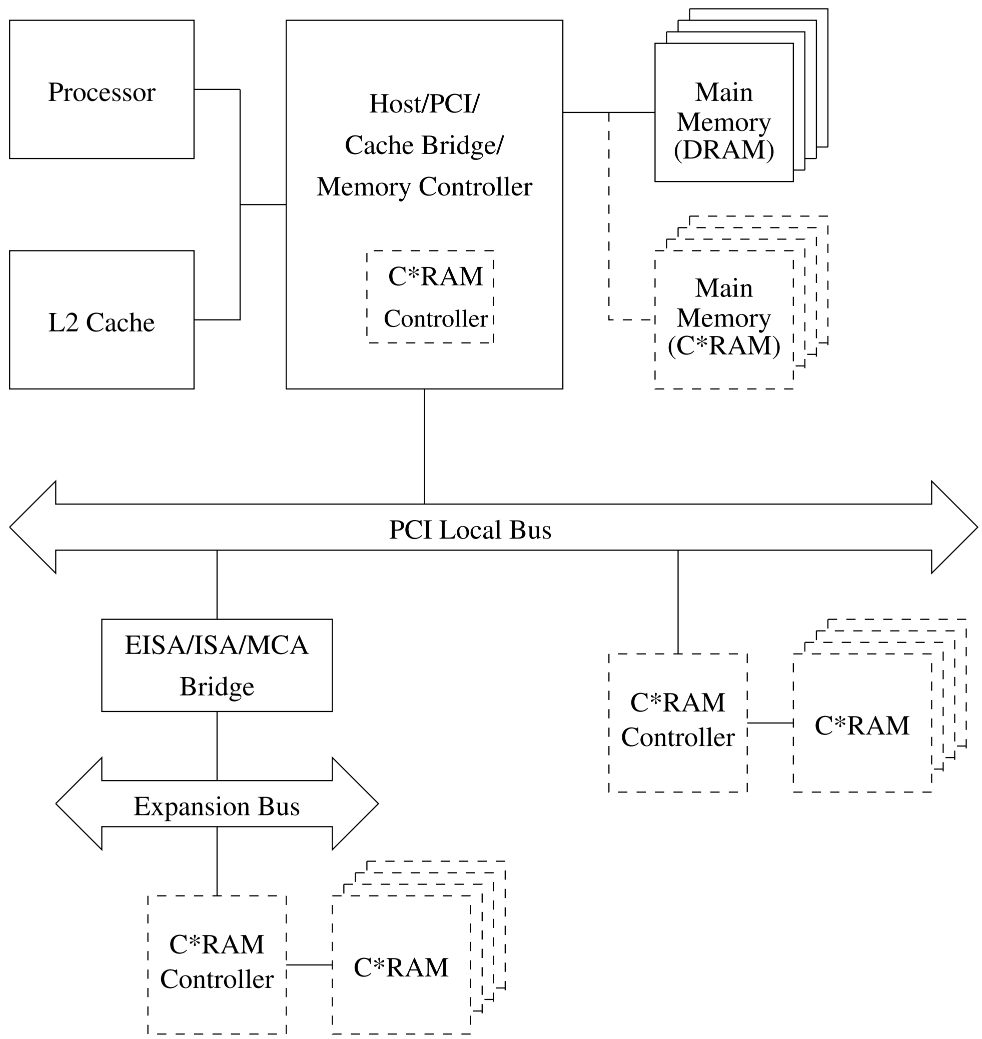


FIGURE 5.5 C•RAM system integration possibilites

5.4 Summary

In Section 5.1, we looked at a bit-parallel extension to the C•RAM design, and derived new performance values for basic arithmetic. Addition performs better on a bit-parallel architecture than a bit-serial one since fewer memory accesses are required, and the PEs are memory bandwidth limited. Multiplication performs better on the bit-serial architecture since it does not use PEs to compute placeholder additions. Since bit-parallel PEs require additional hardware, and the memory bandwidth limitations of DRAM are not too severe, we concluded that the bit-serial architecture is probably the most appropriate for this design.

Memory bandwidth to the Processing Elements can be improved with more efficient use of inactive sense amplifiers. Synchronous DRAM control circuitry allows more efficient memory bank management, and could potentially be used as the front-end interface in future C•RAM designs.

It is intended that C•RAM will eventually be integrated into desktop systems as main memory on the motherboard, but until then integration on expansion PCI or EISA buses will suffice for demonstration purposes. A C•RAM controller is required for translation of high level microroutine commands from the host to low level array instructions to the PEs. On the motherboard, it would be an enhanced memory controller with a writeable control store for microroutines. A compiler is required to provide physical address abstractions to the user, and allow access to vector processing through popular programming languages.

CHAPTER 6 Conclusions

"The decisive part of the device,
determining more than any other part
its feasibility, dimensions, and cost,
is the memory."

-John von Neumann

When this research period began, it was known that processing in memory was an inexpensive and powerful computing architecture. The abundance of memory in computers combined with their high internal bandwidth make them the ideal component to house the parallel processing needed by certain applications. In earlier C•RAM research, Elliott and Cojocaru investigated different Processing Element architectures, designed and fabricated small prototype chips, and showed favourable performance figures.

This thesis advances the C•RAM project in two directions. An industry standard Dynamic RAM was changed into a JEDEC compatible Computational RAM. Secondly, it was shown that this chip could be used for fast manipulation of databases, and for performing the most computationally heavy components of MPEG-2 video compression in real time.

In Chapter 3, the C•RAM chip was introduced, and an outline of the design steps taken during the period at IBM Microelectronics was presented. Processing Elements, based on those used in a previous prototype, were designed and laid out on an 8 Sense Amplifier pitch and interfaced with the predesigned memory array. The area and power consumption increase that resulted was shown to be under 15%.

A prime objective of this design iteration is to make the C•RAM industry acceptable, which was easily achieved by starting with an industry accepted design and changing it very little. Since it operates exactly like its DRAM predecessor, C•RAM can immediately replace all 16Mb Dynamic RAMs with no supporting hardware. Operation of the PEs requires a microcoded controller, which would be implemented on the motherboard as an extended DRAM controller.

In Chapter 4, the C•RAM's performance was analyzed. A small library of arithmetic microroutines was generated, and DRAM's page mode feature (which increases the memory-to-PE bandwidth by treating bitlines as high speed storage elements) was shown to give a threefold performance increase. Algorithms for parallel searches and sorts were generated, and compared to standard single processor techniques. Methods for performing Discrete Cosine Transforms and motion estimation were shown, and the performance figures indicated that real-time MPEG-2 compression is possible with C•RAM.

We are convinced that the Processing Element's size to performance ratio is about right. Memory bandwidth generally determines the performance of an architecture, and since the PEs and their memory (in page mode) cycle at about the same rate, it would not be wise to make the PE substantially bigger. To that end, it was shown in Chapter 5 that bit-parallel processing, an architecture requiring more hardware but fewer memory accesses per addition, is not worthwhile in DRAM.

Memory bandwidth to the PEs (and therefore performance) can be improved further with better management of memory bank accesses. We also saw in Chapter 5 that bank access flexibility can be obtained by either reworking the Asynchronous DRAM's bank management scheme, or by using a Synchronous DRAM interface.

In the immediate future, the C•RAM project should proceed with the construction of a working real-time image processing prototype. Full implementation of real-time MPEG-2 compression is a challenging next step which will require a compatible controller, compiler software, and a complete algorithm.

Other application possibilities should be explored. It was mentioned briefly in Section 4.3 that C•RAM could be used for financial risk analysis and portfolio value forecasting. The Deep Blue group at IBM have chosen financial modelling as one of the next projects for their massively parallel RS/6000 machine [Tan97].

An architecture similar to C•RAM's was discussed in [Fol90] as an integral part of a graphics rendering engine. Accelerix has recently fabricated a processor-in-memory chip with 4096 pixel processors and an onchip controller which performs both as a graphics controller and frame buffer [Fos96]. Research that determines C•RAM's place in a full rendering pipeline and proposes an implementation strategy would be useful.

The logical next step for C•RAM itself is the implementation on a 64Mb or larger Dynamic RAM. During this design iteration, tradeoffs between using a static ALU and a dynamic ALU should be determined. It is unclear whether one should optimize for area, power, and performance at the expense of rigid timing constraints and additional control complexity.

References

[Asp90] Aspray, William. John von Neumann and the origins of modern computing. Boston: Massachusetts Institute of Technology Press, 1990.

[Bat80] Batcher, Kenneth E. Massively Parallel Processor. *IEEE Transactions on Computers*. Volume C-29. No 9. September 1980.

[Bay96] Bayko, John. Great Microprocessors of the Past and Present. Version 9.9.2. University of Regina, 1996.
http://www.cs.uregina.ca/~bayko/cpu.html

[Che77] Chen, Wen-Hsuing. et al. A Fast Computational Algorithm for the Discrete Cosine Transform. *IEEE Transactions on Communications*. Volume Com-25. No 9. September 1977.

[Coj95] Cojocaru, Christian. Computational RAM: Implementation and Bit-Parallel Architecture. Masters Thesis. Carleton University. January 1995.

[Dia97] Diamond Multimedia. MMX - A Diamond Multimedia White Paper.
http://www.diamondmm.com/product-support/white-papers/visualization/mmx

[Ell90] Elliott, D.G., W.M. Snelgrove. C•RAM: Memory with a Fast SIMD Processor. Proceedings of the Canadian Conference on VLSI, pp 3.3.1-3.3.6. Ottawa ON, October 1990.

[Ell92] Elliott, D.G., W.M. Snelgrove, and M. Stumm. Computational RAM: A Memory-SIMD Hybrid and its Application to DSP. IEEE 1992 Custom Integrated Circuits Conference, pages 30.6.1-30.6.4. Boston MA, May 1992.

[Ell95] Elliott D.G., W.M. Snelgrove, C. Cojocaru, and M. Stumm. A PetaOp/s is currently feasible by computing in RAM. PetaFLOPS Frontier Workshop at the IEEE Frontiers of Massively Parallel Computation Symposium. McLean VA, February 1995.

[Ell96] Elliott, Duncan. Computational RAM: Merged Processor Logic and DRAM. Private Presentation at IBM Burlington Vermont. September 1996.

[Ell97] Elliott, Duncan G. Computational RAM: A Memory-SIMD Hybrid. Ph.D. Thesis in Progress. University of Toronto. October 1997.

[Fol90] Foley, James D. et al. Computer Graphics: Principles and Practice. 2nd Edition. Reading, Mass.: Addison-Wesley Publishing Company, 1990.

[Fos96] Foss, Richard C. Implementing Application Specific Memory. ISSCC '96 Digest. p 260ff. San Francisco. Feb 8-10 1996.

[Fou90] Fountain, Terry J. and Malcolm J. Shute. Multiprocessor Computer Architectures. New York: Elsevier Science Publishing Company, 1990.

[Fox94] Fox, Geoffrey C. et al. Parallel Computing Works. Section 7.1: Embarrassingly Parallel Problem Structure. San Francisco: Morgan Kaufmann Publishers, 1994.

http://www.netlib.org/utk/lsi/pcwLSI/text/node132.html

[Gea97] Gealow, Jeffrey Carl. An Integrated Computing Structure for Pixel-Parallel Image Processing. Sc.D. Thesis. Massachusetts Institute of Technology. June 1997.

[Gea96] Gealow, Jeffrey C. et al. System Design for Pixel-Parallel Image Processing. *IEEE Transactions on VLSI Systems.* Vol 4. No 1. March 1996.

[Gok95] Gokhale, Maya et al. Processing in Memory: The Terasys Massively Parallel PIM Array. *Computer.* April 1995.

[Gol72] Goldstine, Herman H. The Computer from Pascal to von Neumann. Princeton, NJ: Princeton University Press, 1972.

[Gus88] Gustafson, J.L. Reevaluating Amdahl's Law. Chapter for book. Parallel Processing for Supercomputers and Artifical Intelligence, edited by Kai Hwang. New York: McGraw-Hill, 1989.
http://www.scl.ameslab.gov/Publications/AmdahlsLaw/Amdahls.html

[Gok95] Gokhale, Maya et al. Processing in Memory: The Terasys Massively Parallel PIM Array. *Computer.* April 1995.

[Gus97] Gustavson, Stefan. MMX- Myths and Facts. April 1997.
http://www.isy.liu.se/~stefang/mmx.html

[Ham90] Hamacher, V. C., Zvonko Vranesic, and Safwat Zaky. Computer Organization. Third Edition. New York: McGraw-Hill, 1990.

[Hen90] Hennessy, John L., and David A. Patterson. Computer Architecture: A Quantitative Approach. San Mateo, CA: Morgan Kaufmann Publishers Inc., 1990.

[Her95] Herrmann, Frederick P. et al. A 256-Element Associative Parallel Processor. *IEEE Journal of Solid State Circuits*. Vol 30. No 4. April 1995.

[Hor90] Hord, R. Michael. Parallel Supercomputing in SIMD Architectures. Boston: CRC Press, 1990.

[Int96] Intel. The Pentium Pro Processor Fact Sheet. Santa Clara, CA. 1996.
http://www.prax.com/wpapers/factsht.htm

[Kal90a] Kalter, Howard L. et al. A 50-ns 16-Mb DRAM with a 10-ns Data Rate and On-Chip ECC. *IEEE Journal of Solid-State Circuits*. Vol 25. No 5. October 1990.

[Kal90b] Kalter, Howard L. et al. A 50ns 16Mb DRAM with a 10ns Data Rate. pp. 232-233. Proceedings of IEEE International Solid States Circuits Conference. 1990.

[Kal96] Kalter, Howard L. Private Communication. IBM Microelectronics. August 22, 1996.

[Kli82] Klingman, Edwin E. Microprocessor Systems Design. Volume II. Englewood Cliffs, NJ: Prentice-Hall Incorporated, 1982.

[Kri89] Krishnamurthy, E.V. Parallel Processing: Principles and Practice. New York:

Addison-Wesley Publishing Company, 1989.

[Le96] Le, Thinh M., W.M. Snelgrove, S. Panchanathan. Computational RAM implementation of MPEG-2 for real-time encoding. SPIE Proceedings of Multimedia Hardware Architecture ‘97. pp.182-193. February 1997.

[Lee94] Lee, J.A.N. Biography of John von Neumann. History of Computing Archive. Department of Computer Science. Virginia Tech. 1994. http://ei.cs.vt.edu/~history/VonNeumann.html

[Lou82] Loucks, Wayne M. et al. A Vector Proceessor Based on One-Bit Microprocessors. *IEEE Micro*. Vol 2. No 1. February 1982.

[McK97] McKenzie, Robert N., W. Martin Snelgrove, and Duncan G. Elliott. Computational RAM: SIMD processing in DRAM. Presented at the CASCON Conference. Toronto, Ontario. November 1997.

[McL86] McLucky, K. and Angus Barber. Sorting Routines for Microcomputers. London: McMillan Education Limited, 1986.

[Nya97] Nyasulu, Peter M. A Computational RAM controller. Departmental Seminar. Department of Electronics, Carleton University. September 24, 1997.

[Pri90] Prince, Betty. Semiconductor Memories. Second Edition. New York: John Wiley & Sons, Inc., 1991.

[Sch88] Schmitt, Lorenz A. et al. The AIS-5000 Parallel Processor. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol 10. No 3. May 1988.

[Sha95] Shanley, T. and Don Anderson. PCI System Architecture. Third Edition. Reading, Massachusetts: Addison-Wesley Publishing Company, 1995.

[Sne95] Snelgrove, W. Martin. VLSI Circuits. Text in preparation. 1995.

[Sto87] Stone, Harold S. High Performance Computer Architecture. Reading, Mass.: Addison-Wesley Publishing Company, 1987.

[Tan90] Tanenbaum, Andrew S. Structured Computer Organization. 3rd Edition. New York: Prentice Hall Inc., 1990.

[Tan97] Tan, C.J. Parallel Processing and the IBM Deep Blue Chess Computer. Keynote Address at CASCON 97. Toronto. November 1997.

[Vra89] Vranesic, Z., and S. Zaky. Microcomputer Structures. New York: Saunders College Publishing, 1989.

[Wak90] Wakerly, John F. Digital Design: Principles and Practices. New York: Prentice Hall Inc., 1990.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                          Bit-Parallel C*RAM Simulator                    %
%                              RMcKenzie 1997                              %
%                                                                          %
%   Bit Conversion functions                                               %
%   ------------------------                                               %
%   hexbin          Hexadecimal to Binary Conversion                       %
%   hexdec          Hexadecimal to Decimal Conversion                      %
%   dechex          Decimal to Hexidecimal Conversion                      %
%   decbin          Decimal to Binary Conversion                           %
%   binhex          Binary to Hexadecimal Conversion                       %
%   bindec          Binary to Decimal Conversion                           %
%                                                                          %
%   Basic DRAM-mode & related procedures                                   %
%   ------------------------------------                                   %
%   ocread          Offchip read of row data                               %
%   ocwrite         Offchip write to row                                   %
%   ocreadd         Offchip read of row data in decimal format             %
%   extract         Prints decimal equivalent of column data               %
%   printmem        Offchip read of large number of rows                   %
%   rand8bit        Generates an 8-bit random number                       %
%   randrows        Writes random data to any group of rows                %
%   widen           Widens a string by adding "0"s to left side            %
%   flipstr         Reverses a string: "12345" becomes "54321"             %
%                                                                          %
%   Basic C*RAM-mode procedures                                            %
%   ---------------------------                                            %
%   rop             Read Operate                                           %
%   ropw            Read Operate Writeback                                 %
%   wr              Write                                                  %
%                                                                          %
%   Advanced C*RAM-mode procedures                                         %
%   ------------------------------                                         %
%   add             Performs nbit addition R=A+B                           %
%   subtr           Performs nbit subtraction R=A-B                        %
%   mult            Performs nbit multiplication R=A*B                     %
%   divide          Performs nbit divide Q=A/B remainder R                 %
%   add2op          Performs nbit 2-operand addition R+=A                  %
%   sub2op          Performs nbit 2-operand subtraction R-=A               %
%   blank           Writes zeroes to multiple rows                         %
%   copy            Copies value of A to B                                 %
%   add1            Performs nbit addition A+=1                            %
%   subtr1          Performs nbit subtraction A-=1                         %
%   negate          Calculates A = -A                                      %
%   writeval        Writes an arbitrary value to local memory of PEs       %
%   compare         Calculates boolean variable "A>B"                      %
%   fpadd           Floating point addition R=A+B                          %
%   bp_add          Calculates nbit bit-parallel addition R=A+B            %
%   bp_mult         Calculates nbit bit-parallel multiplication R=A*B      %
%   sort            Performs parallel sort                                 %
%                                                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

const npe  := 15
const nrow := 511
const nmantb := 5
const nexpb := 3

var x           :array 0..npe of string(1)
var y           :array 0..npe of string(1)
var we          :array 0..npe of string(1)
var r           :array 0..npe of string(1)
var prevr       :array 0..npe of string(1)
var prevsum     :array 0..npe of string(1)
var sum         :array 0..npe of string(1)

var m           :array 0..npe,0..nrow of string(1)

var nrop,nwr,nropw      :int := 0
var nocw,nocr           :int := 0


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                   GENERAL MAINTENANCE FUNCTIONS                   %
%  The functions here are simple string and number manipulators    %
%  used frequently in coding.                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function flipstr (s     :string):string
    %Reverses the order of a string. "12345" -> "54321"

    var r   :string:= ""
    for decreasing q:length(s) .. 1
        r := r + s(q)
    end for
    result r
end flipstr

function widen(s       :string,n    :int):string
    %Makes string s n-bits long by adding zeroes to the left side.
    % widen("110",5) --> "00110"
    var r           :string := ""

    for q:1..(n-length(s))
        r := r + "0"
    end for
    r := r + s
    result r
end widen

function contain(cop        :string,f       :string(1)):boolean
    var r           :boolean := false

    for q:1..length(cop)
        if cop(q) = f then
            r := true
        end if
    end for
    result r
end contain

function rand8bit :string
    %Generates a random 8-bit number
    var r           :string:=""
    var t           :int

    for q:0..7
        randint(t,0,1)
        if t = 1 then
            r := r + "1"
        else
            r := r + "0"
        end if
    end for
```

```
    result r
end rand8bit

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                       BASE CONVERSION FUNCTIONS                   %
%                                                                   %
%  The functions shown here are used to convert between binary     %
%  decimal, and hexadecimal.                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function hexdec1 (h     :string(1)):int
    %Converts a 1 digit hex value to decimal
    var r           :int := 0

    if (ord(h) >= 97 and ord(h) <= 102) then
        r := ord(h) - 87
    elsif (ord(h) >= 65 and ord(h) <= 70) then
        r := ord(h) - 55
    elsif (ord(h) >= 48 and ord(h) <= 57) then
        r := strint(h)
    else
        put "ERROR in base conversion: ",h," is not a valid hex digit"
    end if
    result r
end hexdec1

function hexdec (h      :string):int
    %Converts an arbitrary sized hex string to decimal
    var r           :int:=0

    for q:0..length(h)-1
        r += (hexdec1(h(length(h)-q))) * (16**q)
    end for
    result r
end hexdec

function dechex1(d      :int):string(1)
    var r           :string(1) := "x"

    if (d >= 0 and d <= 9) then
        r := intstr(d)
    else
        r := chr(d+87)
    end if
    result r
end dechex1

function dechex (d      :int):string
    var dw          :int := d
    var bhex        :string := ""

    loop
        bhex := bhex + dechex1(dw mod 16)
        dw := dw div 16
        if dw = 0 then
            exit
        end if
    end loop
    result flipstr(bhex)
end dechex

function decbin (d       :int):string
    var bb          :string:=""
    var dw          :int := d

    loop
        bb := bb + intstr(dw mod 2)
        dw := dw div 2
        if dw = 0 then
            exit
        end if
    end loop
    result (flipstr(bb))
end decbin


function bindec1(b      :string(1)):int
    var r       :int := 1
    if b = "0" then
        r := 0
    end if
    result r
end bindec1

function bindec (b      :string):int
    var r           :int :=0
    for q:0..length(b)-1
        r += bindec1(b(length(b)-q)) * 2**q
    end for
    result r
end bindec

function hexbin (h     :string):string
    var d       :int := hexdec(h)

    result decbin(d)
end hexbin

function binhex (b      :string):string
    var dec         :int:= bindec(b)

    result dechex(dec)
end binhex

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                         DRAM MODE FUNCTIONS                       %
%                                                                   %
%  Procedures which perform offchip reads & offchip writes, and    %
%  functions that show decimal equivalents of rows and columns     %
%  are listed here.                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure ocwrite(row       :int,data       :string(npe+1))
    var dw          :string:=widen(data,npe+1)
    %Assumes the #of PEs is npe
    nocw+=1
    for decreasing pe:npe..0
        m(pe,row) := dw(16-pe)
    end for
end ocwrite

procedure ocread(row        :int)
    %Offchip read ... returns the contents of a row
```

```
    var rdata       :string:=""
    nocr+=1
    for decreasing pe:npe..0
        put m(pe,row)..
    end for
    put ""
end ocread

procedure ocreadd(row       :int)
    %Offchip read that prints the decimal value of the row
    var r               :string:=""
    for decreasing pe:npe..0
        r += m(pe,row)
    end for
    put bindec(r):5
end ocreadd

function extract(col,lorow,hirow    :int):int
    %Prints the decimal equivalent of mem(col,hirow...lorow)

    var rd          :string:="0"
    var r           :int := 0

    for decreasing row:hirow..lorow
        rd := rd+m(col,row)
    end for
    r := bindec(rd)
    result r
end extract

procedure printmem(lorow,hirow      :int)
    %Prints out rows lorow..hirow

    for decreasing row:hirow..lorow
        put row:3," ":5..
        ocread(row)
    end for
end printmem

procedure randrows(lorow,hirow      :int)
    %Fills lorow..hirow with random bits

    var rowd            :string
    var r               :int

    for decreasing row: hirow..lorow
        randint(r,0,2**16-1)
        rowd := widen(decbin(r),16)
        ocwrite(row,rowd)
    end for
end randrows


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                       C*RAM MODE FUNCTIONS                         %
%                                                                    %
%  All the code required for low-level simulation of the PEs        %
%  is listed here. It is not necessary to study this stuff.         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure initialize
    for pe:0..npe
        x(pe) := "0"
        y(pe) := "0"
        we(pe):= "1"
        r(pe) := "0"
        for row:0..nrow
            m(pe,row) := "0"
        end for
    end for
end initialize

function alu(op         :string,
             x,y,m      :string(1)):string(1)
    var r           :string(1):= "0"
    var opselect    :string := x+y+m
    var opb         :string := flipstr(widen(hexbin(op),8))
    %                          ^^^^^^^
    % We need to flip the string since "10101010" has op(0) as "1"

    %This code is dreadfully inefficient. I've done it this way for
    %ease of understanding. What I'd do ideally is:
    % r := opb(bindec(opselect)+1)

    if (x="0" and y="0" and m="0") then
        r := opb(1)
    elsif (x="0" and y="0" and m="1") then
        r := opb(2)
    elsif (x="0" and y="1" and m="0") then
        r := opb(3)
    elsif (x="0" and y="1" and m="1") then
        r := opb(4)
    elsif (x="1" and y="0" and m="0") then
        r := opb(5)
    elsif (x="1" and y="0" and m="1") then
        r := opb(6)
    elsif (x="1" and y="1" and m="0") then
        r := opb(7)
    elsif (x="1" and y="1" and m="1") then
        r := opb(8)
    end if
    result r
end alu

procedure activate_addblock
    %Performs the addblock logic operations in extended bit-parallel PE
    %Use of bit-paralllelism is optional, and activated by putting an "a"
    %in the register code.

    var cin             :string(1) := "0"
    var cout            :string(1) := "0"

    for pe:0..npe
        if cin = "0" then
            if x(pe) = "0" then
                if r(pe) = "0" then
                    cout := "0"
                    sum(pe) := "0"
                else
                    cout := "0"
                    sum(pe) := "1"
                end if
            else
                if r(pe) = "0" then
```

```
                    cout :="1"
                    sum(pe) := "0"
                else
                    cout := "0"
                    sum(pe) := "1"
                end if
            end if
        else
            if x(pe) = "0" then
                if r(pe) = "0" then
                    cout :="0"
                    sum(pe) := "1"
                else
                    cout := "1"
                    sum(pe) := "0"
                end if
            else
                if r(pe) = "0" then
                    cout :="1"
                    sum(pe) := "1"
                else
                    cout := "1"
                    sum(pe) := "0"
                end if
            end if
        end if
        cin := cout
    end for
end activate_addblock

procedure activate_bustie
    var zero            :boolean := false

    for pe:0..npe
        if r(pe) = "0" then
            zero := true
        end if
    end for

    for pe:0..npe
        prevr(pe) := r(pe)
        prevsum(pe) := sum(pe)
        if zero then
            r(pe) := "0"
            sum(pe) := "0"
        end if
    end for
end activate_bustie

procedure deactivate_bustie
    for pe:0..npe
        r(pe) := prevr(pe)
        sum(pe) := prevsum(pe)
    end for
end deactivate_bustie

procedure write_x
    for pe:0..npe
        x(pe) := sum(pe)
    end for
end write_x

procedure write_y
    for pe:0..npe
        y(pe) := sum(pe)
    end for
end write_y

procedure write_we
    for pe:0..npe
        we(pe) := sum(pe)
    end for
end write_we

procedure shift_left
    x(0) := "0"             %PE0 gets a "0" on shift_left

    for pe:1..npe
        x(pe) := sum(pe-1)
    end for
end shift_left

procedure shift_right
    y(npe) := "0"           %PEnpe gets a "0" on shift_right

    for decreasing pe:npe-1..0
        y(pe) := sum(pe+1)
    end for
end shift_right

procedure operate(row           :int,
                  op            :string,
                  cop           :string)

    for pe:0..npe
        r(pe) := alu(op,x(pe),y(pe),m(pe,row))
        sum(pe) := r(pe)
    end for

    if contain(cop,"a") then
        activate_addblock
    end if

    if contain(cop,"t") then
        activate_bustie
    end if

    if contain(cop,"x") then
        write_x
    end if

    if contain(cop,"y") then
        write_y
    end if

    if contain(cop,"w") then
        write_we
    end if

    if contain(cop,"l") then
        shift_left
    end if

    if contain(cop,"r") then
```

```
        shift_right
    end if

    if contain(cop,"t") then
        deactivate_bustie
    end if
end operate

procedure seereg (rx        :string)
    %Shows values of selected regsiters for all PEs
    % seereg("xyw") shows the values of npe X-registers, Y-registers,
    %               and WE-registers
    %   x   - X register
    %   y   - Y register
    %   y   - WE register
    %   p   - Current ALU output value

    for q:1..length(rx)
        if rx(q) = "x" then
            put "X":8..
            for decreasing pe:npe..0
                put x(pe)..
            end for
            put ""
        elsif rx(q) = "y" then
            put "Y":8..
            for decreasing pe:npe..0
                put y(pe)..
            end for
            put ""
        elsif rx(q) = "w" then
            put "WE":8..
            for decreasing pe:npe..0
                put we(pe)..
            end for
            put ""
        elsif rx(q) = "p" then
            put "PE":8..
            for decreasing pe:npe..0
                put r(pe)..
            end for
            put ""
        end if
    end for
end seereg

procedure rop (row      :int,op     :string,cop     :string)
    %READ OPERATE
    %Performs an ALU operation using the current xreg yreg and m(row)
    %values. Writes to the applicable registers

    nrop += 1
    operate(row,op,cop)
end rop

procedure wr (row       :int)
    %PE WRITE
    %Writes the current PE result to row

    nwr += 1
    for pe:0..npe
        if we(pe) = "1" then
            m(pe,row) := sum(pe)
        end if
    end for
end wr

procedure ropw(row      :int,op     :string,cop     :string)
    %READ OPERATE WRITE
    %Performs read-operate, then a write to row
    nropw += 1
    rop(row,op,cop)
    wr(row)
end ropw

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    C*RAM MODE ADVANCED USE                        %
%                 ARITHMETIC AND LOGIC PROCEDURES                   %
%                                                                   %
% A collection of useful procedures is listed here. It is in a     %
% format that mimics the controller's instruction sequence to the  %
% Processing Element array. The host initiates the execution of    %
% one of these procedures, and the controller sends sequences of   %
% rop, ropw, and wrs to the array                                   %              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procedure add(alsb,blsb,rlsb,nbit       :int)
    % Performs nbit addition R=A+B

    rop(0,"00","y")                         %Set the carry bit to 0
    for bit:0..nbit-1
        rop(alsb+bit,"aa","x")              %Load 1st bit
        rop(blsb+bit,"96","")               %Calculate sum
        wr(rlsb+bit)
        rop(blsb+bit,"e8","y")              %Calculate carry bit
    end for
%   ropw(rlsb+nbit,"cc","")                 %Write carry-out bit (optional)
end add

procedure add2op(rlsb,alsb,nbit        :int)
    % Performs R+=A
    % Requires opcodes different from addition as the carryout
    % calculation depends upon bbit, carryin, and the sum bit
    % of (abit+bbit+carryin)

    rop(alsb,"00","y")                      %Clear carry bit
    for bit:0..nbit-1
        rop(alsb+bit,"aa","x")              %Load abit
        ropw(rlsb+bit,"96","")              %Load rbit, sum and writeback
        rop(rlsb+bit,"d4","y")              %Calculate carryout
    end for
%   ropw(rlsb+nbit,"cc","")                 %Write carry-out bit (optional)
end add2op

procedure subtr(alsb,blsb,rlsb,nbit       :int)
    % Performs nbit subtraction R=A-B

    rop(0,"ff","y")                 %Set the carry bit to 1
    for bit:0..nbit-1
        rop(alsb+bit,"aa","x")              %Load 1st bit
        rop(blsb+bit,"69","")
        wr(rlsb+bit)
        rop(blsb+bit,"d4","y")              %Calculate carry bit
    end for
```

```
%   ropw(rlsb+nbit,"cc","")                 %Write carry-out bit (optional)
end subtr

procedure sub2op(rlsb,alsb,nbit             :int)
    % Performs R-=A
    % Uses different opcodes from subtraction since we FIRST load and
    % invert the abit from the minuend operand, then load the rbit
    % find the sum and writeback, and finally calculate carryout.

    rop(0,"ff","y")                 %Set the carry bit to 1
    for bit:0..nbit-1
        rop(alsb+bit,"55","x")              %Load abit, invert & wr to X-reg
        ropw(rlsb+bit,"96","")              %Calculate sum
        rop(rlsb+bit,"d4","y")              %Calculate carry bit
    end for
%   ropw(rlsb+nbit,"cc","")                 %Write carry-out bit (optional)
end sub2op

procedure mult(alsb,blsb,rlsb,nbit          :int)
    % Performs nbit multiply R=A*B

    rop(rlsb,"ff","w")
    for bit:0..(2*nbit)-1
        ropw(rlsb+bit,"00","")              %Set partial product to 0
    end for

    for bit:0..(nbit-1)
        rop(blsb+bit,"aa","w")              %If (blsb+bit)=1 then
            add2op(rlsb+bit,alsb,nbit)   %add to partial product
        %End if
    end for
    rop(rlsb,"ff","w")
end mult

procedure blank(lorow,hirow         :int)
    % Puts zeroes in memory locations from lorow to hirow

    for row:lorow..hirow
        ropw(row,"00","")
    end for
end blank

procedure copy(alsb,blsb,nbit       :int)
    % Copies nbit bits of A to B

    for bit:0..nbit-1
        rop(alsb+bit,"aa","")
        wr(blsb+bit)
    end for
end copy

procedure divide (alsb,blsb         :int,
                  rl                :int,
                  nbit              :int)
    % Performs an n-bit divide of A/B
    var rlsb            :int := rl
    var qlsb            :int := rlsb+nbit+1

    copy(alsb,rlsb-nbit,nbit)       %Copy A
    blank(rlsb,rlsb+nbit)           %Clear remainder bits
    blank(blsb+nbit,blsb+nbit)      %Clear top bit in B
```

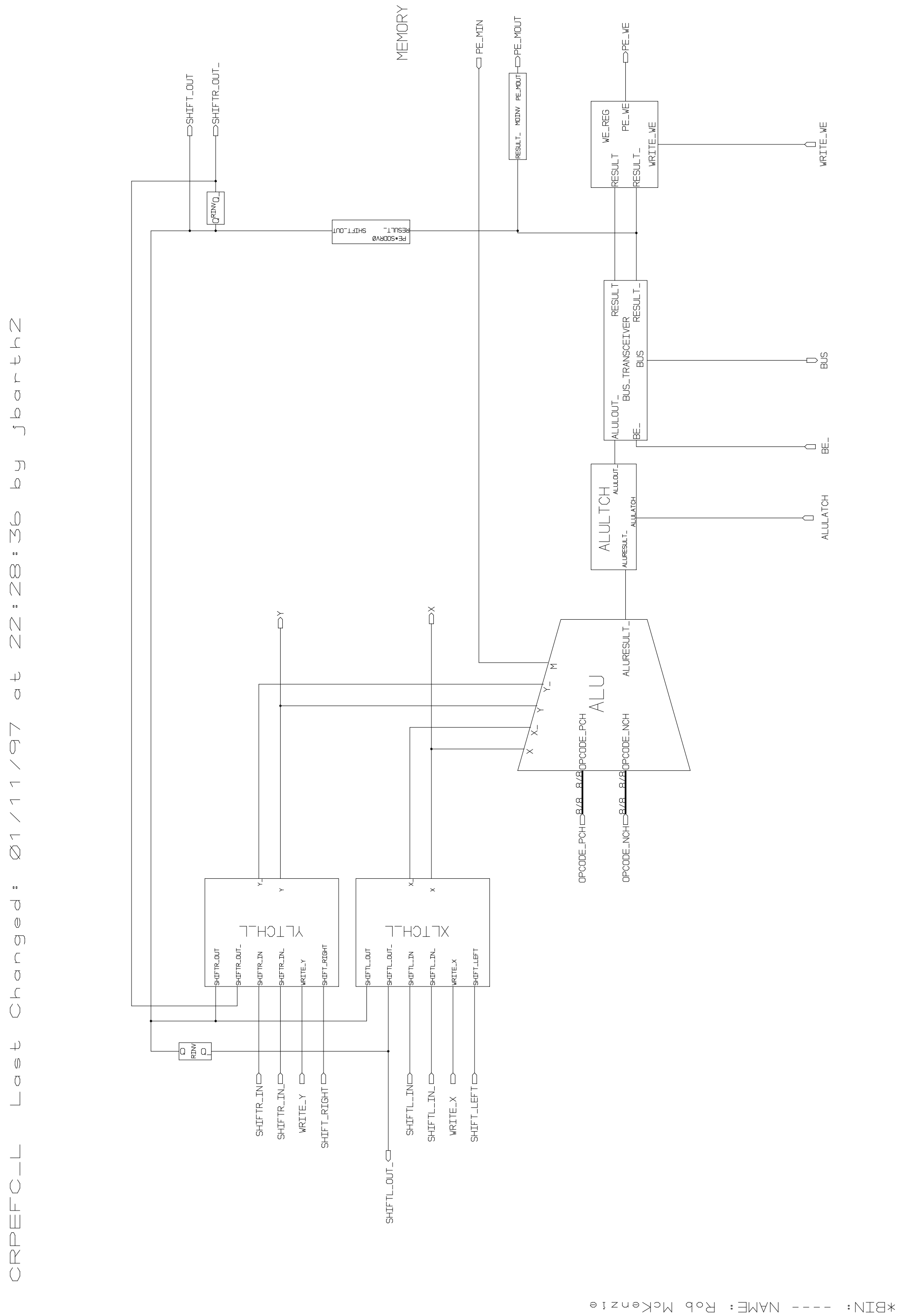
```
    for bit:0..nbit-1
        rlsb -= 1
        qlsb -= 1

        rop(rlsb,"ff","w")
        sub2op(rlsb,blsb,nbit+1)
        rop(rlsb+nbit,"55","x")
        wr(qlsb)                             %Write not(rlsb+nbit) to qlsb

        rop(rlsb+nbit,"aa","w")
        add2op(rlsb,blsb,nbit+1)         %Conditional restore
    end for
    rop(rlsb+nbit,"ff","w")
end divide

procedure add1(alsb,nbit          :int)
    % Adds 1 to A so that A+=1
    % X register holds the carry-out
    % Y register holds the carry-in

    rop(alsb,"ff","y")
    for bit:0..nbit
        rop(alsb+bit,"88","x")          %Calculate carry-out
        ropw(alsb+bit,"66","")          %Calculate sum
        rop(alsb+bit+1,"f0","y")        %Move carry-out to carry-in
    end for
end add1

procedure subtr1(alsb,nbit          :int)
    % Subtracts 1 from A, so that A-=1
    % X register holds the carry-out
    % Y register holds the carry-in
    % (2's compliment of 1 is 111111...111).

    rop(alsb,"ff","y")
    for bit:0..nbit
        rop(alsb+bit,"ee","x")          %Calculate carry-out
        ropw(alsb+bit,"99","")          %Calculate sum
        rop(alsb+bit+1,"f0","y")        %Move carry-out to carry-in
    end for
end subtr1

procedure negate(alsb,nbit          :int)
    % Negates nbit A so that A=(-A)
    % NOTE: negation is not simply bit inversion. It is adding 1 to the
    %       inverted bits. "nbit" must include the sign bit.

    rop(alsb,"ff","y")              %Put 1 in carry register
    for bit:0..nbit-1
        ropw(alsb+bit,"99","x")     %Add carry to inverted bit
        rop(alsb+bit,"0c","y")      %Calculate carry-out
    end for
end negate

procedure writeval(v                :int,
                   dlsb             :int,
                   nbit             :int)
    % Writes the nbit value "v" to local memory in all PEs
    % Requires funky controller hardware to perform the "if..then" bit

    var vbin            :string:=widen(decbin(v),nbit)
```

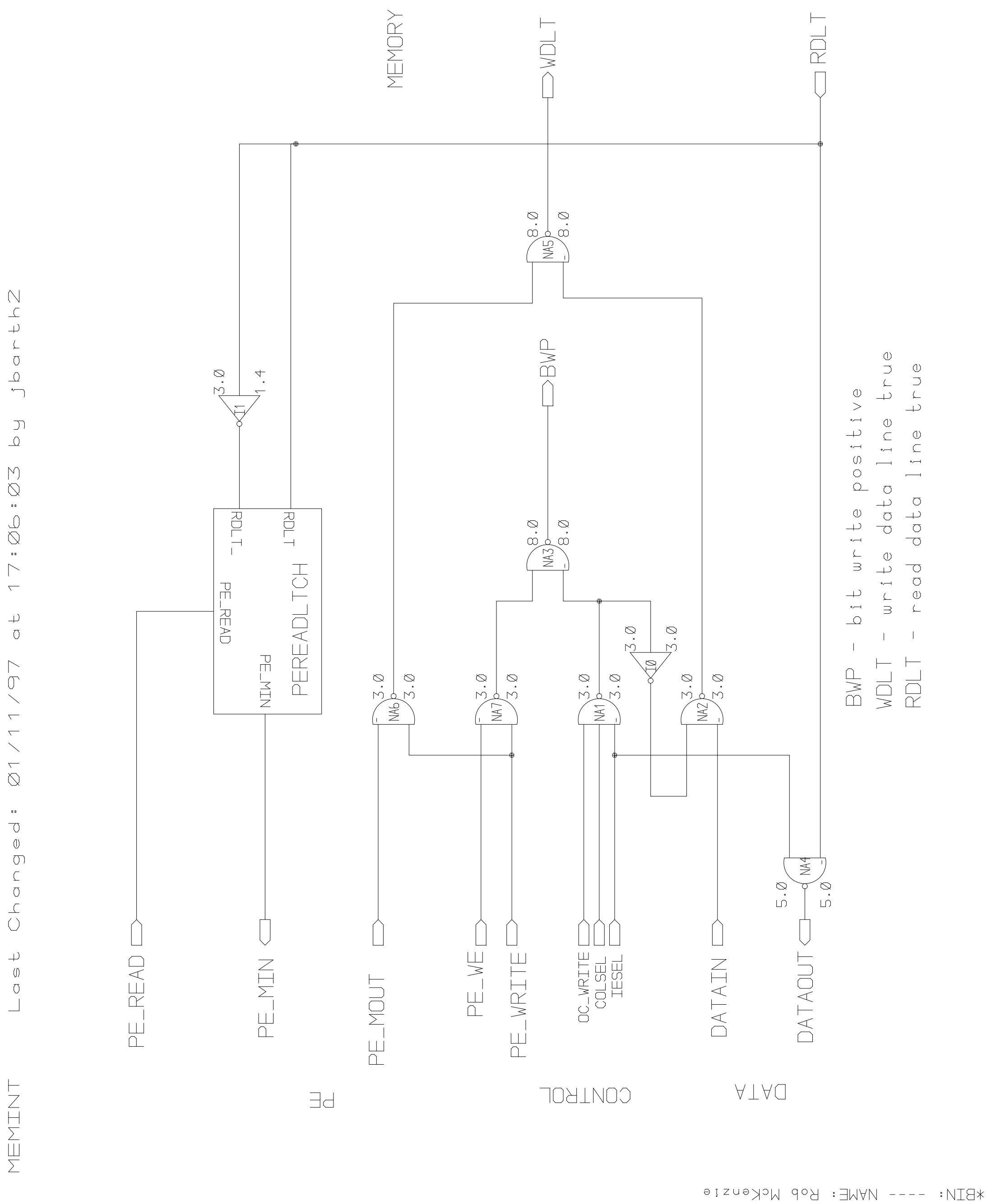
```
    for bit:0..nbit-1
        if vbin(nbit-bit) = "0" then
            ropw(dlsb+bit,"00","")
        else % vbin(nbit-bit)="1"
            ropw(dlsb+bit,"ff","")
        end if
    end for
end writeval

procedure compare (alsb,blsb,nbit         :int)
    % Compares nbit values A and B, and writes the boolean value
    % "A greater than B" in the X-register.

    for bit:0..nbit-1
        rop(alsb+bit,"aa","y")
        rop(blsb+bit,"d4","x")
    end for
end compare

procedure fpadd(amlsb,aelsb,asign,
                bmlsb,belsb,bsign,
                rmlsb,relsb,rsign,
                smlsb,edlsb,incr,mslsb               :int)
    % Performs a floating point add R=A+B
    % Variables:
    % amlsb - Amantissa lsb     aelsb - Aexponent lsb     asign - A signbit
    % bmlsb - Bmantissa lsb     belsb - Bexponent lsb     bsign - B signbit
    % rmlsb - Rmantissa lsb     relsb - Rexponent lsb     rsign - R signbit
    % smlsb - Shifted Mantissa lsb
    % edlsb - Exponent Difference lsb
    % incr  - Increment bit
    % mslsb - Mantissa sum lsb

    %Enable Writes
    rop(0,"ff","w")

    %Stick in ones where desired
    ropw(rmlsb+nmantb,"ff","")
    ropw(smlsb+nmantb,"ff","")

    compare(aelsb,belsb,nexpb)          %Put "a>b" into X-register
    rop(0,"f0","w")                     %Write "a>b" into WE-register
        %a>b
        seereg("w")
        copy(amlsb,rmlsb,nmantb)
        copy(aelsb,relsb,nexpb)
        copy(asign,rsign,1)
        copy(bmlsb,smlsb,nmantb)
        subtr(aelsb,belsb,edlsb,nexpb)  %Subtract exponents

    compare(aelsb,belsb,nexpb)          %Put "a>b" into X-register
    rop(0,"0f","w")                     %Write "b>a" into WE-register
        %b>a
        seereg("w")
        copy(bmlsb,rmlsb,nmantb)
        copy(belsb,relsb,nexpb)
        copy(bsign,rsign,1)
        copy(amlsb,smlsb,nmantb)
        subtr(belsb,aelsb,edlsb,nexpb)  %Subtract exponents
    rop(0,"ff","w")                     %End if

    %If the signs of the two numbers are the same, then we'll do a
    %mantissa add2op when adding, otherwise, we'll do a sub2op.
    %incr = asign nxor bsign
    rop(asign,"cc","x")
    rop(bsign,"a5","")
    wr(incr)

%   Mantissa shifting
%   for decreasing expbit:nexpb-1..6
%       rop(edlsb+expbit,"aa","w")      %If hi order exponent difference bits
%       blank(smlsb,smlsb+nexpb)        %are 1, then we blank the mantissa
%   end for

    for decreasing expbit:2..0
        %We now shift the smantissa 2**expbit positions if the expbit is 1
        %Remember that the top order bit above ShiftedMantissa is a 1

        rop(edlsb+expbit,"aa","w")
        copy(smlsb+2**expbit,smlsb,nmantb-2**expbit)
        blank(smlsb+nmantb-2**expbit+1,smlsb+nmantb)
    end for

    rop(incr,"cc","w")                  %If it's an increment
        add(rmlsb,smlsb,mslsb,nmantb+1)
    rop(incr,"55","w")                  %else it's a decrement
        subtr(rmlsb,smlsb,mslsb,nmantb+1)
    rop(0,"ff","w")                     %end if
    copy(mslsb,rmlsb,nmantb+1)

    %Last thing ... check to see if the top order bit is a zero. If so
    %we need to do a final shift on the result and add1 to the exponent

    rop(rmlsb+nmantb,"55","w")
        copy(rmlsb+1,rmlsb,nmantb)
        add1(relsb,nexpb)
    rop(0,"ff","w")
end fpadd

procedure bp_add(arow,brow,rrow     :int)
    % Performs a bit parallel add R=A+B across npe PEs

    rop(arow,"aa","x")          %Read A, put in X-Reg
    rop(brow,"5a","xa")         %Read B, add, and put result in X-Reg
    ropw(rrow,"f0","")          %Write result to R
end bp_add

procedure bp_mult(arow,brow,mrow,prow,nbit  :int)
    % Performs a bit parallel multiply R=A*B across PEs

    blank(prow,prow)            %Clear product
    rop(arow,"aa","x")          %Load A into X-register
    rop(brow,"aa","y")          %Load B into Y-register

    for bit:0..nbit-1
        rop(mrow,"dd","wt")     %Broadcast Bbit to all PEs
        ropw(prow,"5a","a")     %Calculate partial product
        rop(mrow,"f0","l")      %Shift X-register (A) left
        rop(mrow,"cc","r")      %Shift Y-register (B) right
    end for
end bp_mult

procedure sort(alsb,nbit        :int)
    % Performs a parallel sort. Each PE is given an item in the list,
```

```
    % and they alternate switching between evenPE/oddPE and oddPE/evenPE.

    const lgn := 0                  %local greater than neighbour row
    const mask:= 1                  %mask bits row

    for pe:0..npe
        ropw(lgn,"ff","w")
        for bit:0..nbit-1
            rop(alsb+bit,"aa","xr")
            ropw(lgn,"b2","")
        end for

        rop(lgn,"55","x")           %Load /lgn into X-register
        rop(mask,"a0","yl")         %Calculate (/lgn)and(mask)
                                    % write to YReg and ShiftLeft
        rop(mask,"fc","w")          %Write to WE registers

        %Swap data
        %(WE-register will stop unwanted swaps from taking place)
        for bit:0..nbit-1
            rop(alsb+bit,"aa","lr")     %Load bit ... shift left and right
            rop(mask,"d8","")           %If mask is "1" take Y-reg value
                                        %if mask is "0" take X-reg value
            wr(alsb+bit)
        end for

        rop(mask,"ff","w")
        ropw(mask,"55","")              %Invert mask bits

        %Extra step
        for decreasing q:npe..0
            put extract(q,alsb,alsb+nbit-1):3," "..
        end for
        put ""
     end for
end sort

procedure findlargest (alsb,nbit           :int)
    %Finds the largest element in an array. It uses the bus transceiver
    %to compare bit-by-bit the PE's local value versus the global value.
    %A "1" in the Y-register indicates that a PE is still "in the running",
    %or "on", otherwise the PE is "off". The PE (or PEs) still "on" at the
    %end of execution has the greatest element.

    rop(alsb+nbit,"ff","y")             %Write 1 to Y-Reg of all PEs
                                        % (i.e. turn them all "on")
    for decreasing bit:(nbit-1)..0
        rop(alsb+bit,"77","tx")         %Calculate a 0 if Abit is a
                                        % 1, and PE is still "on".
                                        % Broadcast and write to X-Reg.
        rop(alsb+bit,"48","y")          %If PE is still "on", compare
                                        % broadcasted bit (X) to Abit (M).
                                        % Determine if PE should switch
                                        % "off" and write to Y-Reg.
    end for
end findlargest

procedure findsmallest (alsb,nbit           :int)
    %Finds the smallest element in an array. It uses the bus transceiver
    %to compare bit-by-bit the PE's local value versus the global value.
    %A "1" in the Y-register indicates that a PE is still "in the running",
    %or "on", otherwise the PE is "off". The PE (or PEs) still "on" at the
    %end of execution has the smallest element.

    rop(alsb,"ff","y")                  %Write a 1 to Y-reg of all PEs
    for decreasing bit: (nbit-1)..0
        rop(alsb+bit,"bb","tx")         %Calculate a 0 if Abit is a 0,
                                        % and PE is still "on".
                                        % Broadcast and write to X-Reg.
        rop(alsb+bit,"84","y")          %If PE is still "on", compare
                                        % broadcasted bit (X) to Abit (M).
                                        % Determine if PE should switch
                                        % "off" and write to Y-Reg.
    end for
end findsmallest

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                         %
%                          MAIN PROGRAM CONTROL                           %
%       Mimics the calls given by the host to the controller.             %
%       Some simple examples are given to show proper syntax.             %
%                                                                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
randomize
initialize

% 4-BIT ADDITION
%randrows(0,7)
%add(0,4,8,4)
%printmem(0,11)
%put "Sample Calculation (PE 0)"
%put extract(0,0,3),"+",extract(0,4,7),"=",extract(0,8,11)

% 4-BIT SUBTRACTION
%randrows(0,7)
%subtr(0,4,8,4)
%printmem(0,11)
%put ""
%put "Sample Calculation (PE 0)"
%put extract(0,0,3),"-",extract(0,4,7),"=",extract(0,8,11)

% 16-BIT ADDITION
%randrows(0,31)
%add(0,16,32,16)
%printmem(0,47)

% FLOATING POINT ADD
%randrows(0,17)
%fpadd(0,5,8,9,14,17,18,24,27,28,34,37,38)
%printmem(0,44)

% 4-BIT MULTIPLY
%randrows(0,7)
%mult(0,4,8,4)
%printmem(0,16)

% 4-BIT DIVIDE
%randrows(0,7)
%divide(0,4,13,4)
%for w:0..3
%    %Check out the first 4 PEs' results
%    put extract(w,4,8)," divided by ",extract(w,0,3)
%    put "is ",extract(w,14,17)," remainder ",extract(w,9,13)
%    put ""
```

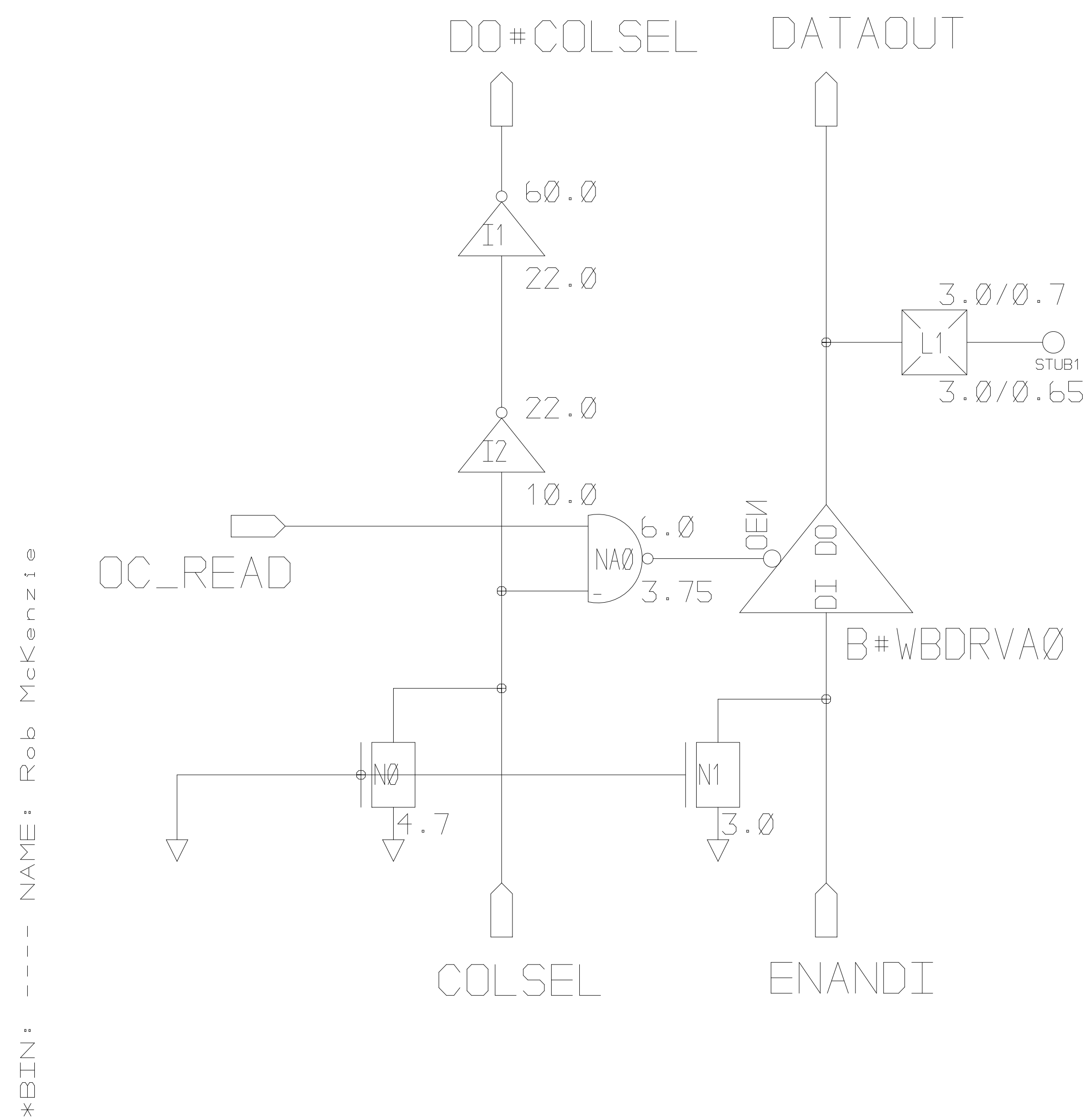
```
%printmem(0,14)

% BIT-PARALLEL ADD
%randrows(1,2)
%bp_add(1,2,0)
%printmem(0,2)
%ocreadd(2)
%ocreadd(1)
%ocreadd(0)

% SHIFT LEFT
%randrows(1,1)
%rop(1,"aa","r")
%rop(1,"cc","r")
%rop(1,"cc","r")
%rop(1,"cc","r")
%rop(1,"cc","r")
%rop(1,"cc","r")
%ropw(2,"cc","")
%printmem(0,2)

% BP_MULTIPLY
%arow = 0
%brow = 1
%mrow = 2
%prow = 3

%ocwrite(0,decbin(210))       %A
%ocwrite(1,decbin(047))      %B
%ocwrite(2,"1")              %Mask (=0000 0000 0000 0001)
%bp_mult(0,1,2,3,8)
%printmem(0,3)
%ocreadd(3)

% SORT
%ocwrite(1,hexbin("aaaa"))          %Write mask bits
%randrows(4,19)
%for decreasing q:npe..0
%    put extract(q,4,19)," "..
%end for
%put ""
%sort(4,16)
%for decreasing q:npe..0
%    put extract(q,4,19)," "..
%end for

% SORT (flip an entire string of numbers:
%       "15 14 13 ... 2 1 0" -> "0 1 2 ... 13 14 15")
%ocwrite(1,hexbin("aaaa"))          %Write mask bits
%ocwrite(4,hexbin("aaaa"))          %Data
%ocwrite(5,hexbin("cccc"))          %Data
%ocwrite(6,hexbin("f0f0"))          %Data
%ocwrite(7,hexbin("ff00"))          %Data
%
%for decreasing q:npe..0
%    put extract(q,4,7):2," "..
%end for
%put ""
%sort(4,4)

% WRITE 8-bit 43s TO ALL PEs AND NEGATE THEM
%writeval(43,0,8)
%for decreasing q:7..0
%    ocread(q)
%end for
%negate(0,8)
%put ""
%for decreasing q:7..0
%    ocread(q)
%end for

% BUS TIE SAMPLE
%ocwrite(0,hexbin("ffff"))
%rop(0,"aa","xt")
%ropw(1,"0f","")
%printmem(0,2)

%FIND LARGEST ELEMENT
%randrows(0,7)
%findlargest(0,8)
%seereg("y")
%for decreasing pe:15..0
%    put extract(pe,0,7)," "..
%end for

%FIND SMALLEST ELEMENT
randrows(4,11)
findsmallest(4,8)
seereg("y")
for decreasing pe:15..0
    if y(pe) = "1" then
        put "*"..
    end if
    put extract(pe,4,11)," "..
end for
put ""
ocwrite(1,hexbin("aaaa"))          %Write mask bits
sort(4,8)


put ""
put "Number of ROps     :",(nrop-nropw)
put "Number of Writes   :",(nwr-nropw)
put "Number of ROpWs    :",(nropw)
put "Number of OCWrites :",(nocw)
put "Number of OCRead   :",(nocr)
```

Appendix B: C•RAM Schematics

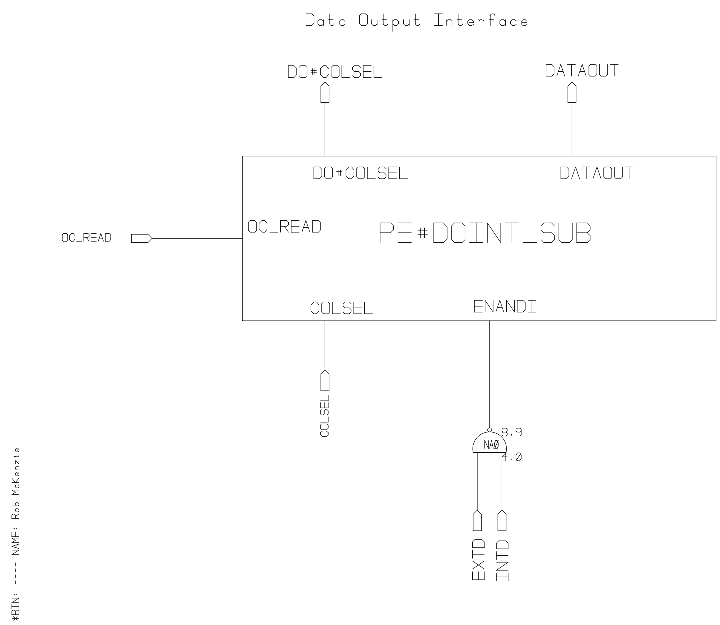



MEMINT
Last Changed: Ø1/11/97 at 17:Ø6:Ø3 by jbarth2
PE_READ
PE_MIN
PE_MOUT
PE_WE
PE_WRITE
OC_WRITE
COLSEL
IESEL
DATAIN
DATAOUT
PE
CONTROL
DATA
PEREADLTCH
MEMORY
WDLT
BWP
RDLT
BWP - bit write positive
WDLT - write data line true
RDLT - read data line true
*BIN: ---- NAME: Rob McKenzie

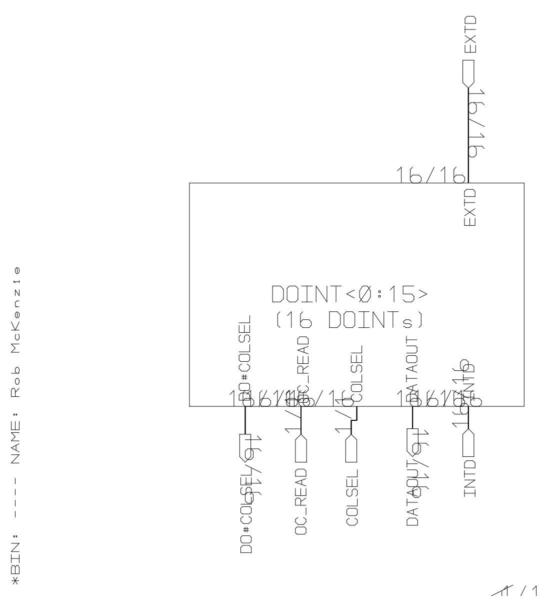
PE#DOINT_SUBLast Changed: Ø1/11/97 at 17:17:42 by jbarth2

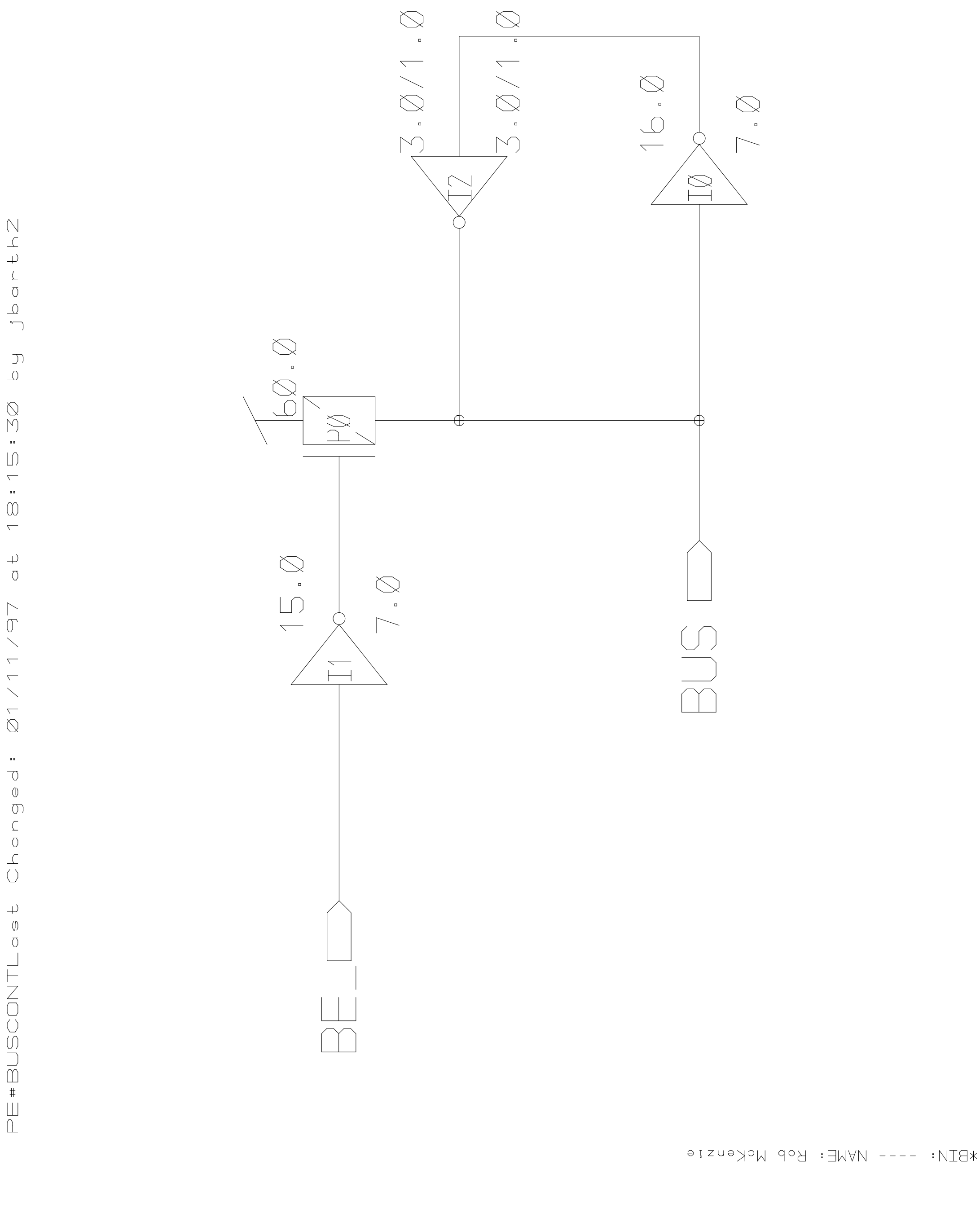


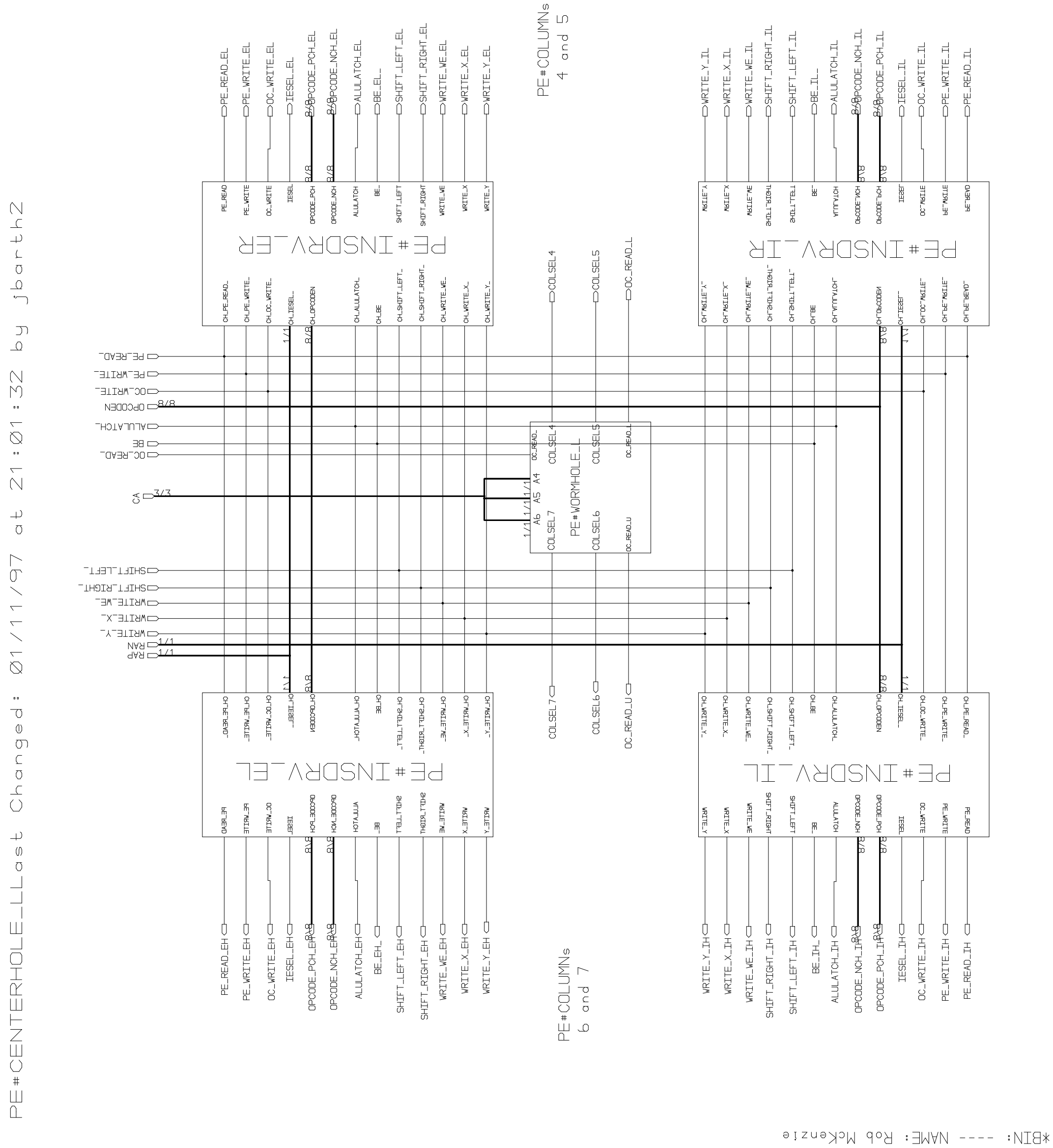
DOINT Last Changed: 01/11/97 at 17:19:01 by jbarth2

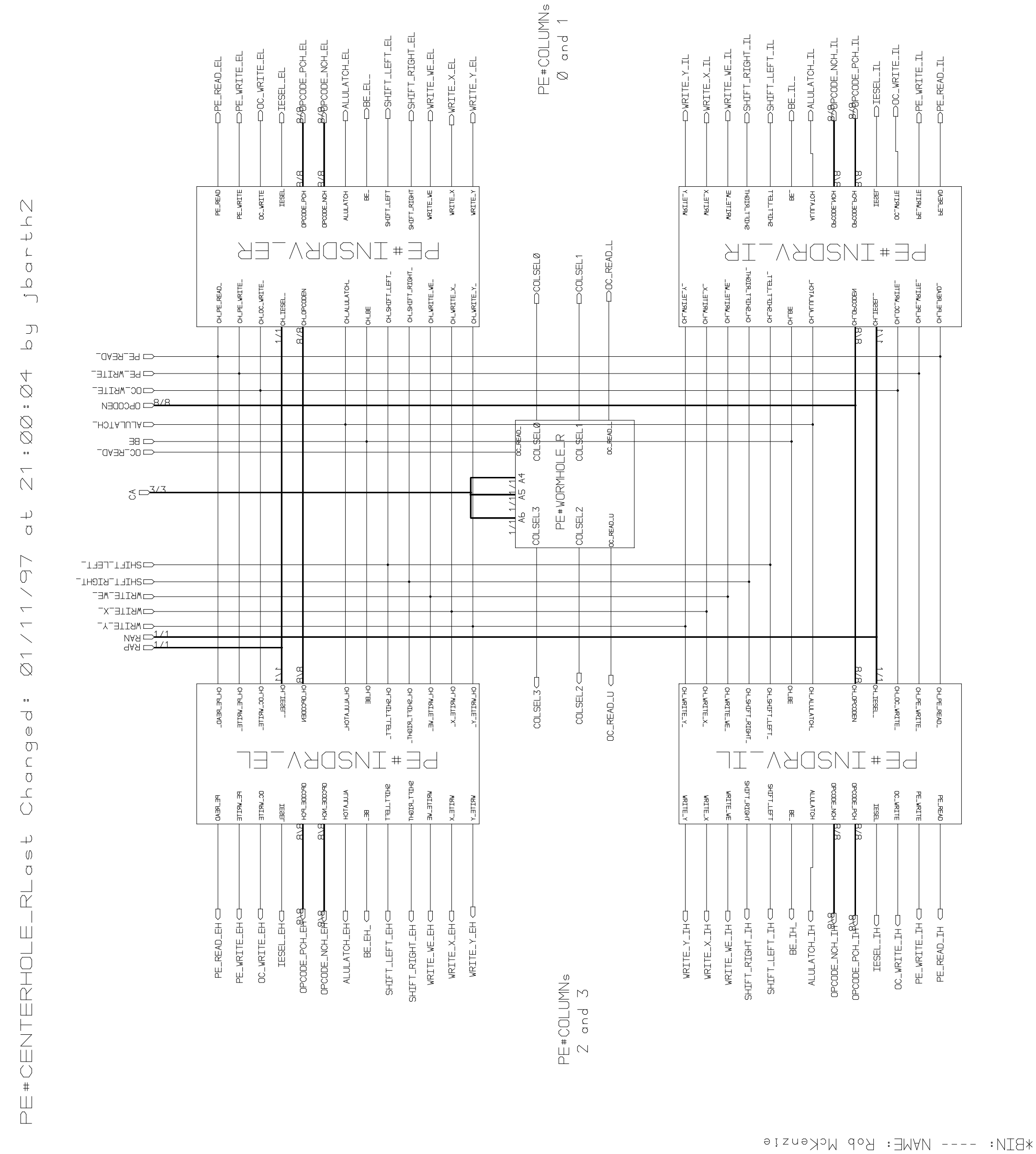


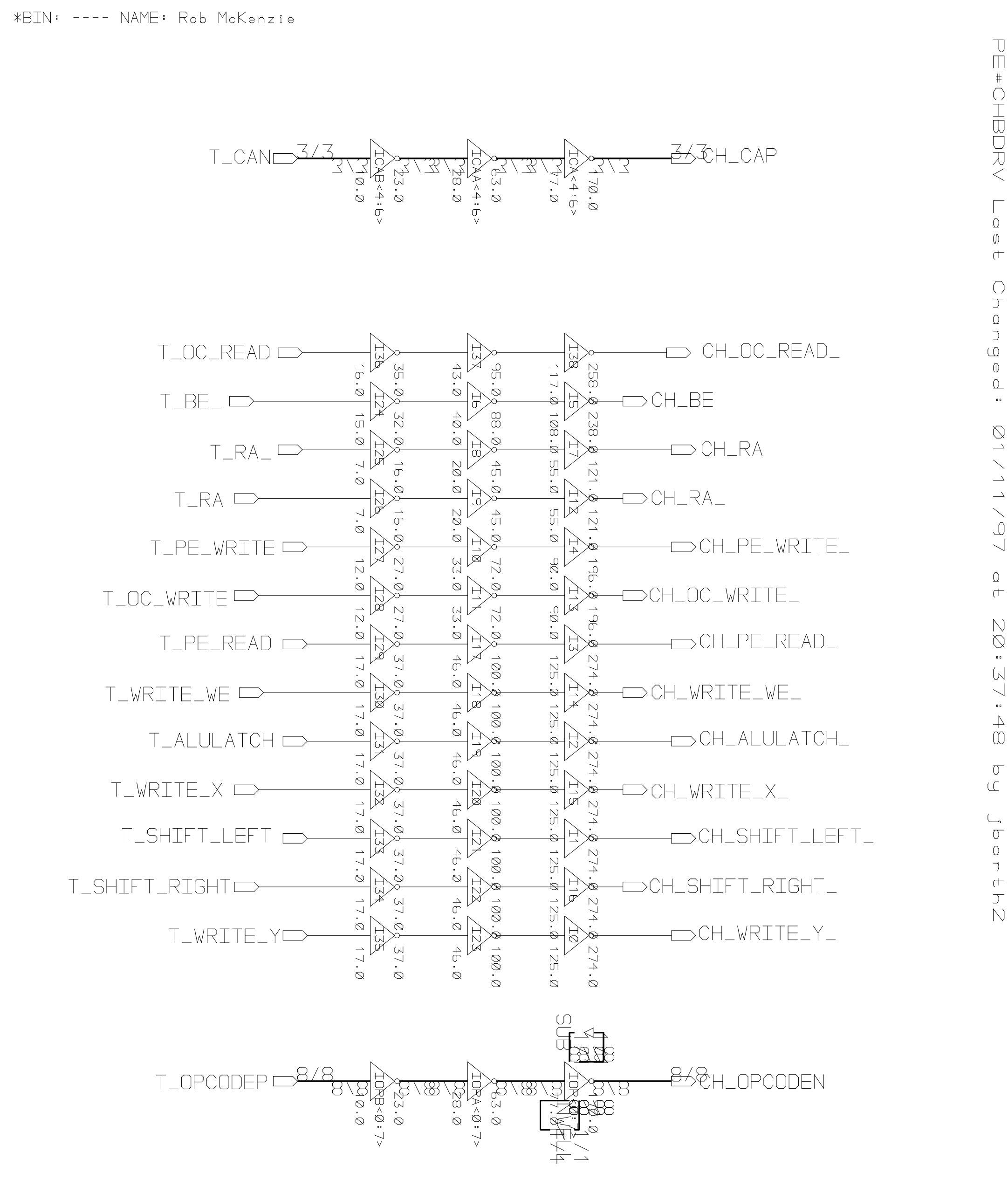
DOINT16 Last Changed: 01/11/97 at 17:21:05 by jbarth2



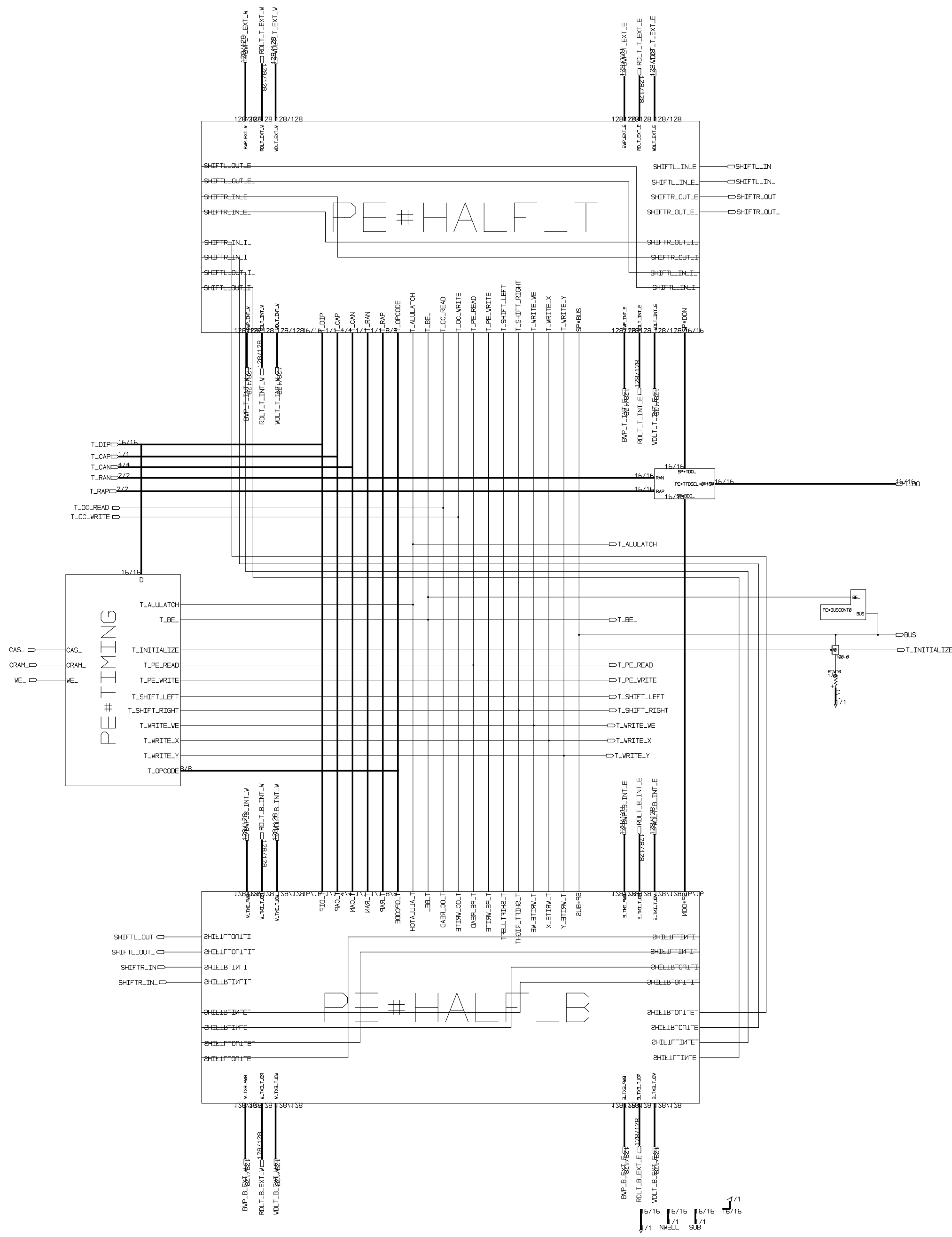

PE#BUSCONTLast Changed: Ø1/11/97 at 18:15:3Ø by jbarth2
3.Ø/1.Ø
3.Ø/1.Ø
16.Ø
7.Ø
I2
IØ
6Ø.Ø
PØ
15.Ø
7.Ø
I1
BUS
BE_
*BIN: ---- NAME: Rob McKenzie


PE#CENTERHOLE_LLast Changed: 01/11/97 at 21:01:32 by jborth2
PE#INSDRV_ER
PE#INSDRV_IR
PE#INSDRV_EL
PE#INSDRV_IL
PE#WORMHOLE_L
PE#COLUMNs 4 and 5
PE#COLUMNs 6 and 7
*BIN: ---- NAME: Rob McKenzie


PE#CENTERHOLE_RLast Changed: Ø1/11/97 at 21:ØØ:Ø4 by jbarth2
PE#COLUMNs Ø and 1
PE#COLUMNs 2 and 3
PE#INSDRV_ER
PE#INSDRV_IR
PE#INSDRV_EL
PE#INSDRV_IL
PE#WORMHOLE_R
*BIN: ---- NAME: Rob McKenzie

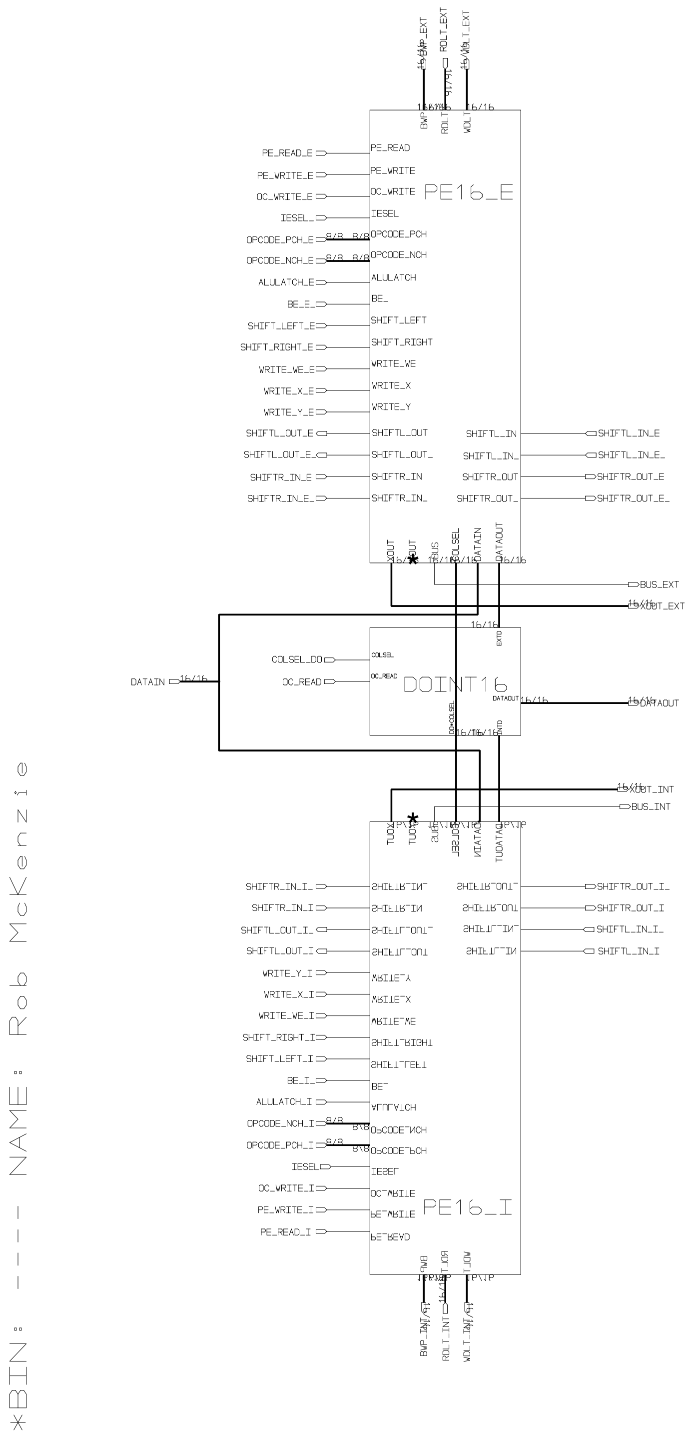

*BIN: ---- NAME: Rob McKenzie
PE#CHBDRV Last Changed: 01/11/97 at 20:37:48 by jbarth2
T_CAN
CH_CAP
T_OC_READ
CH_OC_READ_
T_BE_
CH_BE
T_RA_
CH_RA
T_RA
CH_RA_
T_PE_WRITE
CH_PE_WRITE_
T_OC_WRITE
CH_OC_WRITE_
T_PE_READ
CH_PE_READ_
T_WRITE_WE
CH_WRITE_WE_
T_ALULATCH
CH_ALULATCH_
T_WRITE_X
CH_WRITE_X_
T_SHIFT_LEFT
CH_SHIFT_LEFT_
T_SHIFT_RIGHT
CH_SHIFT_RIGHT_
T_WRITE_Y
CH_WRITE_Y_
T_OPCODEP
CH_OPCODEN

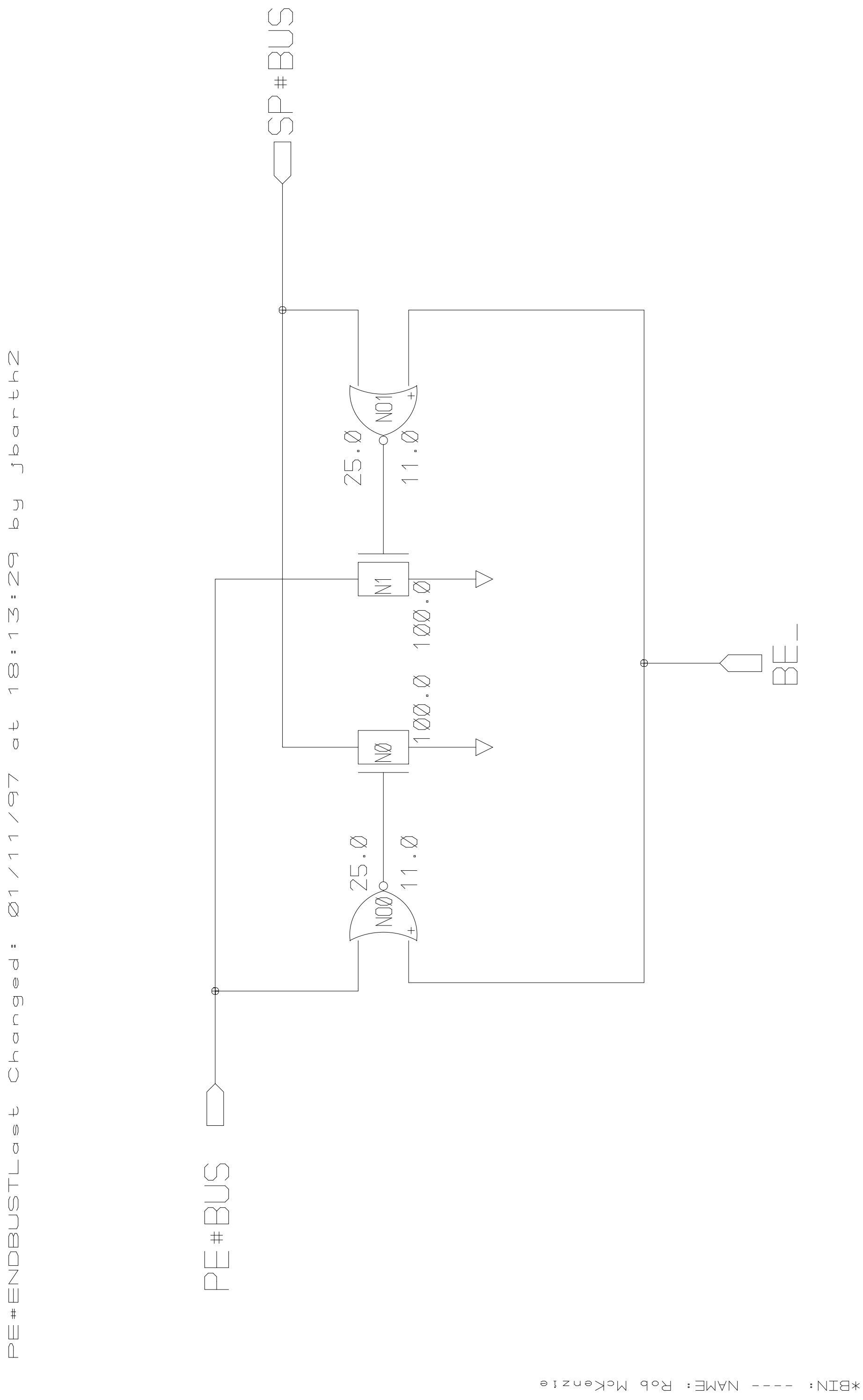
PE#CHIP Last Changed: 01/11/97 at 20:49:24 by jbarth2

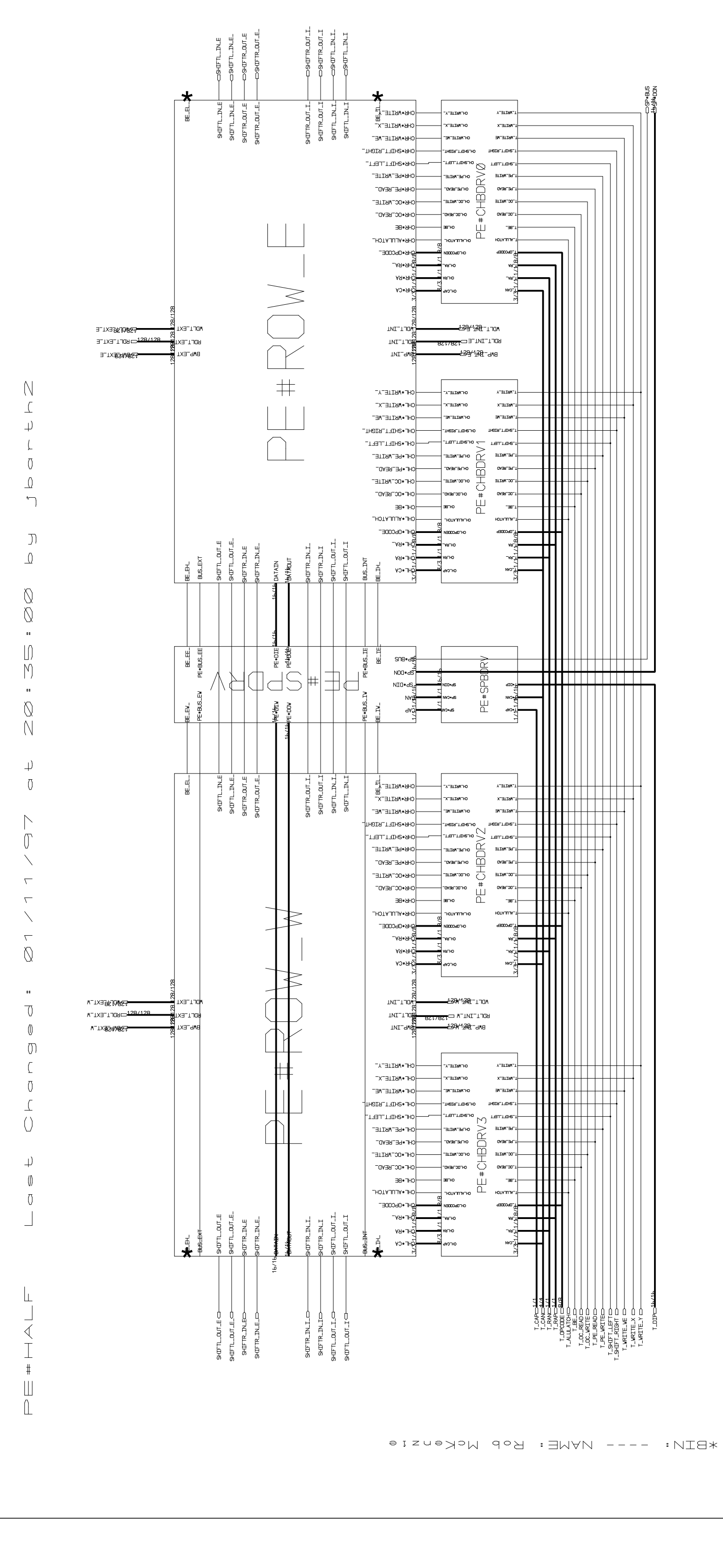


*BIN: --- NAME: Rob McKenzie

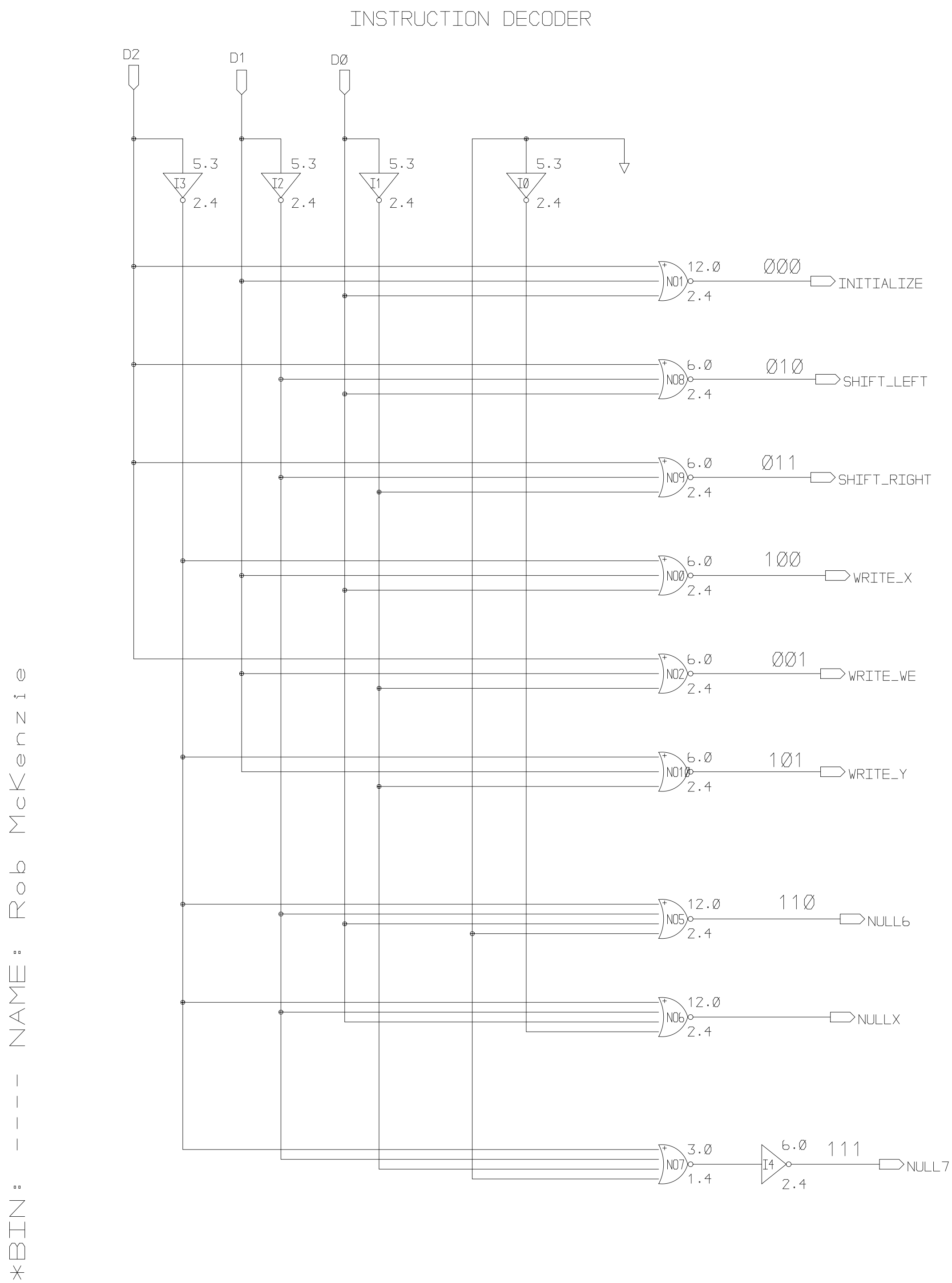
PE#COLUMN Last Changed: 01/11/97 at 16:26:03 by jbarth2



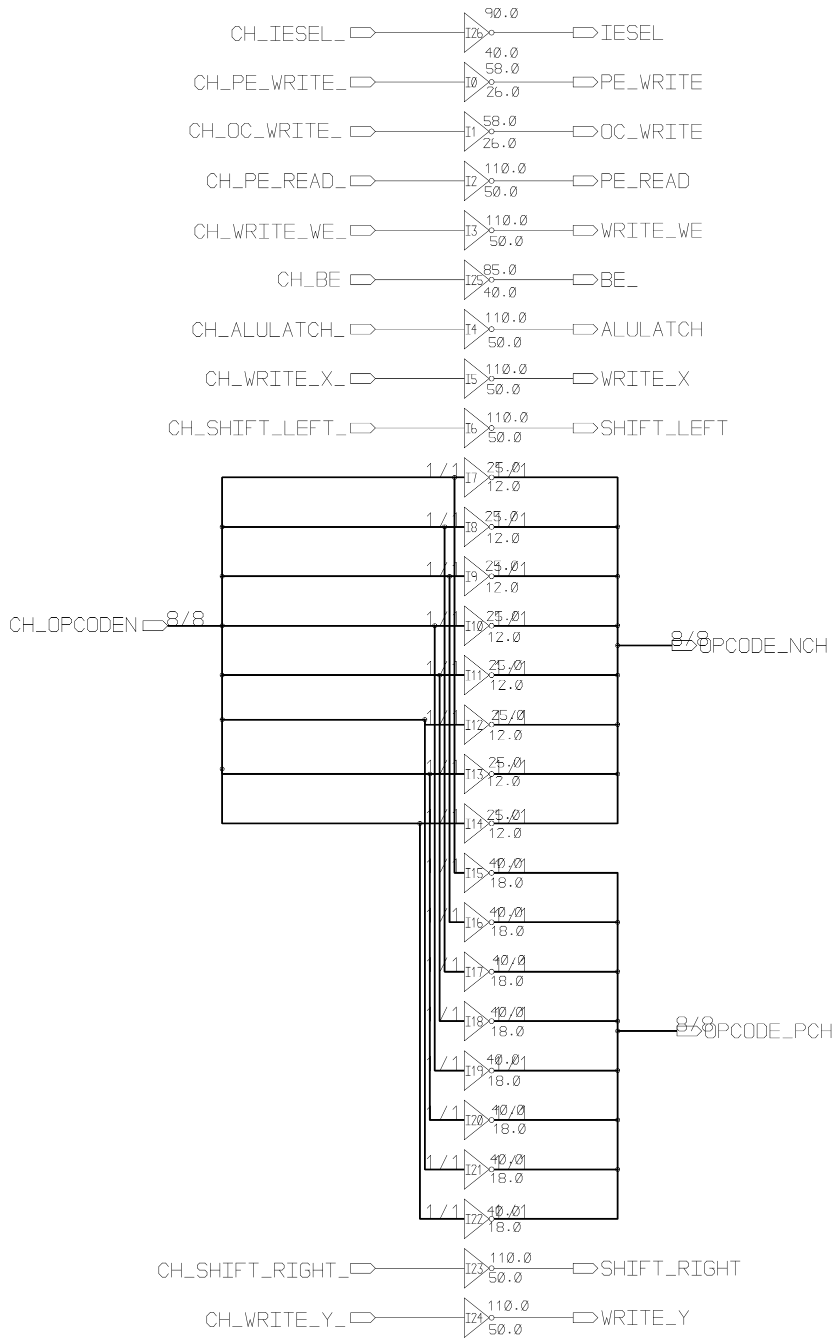

PE#ENDBUSTLast Changed: 01/11/97 at 18:13:29 by jbarth2
SP#BUS
N01
25.0
11.0
N1
100.0
N0
100.0
N00
25.0
11.0
BE_
PE#BUS
*BIN: ---- NAME: Rob McKenzie


PE#HALF Last Changed: 01/11/97 at 20:35:00 by jbarth2
*BIN: ---- NAME: Rob McKenzie
PE#ROW_E
PE#ROW_W
PE#SPDRV
PE#CHBDRV0
PE#CHBDRV1
PE#SPBDRV
PE#CHBDRV2
PE#CHBDRV3

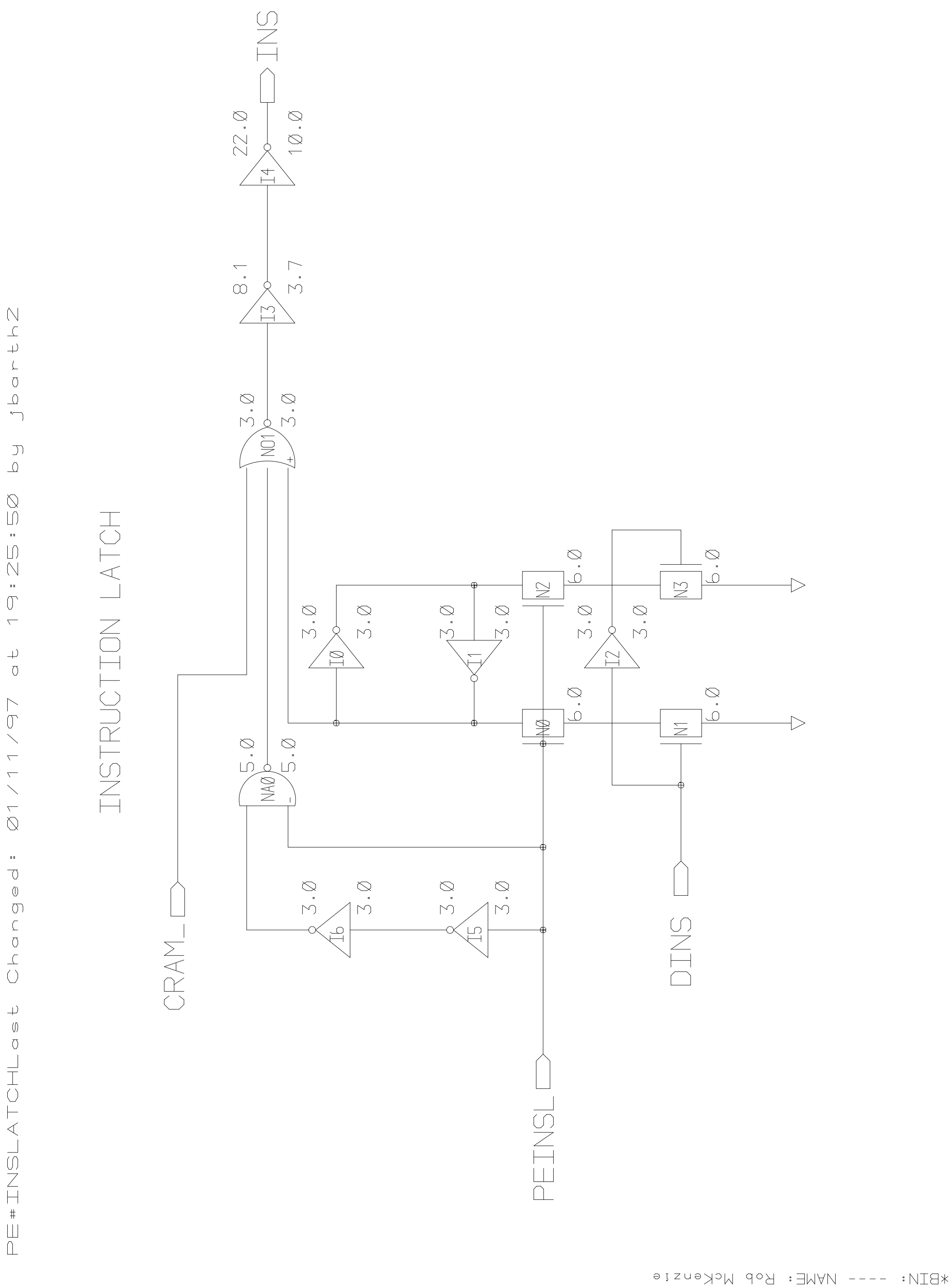
PE#INSDEC Last Changed: 01/11/97 at 19:24:26 by jbarth2



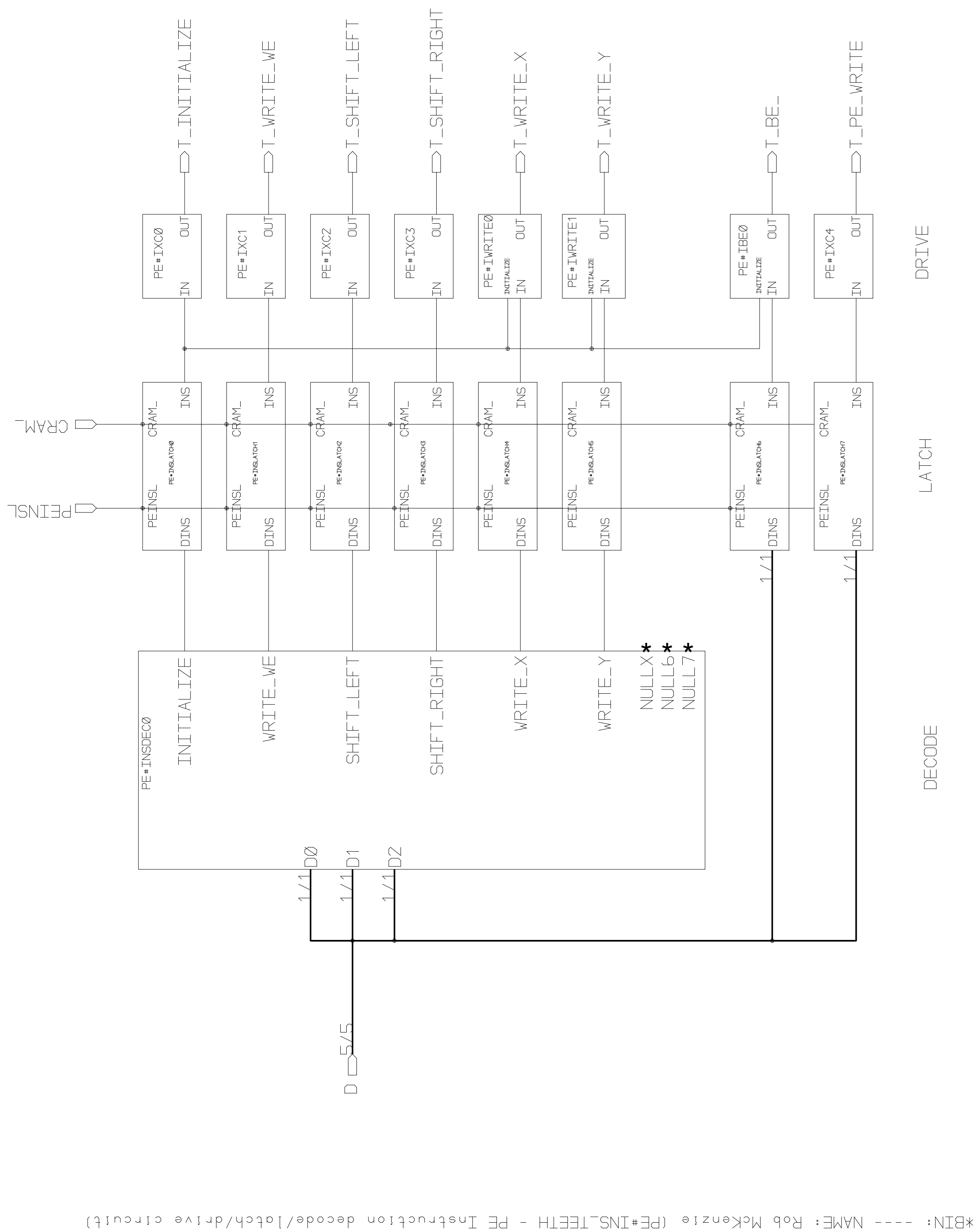
PE#INSDRV Last Changed: 01/11/97 at 20:56:38 by jbarth2



*BIN: ---- NAME: Rob McKenzie

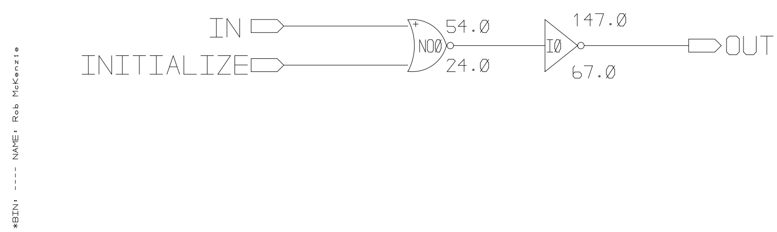


PE#INSLATCHLast Changed: Ø1/11/97 at 19:25:5Ø by jbarth2
INSTRUCTION LATCH
CRAM_
PEINSL
DINS
INS
NAØ
NO1
IØ
I1
I2
I3
I4
I5
I6
NØ
N1
N2
N3
*BIN: ---- NAME: Rob McKenzie


T_INITIALIZE
T_WRITE_WE
T_SHIFT_LEFT
T_SHIFT_RIGHT
T_WRITE_X
T_WRITE_Y
T_BE_
T_PE_WRITE
PE#IXC0
PE#IXC1
PE#IXC2
PE#IXC3
PE#IWRITE0
PE#IWRITE1
PE#IBE0
PE#IXC4
IN
OUT
INITIALIZE
DRIVE
CRAM_
PEINSL
INS
DINS
LATCH
PE#INSDEC0
INITIALIZE
WRITE_WE
SHIFT_LEFT
SHIFT_RIGHT
WRITE_X
WRITE_Y
NULLX
NULL6
NULL7
DECODE
D0
D1
D2
1/1
D
5/5
*BIN: ---- NAME: Rob McKenzie (PE#INS_TEETH - PE Instruction decode/latch/drive circuit)

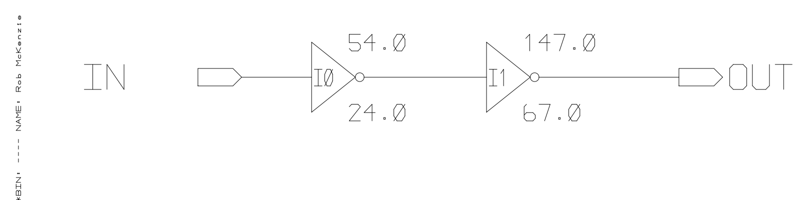
PE*IWRITE Last Changed: 01/11/97 at 19:28:53 by jbarth2

WRITE_X/WRITE_Y Cross-Chip Instruction Driver



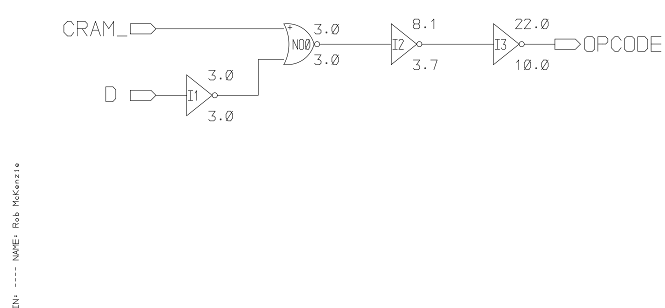
PE*IXC Last Changed: 01/11/97 at 19:26:43 by jbarth2

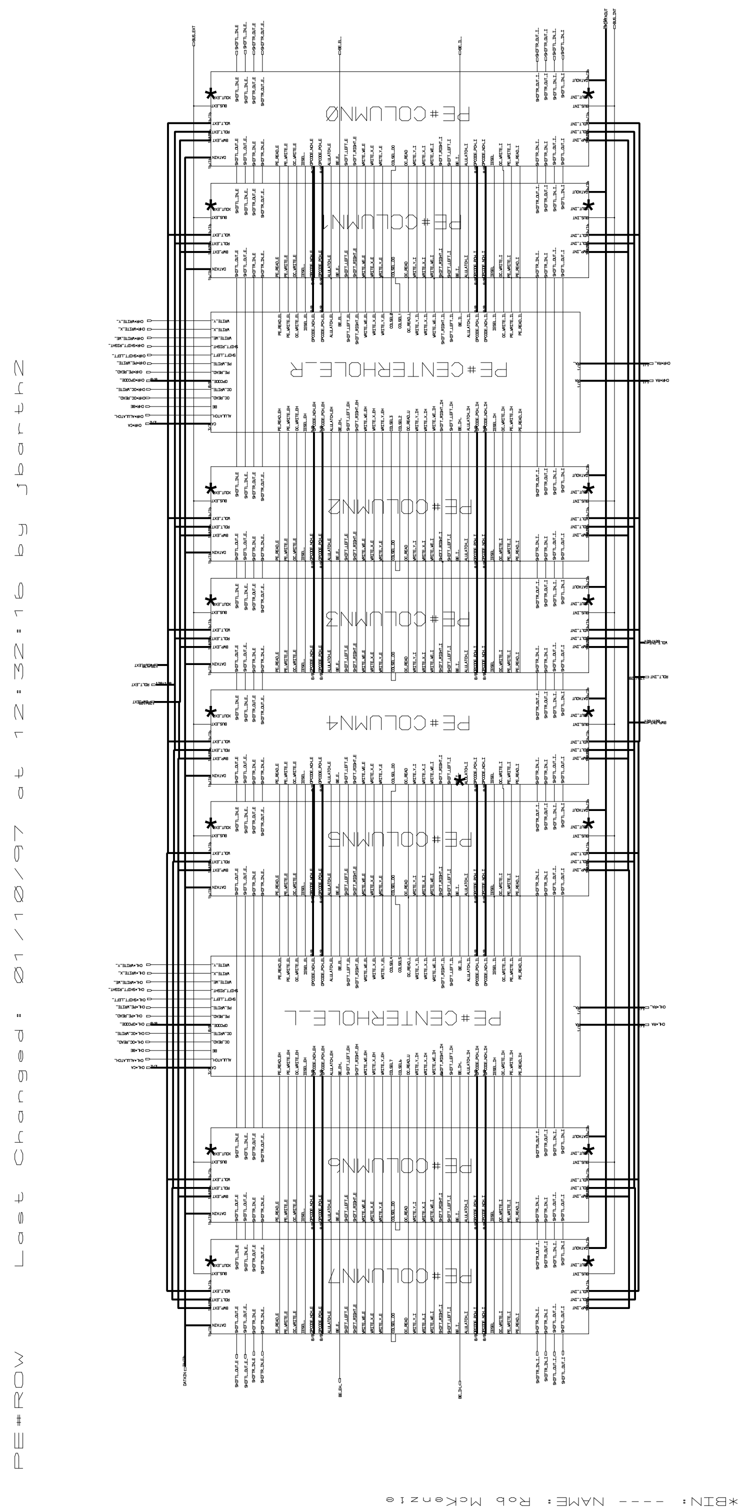
cross-chip PEinstruction driver

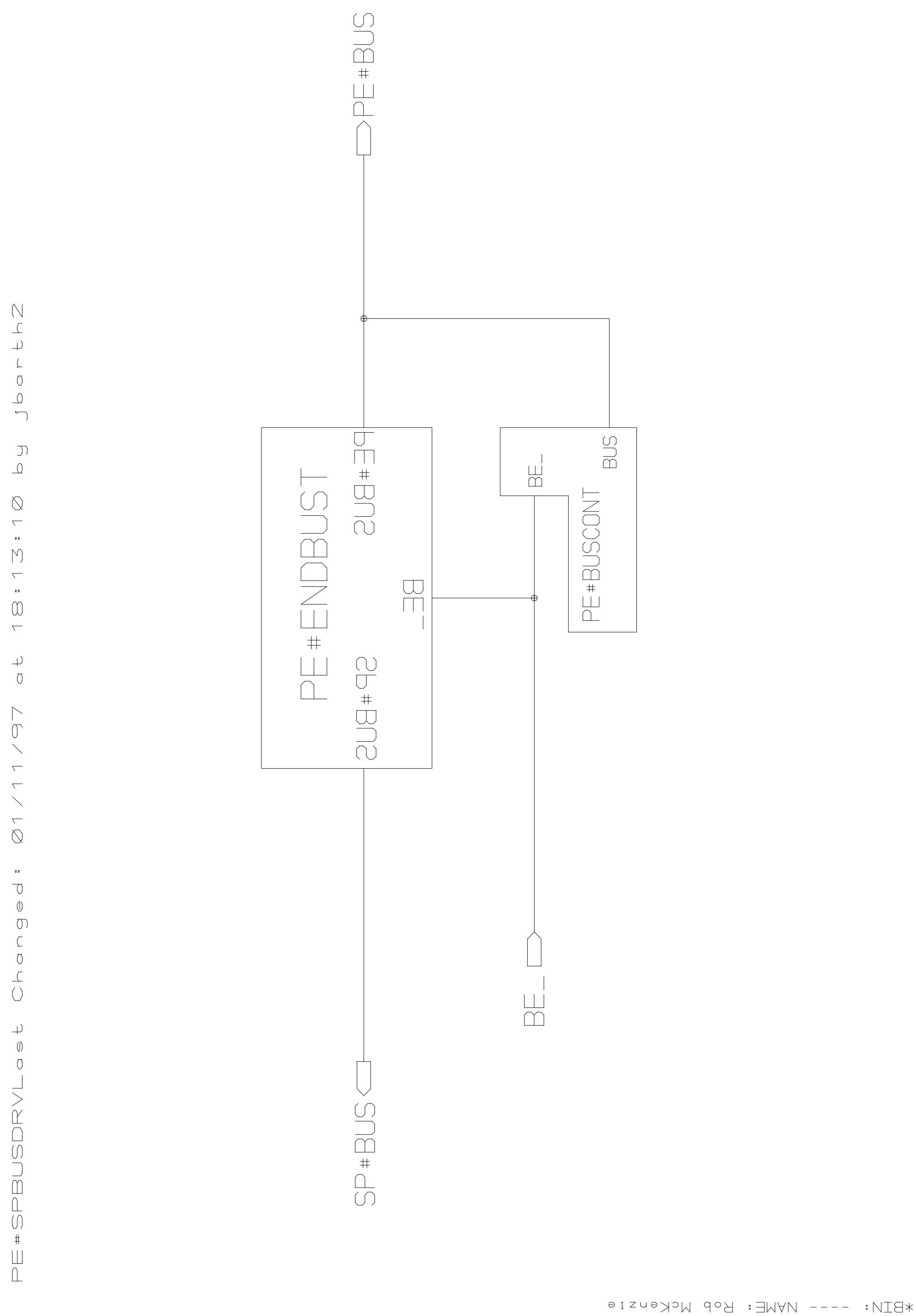


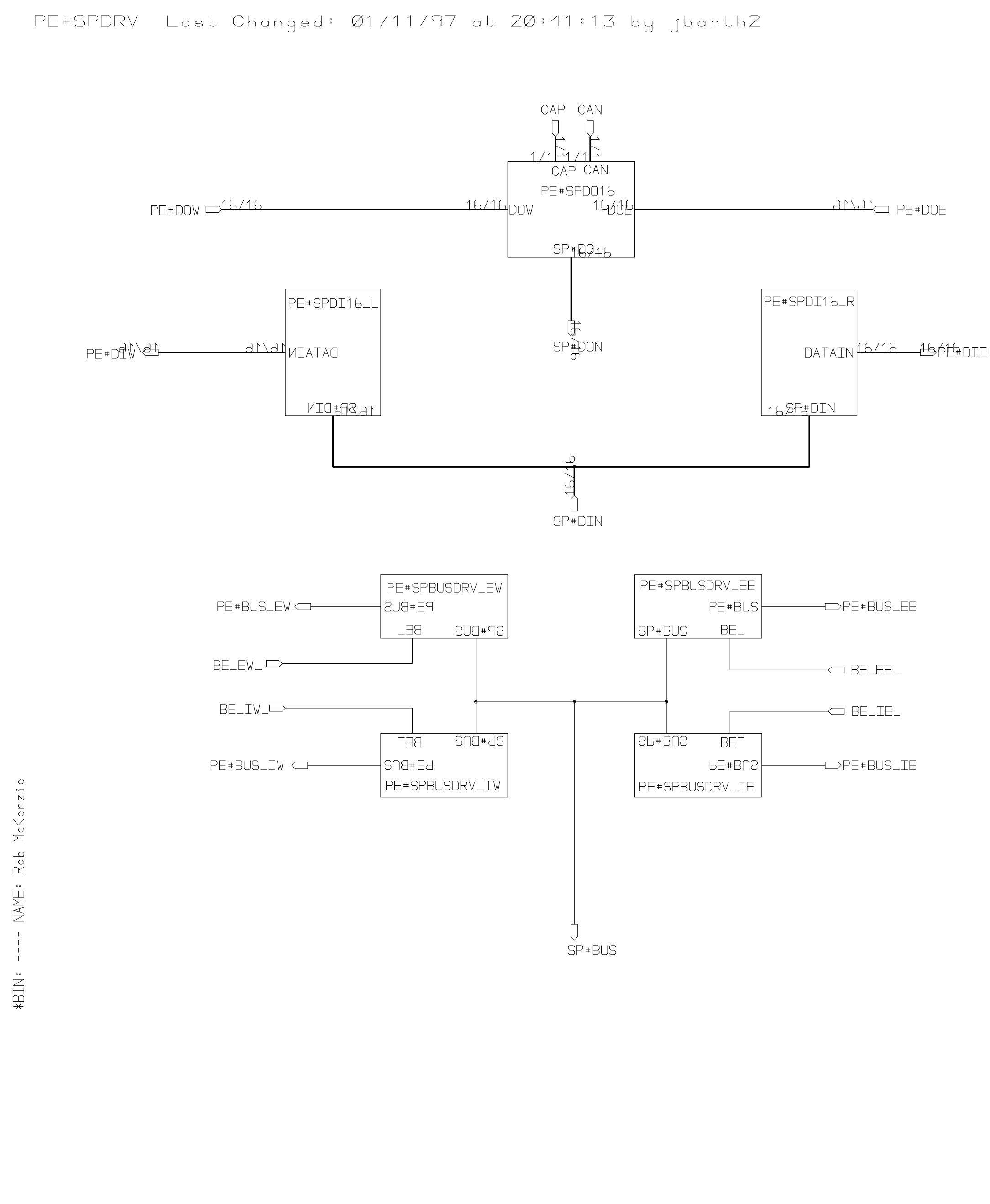
PE*OPLATCHLast Changed: 01/11/97 at 19:34:56 by jbarth2

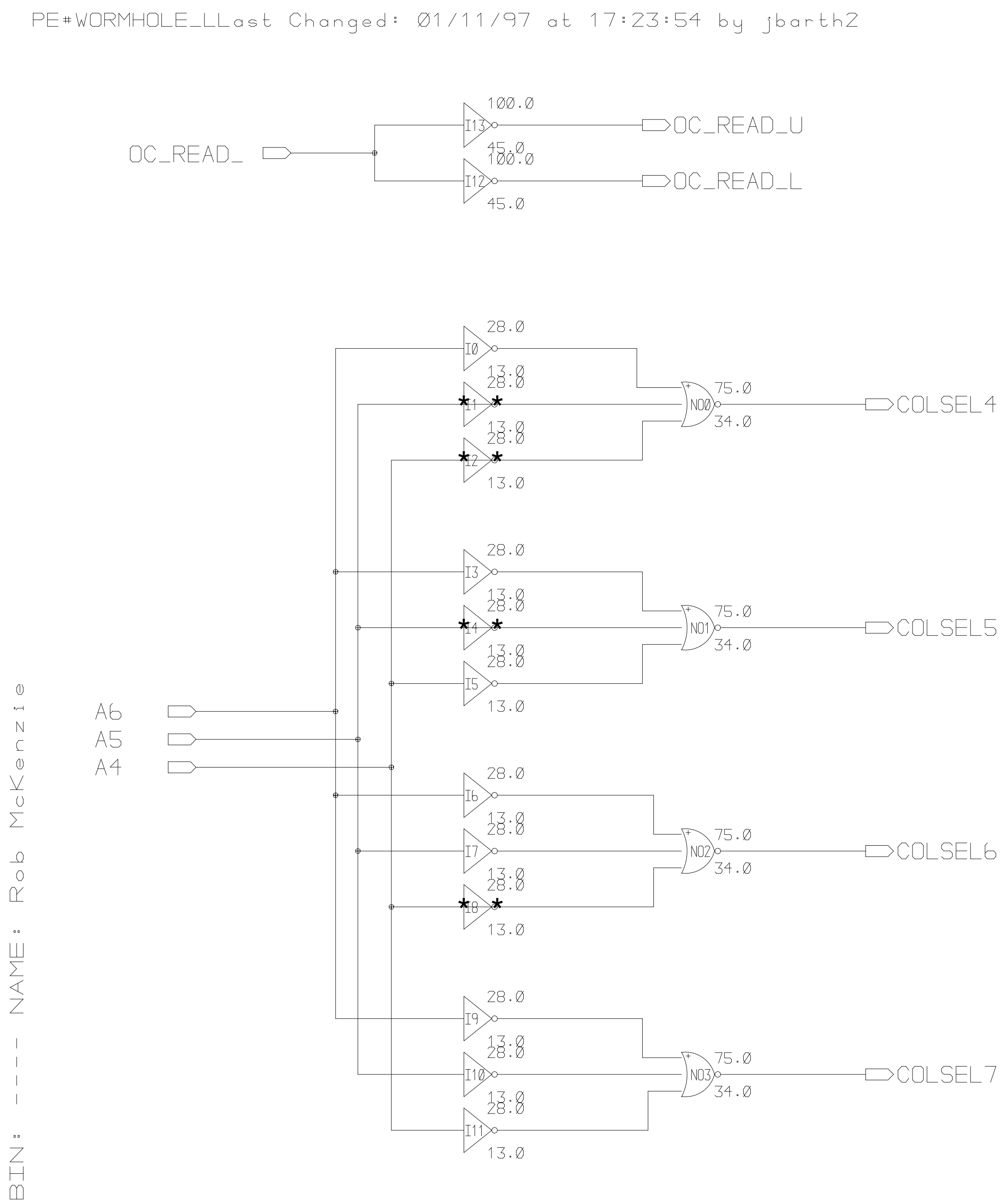
OPCODE Latch



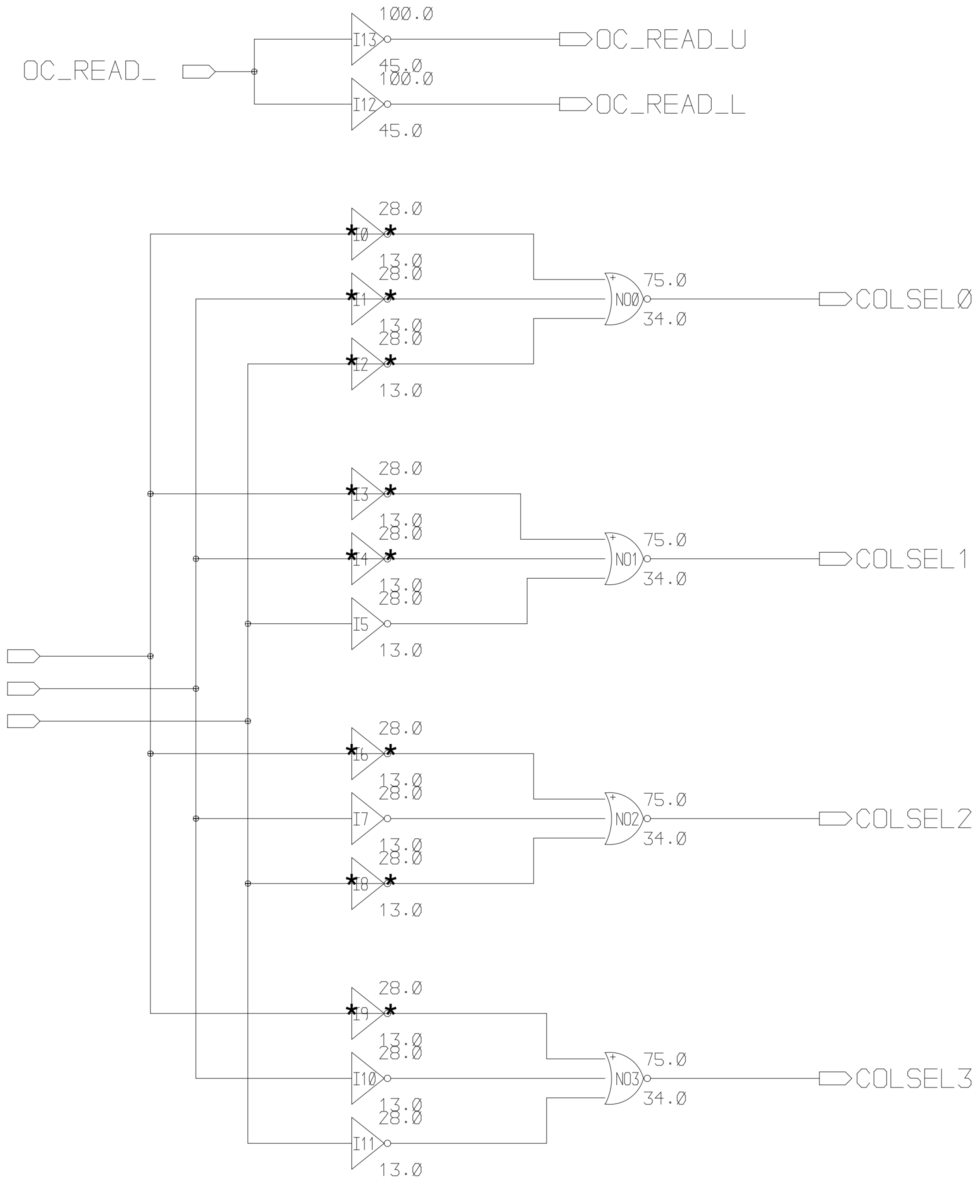

PE#ROW Last Changed: 01/10/97 at 12:32:16 by jbarth2
PE#COLUMN0
PE#COLUMN1
PE#CENTERHOLE_R
PE#COLUMN2
PE#COLUMN3
PE#COLUMN4
PE#COLUMN5
PE#CENTERHOLE_L
PE#COLUMN6
PE#COLUMN7
*BIN: ---- NAME: Rob McKenzie


PE#SPBUSDRVLast Changed: Ø1/11/97 at 18:13:1Ø by jbarth2
PE#BUS
SP#BUS
PE#ENDBUST
PE#BUS
SP#BUS
BE_
PE#BUSCONT
BE_
BUS
BE_
*BIN: ---- NAME: Rob McKenzie


PE*SPDRV Last Changed: 01/11/97 at 20:41:13 by jbarth2
CAP
CAN
PE*SPDO16
PE*DOW
DOW
DOE
PE*DOE
SP*DO
PE*SPDI16_L
PE*DIW
DATAIN
SP*DON
PE*SPDI16_R
DATAIN
PE*DIE
SP*DIN
SP*DIN
PE*SPBUSDRV_EW
PE*BUS_EW
BE_EW_
BE_IW_
PE*BUS_IW
PE*SPBUSDRV_IW
PE*SPBUSDRV_EE
PE*BUS
SP*BUS
BE_
PE*BUS_EE
BE_EE_
BE_IE_
PE*BUS_IE
PE*SPBUSDRV_IE
SP*BUS
*BIN: ---- NAME: Rob McKenzie

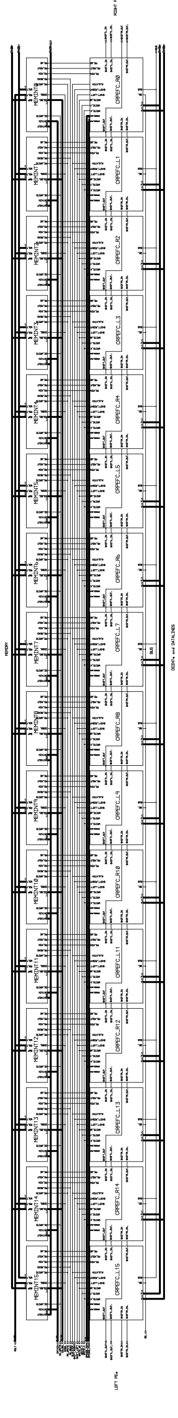

PE#WORMHOLE_LLast Changed: Ø1/11/97 at 17:23:54 by jbarth2
OC_READ_
I13
I12
100.0
45.0
OC_READ_U
OC_READ_L
A6
A5
A4
I0
I1
I2
I3
I4
I5
I6
I7
I8
I9
I10
I11
28.0
13.0
NO0
NO1
NO2
NO3
75.0
34.0
COLSEL4
COLSEL5
COLSEL6
COLSEL7
*BIN: ---- NAME: Rob McKenzie

PE#WORMHOLE_RLast Changed: Ø1/11/97 at 17:29:57 by jbarth2

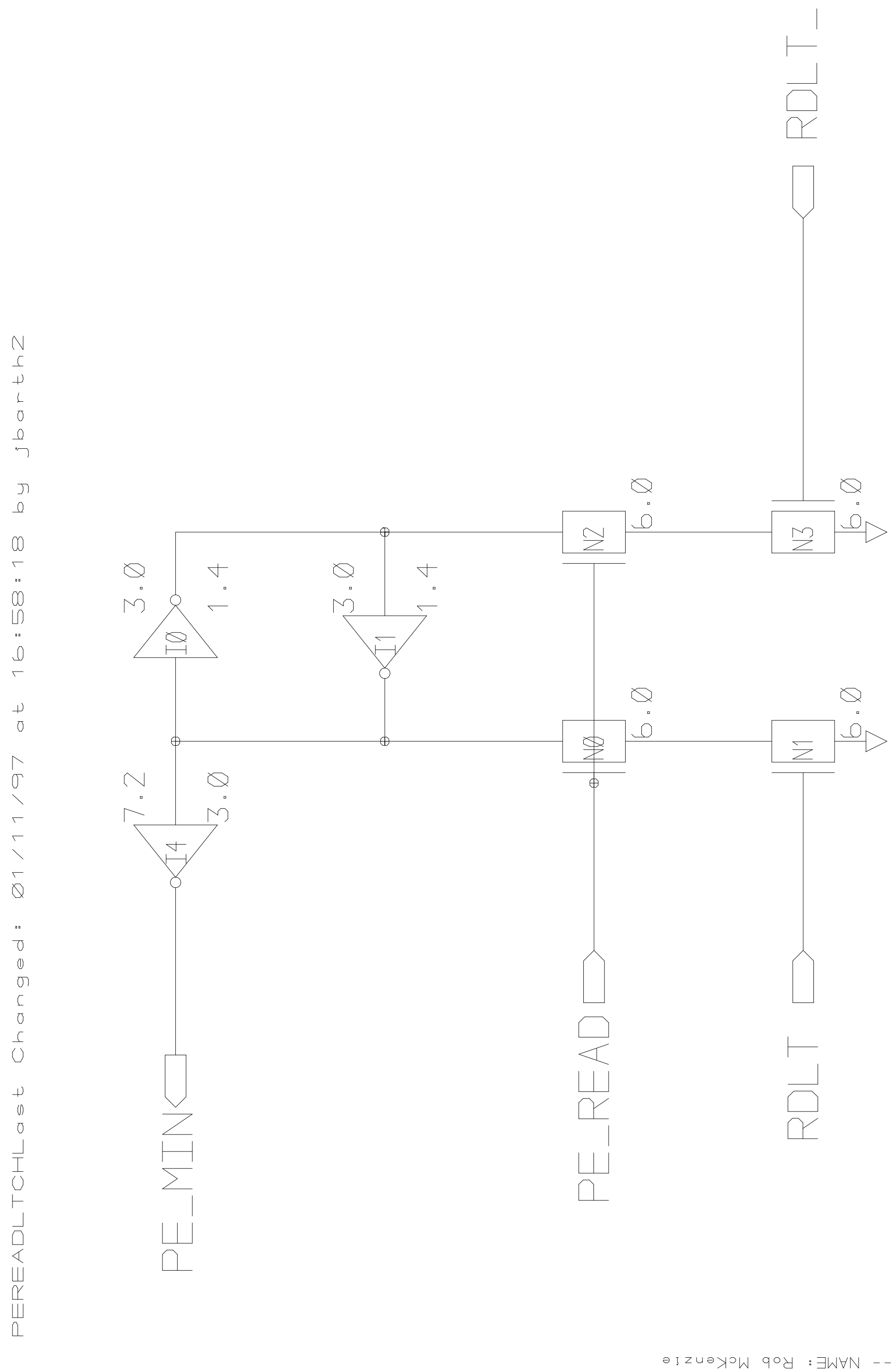


*BIN: ---- NAME: Rob McKenzie

PE16 Last Changed: Ø1/11/97 at 16:26:35 by jbarth2



*BIN: ---- NAME: Rob McKenzie


PEREADLTCHLast Changed: Ø1/11/97 at 16:58:18 by jbarth2
PE_MIN
PE_READ
RDLT
RDLT_
I4
7.2
3.Ø
I0
3.Ø
1.4
I1
3.Ø
1.4
NØ
6.Ø
N1
6.Ø
N2
6.Ø
N3
6.Ø
*BIN: ---- NAME: Rob McKenzie