A Video Controller and Distributed Frame Buffer for the J-Machine

by

Eric Lawrence McDonald

S.B., M.I.T.
(1990)

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE
IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

Massachusetts Institute of Technology
January 1995

©1995 Massachusetts Institute of Technology. All rights reserved.

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
January, 1995

Certified by ______________________________
Prof. William J. Dally
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ______________________________
Prof. Frederic Morgenthaler
Chairman, Departmental Committee on Graduate Students

Eng.

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

APR 13 1995

LIBRARIES

A Video Controller and Distributed Frame Buffer for the J-Machine

by

Eric Lawrence McDonald

Submitted to the
Department of Electrical Engineering and Computer Science
on January 20, 1995, in partial fulfillment of
the requirements for the Degree of Master of Science in
Electrical Engineering and Computer Science

Abstract

A high-bandwidth video system for the J-Machine concurrent computer has been developed. The system integrates a physically distributed frame buffer with a commercial video controller chip to provide a unique research tool in the field of computer graphics. Combined with the fine-grain programming mechanisms of the J-Machine, the flexibility of the system allows research into a number of distributed graphics algorithms without the dampening effects of low video memory bandwidth. The video system is scalable to provides a range of price/performance tradeoffs. Applications for the J-Machine video system include scientific visualization, animation, and high-speed photorealistic rendering.

Thesis Supervisor: Prof. William J. Dally
Title: Associate Professor of Electrical Engineering and Computer Science

Keywords: Video, Distributed, Graphics, J-Machine.

Acknowledgments

Many people have given me invaluable help and guidance throughout my years here at M.I.T, and I'd like to offer them my whole-hearted thanks.

I would first like to thank my thesis advisor, Prof. Bill Dally, for giving me such a challenging research project. His wealth of knowledge and expertise helped me though some critical design decisions, and yet he kept enough distance to let me find solutions to the exciting problems on my own.

Several members of the CVA group offered their wisdom as well. My first few months with the group were spent under the tutelage of Mike Noakes, who was more than generous with his time when I was trying to learn as much as possible about the J-Machine hardware. Toward the end of the project, the debugging advice offered by Andrew Chang helped me untangle some crucial puzzles. And the advice, opinions, and humor of Stuart Fiske and Rich Lethin have helped me put some perspective on where I've been and where I'm going.

The faculty at M.I.T. is tremendous, and I would like to extend my particular thanks to Profs. Don Troxel and Al Drake. As a TA in their courses, I experienced some of my most worthwhile and rewarding moments here at the institute.

Many, many thanks go out to my family. The love and support from my parents have given me all the confidence I need to make a difference in life. My brother and sister have already demonstrated the successful lives that a loving family can foster. The warmth of my family, and Jellybean (my cat, not the Machine), have always been able to keep my spirits aloft.

My deepest gratitude goes out to my best friend Dave. His friendship, tolerance, insight, and humor have helped me to appreciate the better parts of life. The experiences we've shared since high school have instilled in me the sense of optimism and hope that I possess today. It is with excitement and wonder that I wait for the future to unfold.

To Mom and Dad

Contents

1 Introduction **13**
1.1 Overview . . . 13
1.2 Background . . . 14
1.2.1 The J-Machine . . . 14
1.2.2 The Demands of High-speed Rendering . . . 15
1.3 Video System Hardware Overview . . . 16
1.4 Prior Work . . . 17
1.5 Results . . . 17
1.6 Thesis Outline . . . 18

2 Pixel Storage Node **19**
2.1 Purpose . . . 19
2.2 Module Descriptions . . . 20
2.2.1 MDP Module . . . 21
2.2.2 DRAM Module . . . 22
2.2.3 FIFO Module . . . 23
2.2.4 MREAD Module . . . 26
2.2.5 MUX Module . . . 28
2.2.6 VRAM Module . . . 30
2.3 Fabrication . . . 35

3 Video Controller **37**
3.1 Purpose . . . 37
3.2 Module Descriptions . . . 37
3.2.1 MDP Module . . . 38
3.2.2 DRAM Module . . . 39
3.2.3 FIFO Module . . . 39
3.2.4 ERM Module . . . 42
3.2.5 XLM Module . . . 44
3.2.6 SCM Module . . . 47
3.2.7 TIM Module . . . 54
3.2.8 MPU Module . . . 56
3.2.9 Bt463 . . . 58
3.3 Fabrication . . . 58

4 Configuring the Video System **59**
4.1 Number of Pixel Storage Nodes 59
4.2 Display Resolution 60
4.3 Starting an Application 60
4.4 Initializing the XLM 62
4.4.1 Single-line frame buffer 63
4.4.2 Single-screen frame buffer 63
4.4.3 Multiple-screen frame buffer 65
4.5 Initializing the SCM 65
4.6 Classes of Pixel Messages 65
4.6.1 Address/Pixel List 66
4.6.2 Raster Operations 66
4.6.3 Proxy Messages 67
4.7 Increasing Effective Storage 68
4.8 Hardware Scrolling and Stretching 68
4.9 Stereo Monitors 69

5 Conclusion **71**
5.1 Summary 71
5.2 Further Work 72

A PSN Schematics and PAL Files **73**

B VC Schematics and PAL Files **108**

List of Figures

1.1 Small Video System for the J-Machine . . . 16

2.1 Pixel Storage Node Hardware Modules . . . 20
2.2 PSN FIFO Block Diagram . . . 24
2.3 Logic Analyzer Plot of EMI Write Cycle . . . 25
2.4 MREAD Block Diagram . . . 27
2.5 MUX Block Diagram . . . 29
2.6 Single-bit VRAM Layer . . . 30
2.7 VRAM Block Diagram . . . 31
2.8 Flow Chart for VRAM Controller . . . 32
2.9 Timing Diagram - VRAM Refresh Cycle . . . 32
2.10 Timing Diagram - VRAM Row Transfer . . . 33
2.11 Timing Diagram - VRAM Write . . . 34
2.12 Timing Diagram - VRAM Read . . . 34

3.1 Video Controller Hardware Modules . . . 38
3.2 VC FIFO Block Diagram . . . 40
3.3 Data transfer between FIFO and MPU . . . 41
3.4 ERM Block Diagram . . . 42
3.5 XLM Block Diagram . . . 45
3.6 Shift Clock Timing for a One-Of-Four PSN . . . 49
3.7 Shift Clock Timing for a One-Of-Eight PSN . . . 49
3.8 General Structure of SCM module . . . 51
3.9 Preventing LdClk Timing Violations with a Delay Line . . . 54
3.10 MPU Block Diagram . . . 57

4.1 Displaying the Same Color Using True-Color and Pseudo-color Modes . . . 62
4.2 Single-buffer VRAM Usage with Four PSNs per Scan Line . . . 64
4.3 Multi-buffer VRAM Usage with Eight PSNs per Scan Line . . . 64
4.4 Increasing Effective Storage through Pixel Compaction . . . 69
4.5 Using the XLM to Scroll an Image Vertically . . . 69

List of Tables

2.1 Effects of Increasing the Number of PSNs 22
2.2 EMI Address Multiplexing Scheme 24
2.3 MREAD Memory Map 26
2.4 MUX Selection 29

3.1 FIFO Interpretation of EMI Address Bits 17 & 16 40
3.2 ERM Memory Map 43
3.3 XLM Usage of Data and Address Bits From FIFO 46
3.4 SCM Usage of Data and Address Bits From FIFO 51
3.5 MPU Usage of Data Bits From FIFO 56

Chapter 1

Introduction

My goal is simple. It is complete understanding of the universe, why it as it is and why it exists as all.
– Stephen Hawking

There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another which states that this has already happened.
– Douglas Adams in *The Restaurant at the End of the Universe*

1.1 Overview

This thesis describes a video system that has been developed for the J-Machine concurrent computer. The video system, also called the Distributed Frame Buffer (DFB), allows the J-Machine to be used as a research tool in the field of distributed graphics processing. The DFB provides the J-Machine with high-bandwidth, multiple-buffer, high-resolution video capability required for applications such as scientific visualization, animation, and photorealistic graphics rendering.

The J-Machine itself is a fine-grain, concurrent computer that provides efficient mechanisms to support several proposed models of concurrent computation. While it has always been possible to use the J-Machine for the calculations involved in graphics applications, the results previously had to be fetched with significantly low-bandwidth bit-serial retrieval. The DFB allows the rapid display of information-rich video generated by the J-Machine processor array.

The DFB is both distributed and scalable. Distributing the physical location of the pixel frame buffers eliminates the bottleneck created by a single video bus and thus provides greater bandwidth into the video system. The scalable nature of the DFB allows J-Machine owners to balance cost against performance in accordance with their particular graphics requirements. The DFB is extremely flexible, with many configuration options under software control.

Before discussing the video system in detail, some background information is needed on both the J-Machine architecture and modern-day graphics requirements.

1.2 Background

1.2.1 The J-Machine

The DFB has been targeted for use with the J-Machine, a fine-grain concurrent computer built by MIT's Concurrent VLSI Architecture Group [9]. A 1024-node version of the J-Machine has been operating since mid-1993.

The J-Machine Philosophy

The J-Machine is a concurrent computer; rather than optimizing execution on a single node, the emphasis is on dividing a job efficiently across many nodes. Moreover, it is a *fine-grain* concurrent computer, allowing programs to be efficiently partitioned into tasks (with an average size of a few dozen instructions) and data objects (with an average size of a few dozen words). The J-Machine provides hardware mechanisms that support several distinct concurrent programming models. For example, the hardware provides efficient synchronization and object translation across multiple nodes.

The processing elements in the J-Machine, called Message Driven Processors (MDPs), are connected through a very low latency, synchronous, three-dimensional mesh network. Since inter-node communication in a fine-grain concurrent computer occurs as frequently as procedure-calling in sequential programs, low overhead for message sends is desirable. To this end, MDP message sends are user-level instructions and do not require bulky system-level calls. Furthermore, the delivery mechanism and message buffering is performed by dedicated logic on the MDP chip, freeing up the main processor for program execution.

Although parallel graphics processing is becoming commonplace, very little research in the field has been done with the fine granularity provided by the J-Machine. The J-Machine and its video system now provide fertile ground for the study of this model.

The J-Machine Hardware

The J-Machine contains up to 65,536 computing nodes, each with an MDP and 256 KWords of external DRAM memory. An MDP chip consists of an ALU, register files, 4K 36-bit words of internal memory, a network interface, and control logic for the external DRAM. The network that links the MDP nodes is provably deadlock-free.

An MDP also contains a diagnostic port used for initializing the chip, halting and examining its state, and reading and writing to its memory. Although any MDP in the J-Machine array can be individually accessed through its diagnostic port, such access is bit-serial and thus slow. It is through this serial port that images from a previous ray-tracing experiment [4] were retrieved.

A single J-Machine processor board can accommodate 64 MDP nodes. Within a board, routing of the x and y network channels is accomplished with copper traces. Between boards, network routing takes place over ribbon cables (x and y dimensions) and elastomeric

connectors (z dimension). The J-Machine chassis can accommodate up to 16 processor boards with an additional 32 peripheral boards.

1.2.2 The Demands of High-speed Rendering

Modern graphics applications such as 3-dimensional rendering, photorealistic animation, and scientific visualization demand far higher performance than non-specialized computing environments can suitably provide. Graphics users must often accept either a slow response time or lower resolution to accomplish work at a tolerable rate. Two obstacles need to be overcome to meet the demands of today's high-performance graphics: pixel processing power and frame buffer memory bandwidth [5].

Pixel processing requirements

The rendering of three-dimensional scenes relies heavily on two fundamental types of data processing: *geometry processing* is used to transform the objects in a scene according to the observer's viewpoint, and *rasterization* is needed to scan-convert the transformed objects into pixel representations. The amount of such computation needed in a real-time high-quality animation sequence can be staggering. For example, rendering a high-quality Gouraud-shaded scene of 100,000 polygons updated at 30 Hz requires about 350 million floating-point operations per second and 750 million integer operations per second [7]. Standard uniprocessor designs simply cannot satisfy such computational requirements.

An obvious way to overcome the processing barrier is to distribute the computational problem among multiple processors. This can be accomplished with parallel processing, hardware pipelining, or a combination of both techniques.

A video system that used a purely parallel approach was proposed as early as 1979 at UNC [6]. In this system, all polygons were broadcast to a set of processing elements, but each processing element only computed those pixels for which it was responsible. This parallel processing was used for both geometry processing and rasterization.

More often, specialized hardware pipelines are used in conjunction with parallel processing. In the Pixel Machine [10], a pipeline of nodes was employed to perform the intrinsically-sequential rendering operations that would otherwise reduce the efficiency of its parallel array of processors. Both the pipeline unit and parallel array were connected to each other and a host computer through a single bus, which posed a potential bottleneck. A system developed at SGI [1] used a more tightly integrated sequence of hardware pipeline and parallel processing stages. However, as is typical with designs containing hardware pipelines, this system was somewhat constrained to a particular rendering algorithm.

The J-Machine video system uses a more flexible approach to satisfy the pixel processing requirements. The entire J-Machine processor array is used to perform pixel processing calculations. The nodes are grouped into functional units which can then be linked in a pipeline or parallel array. The allocation of nodes to the stages of the rendering algorithm is arbitrary, as is the rendering algorithm itself. No specialized rendering hardware is used to perform the data processing. Processing can be done at either the pixel level or object level. However, this flexibility brings with it the non-trivial challenges of synchronization and load balancing.

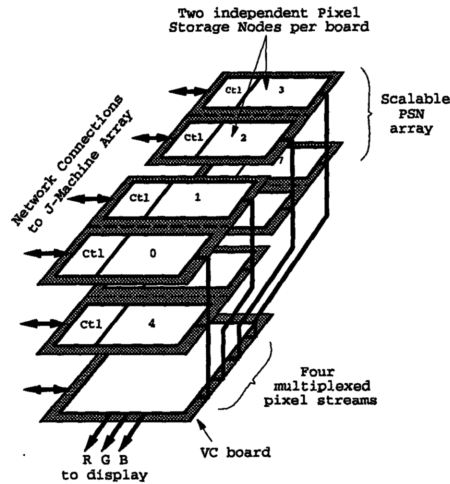


Figure 1.1: Small Video System for the J-Machine

Frame buffer bandwidth

The second potential bottleneck in rendering architectures is the bandwidth of the video memory used to store the computed image. To meet this high demand, all of the rendering architectures mentioned thus far have used memory interleaving in one form or another. The distributed nature of the J-Machine's frame buffer provides a similar mechanism to meet these bandwidth requirements. The maximum configuration of the video system can support 140 million 32-bit writes to the frame buffer per second.

The frame buffer is scalable, so the exact size of the video system can be tailored to the performance requirements of individual installations. Although fewer frame buffer boards reduces total memory bandwidth, the dollar cost of the system is reduced.

1.3 Video System Hardware Overview

The J-Machine video system consists of two types of printed circuit boards. The first type contains two independent and identical nodes called Pixel Storage Nodes (PSNs); multiple PSNs comprise the distributed frame buffer storage. The second type is called the Video Controller (VC) board. There is only a single VC per system; its function is to coordinate the real-time transfer of pixel data and to display images on an RGB monitor.

Each type of board connects to an edge of the J-Machine processor array. Figure 1.1 shows one possible configuration of the video system, consisting of an array of eight Pixel Storage Nodes and a single Video Controller Board.

The VC must repeatedly retrieve pixel data for every line of each screen refresh cycle; no buffering is performed on the VC itself. The video specification provides for up to a 1280 x 1024 pixel screen with 24-bit deep pixels. The RGB video adheres to RS-343 synchronization standards [3], making it compatible with most high-performance monitors.

Although there is only one VC per video system, synchronization between multiple VCs

is possible if multiple video systems are installed in a single J-Machine. This permits the generation of high-resolution three-dimensional images on a stereo monitor.

As shown in the figure, the aggregate PSN array is ultimately responsible for supplying four pixel streams to the VC. Although the resulting data bus is four times the size of the pixels (the video controller IC has a massive 128-bit input port), the buses need operate only at one-quarter the 120 MHz pixel rate. So instead of retrieving a single pixel every 120 MHz clock cycle, four pixels are retrieved every 30 MHz clock cycle. Because of this arrangement, the video system requires at least four PSNs and a single VC. If more than four PSNs are used, each pixel stream bus is shared in a time-multiplexed fashion, with the VC providing bus arbitration.

A typical screen update proceeds as follows: an application is distributed throughout MDPs in the J-Machine cube. Each MDP is responsible for computing pixels for a certain region of the display. The MDPs have halted at a barrier synchronization because they are waiting for the recently-computed frame to be displayed. When they receive word from the VC that the end-of-frame has arrived, they can send their pixel updates to the frame buffer[1]. These pixel messages are sent to one or more of the PSNs over the J-Machine interprocessor network. Before every scan line in the new frame, the VC instructs the PSNs to download a row of pixel data into shift registers. The VC sequences through this data for the duration of the scan line. It fetches pixels from four PSNs at a time and spreads them across the next four display columns. When the end of a horizontal scan line is reached, the PSNs are instructed to download another row of pixel data. All communication between the PSNs and the VC takes place over a dedicated set of ribbon cables. When every scan line has been displayed, the process repeats.

1.4 Prior Work

This research builds upon initial work performed by S. Zamani [12]. In his Bachelor's thesis, Zamani proposed a design to deal with some of the fundamental architectural requirements of a video system for the J-Machine. The J-Machine video system includes some of those early design ideas, specifically with respect to the Pixel Storage Node.

In addition, other peripheral boards have been developed for the J-Machine. A SCSI interface was implemented to provide scalable non-volatile disk storage [11]. An SBus interface was built to allow a Sparc workstation to function as the front end to the J-Machine [8]. Since all of the peripheral boards must integrate into the J-Machine, they share a set of common logic.

1.5 Results

The J-Machine video system achieves the high-bandwidth required by modern graphics applications, but it has some weaknesses. Most notably, the various levels of interleaving required to achieve this bandwidth (discussed in Chapter 3) presents a convoluted mapping between position in memory and location on the display. Although a software library

[1] If all updates can be computed and delivered to the frame buffer in time, only a single buffer is needed. Otherwise, at least two buffers are needed and the VC is instructed which buffer to use for the current frame.

can abstract away any confusion, it will also incur a performance penalty. A truly high-performance application therefore needs to understand the hardware at a fundamental level. For less demanding applications, the J-Machine video system can be quite useful.

1.6 Thesis Outline

Chapter 2 discusses the architecture of the Pixel Storage Node and some of the tradeoffs that were made in the interest of design simplicity. Chapter 3 describes the Video Controller board and the flexible graphical environment it provides. Chapter 4 presents a range of system configurations, each with its own performance and cost. Chapter 5 closes by discussing potential future research issues and summarizing the results of this thesis.

Chapter 2

Pixel Storage Node

Do, or do not. There is no try.
– YODA

Even if you are on the right track,
you'll get run over if you just sit there.
– UNKNOWN

2.1 Purpose

Each Pixel Storage Node (PSN) provides a portion of the physical storage for the pixels in the video frame buffers. The relation between a storage location on a PSN and a position on the screen is not fixed by the hardware. Instead, it is configurable by the user via the Video Controller board. This mapping cannot be completely arbitrary, however, and will be discussed in Chapter 3.

The pixels of an image are typically generated by computations in the J-Machine processor array, transferred to one or more of the PSNs, and then fetched repeatedly by the VC. The native J-Machine inter-processor network provides the data path from the J-Machine to the PSN; this path is used when pixel values need to be modified. The second data path – from the PSNs to the VC – uses a separate set of ribbon cables. This path is used very frequently, since the Video Controller must refresh the CRT display.

There are two distinct methods for generating pixel data. One method requires that the precise pixel address and pixel value be computed by the J-Machine, and then transferred pixel-for-pixel to the PSNs. Such a message might say "put pixel value 0x3bf in VRAM location 0x0201." The second method requires only that general raster operations (ROPs) be generated by the main J-Machine array; the PSNs then interpret the ROPs locally to generate and store the exact pixel values. Such a message might say "draw a blue line from display coordinates (0,0) to (50,100)." The list of supported ROPs appears in section 4.6, along with a discussion of the tradeoffs associated with these two methods.

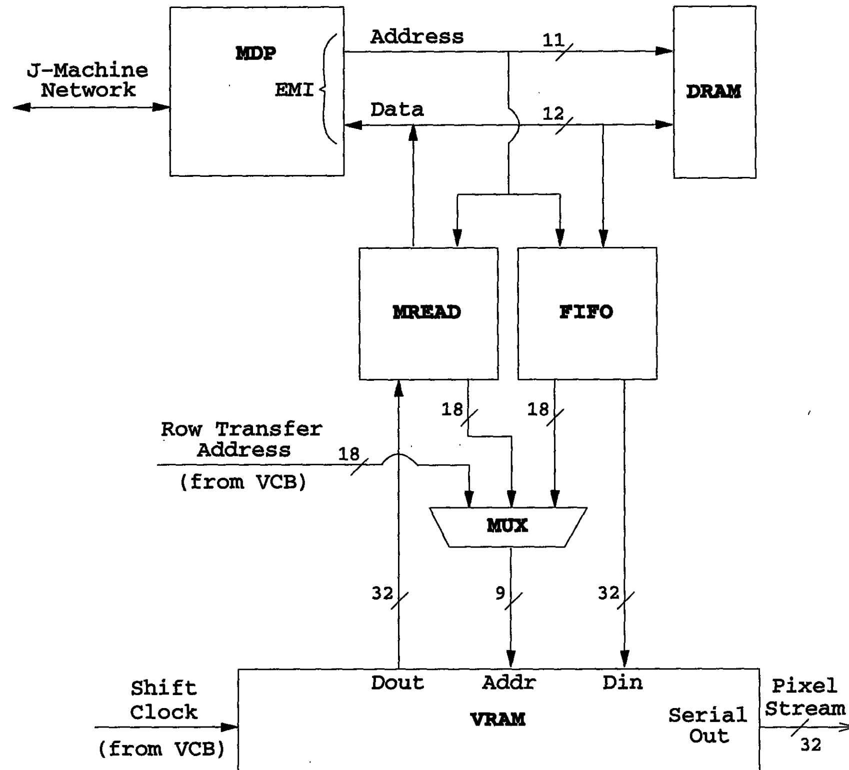


Figure 2.1: Pixel Storage Node Hardware Modules

2.2 Module Descriptions

Figure 2.1 shows the modules that comprise a Pixel Storage Node. Various control signals between the modules have been omitted for clarity. Pixel messages flow in from the J-Machine through the network connection on the upper left. The MDP decodes each message and writes address/pixel pairs out to its external memory interface (EMI), where they are placed into either standard external DRAM or into a FIFO directed toward the video RAM (VRAM). If the MDP tries to read from its EMI, the request will be serviced either by DRAM or by the MREAD module (which can read directly from the VRAM), depending on the address. A MUX is needed to select the address lines for the VRAM, since an address can come from the FIFO module (pixel write), the MREAD module (pixel read), or the Video Controller Board (in the form of a row-transfer request, discussed in section 2.2.6). The dual-ported VRAM has a serial output port which is constantly supplying pixel data to the video controller. The video controller uses a "shift clock" to sequence through this data as it retraces the display.

The remaining sections of this chapter discuss each of the modules in more detail. PAL program listings and full schematics for all modules appear in the appendices.

2.2.1 MDP Module

The MDP provides a data link between the J-Machine processor array and the rest of the PSN. Its 20 MByte/sec[1] port is connected via ribbon cable to an MDP along one of the J-Machine processor board edges. It is over this path that incoming pixel messages, generated somewhere within the J-Machine array, arrive at the MDP. The PSN must somehow retrieve this pixel data from the MDP to complete the pixel data path.

The MDP's external memory interface (EMI) is used to accomplish this task. Normally, the EMI allows the MDP to manage up to 1 Mword (36-bit words) of external DRAM storage. However, if the MDP copies data from network messages to its EMI interface, the data can instead be grabbed off the EMI bus and used by the rest of the PSN.

The network protocol used by the MDP to communicate with neighboring MDPs is obviously well-defined, so the question arises: why dedicate an expensive MDP to carry out this rather trivial task of copying data, instead of using some dedicated programmable logic? The two primary reasons are:

- The handshaking and token-passing network protocol was deemed too complicated to attempt to implement with programmable logic. It is also not clear that PALs or other PLDs could achieve the goal with the given timing constraints. For these reasons, the PSN and two other peripheral boards – the Video Controller board discussed later and the SCSI Interface Board used for the disk array – use an MDP for this data-bridging function. The single J-Machine peripheral board that obviates the use of an MDP is the SBus interface board; the tradeoff there, however, is that two-way communication is achieved by requiring connections to *two* MDPs in the processor array (one for each direction).

- Without an MDP available to process the incoming network messages, the format of incoming pixel messages needs to be rather tightly specified when the PLDs are programmed. An MDP, on the other hand, allows both pixel messages and raster operations to be sent to the PSNs (as discussed in section 2.1). This flexibility would not be possible without a processing engine – one more complicated than a finite state machine implemented in PLDs, for example.

A key disadvantage to using an MDP in this fashion is that its EMI interface has a paltry bandwidth of 8.75 MBytes/sec. The primary reason for this low bandwidth was a lack of I/O pins on the 168-pin PGA package used for the MDP. Because of the pin limitation, two MDP cycles are required to specify an external DRAM address, and three cycles are required to fetch 36 bits of data over 12 data lines (although the first data cycle is coincident with the second address cycle). Additional delay is caused by the need to copy each pixel out of the MDP's network buffer, into a register, and out to its EMI.

Thus, the pixel bandwidth into a single node is limited not by the J-Machine's network bandwidth but by the MDP's EMI bandwidth. It is important to note, however, that one can compensate for this lower bandwidth by scaling up the number of PSNs in the video

[1]Numerical figures in this thesis assume a J-Machine operating frequency of 20 MHz. As such, the network clock is 20 MHz while the MDP processor clock is one-half this rate. In theory, the operating frequency can be increased to 32 MHz.

PSN Nodes	*Aggregate EMI Bandwidth*	*Refresh rate (32-bit pix)*	*Storage Cap. (32-bit pix)*
4	35 MB/s	8.75 fps	2 frames
8	70 MB/s	17.5 fps	4 frames
16	140 MB/s	35.0 fps	8 frames
32	280 MB/s	70.0 fps	16 frames

Table 2.1: Effects of Increasing the Number of PSNs

system. Table 2.1 shows the effect scaling has on the pixel bandwidth into the video system. Also shown is the maximum frame storage and refresh rate assuming 32-bit pixels and a 1024×1024 display.

Realistic animation demands at least 15 frames per second to provide the illusion of continuous motion. Thus table 2.1 indicates that the video system requires at least 8 PSNs to meet the lower threshold for full-screen, 32-bit animation. However, if the rendering algorithm can pack multiple pixel values into a single 32-bit word (as discussed in Chapter 4), animation is still possible in a system with 4 PSNs. In essence, speed can be achieved at the expense of simultaneous color range.

The MDP module also requires a host of support components to integrate it into the J-Machine diagnostic and routing network. The schematics and PAL files that provide this support appear in the appendices; a more detailed description can be found in [8], [9], and [11].

2.2.2 DRAM Module

The external DRAM connected to the MDP module could potentially be used in two ways:

- As normal MDP external memory, available for program and data storage. Data intended for the VRAM would be addressed beyond the address range of the DRAM. No installed J-Machine node has more than 256 KWords of external DRAM, so it makes little sense to install more than that on the peripheral boards. This leaves 768 KWords of unused address space that can be memory mapped into VRAM space.
- As shadow video memory, to be used for duplicate pixel storage only. All EMI writes would be directed into both the DRAM and the VRAM, but EMI reads would be serviced by the DRAM. This would allow the MDP to use rendering algorithms that require the current value of a pixel before updating it. (Even though the MREAD module provides a read path between the MDP and VRAM, it is much slower than the DRAM path for reasons revealed in section 2.2.4.) Obviously, this scheme requires at least as much DRAM memory as VRAM memory per node; since 512 KWords of VRAM are installed per PSN, this would also be the minimum DRAM storage.

The first method was chosen over the second. A high-performance application uses many more J-Machine nodes than there are PSN nodes. To minimize the effects of low EMI bandwidth, a read-modify-write algorithm should be distributed across this larger set

of nodes; the shadow memory model therefore loses its appeal. Note that a PSN can still use its DRAM to locally implement a read-modify-write algorithm, but it must write to DRAM and VRAM with separate instructions.

2.2.3 FIFO Module

Design Considerations

The FIFO module serves to decouple the timing constraints of the MDP (which supplies the pixel data) from those of the VRAM (which consumes the data).

Since the EMI was designed solely as a memory interface, it has none of the provisions of a more general-purpose peripheral interface such as request, acknowledge, and ready lines. Thus, it is useful only for devices capable of consuming or producing data when the MDP demands it.

This requirement makes it impossible to directly connect the VRAM to the EMI bus. The VRAM may be in the midst of a refresh cycle, or may be servicing a row-transfer request for the video controller. Disruption of either of these activities would produce undesirable results, but the EMI interface has no provisions for detecting this not-ready condition. Without some kind of queue, any writes during this period need to be ignored by the VRAM module.

Since external DRAM memory should still be available to the MDP, the FIFO allows both DRAM and VRAM accesses by using a memory-mapping scheme: addresses less than 0x80000 correspond to the DRAM and addresses greater than or equal to 0x80000 correspond to the VRAM. To implement this scheme, the FIFO controller must understand the multiplexing mentioned in section 2.2.1. During an EMI access, the eleven address lines are time-multiplexed to provide a 20-bit wide address with two bits of phase, and the twelve data lines are time-multiplexed to produce 36-bit wide data. The two-cycle address-multiplexing scheme is shown in table 2.2. The three-cycle data-multiplexing scheme is simply the three 12-bit portions of the data word from lowest to highest significance. Since the VRAM's data port is 32 bits wide, the FIFO must assemble the three 12-bit data chunks appropriately. Note that the FIFO drops the four most significant bits, which are normally used as a tag for the data word; the video controller neither requires nor recognizes more than 32 bits of pixel data.

The FIFO module must take care to disable the external DRAM when an EMI access is intended for the VRAM. It therefore suppresses the write signal when the EMI address is greater than 0x7ffff. It should also disable the DRAM when a read is made from this upper address range, since the MREAD module is then responsible for driving the bus.

Implementation

As shown in figure 2.2, the FIFO module consists of separate address and data FIFOs, all coordinated by a 22v10B PAL. When the PAL detects an EMI write access to an address greater than 0x7ffff, it sequentially strobes the shift-in control lines of the FIFOs (`SIAddr`, `SIWord0`, `SIWord1`, `SIWord2`) to capture both the address and data. An FSM in the VRAM module can detect the subsequent data ready signal produced by the FIFOs and will eventually shift out the address/data pair.

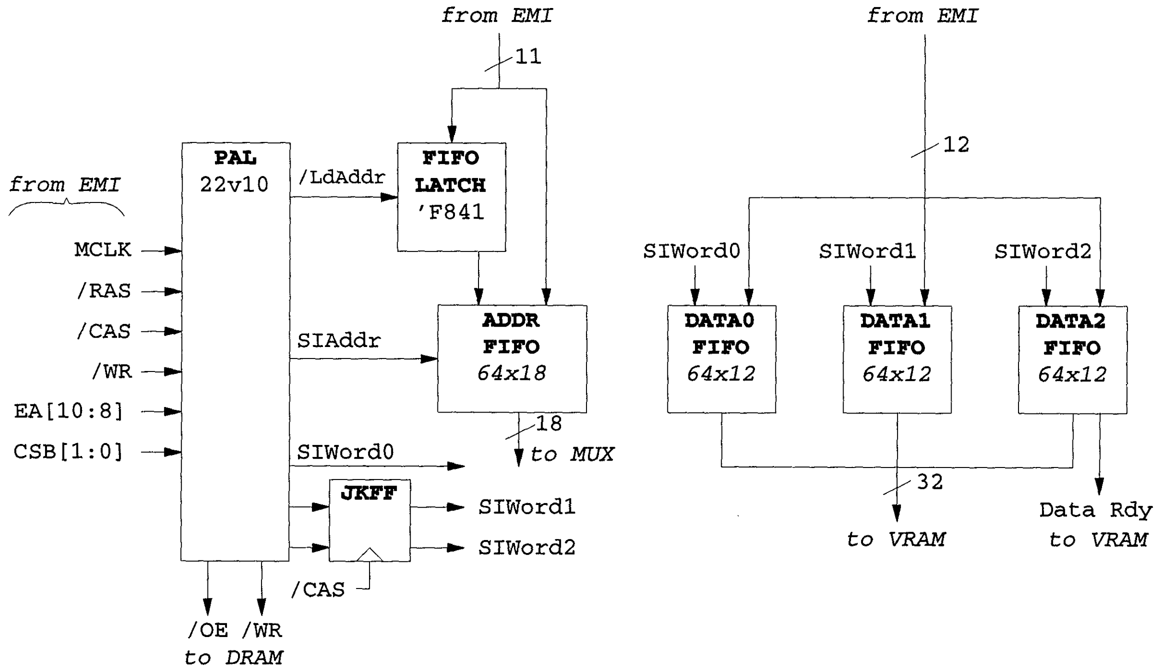


Figure 2.2: PSN FIFO Block Diagram

EMI line	*1st cycle*	*2nd cycle*
EA10	A19	A18
EA9	A17	A16
EA8	A15	A14
EA7	A13	A12
EA6	A11	A10
EA5	A9	A8
EA4	A7	A6
EA3	A5	A4
EA2	A3	A2
EA1	A1	CSB1
EA0	A0	CSB0

Table 2.2: EMI Address Multiplexing Scheme

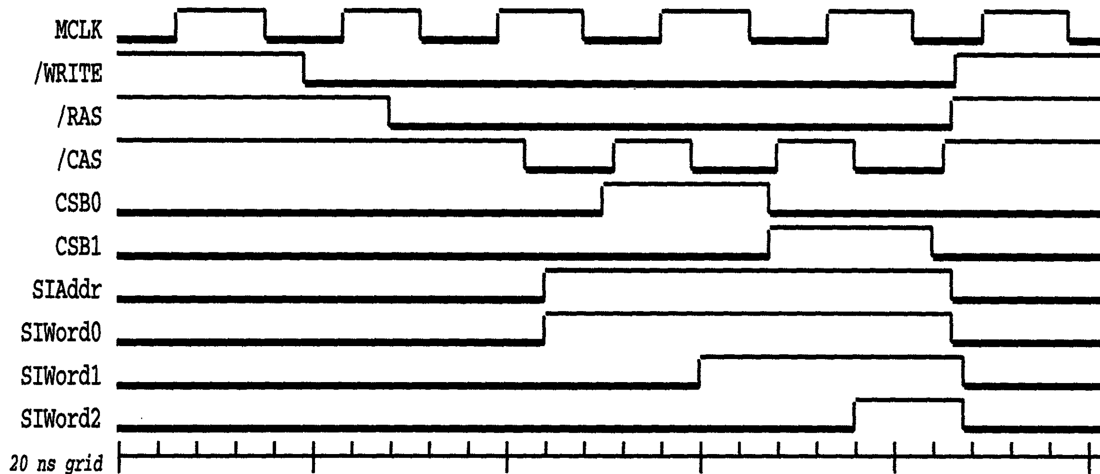


Figure 2.3: Logic Analyzer Plot of EMI Write Cycle

Figure 2.3 is the logic analyzer output produced by an EMI write to an address greater than 0x7ffff. `MCLK` is the MDP's processor clock. The A19 bit appears on the `EA10` line (not shown) when $\overline{\texttt{RAS}}$ first falls. If it is high and $\overline{\texttt{WRITE}}$ is low, the PAL will proceed to monitor the $\overline{\texttt{CAS}}$ and `CSB` lines. When $\overline{\texttt{CAS}}$ first drops, the PAL shifts in the EMI address (`SIAddr`) and the first twelve bits of the data (`SIWord0`). As the `CSB` cycle counter increases during subsequent $\overline{\texttt{CAS}}$ cycles, the second and third twelve-bit words will be shifted into their respective FIFOs.

The FIFO shift signals must be registered because they are derived from the combinational `CSB` lines. However, `MCLK` doesn't provide a suitable clock for these registers; as seen in the plot above, the rising edge of `MCLK` occurs before the falling edge of $\overline{\texttt{CAS}}$, which is the time the data is valid. And the falling edge of `MCLK` occurs too late into the $\overline{\texttt{CAS}}$ cycle to satisfy the FIFO hold time requirements. Thus, an external JK flip-flop, clocked by $\overline{\texttt{CAS}}$, is needed to align `SIWord1` and `SIWord2` with the falling edges of $\overline{\texttt{CAS}}$.

The address and data FIFOs are 64 words deep, which is more than enough space to queue the pixel data. In the worst-case scenario, the VRAM would require 13 clock cycles to perform an entire row-transfer (and subsequent memory refresh) before even recognizing the FIFO data. It would then take 13 cycles to process the data and shift it out of the FIFO. However, any further data would only require 13 cycles between shift outs, since row-transfers occur infrequently. These cycles are based on a 40 MHz clock, quadruple the speed of the MDP processor clock. And since the MDP needs at least six of its cycles between successive writes, the VRAM will quickly drain the FIFO should it begin to fill up.[2]

Although the EMI's $\overline{\texttt{RAS}}$ and $\overline{\texttt{CAS}}$ control lines pass unimpeded to the DRAM chips, the $\overline{\texttt{WR}}$ signal is routed through the PAL so that it can be squished when the address is greater than 0x7ffff. And while most J-Machine nodes permanently enable their DRAM outputs,

[2]The FIFO depth was selected before optimizing the time required by a row-transfer and before the 40 MHz clock was chosen.

Address	*Interpretation*
`0x80000 - 0xbffff`	Request VRAM data from *Address* minus 0x80000
`0xc0000 - 0xcffff`	Poll `DataREQ` line and view data returned from last request. Also selects bank for next read to Bank A.
`0xd0000 - 0xdffff`	Same as above, but selects bank for next read to Bank B.

Table 2.3: MREAD Memory Map

here the $\overline{\texttt{OE}}$ signal is generated by the FIFO PAL so as to avoid bus contention during MREAD cycles.

2.2.4 MREAD Module

Design Considerations

The MREAD module allows the MDP to read pixel data directly out of VRAM. In theory this is unnecessary because the MDPs themselves (both in the PSN and in the J-Machine) are responsible for setting the VRAM contents in the first place – they presumably can maintain a copy of what they send into the VRAM. But for debugging purposes this module provides an essential feedback mechanism. During testing it is always a good idea to be able to read back what one has written.

It is impossible to satisfy the MDP's request for VRAM data during the same memory cycle it is issued. The VRAM module may be busy accepting data from the FIFO or performing a row-transfer for the video controller. But even if the VRAM were idle, the EMI address lines could not be decoded fast enough to initiate a VRAM read cycle that could complete in time. The MREAD module must therefore use a multi-phase approach: the MDP first issues an EMI read to a memory-mapped address range. It then monitors a flag in a separate memory-mapped location to determine when the pixel data is ready. The data is then retrieved in a final read cycle. This reduces the read bandwidth from the VRAM to the MDP, but is acceptable because the MREAD interface was implemented primarily for testing purposes.

Implementation

Figure 2.4 is a block diagram of the MREAD module. Much like the FIFO module, a controlling PAL monitors the EMI lines of the MDP. When the PAL detects an EMI read to an address greater than 0x7ffff, the PAL latches the multiplexed address with $\overline{\texttt{LdRAS}}$ and $\overline{\texttt{LdAddr}}$, and then raises its `DataREQ` line to the VRAM module. Since it will take the VRAM module some time to supply the pixel, the value driven onto the EMI bus for that initial read will be the last pixel fetched, not the one just requested. Eventually the VRAM module will place the newly-read pixel into the `DATA` registers with $\overline{\texttt{PixLd}}$ and then assert `DataACK` to indicate completion. The MREAD PAL releases `DataREQ` when it sees this, so after first initiating a read the MDP should poll `DataREQ` and wait for it to drop before going back to read the returned pixel.

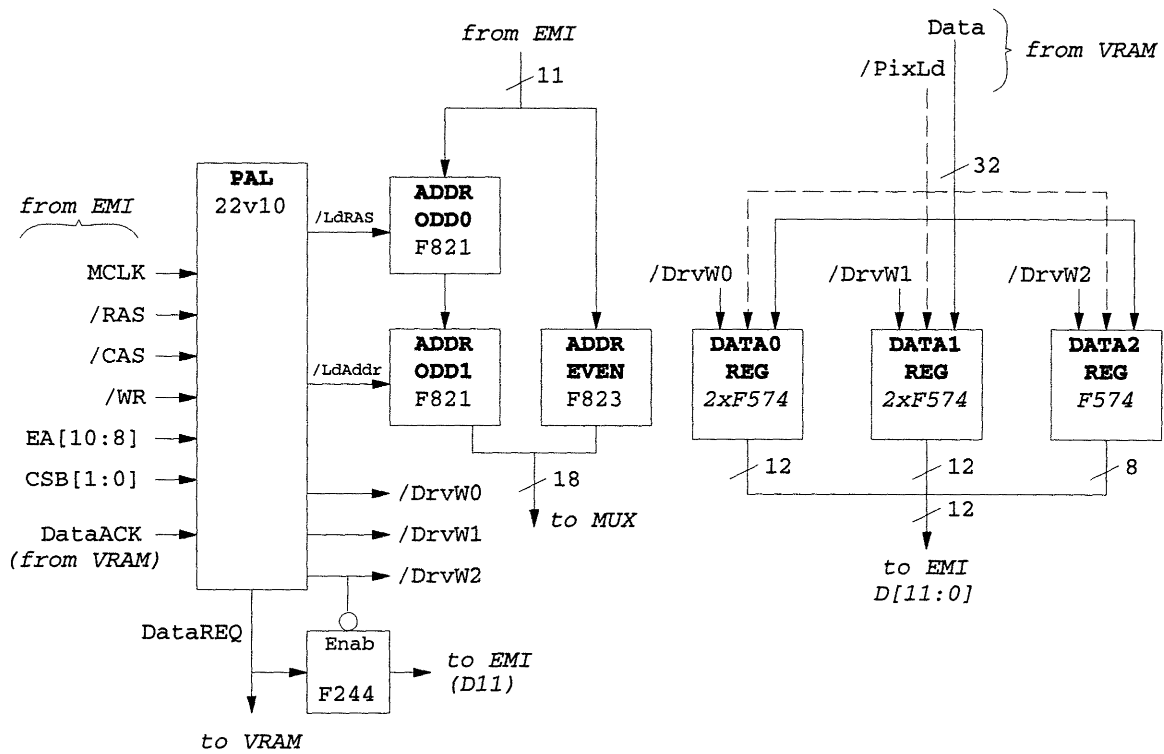


Figure 2.4: MREAD Block Diagram

Since the MDP will typically be polling the MREAD module after issuing a request, the MREAD module must not interpret *every* MREAD access as a VRAM data request. To do so would cause the PAL to re-assert `DataREQ` each time the MDP polled `DataREQ`, and this could create a race condition with the `DataACK` signal being returned by the VRAM module. The MREAD PAL therefore maps the address space as shown in table 2.3. Only MDP reads in the 0x80000 to 0xbffff address range will fetch the VRAM contents of that address (minus 0x80000). This allows an 18-bit address range for each VRAM bank (the two VRAM banks are discussed in section 2.2.6). To decide which VRAM bank to read, the MREAD PAL looks back to the last read made from an address greater than 0xbffff. If it was in the 0xc0000 page, VRAM bank A will be used on the next read cycle. If it was the 0xd0000 page, VRAM bank B will be used instead. A read from either of these pages will also return the `DataREQ` status in bit 35 of the data, without initiating another VRAM read cycle.

When the MREAD PAL drives data onto the bus with its `DrvW[2:0]` signals, it relies on the PAL in the FIFO module to disable the external DRAM to avoid bus contention. Thus, the FIFO PAL must monitor for both FIFO and MREAD accesses to control the DRAMs appropriately.

2.2.5 MUX Module

Design Considerations

The MUX module selects one of three sources for the address lines of the VRAMs. An address may come from the FIFO as a place to store a given pixel, from the MREAD module as an address to read, or from the VC as a row-transfer address. To conform to the VRAM address interface, the 18-bit address supplied by each of these sources must be broken up into a nine-bit column and a nine-bit row.

Implementation

The VRAM module provides the three MUX control signals shown in figure 2.5. When `Pix` is high the address will come from the `ADDR` FIFOs. When it is low the address will come from either the VC or the MREAD module, depending on the state of `Line`. The VRAM expects its address to be multiplexed into a row and column, so when `CAS` is high the lower 9 bits of the address will be selected; otherwise the upper 9 bits will be selected. Table 2.4 summarizes this behavior.

Although straightforward in function, the MUX module requires eight chips as implemented with simple discrete multiplexors. The upper mux in the figure requires five quad 2-to-1 chips, and the lower one requires three quad 4-to-1 chips. The chip count could have been reduced with more elaborate components, but at a higher dollar cost. If the area on the circuit board had been smaller, this module would have been the first to get streamlined.

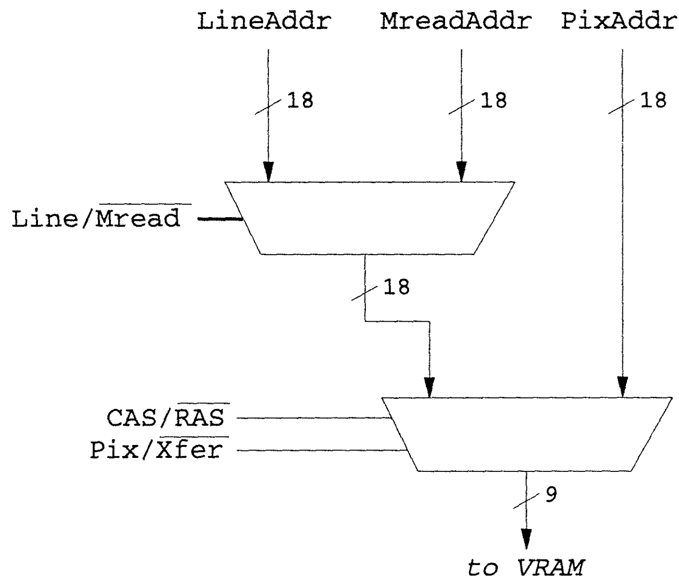


Figure 2.5: MUX Block Diagram

CAS/$\overline{\text{RAS}}$	Pix/$\overline{\text{Xfer}}$	Line/$\overline{\text{Mread}}$	*Output*
0	X	X	Upper nine bits (row)
1	X	X	Lower nine bits (column)
X	0	0	Mread address
X	0	1	Row transfer address
X	1	X	Pixel write address

Table 2.4: MUX Selection

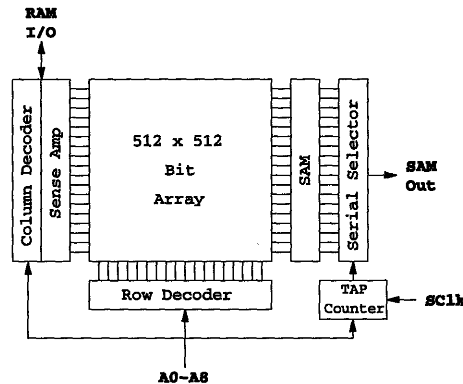


Figure 2.6: Single-bit VRAM Layer

2.2.6 VRAM Module

Design Considerations

The video memory is the primary resource of a PSN, and the complexity of the VRAM module reflects this importance. The VRAM controller must communicate with three other modules – the FIFO, MREAD and XFER modules – to arbitrate their access to the video storage. FIFO and MREAD accesses have already been discussed; XFER accesses are generated once per scan line and will be described in section 2.2.4.

Contention for the video memory would have been exacerbated had we needed to worry about the video controller's requirement to refresh the display. Fortunately, today's dual-ported Video RAM (VRAM) architecture eliminates this potential bottleneck. Figure 2.6 is a simplified diagram of a single-bit[3] layer of VRAM storage. A standard array of RAM is supplemented by the Serial Access Memory (SAM) shown on the right. A "read transfer cycle" copies an entire row from the RAM array into the SAM[4], after which time the SAM can be accessed completely independently (and faster) than the RAM array. The TAP is a 9-bit counter that indexes one of the 512 copied values and sends it to the output port. The TAP increments on the rising edge of `SClk` (shift clock), eliminating the need for separate address lines. This sequential access approach is not as flexible as the general random access used on the RAM side, but it is targeted at video applications where most accesses are in fact sequential.

Implementation

Figure 2.7 is a block diagram of the VRAM module. The first thing to notice is that the memory is segmented into two 256K banks. This is necessary to satisfy the video controller's read cycle time in a minimum video system. In a system with only 4 PSNs,

[3]The single layer shown is replicated four times in the VRAM chips used in this project. The chips are organized in groups of eight to provide 32-bit wide pixels.

[4]A "write transfer cycle" transfers data in the other direction, but this capability isn't used.

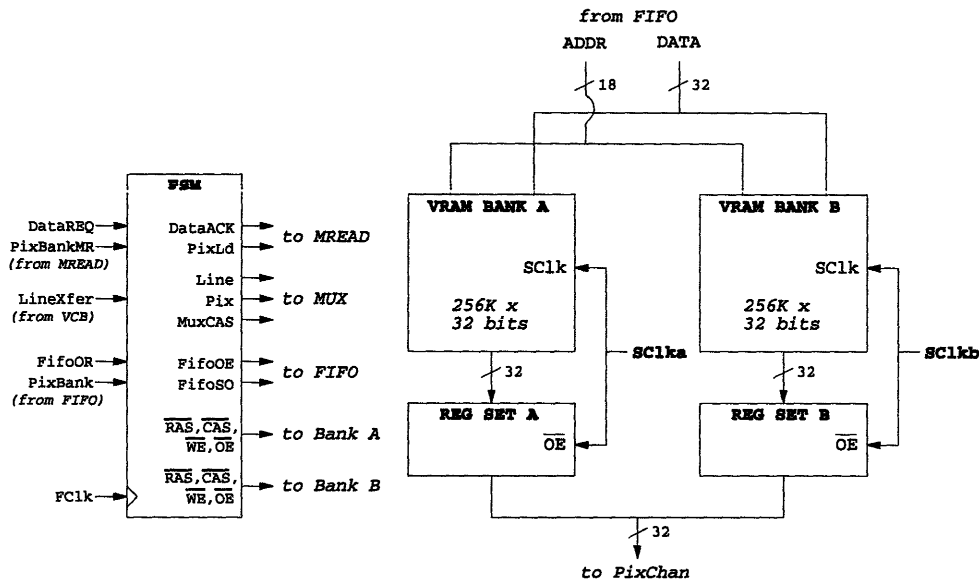


Figure 2.7: VRAM Block Diagram

the time between successive reads from a given PSN is only 23 ns, whereas the VRAM serial port requires at least 30 ns between reads. Thus, subsequent reads from a given PSN are alternately satisfied by banks A and B. Although the timing can be more relaxed in a system with more PSNs, the PSN is hardwired to always deliver pixels to the VC in this alternating fashion.

This mode of operation is nearly transparent to the user. The FSM determines the bank in which to store a given pixel by looking at the **PixBank** bit, which is actually the least significant bit of the pixel's address. Thus, pixels written to even addresses are placed in bank A, and those written to odd addresses are placed in bank B.

However, when using the MREAD module to read data back from the VRAM, the bank to read must be set explicitly as described earlier in section 2.2.4 and table 2.3. Since the MREAD module is principally used only during testing, this inconvenience is not important.

Pixels fetched from the SAM ports of the VRAMs are put into registers before being enabled onto the common **PixChan** bus. Notice that the **SClk** signal that provides the shift clock for the SAM is also used as the output enable for the register. Thus, the low-to-high transition of **SClk** transfers data from the VRAM to the register and increments the VRAM's TAP, but the pixel isn't driven onto the pixel bus until the next time **SClk** goes low. This design eliminates the need for a separate output-enable going to each register, but causes the **SClk** signal to assume two roles. The implications of this design are discussed in the next chapter.

FClk is a 40 Mhz clock signal supplied by the VC. It is asynchronous to the J-Machine clocking network, and allows the VRAM controller to run at a higher speed. More important, it is not selectable by the J-Machine user (who can choose between three system clocks). If the FSM frequency were not fixed, it would be difficult to arrange those portions of the state diagram that must meet certain VRAM timing constraints.

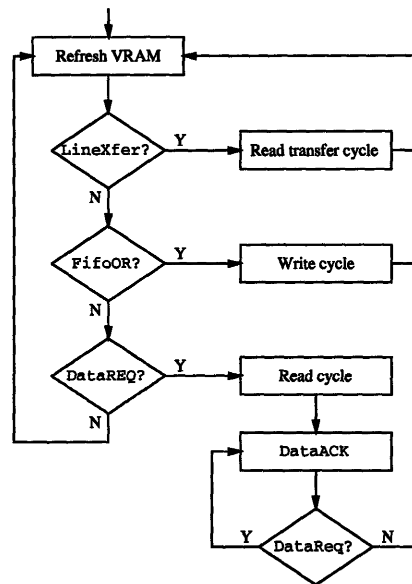


Figure 2.8: Flow Chart for VRAM Controller

The flow chart in figure 2.8 reveals that the most frequent operation of the FSM is the VRAM refresh cycle, which is required to maintain the integrity of the volatile memory. An internal refresh function provided by the VRAM chips reduces this task to the simple $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ operation shown in figure 2.9. An internal address counter eliminates the need to supply a row address for each refresh cycle.

In the last three refresh states, the controller sequentially checks for requests coming from the XFER, FIFO, and MREAD modules respectively. The XFER module gets first priority because the row of data must be transfered to the serial port before the beginning of the next scan line. There is no danger of permanently blocking requests from FIFO and MREAD because XFER requests are generated only once every 16 μs.

Figure 2.10 is a timing diagram for the row-transfer operation. The VRAM recognizes a row-transfer request when $\overline{\text{DTOE}}$ drops before $\overline{\text{RAS}}$ does. The FSM drives `Line` high to select

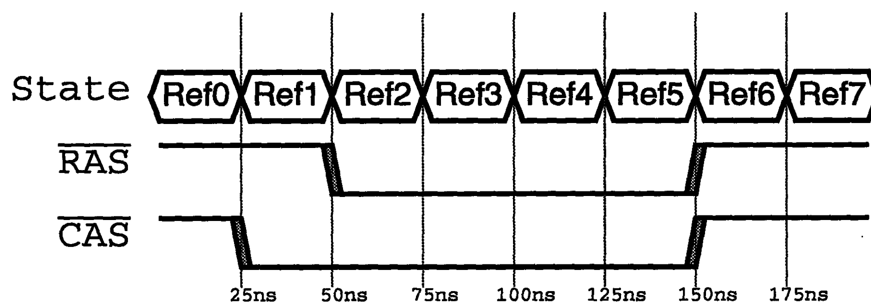


Figure 2.9: Timing Diagram - VRAM Refresh Cycle

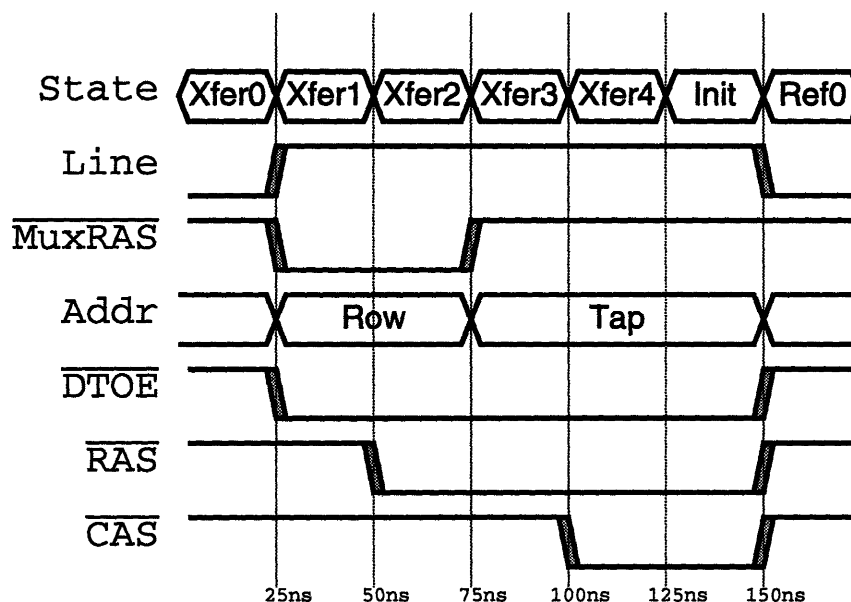


Figure 2.10: Timing Diagram - VRAM Row Transfer

the transfer address from the MUX. The row to be transferred is specified by the nine bits latched on the falling edge of $\overline{\text{RAS}}$. The TAP counter is initialized to the nine bits latched on the falling edge of $\overline{\text{CAS}}$.

Without an outstanding row-transfer request, the FSM next looks for the FifoOR condition. This indicates that one or more pixels is waiting to be transferred from the FIFO to VRAM. Although there may be a continuous flood of FIFO data, the controller always performs a VRAM refresh and possibly a XFER cycle between successive writes. The write operation is depicted in figure 2.11. The DATA FIFO outputs are directly connected to the VRAM, so the pixel data will be valid by the time FifoOR is raised. The address must be selected in the normal multiplexed fashion. At the end of the write, the VRAM uses FifoSO to shift the data and address out of the FIFOs.

Lowest in priority is the MREAD module, which may be attempting to read a value out of VRAM. Requests from MREAD will be deferred if there are any outstanding XFER or FIFO requests, but this is acceptable given the debugging purpose of MREAD. As shown in figure 2.12, the VRAM distinguishes a VRAM read from a row-transfer based on the state of $\overline{\text{DTOE}}$ when $\overline{\text{RAS}}$ falls. The FSM releases $\overline{\text{FifoOE}}$ to disable the outputs of the DATA FIFO, allowing the VRAM to use the common bus to drive data into MREAD. This data is latched by the $\overline{\text{PixLd}}$ signal.

Following the read, the FSM maintains DataACK and awaits the fall of DataREQ, indicating that MREAD has recognized the completion of the read cycle[5]. This REQ/ACK handshaking is necessitated by the the different operating frequencies of the MREAD and VRAM modules. In operation, the loop only lasts for about two MREAD cycles. Note that a hardware- or software-level failure in one of the PALs engaging in the handshaking could

[5]This acknowledgment is presumably being polled by the MDP as discussed in section 2.2.4

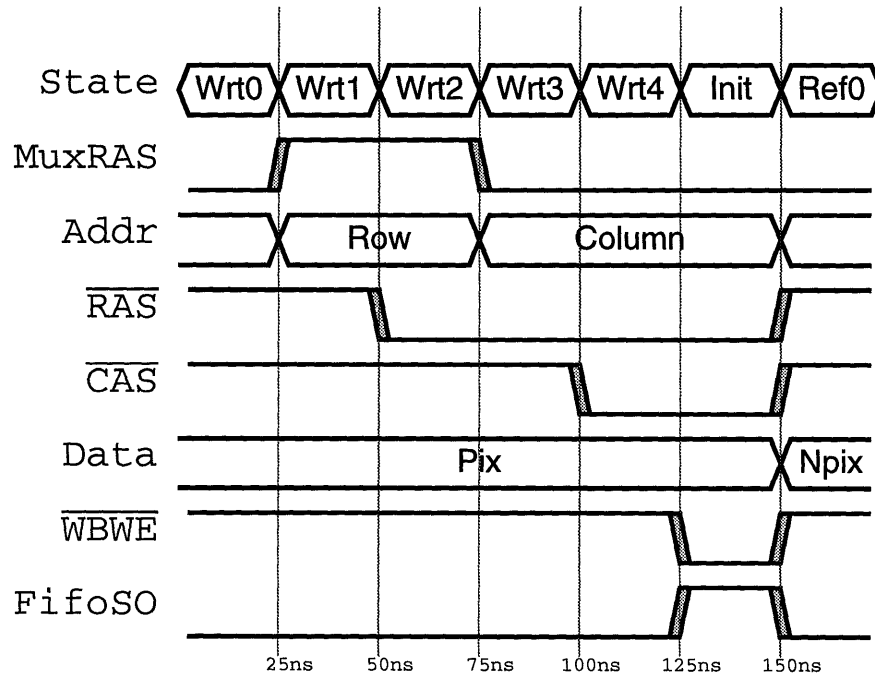


Figure 2.11: Timing Diagram - VRAM Write

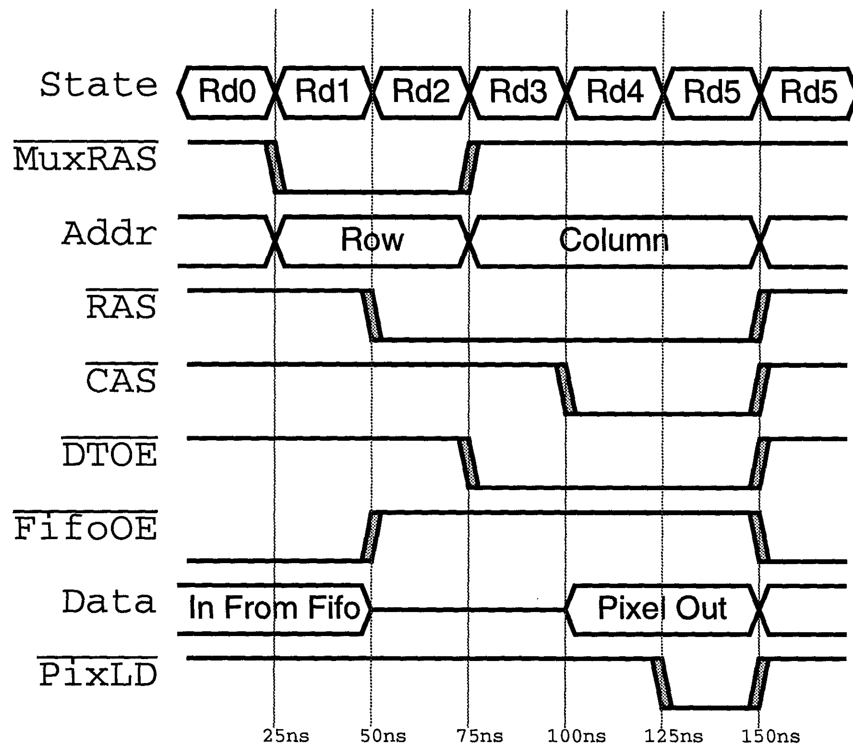


Figure 2.12: Timing Diagram - VRAM Read

cause MREAD to loop infinitely, allowing the VRAM contents to decay and producing garbage on the display.

2.3 Fabrication

The form factor used for J-Machine peripheral circuit boards is rather large (11 inches wide by 15 inches long), so two PSNs fit onto a single circuit board dubbed the Pixel Storage Board (PSB). Thus, as discussed in section 1.3, the minimum video system configuration requires two PSBs (four PSN nodes) and a Video Controller board. The PSB has a ground plane, power plane, and six signal layers.

The PSB has gone through two fabrication runs. The first version of the board differed from the second in two significant ways: each node had only a single bank of VRAM, and the two nodes were laid out vertically along the length of the board. Several other changes were made, most of them bug fixes or format changes.

The effect of the first difference was that twice as many nodes were needed in the minimum configuration in order to meet the video controller timing requirements. A type of "pixel funneling" was experimented with along the lines proposed by [12], but this was changed in favor of the dual-bank/common-pixel-bus approach now in use.

The nodes were rearranged for the second version to keep the two MDPs fairly close to the ribbon-cable connector that couples them to the J-Machine. In the first version, one of the MDPs was about 12 inches away from the connector. It is uncertain whether this layout posed a potential reliability problem, but it was decided to keep the distance between MDPs as small as possible.

The clock signals were hand-routed to minimize signal reflection. The MDP clock trace was made as identical as possible to the MDP clock trace on the J-Machine processor boards in terms of length, signal layer usage, and number of buffers used.

Aside from the MDP and an assortment of registers, latches, and PALs, the PSB required four specialized types of ICs:

- *High speed FIFOs.* To easily interface with the EMI bus, these were needed in 9-bit wide versions for `ADDR` and in 12-bit wide versions for `DATA`. The `DATA` FIFOs also had to have an output enable for reasons discussed in section 2.2.4. Fortunately, Cypress Semiconductor manufactures FIFOs that meet almost all of these requirements (an 8-bit and 4-bit FIFO were coupled for `DATA`).

- *VRAM.* The Toshiba TC524256BZ-80 chip provided all of the functionality described in section 2.2.6. Due to their sheer number, the combined cost of the VRAM was higher than any other portion of the PSN except the MDP. At approximately $12 per megabit, each 16-megabit PSN requires about $200 worth of VRAM.

- *Pixel bus drivers.* The registers/drivers needed between the VRAM and the pixel bus had to combine high speed with high output drive. The Signetics ABT series 74ABT646 provided the solution. The outputs of these tri-state registers can sink 64 mA and source 32 mA, with an average enable/disable time of 5 ns. Although these chips are only used to *drive* the bus, they are actually bidirectional. Jumpers were

added to the PSN so that a future research project could use this input capability to implement a frame grabber or similar input device.

- *Clock receivers.* The FCLK and $\overline{\texttt{OE}}$ signals used by the VRAM module are supplied by the VC. To protect these signals from EMI interference, differential signal pairs are used on the ribbon cables. AT&T manufactures a series of drivers/receivers that use balanced pseudo-ECL levels – ECL levels shifted by 5V to run on a single +5V power supply. The PSB uses a 41LR quad receiver to convert the pseudo-ECL signals to TTL levels. On the other end, the VC uses a 41LP driver to convert and drive the signals. These chips are also used to transport the MDP clock.

Chapter 3

Video Controller

Where a calculator on the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and perhaps weigh 1.5 tons.
– POPULAR MECHANICS, March 1949

My primary and cerebral functions are now operating almost entirely from within the computer. They have expanded to such a degree that it would be impossible to return to the confines of my human brain. Any attempt to do so would mean my death.
– LT. BARCLAY in *Star Trek: The Next Generation* Episode #93

3.1 Purpose

The Video Controller (VC) coordinates the flow of pixels in the system and produces an image on the display. While the Pixel Storage Nodes are busy delivering new pixels to the VRAM buffers, the VC determines which pixels to fetch, how to interpret them, and where to place them on the screen. All of the VC modules are configurable – some directly by software – providing multiple graphical environments.

At the heart of the VC is a Brooktree Bt463 high-performance RAMDAC. When supplied with digital-level pixels and timing signals, this chip generates the analog RS-343 video signals accepted by many monitors. Every VC module supports the Bt463 in one way or another. The Timing Module and MPU module directly interface to the Bt463 to provide clocking and control. The XferAddr Lookup Module and Shift Clock Module provide indirect support by controlling the flow of pixel data. The MDP and FIFO modules direct the progress of the other modules. And the EMI Readback Module allows the system status to be monitored.

3.2 Module Descriptions

Figure 3.1 is a block diagram of the VC. An MDP acts as a link between the J-Machine and the VC. A FIFO provides a data path from the MDP to the other modules without

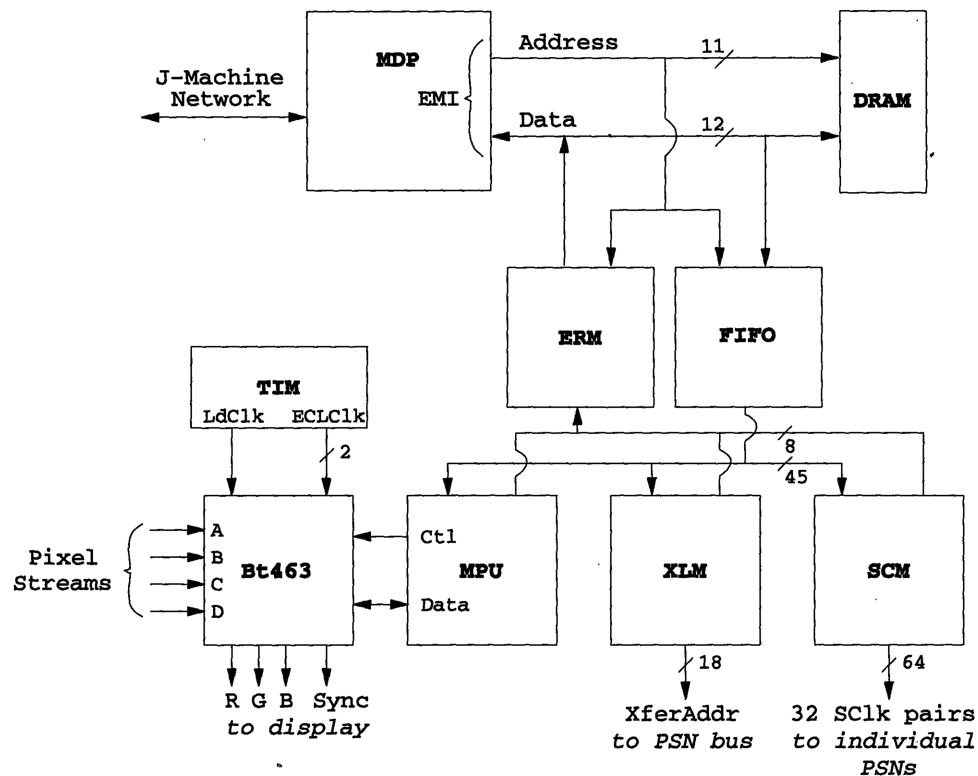


Figure 3.1: Video Controller Hardware Modules

constraining their timing requirements. The ERM allows the MDP to check on the status of the modules. The MPU interfaces to the Bt463 and provides dynamic control over numerous display options. The XLM dictates which portion of VRAM storage should be displayed for each scan line, and the SCM selects those VRAMs that should take part in this process. Finally, the TIM module generates the system clock, pixel clock, and monitor synchronization signals.

3.2.1 MDP Module

The VC employs the same method to communicate with the J-Machine as the Pixel Storage Node: an MDP translates certain incoming network messages into EMI accesses, which are acted upon by VC modules that spy on the EMI bus. Much of the discussion in section 2.2.1 applies, but the traffic seen by the VC's MDP and the PSN's MDP differs in two respects:

- The rate of incoming messages is much smaller on the VC. For many applications, the only communication with the VC will be to perform one-time initialization of the display options. Even those applications which modify these options between successive video frames send only a handful of messages per frame.

- Messages are sent in both directions. In contrast to the one-way flow of pixels into a PSN, the VC may be asked to reply with messages indicating the beginning of video blanking periods or the current status.

With these differences in mind, there is little incentive to use anything other than an MDP as a link to the J-Machine. Despite its low bandwidth, the EMI interface does not introduce a bottleneck during the execution of practical graphic applications. One might take issue with the cost of an MDP versus any alternatives, but the ease with which messages can be injected into the network by the MDP overwhelms this objection. Still, some operations performed by the VC MDP are less than elegant: for instance, the MDP must perform polling to obtain status updates since it lacks a general-purpose interrupt mechanism.

3.2.2 DRAM Module

The MDP is equipped with the standard ration of 256 KWords of external DRAM memory. This can be used for program or data storage, but is unused in the current VC software driver, which was written in assembly and fits within the MDP's internal RAM.

3.2.3 FIFO Module

Design Considerations

The VC's FIFO has the same task as the PSN's FIFO: monitor the EMI bus for transactions with addresses greater than 0x7ffff, and push the assembled address and data into FIFOs for use by the rest of the VC. However, there are now multiple consumers of FIFO data; the MPU, XLM, and SCM modules are all possible destinations. The FIFO module must be split into two parts:

- A controller that inserts data into the FIFOs. This controller operates with the same clock as the MDP so as to decode the EMI signals properly.
- A controller that detects data ready on the other end of the FIFO and directs it to the appropriate module. This controller must perform handshaking with each module since the receiver may not be ready to accept new data. The receiver modules all operate at a higher clock rate than the MDP, so for efficiency this controller is also operated at the higher rate.

As discussed in later sections, the XLM and SCM modules both make use of to private tables during their operation. The "active table" number can be set to increment automatically at the bottom of every video frame. However, since the modules operate independently, they require a common "reset" signal to align their active table numbers. The MDP initiates this `TabSync` synchronization by strobing a certain EMI address. Since the table counters increment together, this is only necessary at start-up or when the number of tables is changed. The most convenient place to recognize the request to generate `TabSync` is in the FIFO module.

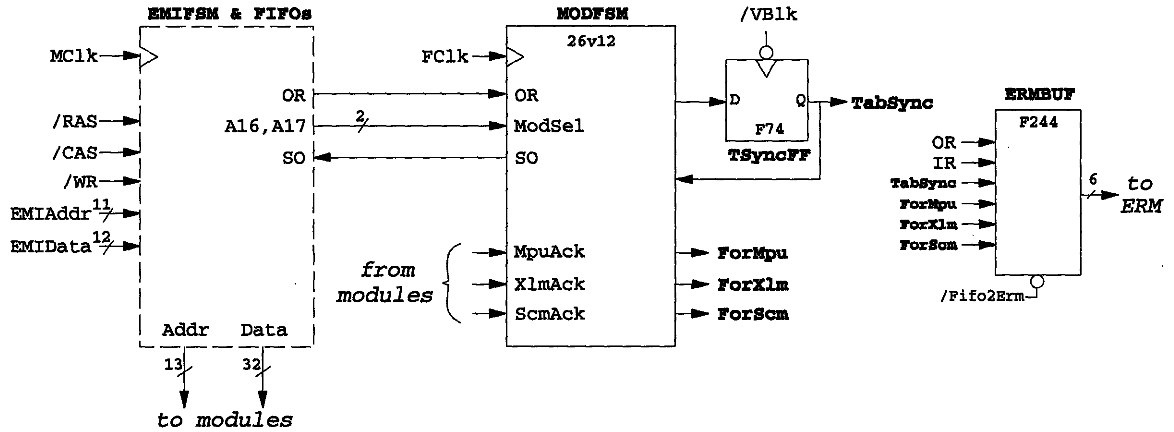


Figure 3.2: VC FIFO Block Diagram

Targeted Module	*ModSel* `A[17:16]`	*First Valid Address*	*Last Valid Address*	*Addr Bits Used*	*Data Bits Used*
MPU	00	0x80000	0x80000	0	32
XLM	01	0x90000	0x91fff	13	23
SCM	10	0xa0000	0xa0fff	12	13
TabSync	11	0xb0000	0xb0000	0	0

Table 3.1: FIFO Interpretation of EMI Address Bits 17 & 16

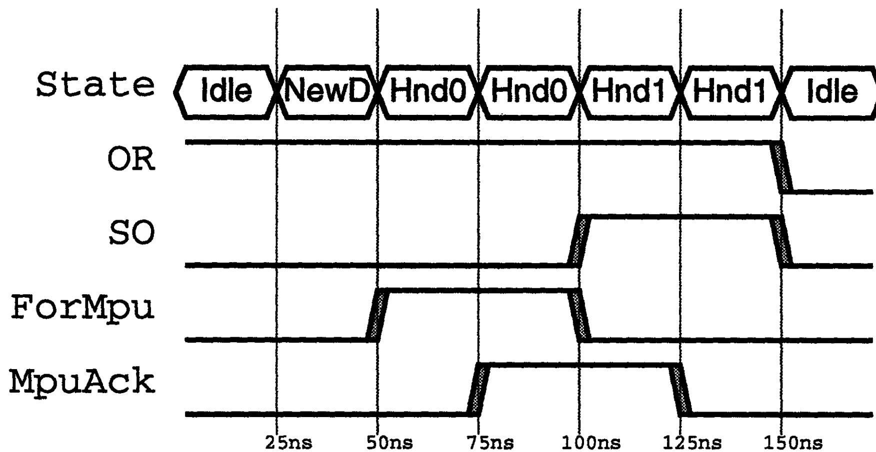


Figure 3.3: Data transfer between FIFO and MPU

Implementation

A block diagram for the FIFO appears in figure 3.2. The dashed block on the left contains a PAL and set of FIFO chips identical to the PSN's FIFO block shown earlier on page 24.

MODFSM is a finite state machine implemented with a 26V12 PAL. Whenever it detects an output ready condition from the FIFOs, it examines bits 16 and 17 of the captured EMI write address[1]. Using these bits, it hands the data and the lower 13 address bits to one of the modules according to the memory map shown in table 3.1. Note that not all modules use all of the address and data bits.

A special "module" that uses neither the address nor data bits is actually the TabSync signal, which is memory-mapped to address 0xb0000. TabSync must be delayed by an external flip-flop to prevent race conditions that can occur if it is asserted too close to the end-of-frame; because the XLM and SCM modules are running independent FSMs, we must prevent the possibility that one module sees TabSync in time to reset its table counter but the other does not. So although MODFSM presents the TabSync request to the flip-flop immediately after receiving it, the flip-flop will pass it through only when the TIM module asserts and then releases $\overline{\text{VBlk}}$, indicating the end of vertical blanking. Positioning TabSync at the beginning-of-frame ensures that both modules have enough time to recognize it.

The MPU, XLM, and SCM modules all share the ADDR and DATA bus outputs of the FIFOs. MODFSM notifies the module that should accept new input with ForMPU, ForXlm, or ForSCM. An example of the data transfer routine executed by the FSM is shown in figure 3.3; upon noticing an output ready condition OR, the FSM decodes the MODSEL bits in state "NewData" and signals the appropriate module in state "Handoff0." It remains in this state until it sees an ACK signal from the module, and then enters "Handoff1" where it waits for the ACK to go away. The FIFO entries are shifted out by the falling edge of SO when the FSM returns to the "Idle" state.

[1]EMI bit 19 must also be set for the FIFO to have accepted the data in the first place.

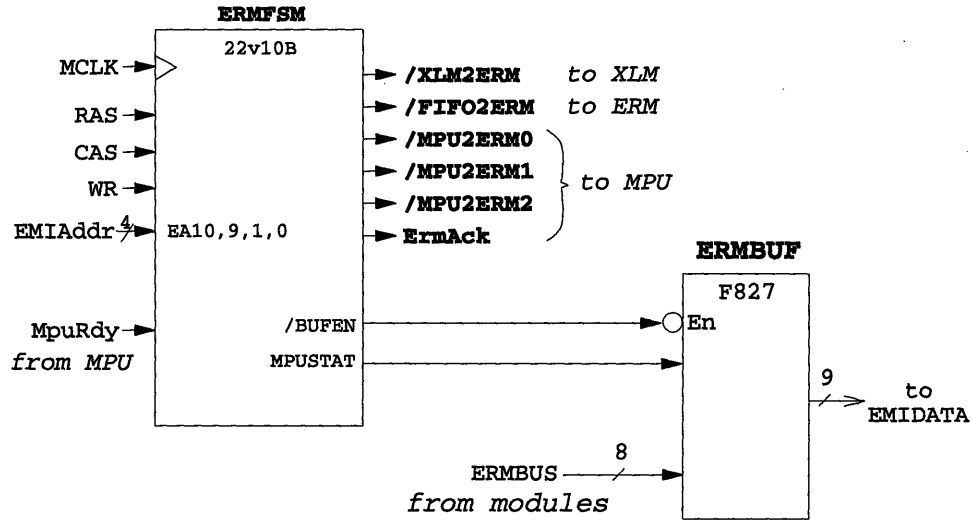


Figure 3.4: ERM Block Diagram

The module selected to receive the new input will not necessary accept it immediately. The module may have previously received data flagged with a request to delay further processing until the end of the current scan line (HHOLD flag) or the end of the current frame (VHOLD flag). In such an event, MODFSM is forced to remain in "Handoff0" until the module has processed the previous data. Because the new data can't be shifted out, the FIFOs can subsequently begin to fill up.

The FIFOs have a depth of 64 words, which is enough storage to queue as many EMI writes as the MDP can produce in a single 16 μs scan line. But if multiple FIFO entries have their HHOLD flag set, or if a single entry has its VHOLD flag set, the FIFOs can potentially fill up. To allow the MDP to check for the FIFO-full condition, the Input Ready (IR) status of the FIFO chips is provided to the ERM. Also provided to the ERM, primarily for debugging purposes, are ForMPU, ForXLM, ForSCM, OR from the FIFO chips, and TabSync. The EMI address from which these status bits can be fetched is revealed in the next section.

A feedback path that allows stalled FIFO data to be pushed out of the way from the head back to the tail of the queue was considered but ultimately dismissed for three reasons. First, out-of-order processing might not be acceptable to the application. Second, the coordination that would be required between MODFSM and EMIFSM (which shifts data into the FIFOs) added too much overhead. Finally, such transfers could get preempted by new EMI writes, creating the risk of data loss.

3.2.4 ERM Module

The MDP, acting on behalf of the application, needs the ability to monitor various status signals within the VC. The EMI Readback Module (ERM) provides this ability, assuming a role similar to the MREAD module in the PSN. But whereas MREAD was designed to assist during debugging, the ERM provides information more useful to an application. For example, it would be difficult for animation programs to know when to update the frame

	Address	*Data Bits*	*Meaning*
FIFO	0xd0000	0	Fifo `OR`
		1	Fifo `IR`
		2	`TabSync`
		3	`ForMpu`
		4	`ForXlm`
		5	`ForScm`
XLM	0xc0000	0-3	`TabCtr`
MPU	0x80000	0-7	Word 0 from Bt463
	or	12-19	Word 1 from Bt463
	0x90000	24-31	Word 2 from Bt463
		8	`MpuStat`

Table 3.2: ERM Memory Map

buffer without being notified when the end-of-frame occurs. And as noted in the previous section, the MDP must be able to verify the readiness of the FIFO to accept new data before sending it.

The ERM consists of a single 22v10B PAL and a tri-state buffer, shown in figure 3.4. `ERMBUF` provides isolation between the ErmBus and the EMI bus. `ERMFSM`, an FSM running on the MDP clock, monitors the EMI for read attempts from addresses greater than 0x7ffff. The FSM interprets these addresses according to the memory map listed in table 3.2. When a read request is detected, the FSM enables two tri-state buffers – `ERMBUF` and a tri-state buffer in the appropriate module – to drive the EMI bus during its three data cycles[2]. `ERMFSM` relies on the FIFO module to prevent bus contention by disabling the external DRAMs during these cycles.

The status signals provided by the FIFO and XLM modules are asynchronous with respect to the MDP clock. It therefore behooves the MDP to read the status from these modules several times in succession to verify signal stability.

Special care is put into reading values back from the MPU module – which provides not status information, but data fetched from the Bt463. As explained in section 3.2.8, the actual request for Bt463 data is initiated by an EMI *write* access through the FIFO module. When the MPU module finally receives the request, it fetches the data and returns it through the ERM.

Due to the delay that follows an MPU request, the MDP must poll the ERM's `MpuStat` flag to determine when the Bt463 data has been fetched. `ERMFSM` asserts `MpuStat` once it has received an `MpuRdy` signal from the MPU. The MDP should poll `MpuStat` using address 0x80000. But the MPU module itself needs to be notified when the data it is providing has been read by the MDP, so that it can re-use its output buffers for any further data requests. So once the MDP has detected `MpuStat` using address 0x80000, it should make

[2]The VC MDP must disable ECC error correction for proper operation.

a final read from address 0x90000. Although the MDP will have already fetched the new value along with MpuStat, only an access to 0x90000 causes ERMFSM to send an ErmAck back to the MPU, freeing it for further processing. The separate addresses are needed to prevent confusion in the handshaking protocol between MREAD and MPU.

3.2.5 XLM Module

Before a horizontal scan line can be displayed on the screen, each VRAM must be instructed to perform a "row-transfer" to copy a row of pixels from its primary memory array to its serial SAM port. This allows rapid, sequential access to the pixels via the SClk shift clock signal. The particular row to transfer and the starting column within that row are jointly called a XferAddr transfer address. It is the responsibility of the XferAddr Lookup Module (XLM) to broadcast the row-transfer request and the 18-bit XferAddr to all VRAMs during horizontal blanking periods.

Instead of fixing the relation between a row number and its XferAddr in hardware, a RAM table allows arbitrary mapping. A single table contains 1024 18-bit entries, one entry for every row of the highest supported resolution. Up to eight tables may be defined in advance to provide eight logical frame buffers. An XLM write operation can change the active table (frame buffer) at any time, but to ease the task of animation the tables can be set to sequence automatically when the end-of-frame occurs. Alternatively, it is possible to use a single table and simply overwrite its contents when necessary. There is more than enough time between video frames to modify all 1024 entries in an XLM table.

Figure 3.5 is a block diagram of the XLM. The registers that accept data from the ADDR and DATA FIFOs are shown on the left. From top to bottom in the middle of the figure are the scan line counter, the active table counter, the valid table registers, and the XLM controller. The SRAM is the large block on the right. In the upper right are shown two flip-flops that are triggered by the horizontal and vertical blanking periods, respectively. They ensure that the XLM controller notices these events, even though it may be busy at the time they occur.

The XLM can execute three operations: an *XLM write* places a new XferAddr entry into SRAM; an *end-of-line update* increments the scan line counter and broadcasts a new XferAddr to the VRAMs; and an *end-of-frame update* increments the active table pointer.

XLM write

When the FIFO's ForXlm line is raised, XLMFSM asserts XlmACK, which simultaneously ACKs the FIFO module and latches in 22 bits from the DATA FIFO and 13 bits from the ADDR FIFO. XLMFSM maintains XlmACK until ForXlm drops, and then decodes the data and address words according to table 3.3.

If either the HHold or VHold flag is set, XLMFSM doesn't immediately perform the write. It instead sets an internal "blocked" flag, which instructs it to process only Eol (end-of-line) and Eof (end-of-frame) events. Once the blocked flag has been cleared by one of these operations, the new data will be written. Thus, HHold and VHold prevent XLM changes from taking place mid-line or mid-screen. If neither HHold nor VHOLD is high, the XLM write commences immediately according to the TYPE of data being received.

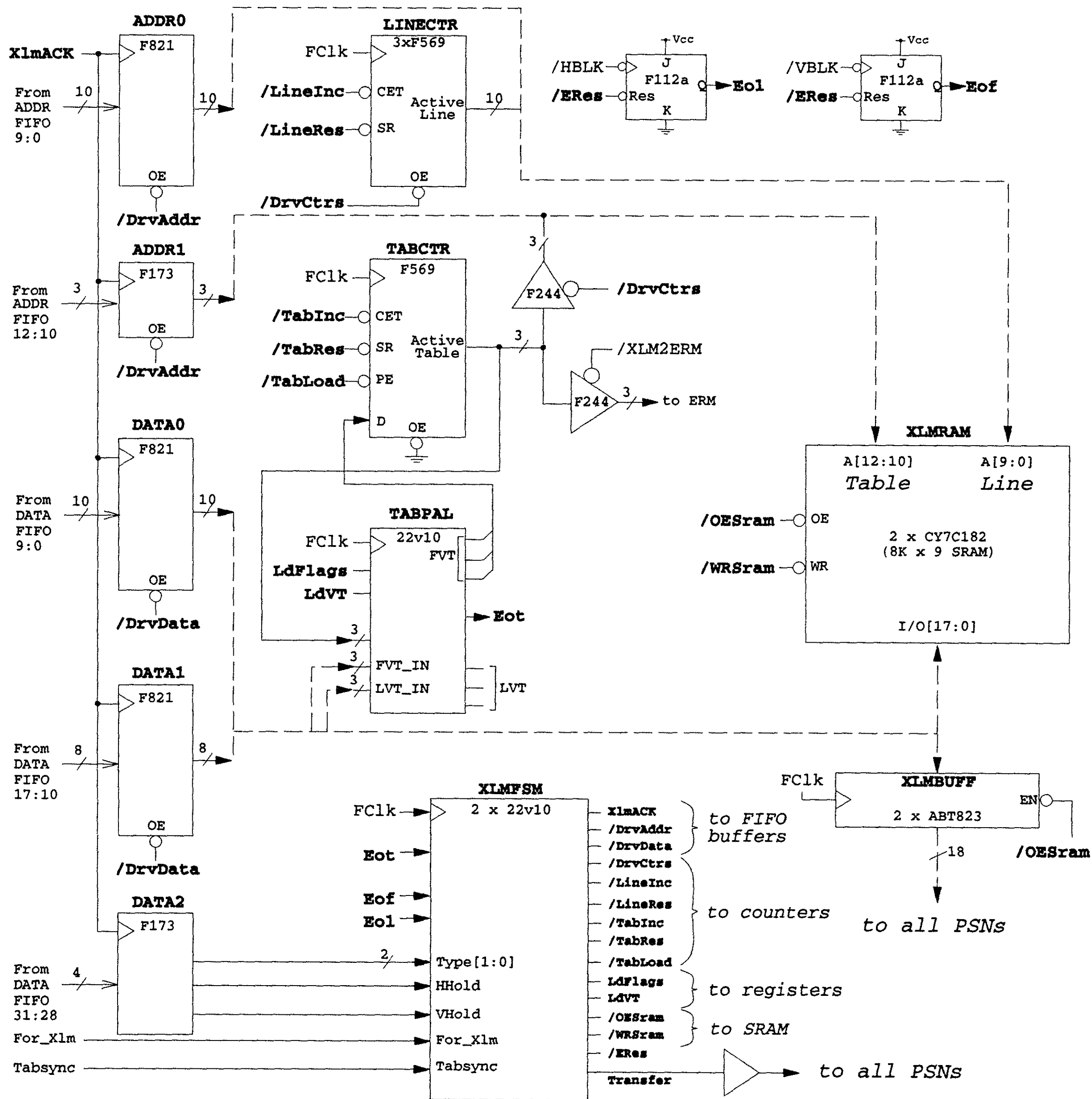
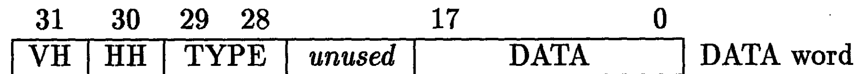


Figure 3.5: XLM Block Diagram



VH	VHold - If set, don't perform write until next vertical blanking period
HH	HHold - If set, don't perform write until next horizontal blanking period
DATA	The TYPE-specific data to be used
TYPE	The type of data being specified: 00 | Xfer Address | Write DATA into SRAM location specified by ADDR bus 01 | Valid Tables | Set LVT register = DATA[2:0], FVT register = DATA[5:3] 11 | Flags | Set FLAGS register = DATA[2:0] (for future use)

TABADDR	These 3 upper bits specify one of eight tables. The active table is cycled between the FVT and the LVT register values.
LINEADDR	These 10 bits specify one of 1024 scan lines. The active line counter is incremented at the end of each scan line.

Table 3.3: XLM Usage of Data and Address Bits From FIFO

- If the TYPE is 00, the 18 bits of DATA specify a `XferAddr`, and the 13 bits of ADDR specify the scan line (lower 10 bits) and table number (upper 3 bits) that should contain this `XferAddr`. The SRAM store is performed with the $\overline{\texttt{DrvAddr}}$, $\overline{\texttt{DrvData}}$, and $\overline{\texttt{WRSram}}$ control lines.
- If the TYPE is 01, the Last Valid Table (LVT) register is loaded from DATA[2:0] and the First Valid Table (FVT) register is loaded from the DATA[5:3] by asserting $\overline{\texttt{DrvData}}$ and `LdVT`. The registers are maintained in `TABPAL`, which also contains a comparator that checks the active table counter against LVT.
- If the TYPE is 11, the lower 2 bits of DATA are stored in a 2-bit register in `TABPAL` by asserting $\overline{\texttt{DrvData}}$ and `LdFlags`. These flags are for "future use" and are currently ignored by the XLM.

Once the FSM has processed the data according to its type, it loops back to wait for another `ForXlm`, `Eol`, or `Eof` signal.

End-of-line update

If `Eol` goes high but `Eof` remains low, `XLMFSM` fetches an 18-bit `XferAddr` from SRAM and instructs every VRAM in the PSN array to use it to perform a row-transfer. If `Eof` is also high, the `XLMFSM` will perform an *end-of-frame* update before the *end-of-line* update.

The XferAddr value at the SRAM location pointed to by TABCTR and LINECTR is latched into the XLMBUFF buffers by asserting DrvCtrs and OESram. Transfer is raised to advertise a valid address on the broadcast bus and to request each of the PSN nodes to perform the row-transfer. LineInc is then asserted to increment the LINECTR in preparation for the next *end-of-line* update.

Next, if XLMFSM's internal "blocked" flag has been set, an *XLM write* is awaiting execution. Either HHold or VHold will be set; if HHold is set, *XLM write* is always performed and the "blocked" flag is cleared; if VHold is set, *XLM write* is not performed unless Eof is also high; if VHold is set but Eof isn't, XLMFSM re-enters the tight loop that checks only for Eof or Eol.

If "blocked" hasn't been set, XLMFSM loops back to wait for another ForXlm, Eol, or Eof. In all cases, ERes is asserted to clear the Eol (and possibly Eof) condition.

End-of-frame update

If Eof is detected, LINECTR and TABCTR are updated before performing the *end-of-line* update (Eol will always be true when Eof is true). LINECTR is reset to zero by asserting LineRes. If Tabsync is high, TABCTR is reset to the value in the FVT register by asserting TabLd. Otherwise, TABCTR is compared against the value in the LVT register by checking Eot. If Eot is high, TABCTR has reached the last valid table and is loaded with FVT. Otherwise, TabInc is asserted to increment TABCTR by one.

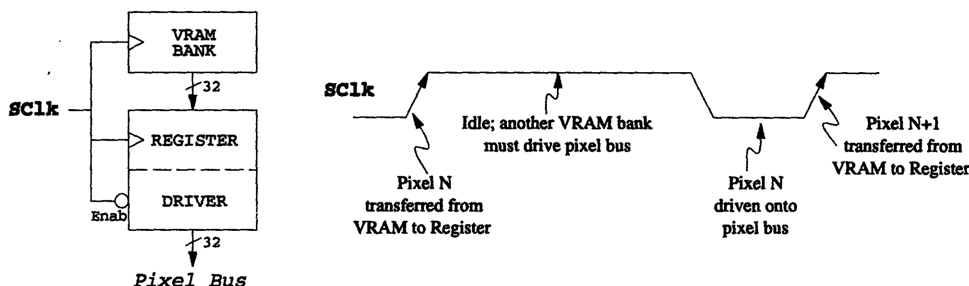
An *end-of-line* update is then performed.

3.2.6 SCM Module

Design Considerations

The Shift Clock Module (SCM) generates the SClk signals used to sequence through the pixels in VRAM. With two VRAM banks per PSN and a maximum of 16 PSNs per video system, the SCM needs to generate and distribute up to 32 separate SClk signals.

The SClk signals do more than shift pixels out of the VRAMs; they determine the very access patterns made into the PSN array and implicitly provide pixel bus arbitration. The figure below shows that between every VRAM and its pixel bus is register/driver chip. SClk is not only fed into the VRAM – it also latches the register *and* is used as the output enable for the driver. So while the low-to-high transition of SClk transfers data from the VRAM to the register, the data is not driven onto the pixel bus until SClk next goes low (which would only be in preparation for the next data transfer). In effect, a pipeline stage has been formed.



The SCM does not support completely arbitrary access patterns made into the set of all VRAM banks. Since each PSN has two VRAM banks driving the same pixel channel, it is pointless to enable both of them onto the channel simultaneously. The two `SClk`s sent to an active PSN are therefore asserted alternately, pulling data from the even VRAM bank one cycle and the odd VRAM bank the next. Indeed, as discussed earlier in section 2.2.6, the VRAM banks were created with this alternating access in mind.

At least four PSNs must contribute pixels to a scan line because precisely four pixels are needed every `LdClk` cycle (`LdClk` is a clock running at one-quarter the true pixel rate; four pixels are loaded into the video controller every `LdClk` cycle, and the video controller spreads the pixels across the next four display columns using the true pixel clock). Therefore, in a minimum configuration with four PSNs, every PSN must contribute to every scan line. But when more PSNs are installed, the update rate of the display can be increased by allowing more nodes to contribute to a single scan line. So the SCM is designed to use either four or eight PSNs per scan line. If eight PSNs are used, four of them deliver pixels during even-numbered `LdClk` cycles and the other four deliver pixels during odd-numbered `LdClk` cycles.

It is important to distinguish the three levels of interleaving:

1. *Interleaving between VRAM banks.* Consecutive pixels from a given PSN are fetched from alternating VRAM banks by interleaving its two `SClk` signals.

2. *Interleaving between PSN nodes.* Four pixels are driven simultaneously by four separate PSNs during every `LdClk` cycle. The video controller ingests these pixels all at once but then distributes them across the next four display columns; this is column interleaving.

3. *Interleaving between groups of PSN nodes.* If eight PSNs are contributing to a scan line, four of them provide pixels during even-numbered `LdClk` cycles and the other four provide pixels during odd-numbered `LdClk` cycles.

If just four PSNs are contributing to a scan line, only the first two types of interleaving occur. Figure 3.6 illustrates the `SClk` timing in this case. Shown are the separate `SClk` signals that drive the two VRAM banks of a single PSN. These two banks alternately drive the connected pixel channel every `LdClk` cycle. Because of the second form of interleaving in the above list, the pixels fetched from the PSN *as a whole* supply columns 0, 4, 8, 12, etc. Because of the first form of interleaving, bank A supplies columns 0, 8, 16, etc. and bank B supplies columns 4, 12, 20, etc. All remaining columns are supplied by the other three participating PSNs.

Since the two `SClk`s shown are complements of each other, one might be tempted to use the true and inverted versions of a single `SClk` supplied by the VC. Although inversion can be accomplished by flipping the `SClk` ECL lines before sending them through the ECL-to-TTL converter, this wouldn't reduce the VC logic needed to generate the `SClks`: the VC must still support the one-of-eight PSN scan line mode.

Figure 3.7 shows the timing for scan lines with eight contributing PSNs. Now, the third form of interleaving means that this PSN *as a whole* supplies columns 0, 8, 16, 24, etc. Of these columns, bank A supplies 0, 16, 32, etc. and bank B supplies 8, 24, 40, etc. The other seven contributing PSNs supply the remaining columns.

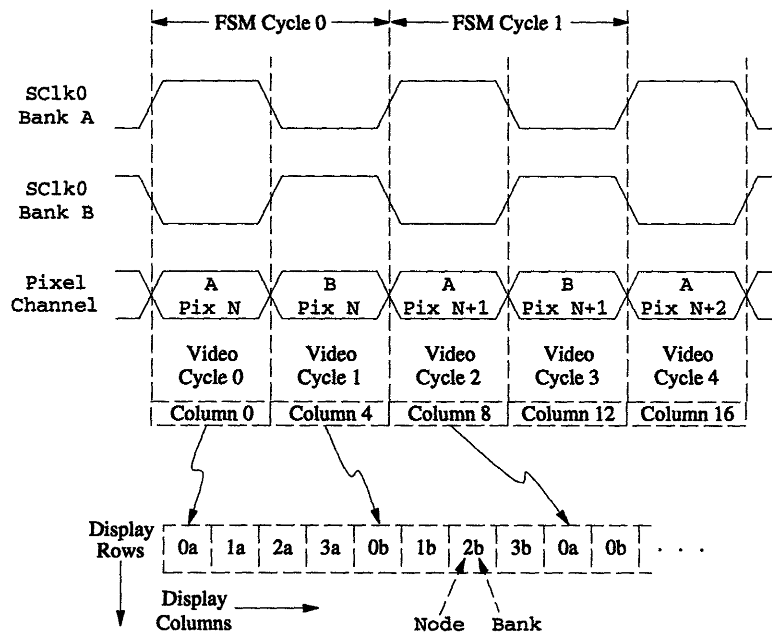


Figure 3.6: Shift Clock Timing for a One-Of-Four PSN

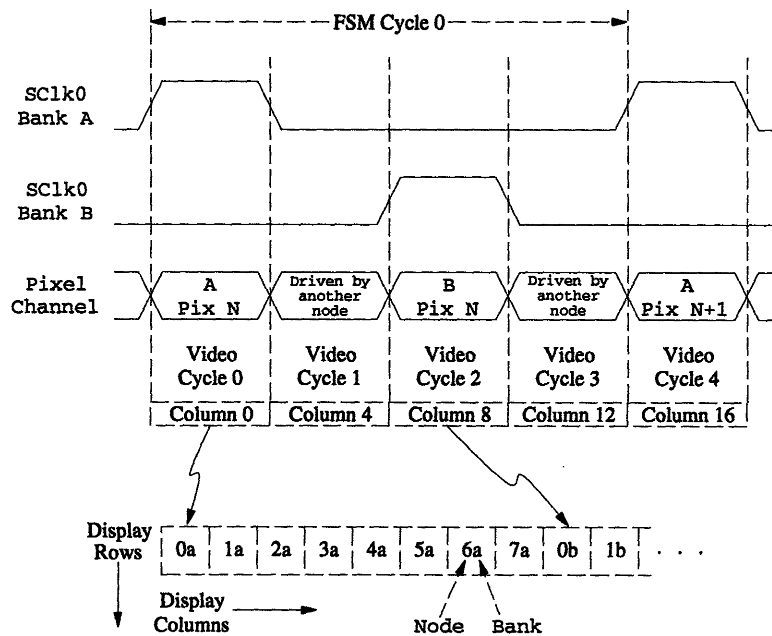


Figure 3.7: Shift Clock Timing for a One-Of-Eight PSN

Although the same four or eight nodes are used for the entire scan line, other scan lines need not use the same nodes. Like the XLM, the SCM uses an SRAM table to fetch the configuration for the current scan line. Unlike the XLM, the SCM does not need to allow up to 1024 unique configurations, one for every scan line; there aren't 1024 ways to access (at most) 16 PSNs. So the SCM allows up to sixteen different configurations to be defined during a single frame, and will sequence through each of them on consecutive scan lines. The SCM wraps around to the first configuration when the last valid (LVL) configuration has been reached, and will continue to cycle through the list for the entire video frame. This list is allowed to have just a single entry if every scan line uses the same PSNs.

In this context, the list of all valid configurations to be sequenced is called a "table." An application may find it desirable to use different tables for different video *frames*. Separate tables can be used to create logical frame buffers, using certain PSNs for one buffer and other PSNs for the next[3]. So the SCM provides space for up to eight tables (each with up to sixteen configuration entries). The active table pointer can be incremented automatically at the end-of-frame. "First Valid Table" and "Last Valid Table" registers specify the range of tables to use; an identical protocol was discussed in the XLM section.

Figure 3.8 depicts the general structure of the SCM. The SRAM in the middle is shared between two blocks clocked at different frequencies. `ScmCtrl` on the left, clocked by the same 40 MHz `FClk` used by most of the VC modules, communicates with the FIFO. One of its responsibilities is to await new `SClk` table data and write it into SRAM.

`ScmCtrl` is also the caretaker of the `Go` flag. Bad table entries can cause damaging bus contention by driving more than one VRAM bank onto the same pixel bus at once. The SCM cannot detect this conflict, but a special measure is taken to prevent contention due to garbage in the SRAM following a power cycle: the SCM will only operate when the `Go` flag has been set. `Go` gets cleared by a system reset or power-cycle, and must be explicitly set by the J-Machine application (after it has initialized the SRAM with valid data).

When `ScmCtrl` sets `Go` at the request of an application, the `SClkGen` block on the right proceeds to fetch data from SRAM to generate up to 32 `SClk` signals. To keep the `SClks` synchronous with the Bt463 video controller, `SClkGen` needs to be clocked by `LdClk`. To determine which `SClks` to assert, `SClkGen` uses that configuration indexed by the line and table counters, which are updated during horizontal and vertical blanking periods respectively. The TTL clocks generated by `SClkGen` are converted to pseudo-ECL logic levels before being transmitted over twisted pairs to the individual VRAM banks.

ScmCtrl Implementation

The overall structure of `ScmCtrl` is similar to the XLM, and includes:

- Registers that accept data from the FIFO;
- An FSM that decodes the FIFO data. Table 3.4 describes how the `DATA` and `ADDR` words are interpreted;
- Flip-flops triggered by the horizontal and vertical blanking periods;

[3]In contrast, separate tables in the XLM allow certain *portions of VRAM* to be used for one logical buffer and other portions for the next.

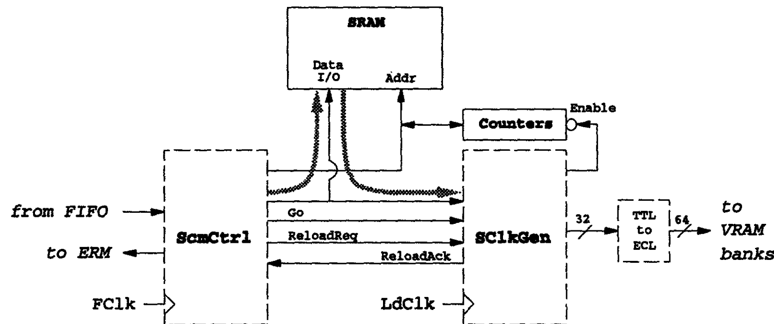
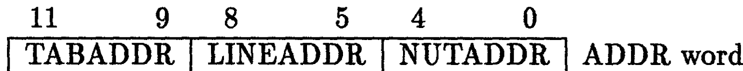


Figure 3.8: General Structure of SCM module

31	30	29 28		3 0	
VH	HH	TYPE	*unused*	DATA	DATA word

VH	VHold - If set, don't perform write until next vertical blanking period		
HH	HHold - If set, don't perform write until next horizontal blanking period		
DATA	The TYPE-specific data to be used		
TYPE	The type of data being specified:		
	00	NUT Entry	Write DATA[3:0] into SRAM location ADDR
	01	Valid Tables	Set registers LVT = DATA[2:0], FVT = DATA[5:3]
	10	Last Valid Line	Set LVL register = DATA[3:0]
	11	Go Flag	Set the Go flag equal to DATA bit 0.



11 9	8 5	4 0	
TABADDR	LINEADDR	NUTADDR	ADDR word

TABADDR	These 3 upper bits specify one of eight tables. The active table is cycled between the FVT and the LVT register values.
LINEADDR	These 4 bits specify one of 16 scan lines. The active line counter is cycled between 0 and the LVL register.
NUTADDR	An index into the Node Usage Table for the given table and line. The Node Usage Table contains 9 entries.

Table 3.4: SCM Usage of Data and Address Bits From FIFO

- PALs which implement the FVT and LVT registers and the Go flag. The PALs also perform line and table counting similar to the way it is accomplished by the XLM. One important difference is that the output enables of these counters are controlled by the SClkGen FSM, not the ScmCtrl FSM. Handshaking takes place between the two FSMs to arbitrate access to the SRAM and counters.

The complete schematics and PAL file listings for the ScmCtrl block appear in the appendix; since it is very similar to the XLM logic, attention shall instead be turned to the SClkGen block.

SClkGen Implementation

The individual SClk signals are best thought in terms of pairs, since each PSN requires two of them for its two VRAM banks. Each of the SClk pairs is implemented as the lower two state bits of a finite state machine that simply loops throughout the entire scan line. Before the scan line begins, the FSMs (one for every PSN) are told how to behave for that line. This "behavior" is determined by three flags:

1. An *active* flag. Unless this flag is set, the FSM remains idle for the entire scan line and the following two flags are ignored.
2. An *odd* flag. Only applicable in *8PSN* mode (below). If set, the initial state for the FSM will be such that the associated PSN supplies pixels during odd-numbered LdClk cycles. Otherwise, the PSN supplies pixels during even-numbered LdClk cycles.
3. An *8PSN* flag. If set, the FSM will loop through the following state transition table:

Q2	Q1	Q0	
0	0	1	(Initial state if *odd* is false)
0	0	0	
0	1	0	
1	0	0	(Initial state if *odd* is true)

If the *8PSN* flag is clear, the FSM will loop through the shorter transition table:

Q2	Q1	Q0	
0	0	1	(Initial state)
0	1	0	

Note that the last two state bits of an "active" FSM create the SClk patterns shown earlier in figures 3.7 and 3.6.

A "Node Usage Table" in SRAM contains the above three flags for every PSN in the system. The table consists of sixteen 2-bit entries and a 1-bit flag using the following format:

3 2	1 0	← *bit*
ConfigN01	ConfigN00	Word 0
ConfigN02	ConfigN03	Word 1
......		
ConfigN15	ConfigN14	Word 7
	8PSN	Word 8

Each `ConfigNxx` field in the table corresponds to a different PSN. The high and low bits of the field specify the *active* and *odd* flags, respectively. The global *8PSN* flag at the end of the table is used by all of the FSMs.

In *8PSN* mode, exactly eight of the 16 entries should have their *active* flag set. Of those eight, four should also have their *odd* flag set. If the *8PSN* flag is not set, then exactly four of the 16 entries should have their *active* flag set, and the *odd* flags are ignored.

Before concluding this section, some examples may help to clarify `SClk` generation:

1. In a system with four PSNs, there is only one valid SCM table because all four PSNs are needed every `LdClk` cycle of each scan line:

`W0:`	1	0	1	0	*Nodes 1 and 0 will be active*
`W1:`	1	0	1	0	*Nodes 3 and 2 will be active*
`W2:`	0	0	0	0	*All other nodes are inactive...*
`W3:`	0	0	0	0	
`W4:`	0	0	0	0	
`W5:`	0	0	0	0	
`W6:`	0	0	0	0	
`W7:`	0	0	0	0	
`W8:`	0	0	0	0	*We're not in 8PSN mode*

This table produces the following access patterns into the VRAM banks:

$$\underbrace{0a\ 1a\ 2a\ 3a}_{LdClk\ 0}\ \underbrace{0b\ 1b\ 2b\ 3b}_{LdClk\ 1}\ \underbrace{0a\ 1a\ 2a\ 3a}_{LdClk\ 2}\ \ldots$$

which uses PSN interleaving within a `LdClk` cycle and VRAM bank interleaving between `LdClk` cycles.

2. In a full video system with 16 PSNs, numerous SCM tables can be defined. The following table retrieves pixels from eight of the 16 PSNs:

`W0:`	1	0	1	0	*Nodes 1 and 0 active during even LdClks*
`W1:`	1	0	1	0	*Nodes 3 and 2 active during even LdClks*
`W2:`	1	1	1	1	*Nodes 5 and 4 active during odd LdClks*
`W3:`	1	1	1	1	*Nodes 7 and 6 active during odd LdClks*
`W4:`	0	0	0	0	*All other nodes are inactive...*
`W5:`	0	0	0	0	
`W6:`	0	0	0	0	
`W7:`	0	0	0	0	
`W8:`	0	0	0	1	*We are in 8PSN mode*

This table produces the following access patterns into the VRAM banks:

$$\underbrace{0a\ 1a\ 2a\ 3a}_{LdClk\ 0}\ \underbrace{4a\ 5a\ 6a\ 7a}_{LdClk\ 1}\ \underbrace{0b\ 1b\ 2b\ 3b}_{LdClk\ 2}\ \underbrace{4b\ 5b\ 6b\ 7b}_{LdClk\ 3}\ \underbrace{0a\ 1a\ 2a\ 3a}_{LdClk\ 4}\ \ldots$$

which interleaves at three levels: the PSNs within a `LdClk` cycle; groups of PSNs between `LdClk` cycles; and VRAM banks between every other `LdClk` cycle.

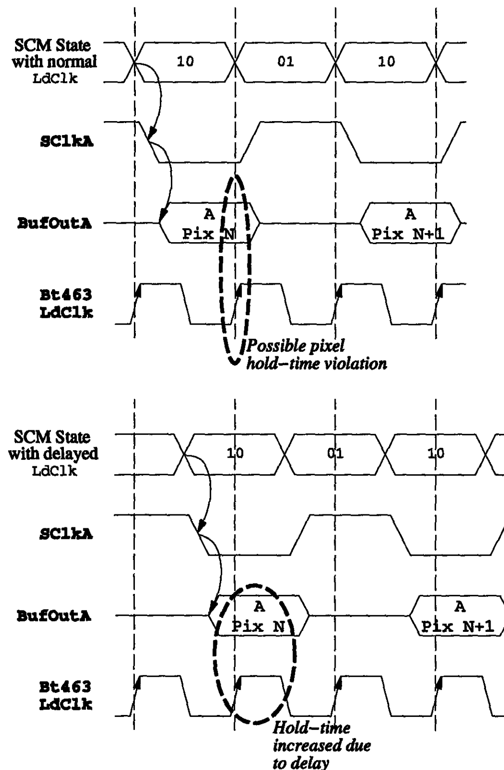


Figure 3.9: Preventing LdClk Timing Violations with a Delay Line

3.2.7 TIM Module

The various timing signals used throughout the video system are supplied by the timing module (TIM). As described in this section, the logic required to generate these signals varies from a simple, self-contained support chip to a more complex FSM with counters and a PROM table.

VidClk and LdClk

The `VidClk` video clock runs at the pixel rate of 111.5 MHz. The only component that requires the full pixel rate is the Bt463 video controller. However, the ECL-level `VidClk` passes through a Brooktree Bt438 support chip on its way to the Bt463.

The Bt438 divides the `VidClk` frequency by four to produce a TTL-level `LdClk` signal that instructs the Bt463 when to accept four new pixel values. The Bt463 requires both `VidClk` and `LdClk` to perform 4:1 pixel multiplexing.

LdClk as delivered to SCM

The `LdClk` signal delivered to the SCM module is used to generate the `SClk` signals sent to the VRAM banks. It is almost identical to the `LdClk` delivered to the Bt463; the only difference is that it is disabled during blanking periods to give the VRAM an opportunity to

perform row-transfers; the VRAM specifications do not allow an active SClk during these operations.

The Bt438 actually produces two versions of LdClk. A version that is always active is sent to the Bt463. Another version that can be disabled is sent to the SCM. The Blank signal described below is used to disable this version of LdClk.

The upper part of the figure 3.9 shows a timing violation that would occur if we were to allow LdClk to *directly* clock the SClk FSMs. The edges of LdClk and SClk would nearly align, which would be unacceptable since SClk is also used as an output enable onto the pixel bus. The hold-time following a rising Bt463 LdClk would not be met. Thus, LdClk is passed through a tapped-delay line before being used by the SClk FSMs. This programmable delay, with a nominal delay of one-half the LdClk period, pushes the output-enable periods further into the next LdClk cycle. As shown in the bottom part of the figure, this allows the hold-time to be met.

Blanking and Sync signals

The Bt463 requires composite Blank and Sync TTL inputs so that it can generate the analog video synchronization signals needed by the monitor. In addition, various modules in the VC need to be alerted when horizontal blanking ($\overline{\mathtt{HBlk}}$) and vertical blanking ($\overline{\mathtt{VBlk}}$) periods begin.

The TIM module generates these four signals with a 13-state FSM, two 16-bit counters, and a small table in PROM. At various stages in the FSM, the table entries are transferred from the PROM to the counters, which then count either LdClk cycles or $\overline{\mathtt{HBlk}}$ cycles (where $\overline{\mathtt{HBlk}}$ itself is generated by the FSM). Those entries in the following table with units of LdClk cycles are based on a 27.875 MHz LdClk (derived from a 111.5 MHz VidClk).

Parameter	Value	Cycles Counted	Time (μs)	Description
HFrontPorch	12	LdClk	0.466	Blanking time between end of scan line and beginning of horiz sync
HSync	38	LdClk	1.40	Duration of horizontal sync pulse
HBackPorch	54	LdClk	1.97	Blanking time between end of horiz sync and active scan line
HActive	339	LdClk	12.20	Duration of active scan line
HLinesActive	897	$\overline{\mathtt{HBlk}}$	–	Number of active horizontal scan lines
HLinesInVFP	6	$\overline{\mathtt{HBlk}}$	–	Number of off-screen scan lines at bottom of display
HLinesInVBP	31	$\overline{\mathtt{HBlk}}$	–	Number of off-screen scan lines at top of display
HActiveInVBlk	411	LdClk	14.78	Time between horiz sync serrations during vertical blanking
HBPaVBP	68	LdClk	2.48	Length of horiz sync serrations during vertical blanking
HLinesInVS	4	$\overline{\mathtt{HBlk}}$	–	Number of horiz sync serrations during vertical sync

31	30	29 28	27	26	25 24	23 16	15 8	7 0
VH	HH	*unused*	RW	WC	C1,C0	Word2	Word1	Word0

VH	VHold - If set, don't perform access until next vertical blanking
HH	HHold - If set, don't perform access until next horiz blanking
RW	Selects either read access (1) or write access (0)
WC	Word Count - Access is a long (1) or short (0)
C1,C0	Control lines fed directly to Bt463; help determine MPU address
Word0	First byte to write in a "long" access (Only byte to write in a "short" access)
Word1	Second byte to write in a "long" access
Word2	Last byte to write in a "long" access

Table 3.5: MPU Usage of Data Bits From FIFO

3.2.8 MPU Module

The MPU module allows the application to read from or write to the MPU interface of the Bt463 video controller. This provides access to things such as the Bt463 control registers and color palettes. Some of these locations are read-only, but most of them are read/write and require initialization to properly set up the Bt463.

A potential source of confusion is the manner in which MPU *reads* are performed by the MDP. The only way to get information (including a request for return data) from the MDP to the MPU is through the FIFO, but the FIFO only accepts data when the MDP *writes* to its EMI. So *both* MPU reads and writes require the MDP to write a value to some memory-mapped address. A flag within the data word itself indicates whether the MPU access should be handled as a read or write.

Figure 3.10 shows the organization of the MPU module. As with most VC modules, some registers (on the left) accept data from the FIFO under the control of an FSM (bottom). Another set of registers (right) provide configuration data read out of the Bt463 to the ERM. The MPU interface of the Bt463 is shown in the middle.

The complete Bt463 MPU interface is detailed in the Brooktree data sheets. Its data bus is only eight bits wide, but some of the configuration entries are 24 bits wide. Thus, depending on the specific location, reads and writes require either one cycle or three cycles. A Word Count (WC) flag in the data sent by the MDP informs the `MPUFSM` how many cycles the access requires, and should be consistent with what the Bt463 expects. The `R/W` input selects either a read or write, and is fed directly from the data word coming from the FIFO.

Table 3.5 details how `MPUFSM` interprets the FIFO data. Note that unlike other modules, none of the `ADDR` FIFO bits are used. The Bt463 has some internal counters which, along with the `C1` and `C0` lines, specify the address. These counters automatically increment after each access, and can be set to specific values through the data lines.

If the MPU access is a read, the appropriate data will be loaded by this module into registers in the ERM module. After the read is complete, the MPU module signals the ERM

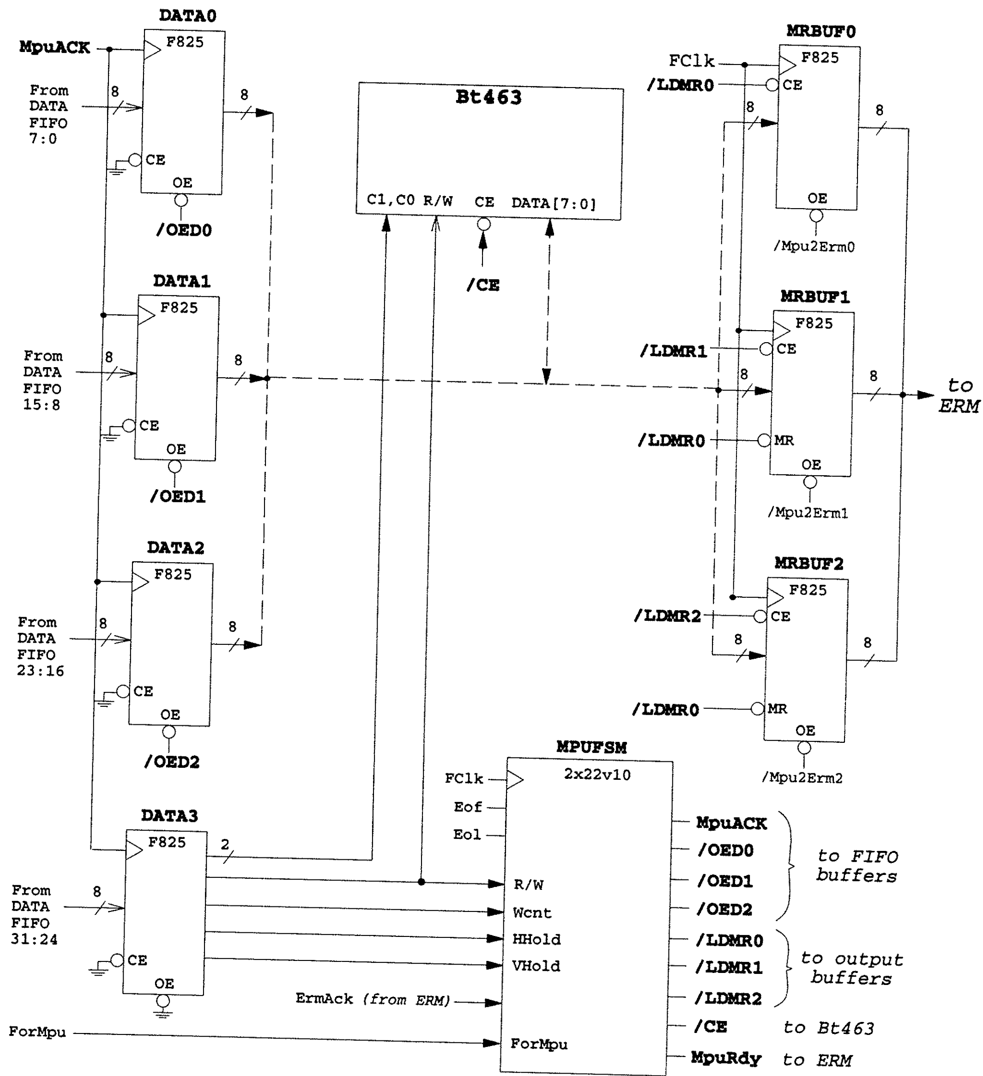


Figure 3.10: MPU Block Diagram

module so that it can in turn provide the data to the MDP. This protocol was described earlier in section 2.2.4.

3.2.9 Bt463

All of the modules described in this chapter have a single unified purpose: to provide control, timing, and pixels for the Bt463 video controller. The Brooktree part was selected because it met two criteria:

- It is configurable. Many RAMDACs have rigid behavior and rely on copious support logic to provide such features as variable colormaps and plane depth. The Bt463, on the other hand, provides internal colormaps and pixel depth that can vary by *pixel*. These two features can be combined to compress multiple pixels into a single 32-bit word; as shown in the next chapter, this compression can be used to increase frame buffer storage and bandwidth.

- It provides 4:1 multiplexing. This scheme requires four pixels to be presented at the same time, but quadruples the period length between fetches. This allows most of the PSN and VC modules to operate at one-quarter the pixel rate, a reasonable speed for TTL logic. If the modules had been required to work at the full 111.5 MHz, the system would require a faster logic family and delicate timing analysis.

The examples in the next chapter demonstrate the flexibility provided by the Bt463 video controller, and the J-Machine video system as a whole.

3.3 Fabrication

Only a single fabrication run of the VC board could be made, but few bug fixes were required. The most visible modification is the addition of a fourth BNC connector next to the red, green, and blue outputs of the Bt463. This connector makes the composite Sync signal (generated by the TIM module) available to the display. The initial design did not include this connector because the Bt463 has the ability to superimpose the Sync signal on top of the green output. Many newer monitors support the "sync on green" standard, but the funds for such a monitor became unavailable. As a result, the system uses an older monitor that requires a separate Sync signal.

Some very specific Bt463 layout recommendations from Brooktree were implemented. For instance, the board fabricator was asked to control the impedance between each of the six signal layers and the power planes to an optimal 75Ω. In addition, two "fingers" delineate the area between the Bt463 and the video output connectors; these fingers do not have any copper in their signal and power layers. They prevent current loops on the board from affecting the video outputs.

Finally, the usual precautions were taken with the clock signals, hand-routing them and providing termination at the end of long runs. The complementary outputs of the various ECL clocks were run parallel to each other to improve their immunity to EMI interference.

Chapter 4

Configuring the Video System

One machine can do the work of fifty ordinary men.
No machine can do the work of one extraordinary man.
– ELBERT GREEN HUBBARD

It is the quality rather than the quantity that matters.
– LUCIUS ANNAEUS SENECA (4 BC - AD 65)

The J-Machine video system supports a variety of configurations. Some of the parameters that establish these configurations need to be changed only when more PSNs are added or a different monitor is used, so they are determined by jumpers and PROMs. Others affect how the application interacts with the system and need to be changed frequently. These are set with software routines that modify various registers and RAM tables.

This chapter presents some sample configurations of the video system. It describes the parameters that can be changed and the effects they produce. It also examines some methods useful in a minimum-sized system to compensate for its smaller bandwidth and storage capacity. This chapter makes liberal use of the terms and concepts discussed in Chapters 2 and 3.

4.1 Number of Pixel Storage Nodes

Up to sixteen PSNs may be installed to increase the aggregate network bandwidth between the J-Machine and video system. Sufficiently parallel applications can use this larger bandwidth to achieve faster updates and smoother animation. The effects of scaling were illustrated earlier in table 2.1. It is not necessary to connect the same number of PSNs to each pixel channel. The VC requires only that at least one PSN connect to each of the four channels.

Two issues should be addressed before installing more than the minimum requirement of four PSNs:

1. *Bandwidth.* Although the maximum frame buffer bandwidth increases with PSN count, it will go unused unless the rendering application has a sufficient degree of parallelism. If only a handful of J-Machine nodes are producing pixel data for a full screen animation, the refresh rate is dominated by computation time, not bandwidth.

2. *Storage capacity.* Improvements in bandwidth cannot be made without an increase in VRAM storage capacity, regardless of the application's desire or ability to use this expensive extra storage.

The second point assumes that one doesn't build socket adapters that allow the 256K-by-4-bit VRAMs to be replaced with less costly VRAMs with a smaller address range. Alternatively, if one is willing to dispense with the flexibility provided by 32-bit words, certain VRAM sockets may be left empty. The value of maintaining 32-bit words is demonstrated later in this chapter.

4.2 Display Resolution

The display resolution of the video system is fixed by the frequency of the pixel clock. This frequency is related to the horizontal resolution (R_H), vertical resolution (R_V), and monitor refresh rate (N_{fps}) by

$$f = \frac{R_H}{\frac{1}{N_{fps} \cdot R_V} - (7 \times 10^{-6})}$$

where $\frac{1}{N_{fps} \cdot R_V}$ is the time allowed per scan line if R_V scan lines must fit within a single screen refresh period ($\frac{1}{N_{fps}}$). Of this time, the RS-343 synchronization signals consume $7\mu s$, with the remaining time available to display R_H columns.

If the pixel clock is changed, so must the TIM module's PROM table entries[1] listed earlier in section 3.2.7. Although the Bt463 is compatible with any pixel clock below 135MHz, an arbitrarily slow pixel clock cannot be used because TIM generates its signals by counting cycles of `LdClk`, whose period is one-quarter that of the pixel clock. If a very slow pixel clock is used, the granularity of the counting process erodes and the synchronization signals fall out of tolerance.

Some monitors look for synchronization signals superimposed on the green video output, while others require a separate sync signal. Although the J-Machine video system was intended for use with the first type of monitor, a `SYNC` connector was added to support the second type. The Bt463 supports both types, selectable with a control register.

The resolution of the monitor currently used by the video system is 1280 columns by 900 rows.

4.3 Starting an Application

This section lists some start-up activities required by a simple J-Machine video application. An interface for each of these activities is provided by a C library call.

[1]The TIM PROM table is large enough to hold 32 separate tables; the active table is specified by jumper pins.

1. *Initializing the Bt463.* These Bt463 registers, accessible through the MPU module, must always be set when an application starts up:

Command_Reg_0	0x40	Select 4:1 multiplexing
Command_Reg_1	0x00	Use upper 4 pixel bits as overlay planes
Command_Reg_2	0x00	No sync on green, no IRE blanking pedestal
Mask_Registers	0xFF	Enable all bits of pixel word

2. *Initializing the window types.* The Bt463 requires that each pixel word be tagged with a 4-bit value called a "window type" (WT). A set of sixteen registers in the Bt463 specify how the word possessing each WT should be interpreted. The WT determines the following attributes of the pixel:

   - Display mode. The display mode can either be true-color (where the R,G, and B values are contained in the pixel itself) or pseudo-color (where the entire pixel is treated as a colormap index from which to retrieve the R,G, and B values).
   - Number of planes. In true-color mode, this corresponds to the number of bits in each R,G, and B field (from 0 to 8). In pseudo-color mode, it corresponds to the number of bits in the colormap index number (from 0 to 9).
   - Starting position within the data word.
   - Starting address of colormap. The colormaps, discussed below, are contained in the Bt463.

   It should be emphasized that the WT tag allows these characteristics to be specified on a pixel-by-pixel basis.

   A simple application using 24-bit true-color pixels might write (`Mode=TrueColor, NumbPlanes=8, Start=0, Colormap=0`) into WT register zero and then ensure that all pixels are tagged with WT=0.

3. *Initializing the colormaps.* Both true-color and pseudo-color display modes use colormaps to translate pixels into values that drive the R, G, and B digital-to-analog converters:

   - In true-color mode, three independent colormaps are indexed by the three pixel fields. A simple application might simply load each 256-entry colormap with values from 0 to 255. An advantage gained by using true-color mode is that all 2^{24} available colors may be specified by a single 24-bit word.
   - In pseudo-color mode, a single colormap is indexed to determine the values for all three DACs. When an application needs to display a pixel with a new color, it allocates a colormap entry and uses its index as the pixel's value. An advantage gained by using pseudo-color mode is that fewer bits are necessary to specify a color.

   Figure 4.1 illustrates the use of colormaps in true-color and pseudo-color modes to produce the same color. The same set of colors is accessible to both modes, but

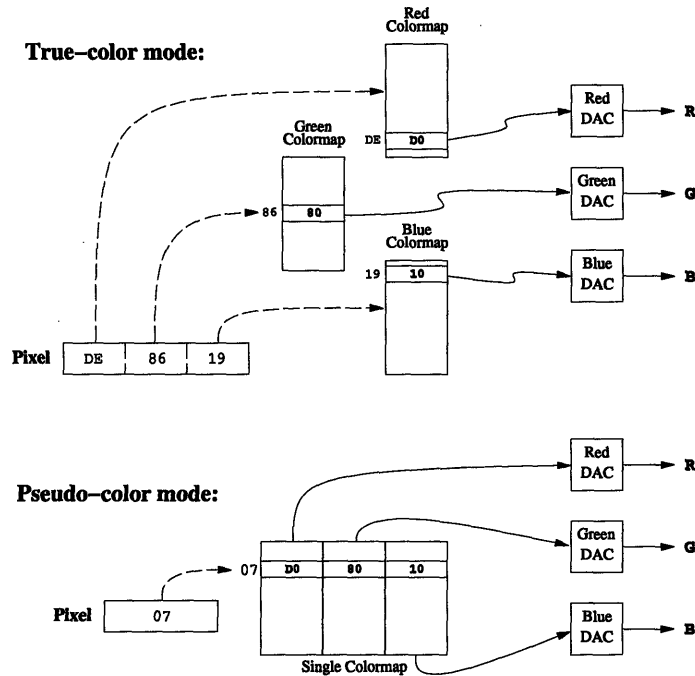


Figure 4.1: Displaying the Same Color Using True-Color and Pseudo-color Modes

in pseudo-color mode only 512 of these colors may be used at once. Note that the true-color example adjusts the values of the R, G, and B fields before sending them to the DACs. The application has set the colormap to implement a technique called "gamma correction," which compensates for non-linearities in the color response of the monitor, as well as non-linearities in intensity as perceived by the human eye.

Multiple colormaps can exist at once within the Bt463, which is why the WT must specify the starting address of the pixel's colormap.

The next two sections describe the steps that are taken after the initialization in this section has been performed.

4.4 Initializing the XLM

At least one XLM table must be initialized to specify the segment of VRAM to display. The table contains a `XferAddr` entry for every scan line of the display. These examples all assume a display resolution of 1280×900, so the last table index used by the XLM is 899.

4.4.1 Single-line frame buffer

A very simple XLM table has all of its entries set to the same `XferAddr`, for example:

```
000: $0
001: $0
002: $0
   ...
898: $0
899: $0
```

Here, row zero of VRAM is used to produce every single scan line. Though not not very practical except for screen fills, this type of table proved useful during testing. One of the first images produced by the video system was a color test pattern; since the pattern simply contained *vertical* bands of color, there was no need to make any line look different from the others.

4.4.2 Single-screen frame buffer

Most of the time, an application would like to allow each scan line to look different from the rest, so the `XferAddr` entries will differ. Assuming that four PSNs are used per scan line, a useful XLM table is:

```
000: $0
001: $200
002: $400
003: $600
   ...
898: $70400
899: $70600
```

Notice that `XferAddr` increases by 0x200 (512 decimal) on successive scan lines. With 1280 columns, the four PSNs are each responsible for providing 320 pixels per scan line, so why increment `XferAddr` by 512? The answer to this question lies in the VRAM architecture.

The VRAM segments `XferAddr` into two parts: the upper nine bits specify which of the VRAM's 512 rows to transfer, and the lower nine bits specify the starting column of the shift register's SAM pointer. Once the SAM pointer reaches the end of the row, it wraps around to the first column of the same row, not the next row. Suppose the second entry in the table above were instead set to 0x140 (320 decimal), which is actually located in the first row of VRAM. Then, only the first $(512 - 320) \times 4 = 768$ columns of the second line would differ from the first. The remaining portion of the second line would begin to duplicate the first line.

Figure 4.2 illustrates the wasted VRAM in this simple configuration. The last 192 words of each VRAM bank become unusable for the first 900 rows. The remaining $1024 - 900 = 124$ rows supply a small amount of off-screen storage, but not enough to fill a screen. Methods to reduce the wasted VRAM are explored later in this chapter.

	Address		
900 active lines	$0	320 active words	192 unused words
	$200	320 active words	192 unused words
	$400	320 active words	192 unused words
	$600	320 active words	192 unused words
	⋮	⋮	⋮
	$70600	320 active words	192 unused words
124 off-screen lines	$70800	320 potentially active words	192 unused words
	$70A00	320 potentially active words	192 unused words
	$70D00	320 potentially active words	192 unused words
	⋮	⋮	⋮

Figure 4.2: Single-buffer VRAM Usage with Four PSNs per Scan Line

	Address	Frame Buffer 1	Frame Buffer 2	Frame Buffer 3	
900 active lines	$0	160 active words	160 active words	160 active words	32 unused words
	$200	160 active words	160 active words	160 active words	32 unused words
	$400	160 active words	160 active words	160 active words	32 unused words
	$600	160 active words	160 active words	160 active words	32 unused words
	⋮		⋮		⋮
	$70600	160 active words	160 active words	160 active words	32 unused words
124 off-screen lines	$70800	160 potentially active words	160 potentially active words	160 potentially active words	32 unused words
	$70A00	160 potentially active words	160 potentially active words	160 potentially active words	32 unused words
	$70D00	160 potentially active words	160 potentially active words	160 potentially active words	32 unused words
	⋮		⋮		⋮

Figure 4.3: Multi-buffer VRAM Usage with Eight PSNs per Scan Line

4.4.3 Multiple-screen frame buffer

Unless the type of pixel compaction described later is used, a system with four PSNs has only enough storage for the single buffer just described. However, if eight PSNs are installed, the following tables can be used to specify three separate frame buffers:

```
000: $0          000: $A0          000: $140
001: $200        001: $2A0         001: $340
002: $400        002: $4A0         002: $540
003: $600        003: $6A0         003: $740
  ...
898: $70400      898: $704A0       898: $70540
899: $70600      899: $706A0       899: $70740
```

Figure 4.3 shows only 32 words per VRAM row are unusable in *8PSN* mode, as compared with the 192 words in *4PSN* mode. Note that the amount of wasted VRAM depends on both the horizontal screen resolution R_H and the number of PSNs per scan line. The fraction of wasted VRAM in *8PSN* mode is:

$$\frac{512 \ mod \ \frac{R_H}{8}}{512}$$

4.5 Initializing the SCM

With only four PSNs, the installed video system must configure the SCM table as shown on page 53; all PSNs will drive during all `LdClk` cycles, and the pixels will be pulled from alternating VRAM banks.

If multiple frame buffers are desired in a video system with more than four PSNs, several models can be used:

1. *Each PSN contains a portion of all buffers.* This is accomplished with appropriate XLM tables as described in the last section.
2. *Each PSN is dedicated to a subset of the buffers.* Multiple SCM tables are needed for this approach, which holds the most benefit for full 16-PSN systems.
3. *A combination of* 1 *and* 2.

The first approach is best suited to applications that have even load distribution throughout the J-Machine processor array; the network traffic to the video system would then be evenly distributed as well. The second approach can be used by applications that divide the processor array into upper and lower portions. The two sub-arrays would then send their messages exclusively to the upper and lower groups of PSN boards, respectively.

4.6 Classes of Pixel Messages

Once the necessary registers and tables have been initialized, the application can begin to write pixels to the frame buffer. The J-Machine messages that encapsulate these pixels are handled by drivers that are loaded into the PSN MDPs. The messages fall into three classes, each with different costs and benefits.

4.6.1 Address/Pixel List

An Address/Pixel List (APL) contains a list of pixel values and the explicit VRAM addresses to which those pixels should be written. For example, the following APL sent to one of the PSNs would produce a 4-pixel long vertical line in the upper left region of the display:

```
0x000: $de
0x200: $de
0x400: $de
0x600: $de
```

The precise column of the line depends on which PSN receives the message and how the XLM was initialized. The precise color of the line depends on the colormap tables and on the display mode of the pixel. When the PSN receives this message, it simply uses each address as an offset from 0x80000 (the memory-mapped address for the beginning of VRAM) and copies each pixel until the list is exhausted. The addresses of this vertical line increment in steps of 0x200 for reasons discussed earlier in section 4.4.2.

Due to the column-wise interleaving of PSNs, a continuous *horizontal* line requires that APL messages be sent to all four (or eight) contributing PSNs. If messages were sent to only one PSN, only every fourth (or eighth) column would be drawn. Assuming that the SCM is in *4PSN* mode, the following APL sent to *all four* PSNs produces a 12-pixel wide horizontal line:

```
0x000: $de
0x001: $de
0x002: $de
0x003: $de
```

Here the addresses only increment in steps of one, but the corresponding columns are implicitly incrementing in steps of four.

4.6.2 Raster Operations

A raster operation (ROP) is a primitive graphic operation performed on the frame buffer data. The supported operations and the contents of their corresponding ROP messages are:

ROP	Message contents
Draw Points	(x_1, y_1, pix_1), (x_2, y_2, pix_2), ...
Draw HPoints	$x, y, pix_1, pix_2, pix_3, ...$ Increments x between pixels
Draw VPoints	Same as above, but increments y between pixels
Draw HLines	(x_1, y_1, pix_1, len_1), (x_2, y_2, pix_2, len_2), ... Increments x by one for len iterations
Draw VLines	Same as above, but increments y

These operations were selected to form the basis of a rudimentary graphics library. They serve as templates for more elaborate routines that can handle objects such as arbitrary lines, triangles, and other polygons.

Just as with APL messages, continuous horizontal lines can only be drawn by sending the same ROP to all four (or eight) PSNs contributing to a scan line. Presently, the line "length" specified in *Draw Hlines* is treated as an iteration count, so the corresponding line is four times as long.

ROPs do not contain actual VRAM addresses; vertices are instead specified using the display (x, y) coordinate system. To perform coordinate-to-address translation, the PSNs need to know the contents of the XLM and SCM tables. In principle, the driver can accommodate any XLM and SCM table, but for testing purposes the driver assumes the tables have been initialized to particular values[2]. The driver also assumes 24-bit pixels are being used; as described later in this chapter, this need not be the case.

The key advantages to using ROPs instead of APLs are:

1. *Simpler interface.* The library provides an abstraction that hides the coordinate-to-address translation from the application.

2. *Smaller messages.* The messages sent to the PSNs are generally much smaller than APLs, causing less congestion of the network and buffers.

On the other hand, several important drawbacks of ROPs are:

1. *Slower pixel updates.* The MDPs on the PSNs must now do more than simply copy data; they must supply varying degrees of computation. Although the routines listed above are not very compute-intensive, every cycle "wasted" on computation cuts into the pixel update rate.

2. *Fixed format.* As noted earlier, the present driver assumes a fixed format for the XLM, SCM, and WT tables. Although a driver can be written to accommodate dynamic changes to these tables, this implies that additional parameters would need to be fetched for every ROP (to determine the current configuration). This would exacerbate the effects of item (1). A compromise solution would have different drivers compiled for different table configurations; the appropriate driver would then be loaded into the PSNs during application set-up.

A very robust application will assume all responsibility for coordinate-to-address translation itself; it will distribute the computation load for ROP processing across its own nodes and use the PSN MDPs simply to copy APL messages into VRAM. This load distribution would help combat a major weakness of the MDP: its lack of native floating-point support.

Less demanding applications can capitalize on the abstraction offered by PSN ROP processing, but at the expense of the pixel update rate.

4.6.3 Proxy Messages

The dimension-ordered J-Machine network routes messages first in the x dimension, then in y, and finally in z. If the PSNs boards were part of the J-Machine cube, this routing algorithm would be sufficient. However, the boards actually jut out from the cube along

[2]Specifically, it assumes the XLM entries are 0x00000, 0x00200, 0x00400, etc. and that the SCM is in *4PSN* mode.

the y dimension, and they do not form z-dimension connections with their neighboring peripheral boards. As a result, J-Machine messages can only reach a PSN if they originate from MDPs on the same z-plane as that PSN. Messages generated on any other plane will reach the appropriate x and y coordinates, but never reach the z destination.

As a workaround, one or more "proxy" J-Machine nodes are allocated on each z plane with a PSN board attached. A node on another plane can send its message directly to the proxy node, which then re-sends the message to the PSN. A proxy message is not a distinct class of pixel message; rather, it is an encapsulation of an APL, or ROP message.

Proxy nodes add a small latency to the time required for pixel message delivery. But as long as there are at least as many proxies as there are PSNs, they do not significantly reduce the bandwidth to the peripheral boards. Care must be taken, however, to evenly distribute the load of proxy messages among the proxy nodes.

A proxy node is also needed to deliver messages to the Video Controller board. However, this proxy is also needed by the VC MDP to return messages to the application.

4.7 Increasing Effective Storage

A video system with four PSNs has only enough VRAM storage for 1.12 frame buffers using 24-bit pixels. This section describes a way to gain effective storage at the expense of color range.

Section 4.3 introduced the concept of window types. A 4-bit WT attached to each 24-bit data word determines how that data word is interpreted, including the size (`NPlanes`) and offset (`Start`) of the pixel value. Normally a true-color application[3] would specify 24-bit pixels so as to access all 16 million colors available. But if it doesn't actually require such a large color range, the size of the pixels can be reduced and multiple pixels can be stored in the same data word. Each pixel would belong to a different frame buffer, and the application would select the active frame buffer by modifying the `Start` field of the WT register.

Figure 4.4 demonstrates how a four-PSN system can use this method to achieve two 12-bit deep frame buffers. As indicated in the table below, this method can be used to achieve as many as 48 frame buffers in a four-PSN system; only two colors are allowed in this extreme case.

Pixel Size	Max number of colors	Effective storage
24	16 million	2 MPixels
12	4 thousand	4 MPixels
1	2	48 MPixels

4.8 Hardware Scrolling and Stretching

The XLM provides a convenient method to perform scrolling of an image without the need to shuffle pixels in and out of VRAM. Vertical scrolling is easily accomplished by rotating

[3]The methods in this section also apply to pseudo-color applications.

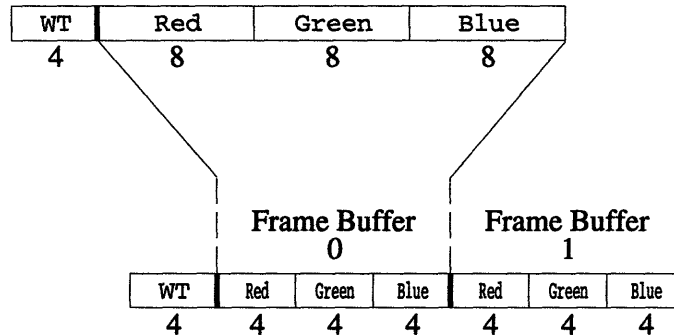


Figure 4.4: Increasing Effective Storage through Pixel Compaction

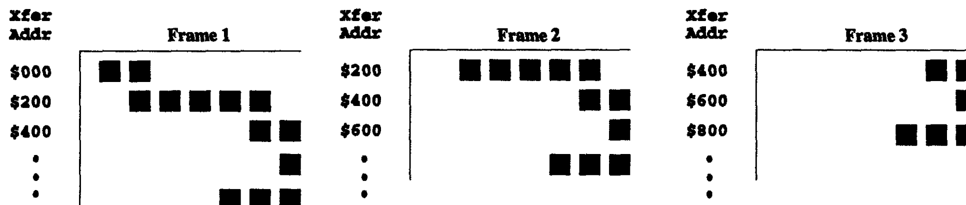


Figure 4.5: Using the XLM to Scroll an Image Vertically

the entries in the XLM table. Figure 4.5 depicts three successive video frames and the XLM tables used to scroll the image upward one row per frame.

Limited horizontal scrolling is also available. Here, the "wasted" VRAM discussed earlier provides a buffer between the left and right edges of the pixel row. The width of this buffer determines the degree of horizontal scrolling available before the image begins to spill across multiple scan lines.

The XLM can also be used to stretch the image vertically. Here, two or more successive table entries would contain the same `XferAddr`. However, due to the fixed nature of `SClk` generation, horizontal stretching is not available.

4.9 Stereo Monitors

Two independent video systems, each with their own VC and array of PSNs, can be combined within the J-Machine to generate three-dimensional images on stereo monitors. To couple the video systems, their VC boards must be synchronized so that one provides images during even frames and the other provides images during odd frames. An observer wears glasses which alternately block the left and right eyes.

Existing jumpers and external connectors can be used accommodate this behavior. In stereo mode, the two VC boards are configured differently:

- One VC is the master; as usual, its Bt438 derives the quarter-rate `LdClk` from the

PixClk oscillator and sends it to the Bt463 and TIM module. But it also passes the signal through a TTL-to-ECL converter, out onto a ribbon cable, and over to the other VC. It sends its local $\overline{\texttt{Reset}}$ signal to the other VC as well.

- The other VC is the slave; although it still requires a local PixClk oscillator, it uses the master's LdClk as its own LdClk. It also feeds the master's $\overline{\texttt{Reset}}$ line to its TIM FSM.

The goal of sharing a single LdClk and TIM $\overline{\texttt{Reset}}$ signal is to align the HBLK, VBLK, and SYNC signals produced by TIM. This insures that the scan lines and frames are in sync. Note that because the slave's LdClk is not directly derived from its PixClk, the edges of the two will not necessarily be aligned. This could potentially wreak havoc with a video controller, which uses both signals. However, the Bt463 has an internal phase-locked loop that resolves any delay between their edges; as long as the periods of PixClk and LdClk do not vary, the PLL will align the signal edges.

Chapter 5

Conclusion

What lies behind us and what lies before us are tiny matters compared to what lies within us.
– RALPH WALDO EMERSON

We shall not cease from exploration; and the end of all our exploring will be to arrive where we started and to know the place for the first time.
– T.S. ELLIOT in *Little Gidding*

5.1 Summary

This thesis described the Distributed Frame Buffer, a video system built for the J-Machine concurrent computer. The DFB provides a high-bandwidth data path from the J-Machine to video memory. The combination of this data path and the aggregate processing power of the J-Machine provides a valuable research tool into high-performance rendering algorithms.

Modern graphics applications such as virtual reality require tremendous processing power to transform scenes according to some observer's viewpoint. After the objects have been transformed, additional processing is needed to scan-convert the objects into pixels. Many existing systems try to satisfy the demands of these two processing stages with a mixture of parallel architecture and hardware pipelines. Unfortunately, such systems must make assumptions about the specific rendering algorithm being used. They are optimized to perform very well if these assumptions are correct, but suffer significantly if they are not.

The J-Machine and its DFB provide a more flexible approach. Rather than fixing the rendering algorithm in hardware, the J-Machine computing nodes can implement the algorithm in software. A unique trait of the J-Machine is its built-in support for fine-grain tasks and data objects. This support distinguishes the J-Machine from other massively-parallel computers and makes it much more suited to graphics-oriented distributed algorithms. Since many algorithms also contain operations that must be serialized, the J-Machine nodes can be organized into a pipeline. The number of nodes allocated to each stage of the pipeline can be modified dynamically to meet varying demands. The DFB is distributed to avoid the bottleneck of a single video bus. It is also scalable, allowing a tradeoff between price

and performance.

The cost of all this flexibility is that careful attention must be paid to load distribution. A distributed graphics algorithm can slow down to a crawl if its computation isn't balanced across the nodes; even though the nodes can often compute their pixels locally, the video frame will not normally be able to advance until it has been completely updated.

The DFB consists of two types of nodes. An array of Pixel Storage Nodes accepts incoming pixel messages from the J-Machine, and a Video Controller node repeatedly fetches frame buffer data from the PSNs to display the pixels on a monitor. For every PSN added to the array, an additional 8.75 MBytes/sec of bandwidth and 2 MBytes of storage is gained.

The PSNs can be configured to understand different classes of pixel messages. Simple address/pixel lists require little processing by the PSN and must be used to achieve maximum bandwidth into the system. More abstract raster operations can be specified instead, but these reduce the frame buffer bandwidth and should be used only by less-demanding applications.

The VC has many configuration options: it can work with monitor resolutions up to 1280×1024; the format of pixel data is flexible, and allows true-color, pseudo-color, and plane depth to be specified on a pixel-by-pixel basis; the portion of VRAM corresponding to an area of the screen can be changed dynamically by the application, allowing the VRAM to be segmented into multiple logical buffers; the VC can accommodate from four to sixteen PSNs; and it can be coupled with another VC to drive stereo monitors.

5.2 Further Work

Now that the hardware foundation exists, much software remains to be developed to validate the performance of the video system under realistic demands. Although a simple graphics library has been developed, a true test of the system requires a full-fledged rendering algorithm executing in the J-Machine cube. A rigorous application would also test the fine-grain mechanisms provided by the J-Machine, a challenge that has not been fully met to date.

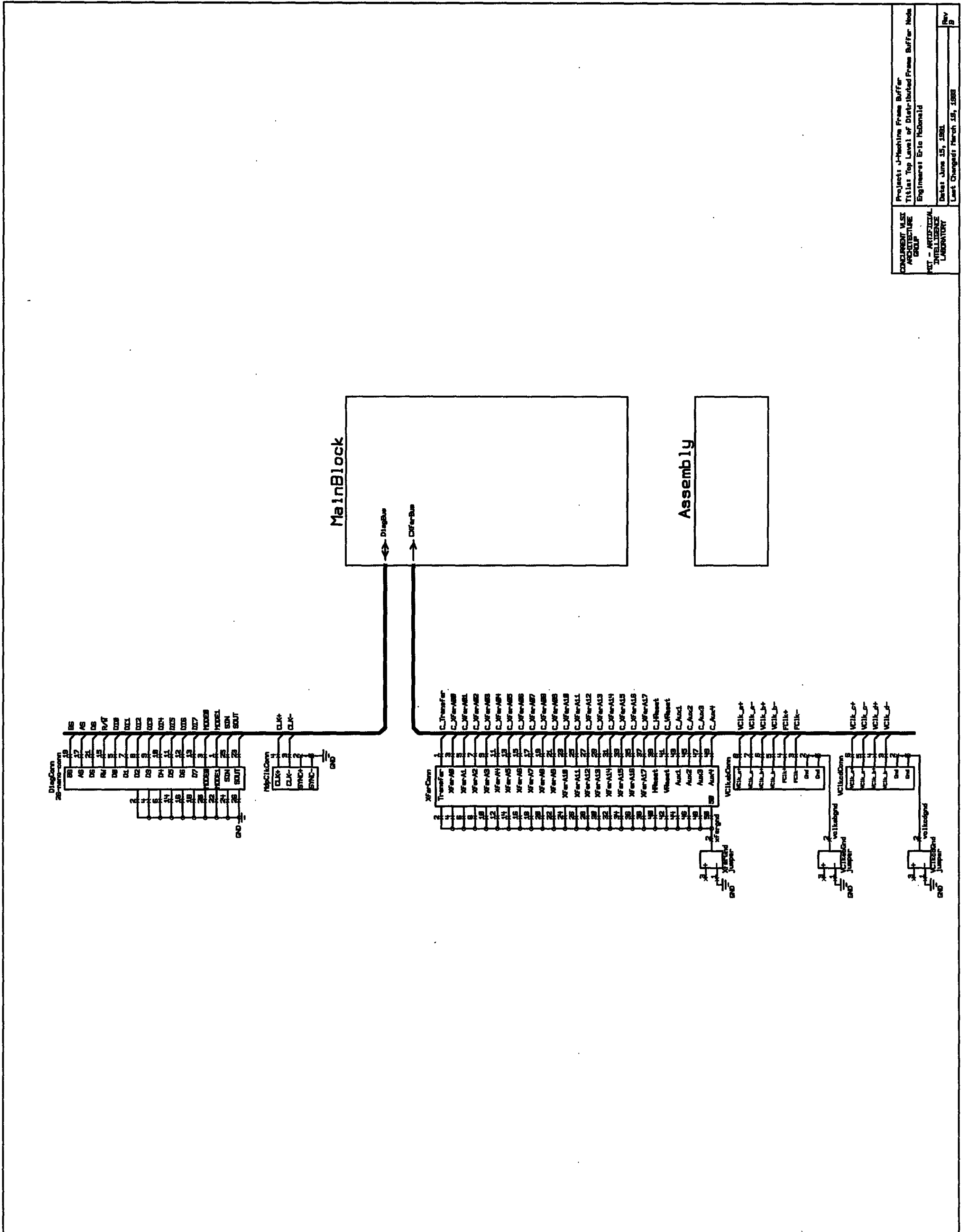
Such an application would also demonstrate how well a heavily-distributed algorithm can compensate for the lack of floating-point hardware in the J-Machine nodes. The first stage of rendering – object transformation – is very floating-point intensive and would require many costly software calls. Would this cripple a demanding J-Machine graphics application, or can the task be sufficiently partitioned and executed in parallel to reduce its overall cost?

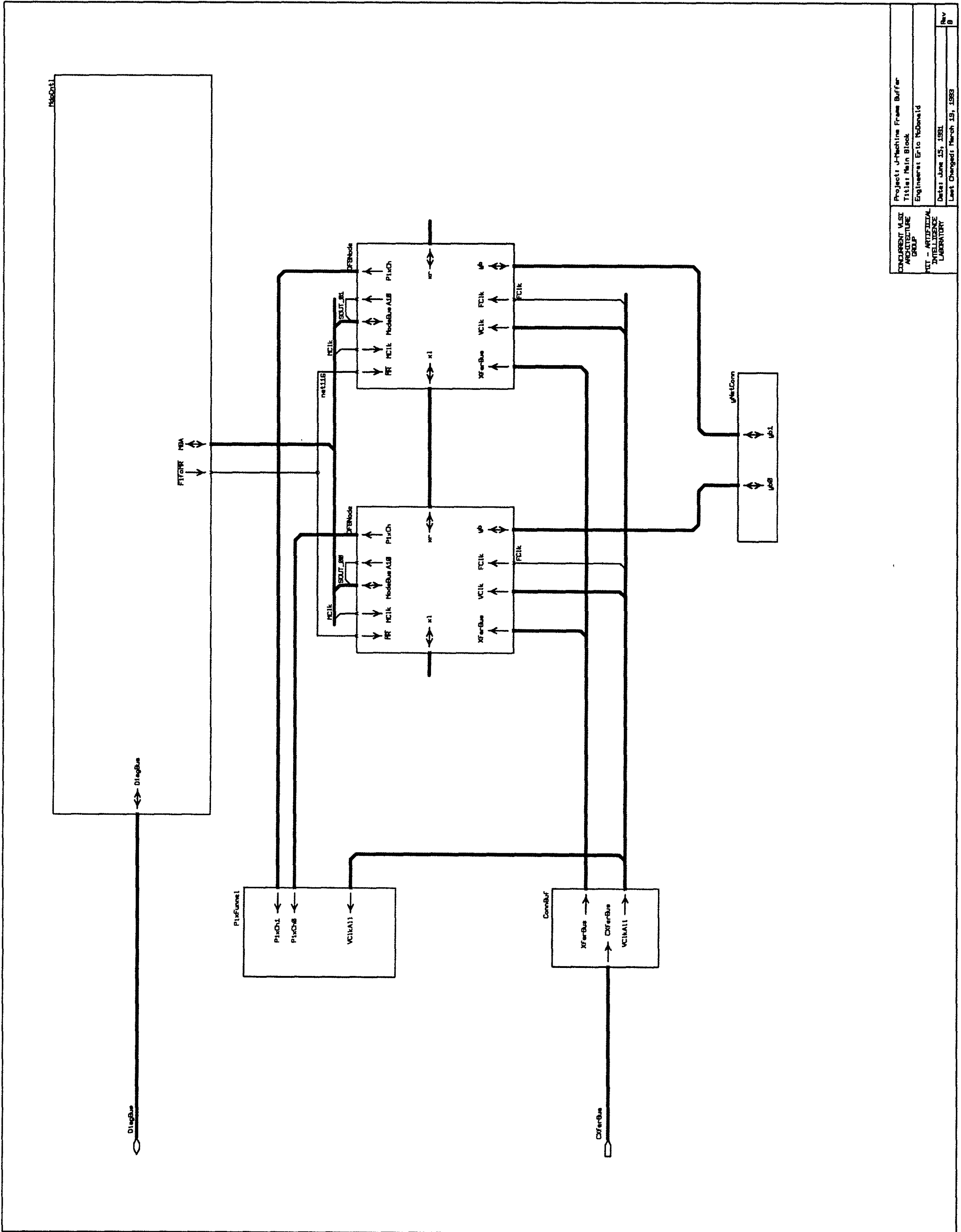
Recent research suggests that the use of multiple frame buffers hampers virtual reality applications by requiring full-frame updates before the next frame can begin [2]. This often introduces enough visual lag during head movement to frustrate the observer; for a given frame buffer bandwidth, it may be more important to update the observer's viewpoint immediately (with partial images) than to supply full (but jerky) updates. The DFB has enough flexibility to investigate this claim more thoroughly.

The fine-grain mechanisms provided by the J-Machine lend themselves naturally to many inherently distributed graphics algorithms. The combination of the J-Machine and the Distributed Frame Buffer can supply a fertile and flexible testing ground for these and future algorithms.

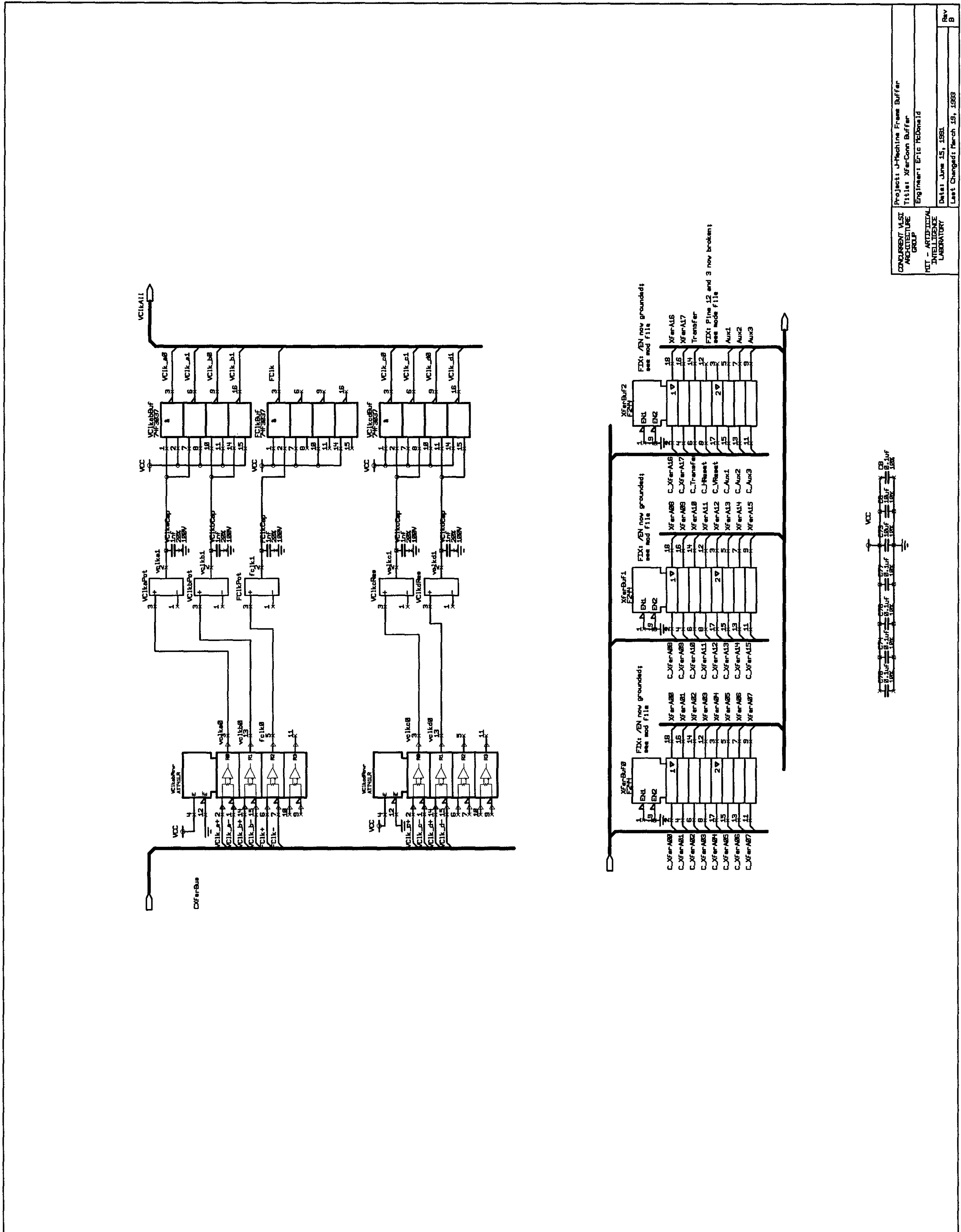
Appendix A

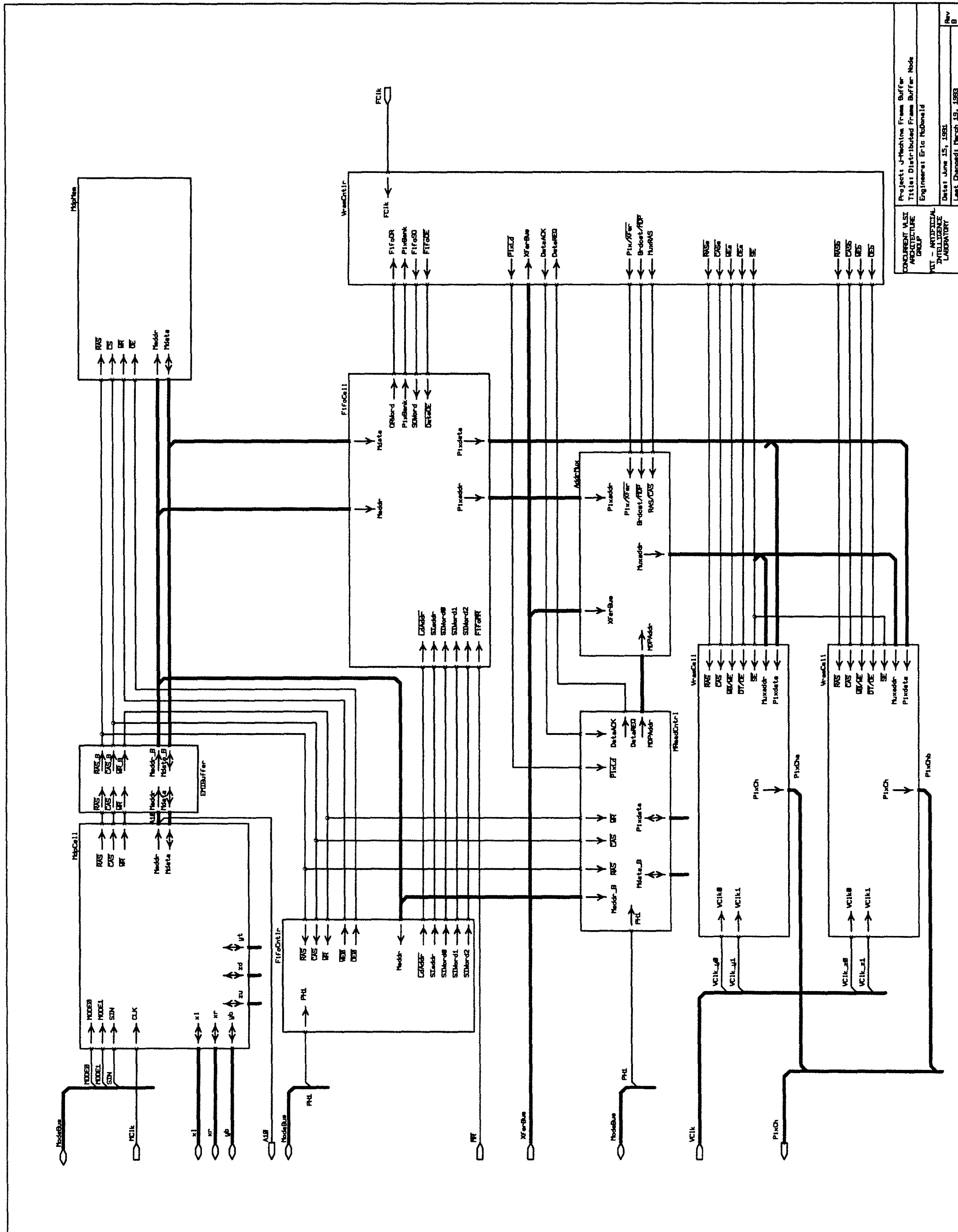
PSN Schematics and PAL Files

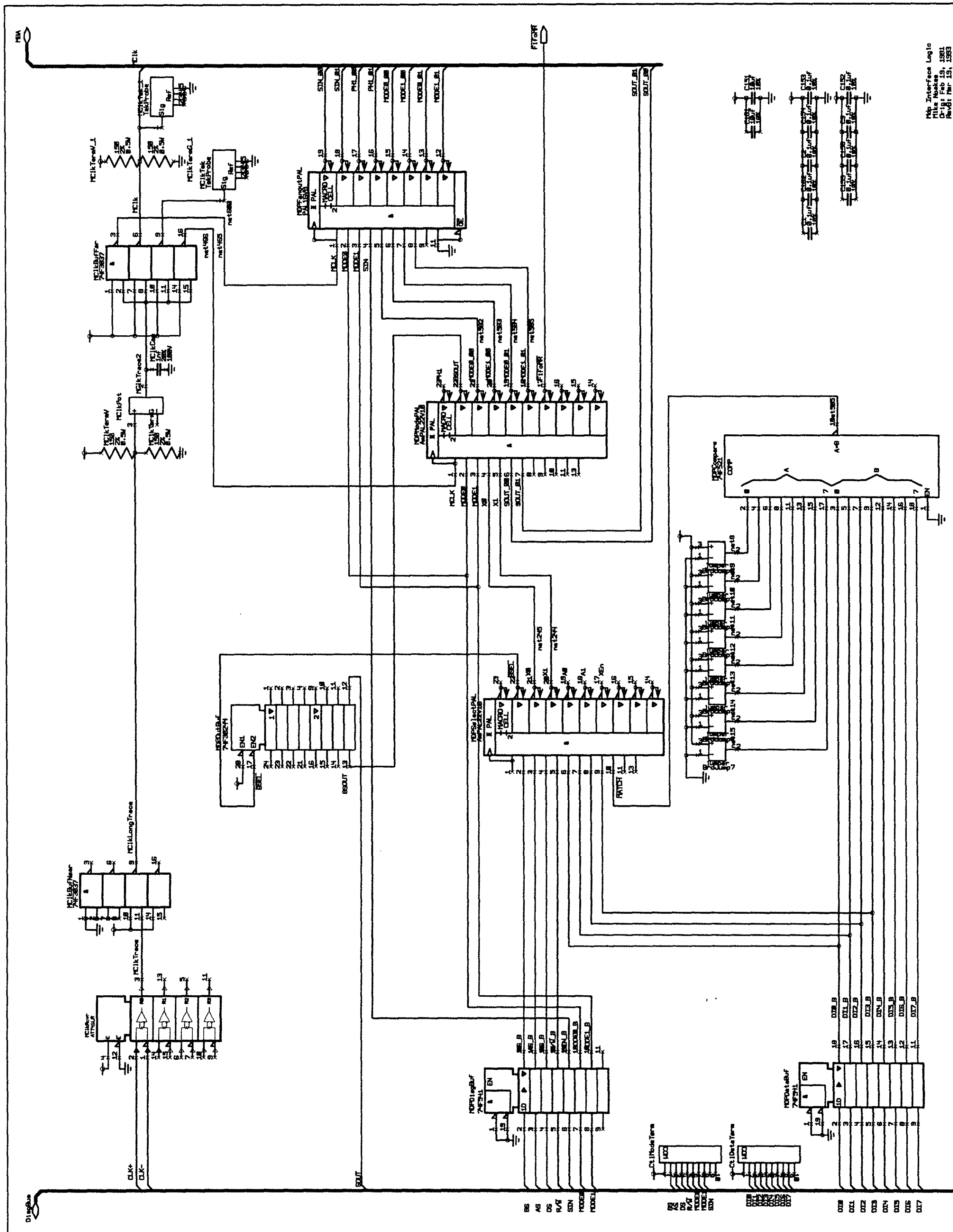

MainBlock
Assembly



Project: J-Machine Frame Buffer
Title: Main Block
Engineer: Eric McDonald
Date: June 15, 1991
Last Changed: March 19, 1993
Rev
B
CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY
MdpCntl
DiagBus
DiagBus
DFBNode
DFBNode
PixCh
ModeBus
MClk
RR
FClk
VClk
XferBus
yNetConn
yb1
yb8
PixFunnel
PixCh1
PixCh8
VClkAll
ConnBuf
XferBus
CXferBus
VClkAll
CXferBus
net116
MBA







mdpmode.abl

```
module mdpmode

title 'MDP Diag Interface - MDPModePal    Eric McDonald    January 15, 1994'

mdpmode  device 'P22V10';

"*** Inputs

Mclk                                    Pin 1;
Mmode0, Mmode1                          Pin 2, 3;
X0, X1                                  Pin 4, 5;
Sout0, Sout1                            Pin 6, 7;

"*** Outputs

!Ph1                                    Pin 23;
BSout                                   Pin 22;
Mode0_0, Mode1_0                        Pin 21, 20 istype 'reg,buffer';
Mode0_1, Mode1_1                        Pin 19, 18 istype 'reg,buffer';
!FifoReset                              Pin 17;

"*** Buses

Mmode    = [Mmode1 , Mmode0];
Mode0    = [Mode1_0 , Mode0_0];
Mode1    = [Mode1_1 , Mode0_1];

"*** Aliases

NOP      = [  0 ,  0  ];
EX       = [  0 ,  1  ];
SH       = [  1 ,  0  ];
RESET    = [  1 ,  1  ];

H, L, X = 1, 0, .X.;

"*** Equations

equations

  " The registers are clocked by MCLK
  [Mode0, Mode1, Ph1].clk = Mclk;

  " Ph1 held hi by reset (stops during Ph1 hi).
  Ph1   := (Ph1 == L) # (Mmode == RESET);

  " The new Mode arrives during ph1.
  " Normally, pass it on ph2.  Hold until next ph1 when reset goes away
  " Note:  (Ph1 == L) means that PH1 is about to go hi
  Mode0 := (X0 == H) & (
           ((Mmode == RESET) & Mmode)                                         #

           ((Mmode != RESET) & (Mode0.fb == RESET) & (Ph1 == H) & Mode0.fb)   #
           ((Mmode != RESET) & (Mode0.fb == RESET) & (Ph1 == L) & Mmode    )  #

           ((Mmode != RESET) & (Mode0.fb != RESET) & (Ph1 == H) & Mmode    )  #
           ((Mmode != RESET) & (Mode0.fb != RESET) & (Ph1 == L) & Mode0.fb)
```

mdpmode.abl

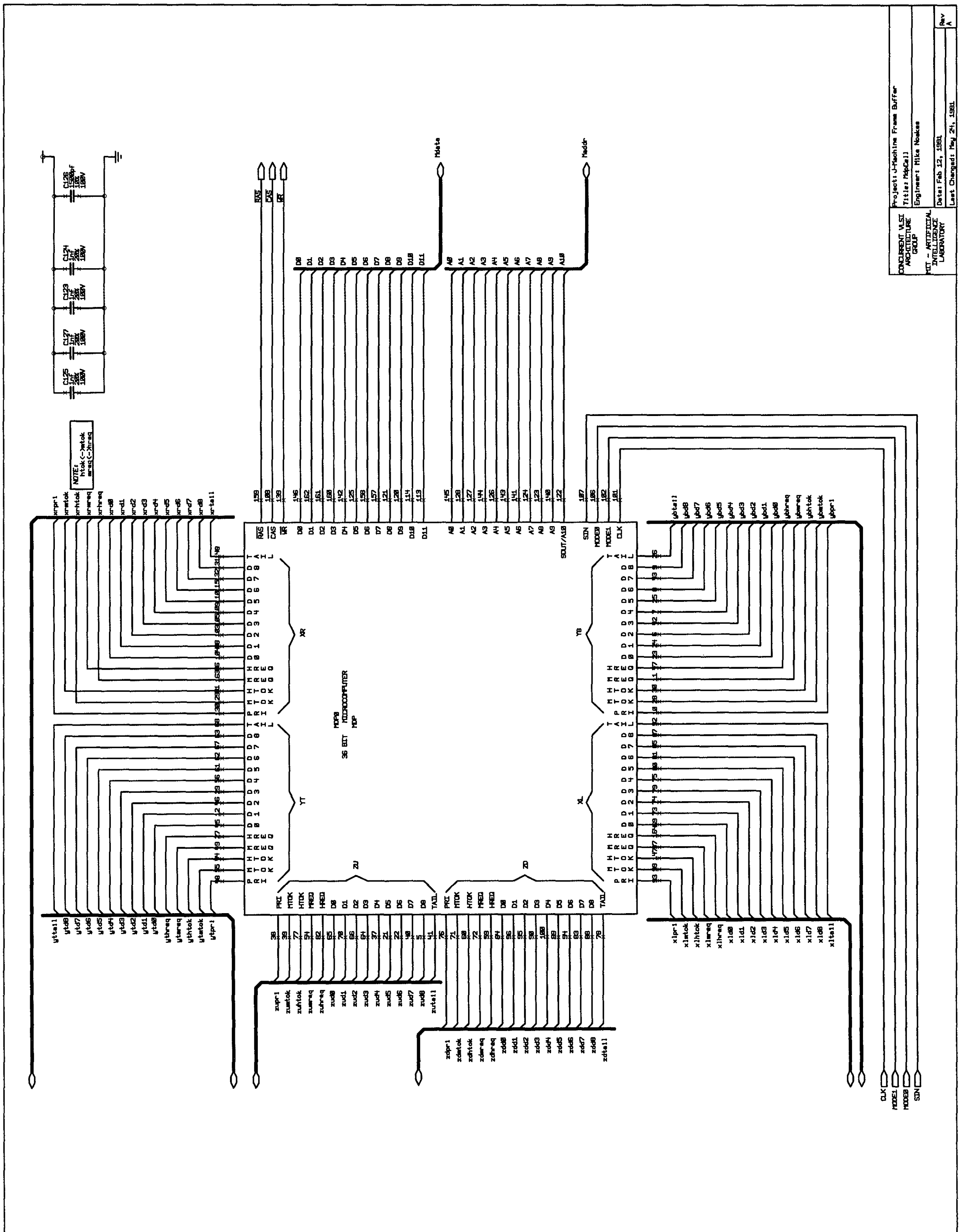
```
          );

  Mode1 := (X1 == H) & (
          ((Mmode == RESET) & Mmode)                                    #

          ((Mmode != RESET) & (Mode1.fb == RESET) & (Ph1 == H) & Mode1.fb)  #
          ((Mmode != RESET) & (Mode1.fb == RESET) & (Ph1 == L) & Mmode    )  #

          ((Mmode != RESET) & (Mode1.fb != RESET) & (Ph1 == H) & Mmode    )  #
          ((Mmode != RESET) & (Mode1.fb != RESET) & (Ph1 == L) & Mode1.fb)
          );

  BSout  = (Sout0 & X0) # (Sout1 & X1);

  FifoReset = ((X0 == H) # (X1 == H)) & (Mmode == RESET) & Ph1;

end mdpmode
```

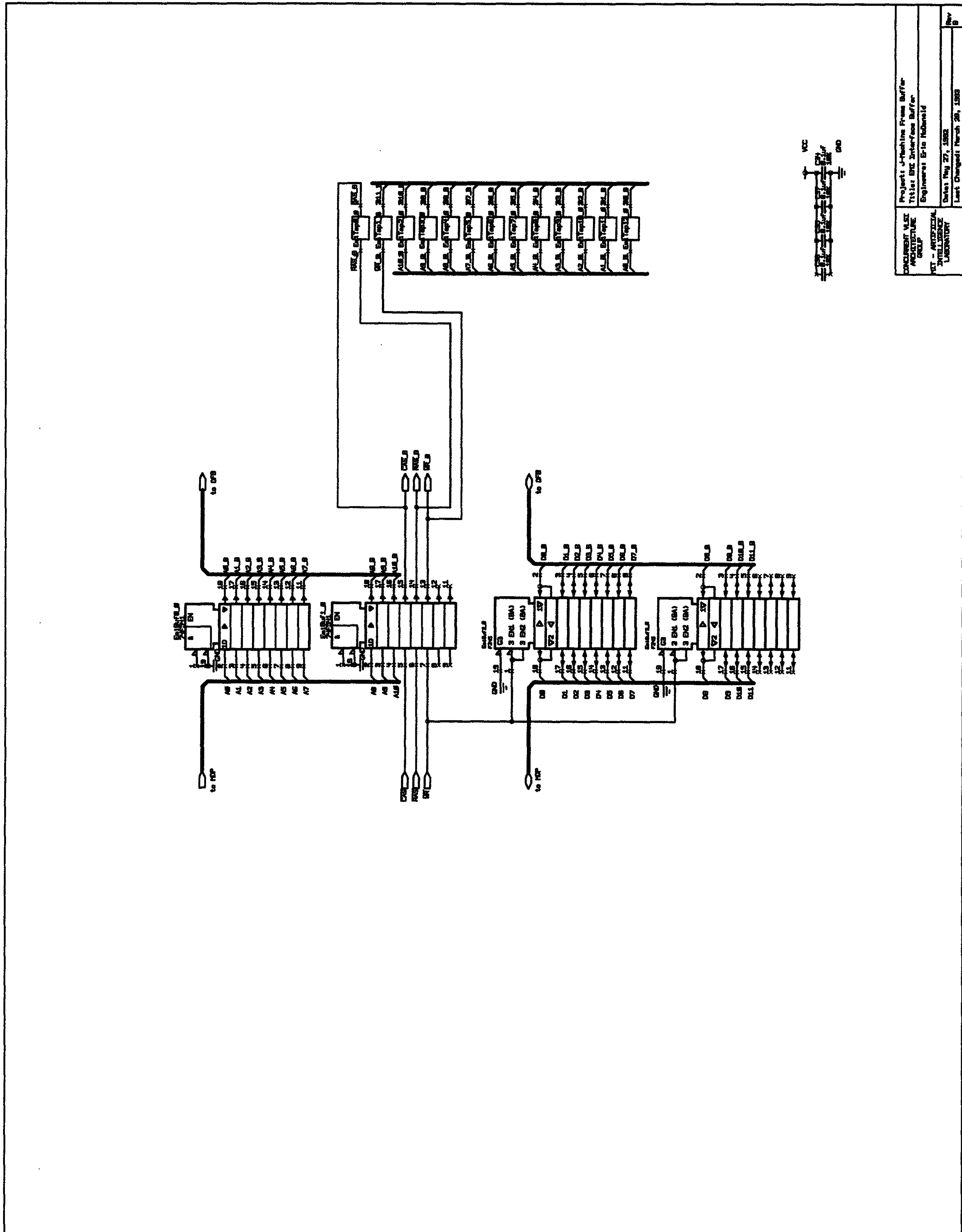
mdpsel.abl

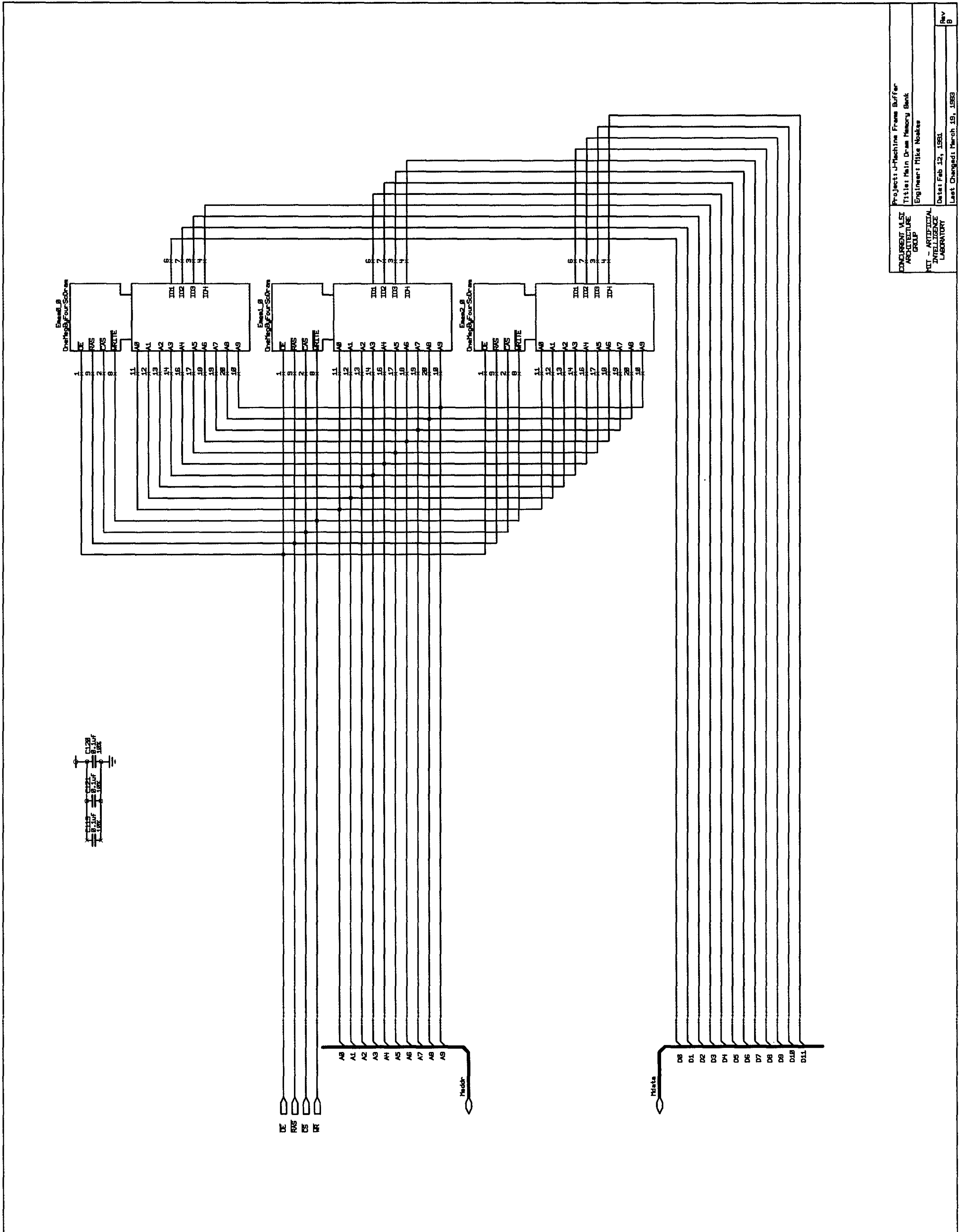
```
module mdpsel

title 'MDP Diag Interface - MDPSelectPal    Eric McDonald    January 15, 1994'

mdpsel device 'P22V10';

"*** Inputs

" These 5 signals are asserted LOW
BS, AS, DS, RW                                  Pin 2, 3, 4, 5;
MATCH                                           Pin 10;

D0, D1, D2, D3                                  Pin 6, 7, 8, 9;


"*** Outputs

" BSel is asserted LOW
BSel                                            Pin 22;
X0, X1                                          Pin 21, 20;
A0, A1                                          Pin 19, 18;
Xen                                             Pin 17;

"*** Bus definitions

ADDR                    = [D1, D0];
ADDRB                   = [A1, A0];
DATA                    = [D1, D0];
XR                      = [X1, X0];

"*** Alias equates

XE_REG                  = [0, 0];
YE_REG                  = [0, 1];
A_TO_D_REG              = [1, 0];
H, L, X                 = 1, 0, .X.;

"*** Output equations

equations

!BSel = (!BS & !MATCH) # (BS & !BSel);

"Store the register address selection.
ADDRB = (!BSel & !AS & ADDR) # ( !(!BSel & !AS) & ADDRB);

"Latch the X register with the appropriate value.
"
"Note that this value is not encoded base 2...instead, each 1/0 bit
"corresponds to one MDP node.
Xen = !BSel & !DS & RW & (ADDRB == XE_REG);
XR = (DATA & Xen) # (XR & !Xen);

end mdpsel
```

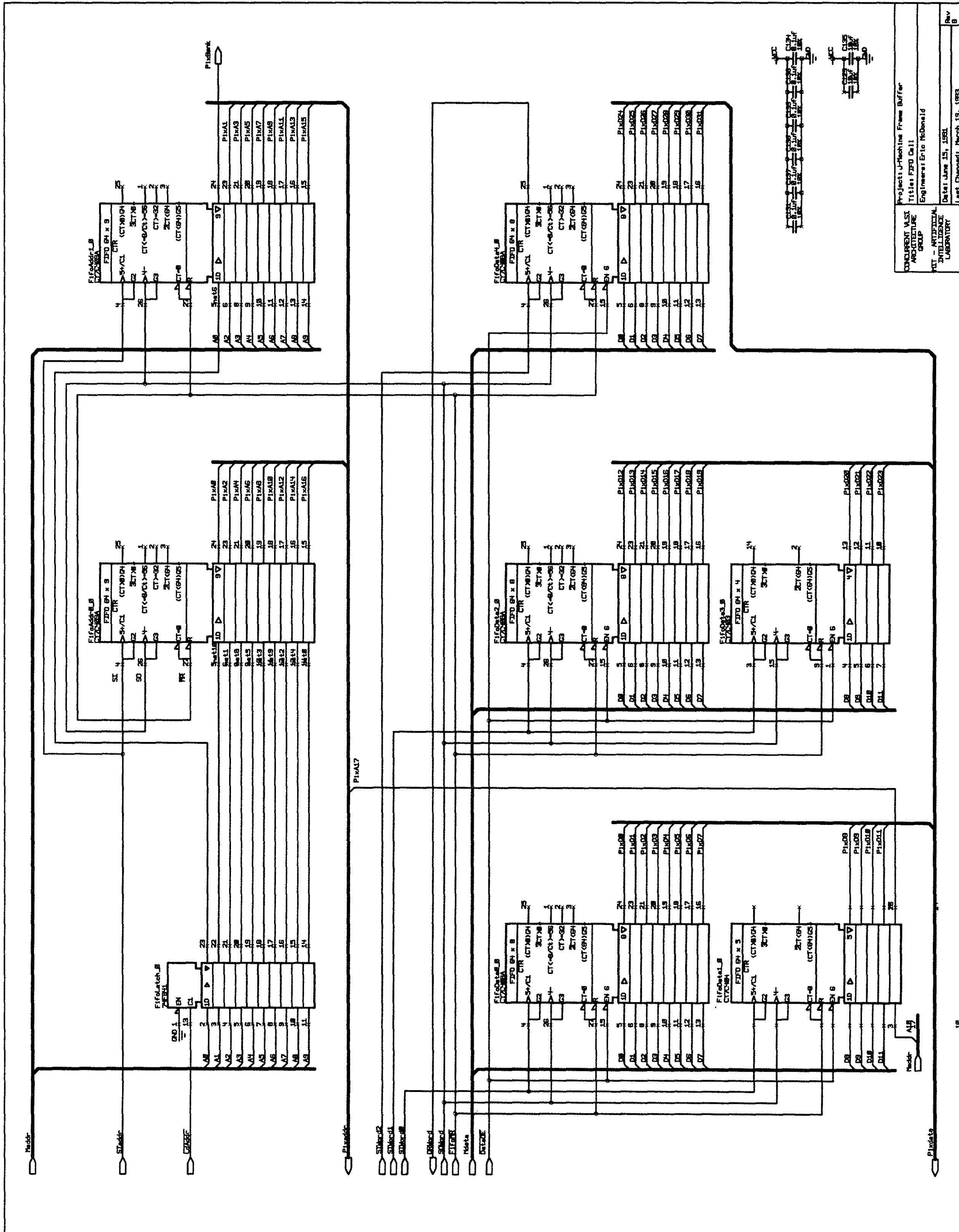
mdpfan.abl

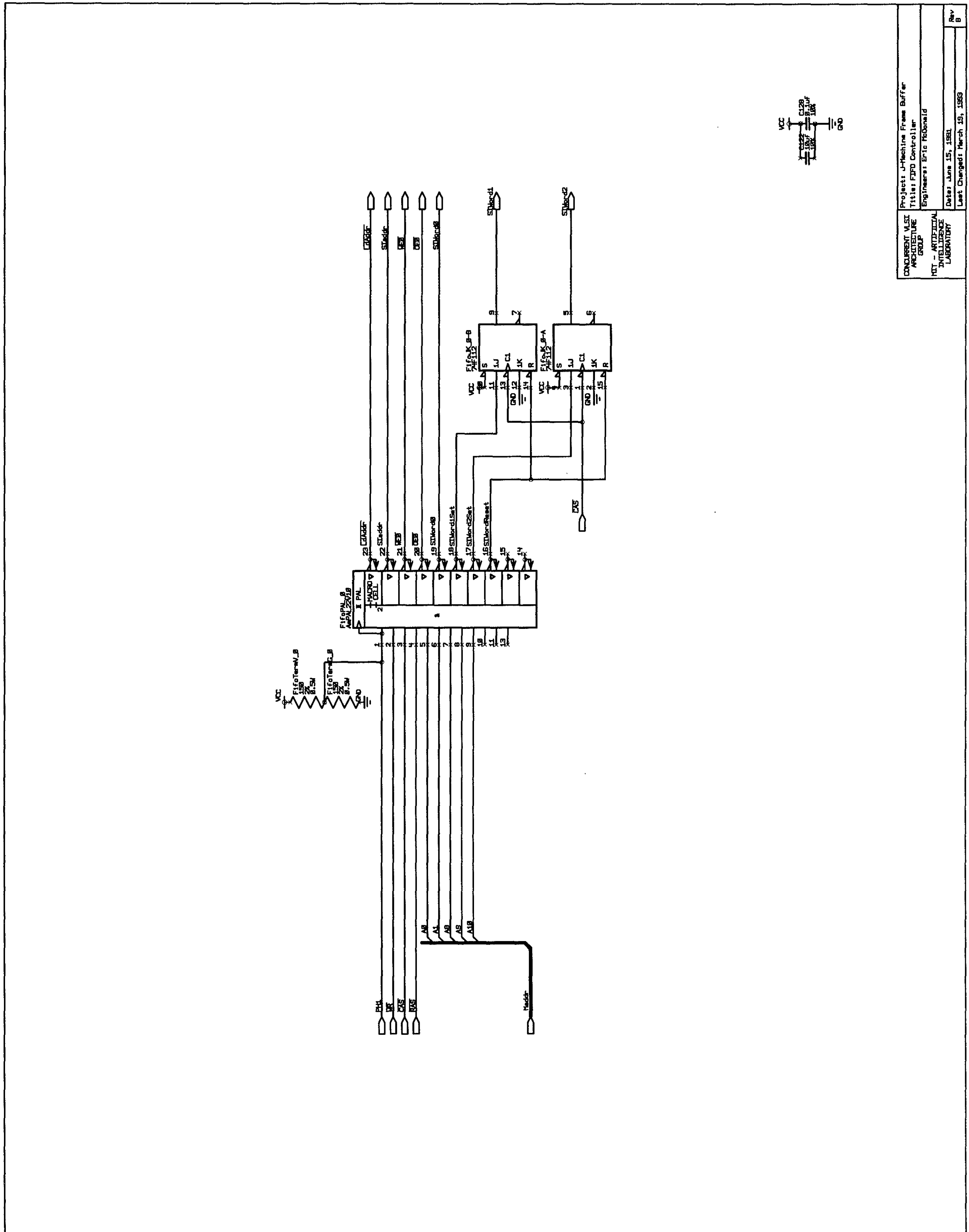
```
module mdpfan

title 'MDP Diag Interface - MDPFanoutPal    Eric McDonald    January 15, 1994'

mdpfan device 'P16V8';

"*** Inputs

Mclk                                    Pin 1;
Mmode0, Mmode1                          Pin 2, 3;
Sin                                     Pin 4;
ModeIn0_0, ModeIn1_0                    Pin 5, 6;
ModeIn0_1, ModeIn1_1                    Pin 7, 8;

"*** Outputs

Sin_0, Sin_1                            Pin 19, 18;
Ph1_0, Ph1_1                            Pin 17, 16 istype 'reg';
Mode0_0, Mode1_0                        Pin 15, 14 istype 'reg';
Mode0_1, Mode1_1                        Pin 13, 12 istype 'reg';

"*** Buses

Mmode   = [Mmode1 , Mmode0];

"*** Aliases

NOP     = [  0 ,  0  ];
EX      = [  0 ,  1  ];
SH      = [  1 ,  0  ];
RESET   = [  1 ,  1  ];
H, L, X = 1, 0, .X.;

"*** Equations

equations

[Ph1_0, Ph1_1].clk = Mclk;

"Just buffer and distribute Sin:

Sin_0 = Sin;
Sin_1 = Sin;

"Just buffer and distribute Mode:
Mode0_0 := ModeIn0_0;
Mode1_0 := ModeIn1_0;
Mode0_1 := ModeIn0_1;
Mode1_1 := ModeIn1_1;

"Calculate and distribute ph1

Ph1_0   := (Ph1_0  == L) # (Mmode == RESET);
Ph1_1   := (Ph1_1  == L) # (Mmode == RESET);

end mdpfan
```


CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY
Project: J-Machine Frame Buffer
Title: MdpCell
Engineer: Mike Noakes
Date: Feb 12, 1991
Last Changed: May 24, 1991
Rev A
MDP8
MICROCOMPUTER
36 BIT
MDP




Project: J-Machine Frame Buffer
Title: Main Dram Memory Bank
Engineer: Mike Noakes
Date: Feb 12, 1991
Last Changed: March 19, 1993
Rev B
CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY


CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY
Project: J-Machine Frame Buffer
Title: FIFO Cell
Engineer: Eric McDonald
Date: June 15, 1991
Last Changed: March 19, 1993

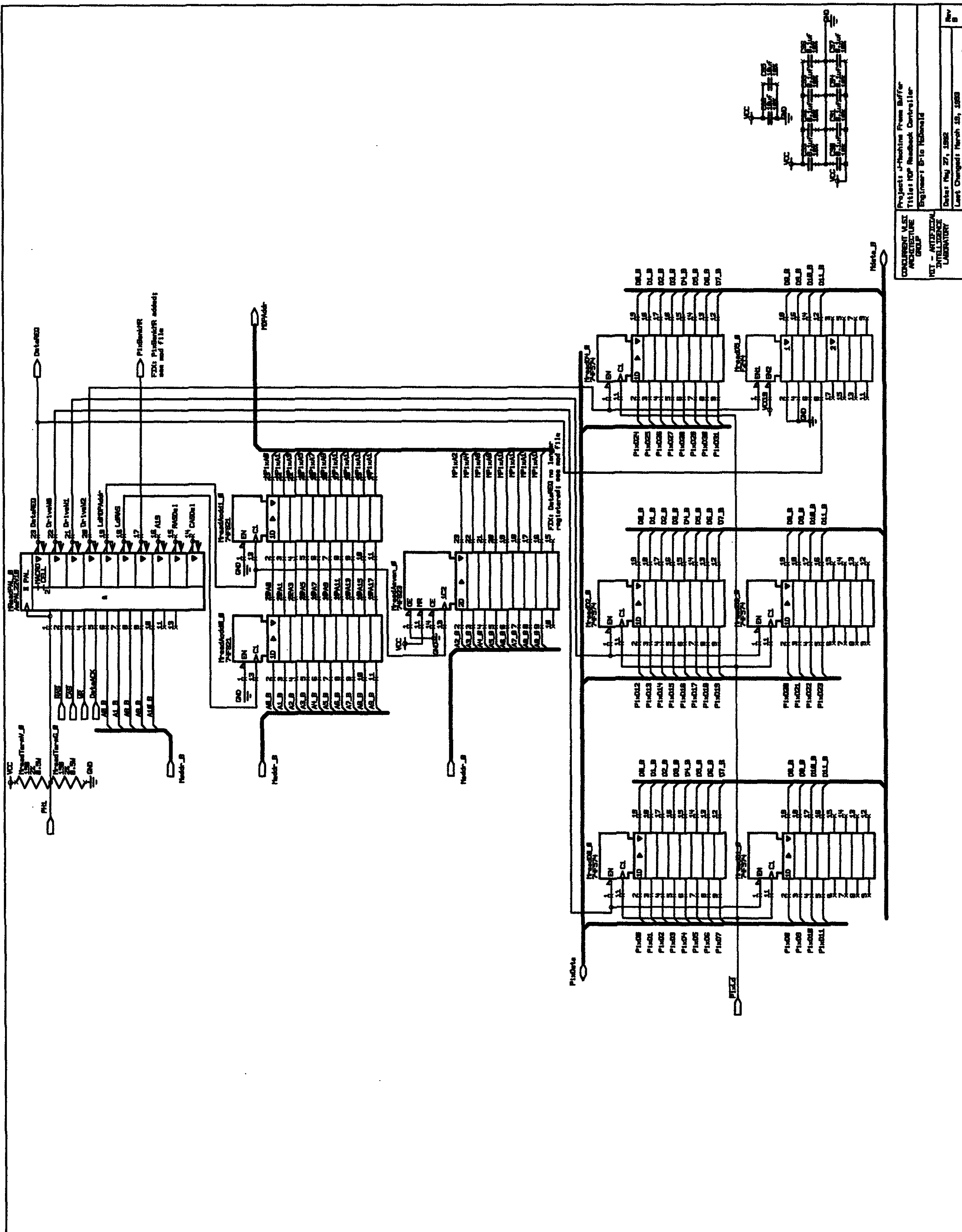

CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY
Project: J-Machine Frame Buffer
Title: FIFO Controller
Engineers: Eric McDonald
Date: June 15, 1991
Last Changed: March 19, 1993
Rev B

fifopal.abl

```
module FifoPal
title 'FIFO Controller Pal

  Eric McDonald     February 15, 1994
  Last revised:     February 15, 1994'

"  DESCRIPTION:
"  ------------
"  This PAL watches all emem accesses made by the MDP.
"  If the access is to an address < $80000, the PAL does nothing but
"    pass along the /WE and /OE signals to the DRAMs.
"  However, accesses to >= $80000 disable the DRAM /WE and /OE lines.
"  For WRITEs, the specified data and some bits of the address
"    are shifted into the appropriate FIFOs.
"  For READs, nothing is done besides disabling /WE and /OE (to allow
"    the MREAD PAL to respond to the read request).

FifoPal  device 'P22V10';

"*** Inputs

PH1                             Pin 1;
WR,CAS,RAS                      Pin 2, 3, 4;  "EMI control signals
CSB0, CSB1                      Pin 5, 6;     "Indicate which word is on bus.
EA10                            Pin 9;

"*** Outputs

"FIFO control lines...
LdAddr                          Pin 23;   "Latch 10 bits of address
SIaddr                          Pin 22;   "Shift in all 20 bits of address
                                          "(current 10 + latched 10)
SIword0                         Pin 19;   "Shift in first 12 data bits
SIword1set                      Pin 18;   "...next 12 data bits     (to JK FF)
SIword2set                      Pin 17;   "...and last 12 data bits (to JK FF)
SIwordReset                     Pin 16;   "Clear SIword{1,2} from JK FF

"EMI DRAM control lines...
WE                              Pin 21;   "/WE line for DRAMs
OE                              Pin 20;   "/OE line for DRAMs

"For internal use
RASDelayed                      Pin 15 istype 'reg';
A19                             Pin 14;

"*** Aliases

DRAM_ACCESS = !A19;          "DRAM takes the lower half of address space
FIFO_ACCESS =  A19;          "..and the Video RAM takes the upper half

H, L, C, X = 1, 0, .C., .X.;
ZERO = [0, 0];
ONE  = [0, 1];
TWO  = [1, 0];
CSB = [CSB1, CSB0];

Word0 = (CSB == ZERO);
```

fifopal.abl

```
Word1 = (CSB == ONE);
Word2 = (CSB == TWO);

RASGoingLow = !RAS & RASDelayed;

"*** Equations

equations

RASDelayed.clk = PH1;
RASDelayed := RAS;

"A19 (provided on EA10 when /RAS drops) will be high for address >= $80000
A19 = (!PH1 & RASGoingLow & EA10) # (!RAS & A19);

"***
"DRAM control lines...
"***
" DRAM /OE and /WE lines are disabled during A19 accesses
OE = A19 # !WR;
WE = A19 # WR;

"***
"FIFO control lines...
"***
"The high-to-low transition of /LdAddr latches the row of the external
"  memory address.  Keep it low for the entire emem access cycle.
!LdAddr = (EA10 & !WR & !RAS & CAS) # (!RAS & !LdAddr);

"The low-to-high transition of SIaddr shifts the latched addr row and the
"  current addr column into the FIFOs.  The subsequent high-to-low return
"  of SIaddr propagates the address through the FIFO.
SIaddr = (A19 & !WR & !CAS & Word0) # (SIaddr & !RAS);
" WARNING: Watch out for a race condition here.  The 'output ready' from
"   the FIFO's comes only from a DATA FIFO, so it is assumed that the
"   ADDR FIFO is in sync with it.  However, SIaddr goes low at 'about'
"   the same time as the SIword's do, so the ADDR FIFO may not have
"   propagated the address all the way through yet.
"
"SIword0 goes right to its FIFO.  SIword{1,2} go to the J inputs of a
" JK Flip Flop whic is clocked off the falling edge of /CAS.
SIword0 = (A19 & !WR & !CAS & Word0) # (SIword0 & !RAS);
SIword1set = (SIword0 & Word1);
SIword2set = (SIword0 & Word2);
!SIwordReset = RAS;

end FifoPal
```

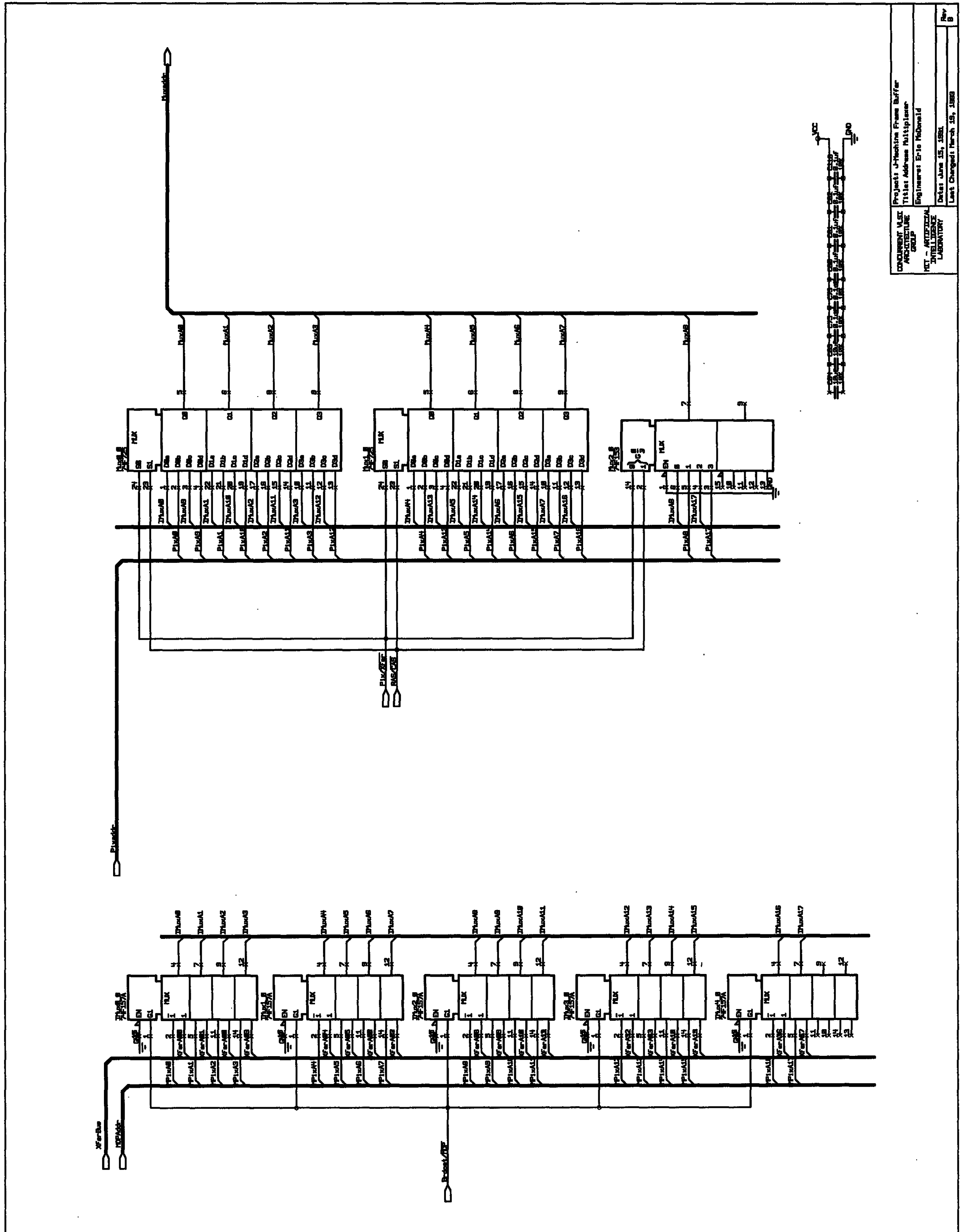


mread.abl

```
module MRead

title 'MDP Readback PAL

  Eric McDonald    July 14, 1992
  Last revised:    April 28, 1994'

" DESCRIPTION:
" ------------
" When the MDP makes a READ access to address >= $80000 (A19=1), we
" initiate a request to the VRAM PAL to supply us with the pixel
" value for the given address in VRAM.

" The request is made with DataREQ and is acknowledged with DataACK once
" the VRAM module has clocked the data into our registers with /PixLd.

" The two valid address ranges for this module are:
"
" $80000 - $bffff       Request pixel data for selected bank from
"                       (address & $3ffff) (strip 2 MSBs)
" $c0000                Poll DataREQ status flag and view data returned
"                       from last read request.  Also selects bank for
"                       next read to Bank A.
" $d0000                Same as above, but selects bank for
"                       next read to Bank B.
" A summary of the bit usage is:
"   A19 A18 A17 A16
"    1   0   X   X   Request data
"    1   1   X   0   Poll and select bank A (PixBankMR = 0)
"    1   1   X   1   Poll and select bank B (PixBankMR = 1)
"
" After requesting the data by addressing the $80000 range,
" the MDP should keep reading from $c0000 or $e0000.  It should
" ignore the data it sees until the DataREQ line goes low again, at
" which point the data is valid.

MRead device 'P22V10';

"*** Inputs

PH1                                     Pin 1;
RAS, CAS, WR                            Pin 2, 3, 4;
VRAMDataACK                             Pin 5;
CSB0, CSB1                              Pin 6, 7;
EA9, EA10                               Pin 9, 10;

"*** Outputs

VRAMDataREQ                             Pin 23 istype 'reg';
DriveWord0, DriveWord1, DriveWord2      Pin 22, 21, 20;
LoadRAS, LoadMDPAddr                    Pin 18, 19;
PixBankMR                               Pin 17 istype 'reg';

"*** Outputs used internally
A19                                     Pin 16;
RASDelayed, CASDelayed                  Pin 15, 14 istype 'reg';
```

mread.abl

```
"*** Aliases

H, L, C, X = 1, 0, .C., .X.;
ZERO = [0, 0];
ONE  = [0, 1];
TWO  = [1, 0];
CSB = [CSB1, CSB0];
RASGoingLow = !RAS & RASDelayed;
RASGoingHigh = RAS & !RASDelayed;
CASGoingLow = !CAS & CASDelayed;
A19Read = !RAS & A19 & WR;

"*** Equations

equations

[VRAMDataREQ,RASDelayed,CASDelayed].clk = PH1;

RASDelayed := RAS;
CASDelayed := CAS;

" A19 appears on EA10 during the initial memory cycle...
A19 = (!PH1 & RASGoingLow & EA10) # (!RAS & A19);

" LoadRAS is asserted during *every* read with A19 == 1
LoadRAS = (WR & EA10 & RASGoingLow)
        # (LoadRAS & !RAS);

" LoadMDPAddr is asserted only during reads with A19 == 1 and A18 == 0
" (A18 appears on EA10 when CAS drops and CSB==0)
" (A16 appears on EA9 when CAS drops and CSB==0)
" VRAMDataREQ depends on this signal.
LoadMDPAddr = (A19Read & !EA10 & CASGoingLow & (CSB == ZERO))
            # (LoadMDPAddr & !VRAMDataREQ);
PixBankMR :=  (A19Read & EA10 & CASGoingLow & (CSB == ZERO) & EA9)
           #  (!(A19Read & EA10 & CASGoingLow & (CSB == ZERO)) & PixBankMR);

" This signals to the VRAM PAL that we have a complete address and would
"   like to read data from the VRAM.
" It also clears the LoadMDPAddr signal.
VRAMDataREQ := (LoadMDPAddr & RAS)
             # (VRAMDataREQ & !VRAMDataACK);

!DriveWord0 = A19Read & !CAS & (CSB == ZERO);
!DriveWord1 = A19Read & !CAS & (CSB == ONE);
!DriveWord2 = A19Read & !CAS & (CSB == TWO);

end MRead
```



xfer.abl

```
module Xfer

title 'Xfer Pal for VRAM FSMs

  Eric McDonald    May 12, 1994
  Last revised:    May 12, 1994'

" DESCRIPTION:
" ------------

Xfer device 'P22V10';

"*** Inputs

FClk                    Pin 1;
Transfer                Pin 2;     " Async. transfer signal from VC
Ack0,Ack1               Pin 3,4;   " VRAM FSMs saw the transfer signal
FFTransfer              Pin 5;     " Transfer signal from Flip Flop

"*** Outputs

Xfer0,Xfer1             Pin 23,22 istype 'reg,buffer'; " Sent to VRAM FSMs
TransferNot             Pin 21 istype 'invert';        " Complement of Transfer
FFReset                 Pin 20 istype 'reg,invert';    " Reset FF


"*** Bits used internally
XferSync                Pin 19 istype 'reg,buffer'; " Sync up the Transfer sig
XferSyncDel             Pin 18 istype 'reg,buffer';
OKToReset               Pin 17 istype 'reg,buffer';

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

"*** Equations

equations

[XferSync,XferSyncDel,OKToReset,Xfer0,Xfer1].clk = FClk;

TransferNot = !Transfer;        " Async sets the FF

XferSync := FFTransfer;         " Sync the FF output to our clock
XferSyncDel := XferSync;

Xfer0 := (XferSync & !XferSyncDel)
       # (Xfer0 & !Ack0);

Xfer1 := (XferSync & !XferSyncDel)
       # (Xfer1 & !Ack1);

OKToReset := XferSync           " OK to reset after we've seen the signal
           # (OKToReset & XferSync);

!FFReset := OKToReset & TransferNot.pin; " Must be sure not to assert
                                         " /FFReset and /TransferNot simult.
```

vc0.abl

```
module VC0

title 'VRAM Control PAL #0

  Eric McDonald    July 14, 1992
  Last revised:    April 28, 1994'

" DESCRIPTION:
" ------------
" Most of the time this FSM is simply providing refresh strobes to the VRAMs.
" However, it also services three kinds of requests:
"   Broadcast - VC has asked us to transfer a row in VRAM to the serial ports
"   FifoOR    - FIFO has data to be written into VRAM
"   DataREQ   - The MDP (via the MREAD module) would like us to read a
"               piece of data out of VRAM

VC0 device 'P22V10';

"*** Inputs

FClk           Pin 1;
FifoOR         Pin 2;     " FIFO has data ready
PixBank        Pin 4;     " The FIFO data should go into this bank
DataREQ        Pin 3;     " MDP wants to read data back from the VRAM
Transfer       Pin 5;     " VC wants us to transfer a row of data
PixBankMR_0    Pin 6;     " The MDP0 readback data should come from this bank
PixBankMR_1    Pin 7;     " The MDP1 readback data should come from this bank

"*** Outputs

DataACK        Pin 23 istype 'reg,buffer';  " ACK back to MDPRead PAL
PixLoad        Pin 22 istype 'reg,buffer';  " Latch data into MDPRead buffers
FifoDisable    Pin 21 istype 'reg,buffer';  " When high, disables FIFO data
Broadcast      Pin 20 istype 'reg,buffer';  " Select Brdcst or MDPRead addr
Pixel          Pin 19 istype 'reg,buffer';  " Pixel or Brdcst/MDP address
Q0,Q1,Q2,Q3,Q4 Pin 14,15,16,17,18 istype 'reg,buffer';  " State bits

"*** State declarations
@include 'vc.sta'

"*** Equations

equations

[current_state,DataACK,PixLoad,FifoDisable,Broadcast,Pixel].clk = FClk;

state_diagram current_state;

  State Init:
        goto Refresh0;

  State Refresh0:
"       !CASa := 1;  !CASb := 1;
        goto Refresh1;
  State Refresh1:
"       !CASa := 1;  !CASb := 1;
"       !RASa := 1;  !RASb := 1;
```

vc0.abl

```
        goto Refresh2;
  State Refresh2:
"       !CASa := 1;  !CASb := 1;
"       !RASa := 1;  !RASb := 1;
        goto Refresh3;
  State Refresh3:
"       !CASa := 1;  !CASb := 1;
"       !RASa := 1;  !RASb := 1;
        goto Refresh4;
  State Refresh4:
"       !CASa := 1;  !CASb := 1;
"       !RASa := 1;  !RASb := 1;
        goto Refresh5;
  State Refresh5:
        if (Transfer) then Xfer0;
          else Refresh6;
  State Refresh6:
        if (FifoOR) then Write0;
          else Refresh7;
  State Refresh7:
        if (DataREQ) then Read0;
          else Refresh0;

  State Xfer0:
        Broadcast := 1;
"       !OEa := 1; !OEb := 1;           First bring /DT low
"       MuxRAS := 1;
        goto Xfer1;
  State Xfer1:
        Broadcast := 1;
"       !OEa := 1; !OEb := 1;
"       !RASa := 1; !RASb := 1;         Then bring /RAS low
"       MuxRAS := 1;                      with valid row
        goto Xfer2;
  State Xfer2:
        Broadcast := 1;
"       !OEa := 1; !OEb := 1;
"       !RASa := 1; !RASb := 1;         Relase MuxRAS and let settle
        goto Xfer3;
  State Xfer3:
        Broadcast := 1;
"       !OEa := 1; !OEb := 1;
"       !RASa := 1; !RASb := 1;
"       !CASa := 1; !CASb := 1;         And finally bring /CAS low with SAM
        goto Xfer4;
  State Xfer4:
        Broadcast := 1;
"       !OEa := 1; !OEb := 1;
"       !RASa := 1; !RASb := 1;
"       !CASa := 1; !CASb := 1;
        goto Init;

  State Write0:
        Pixel := 1;
"       MuxRAS := 1;                         Drive row address and let settle
        goto Write1;
  State Write1:
```

vc0.abl

```
        Pixel := 1;
"       !RASa := !PixBank; !RASb := PixBank;  Bring /RAS low with valid row
"       MuxRAS := 1;
        goto Write2;
  State Write2:
        Pixel := 1;
"       !RASa := !PixBank; !RASb := PixBank;  Drive col address and let settle
        goto Write3;
  State Write3:
        Pixel := 1;
"       !RASa := !PixBank; !RASb := PixBank;
"       !CASa := !PixBank; !CASb := PixBank;  Bring /CAS low with valid column
        goto Write4;
  State Write4:
        Pixel := 1;
"       !RASa := !PixBank; !RASb := PixBank;
"       !CASa := !PixBank; !CASb := PixBank;
"       !WEa  := !PixBank; !WEb  := PixBank;
        goto Write5;
  State Write5:
"       !RASa := !PixBank; !RASb := PixBank;
"       !CASa := !PixBank; !CASb := PixBank;
"       FifoSO := 1;
        goto Init;

  State Read0:
"       MuxRAS := 1;                              Drive row addr & let settle
        goto Read1;
  State Read1:
"       MuxRAS := 1;
"       !RASa := !PixBankMR; !RASb := PixBankMR;  Bring /RAS low w/valid row
        FifoDisable := 1;
        goto Read2;
  State Read2:
"       !RASa := !PixBankMR; !RASb := PixBankMR;  Drive col addr & let settle
"       !OEa  := !PixBankMR; !OEb  := PixBankMR;
        FifoDisable := 1;
        goto Read3;
  State Read3:
"       !RASa := !PixBankMR; !RASb := PixBankMR;
"       !CASa := !PixBankMR; !CASb := PixBankMR;  Bring /CAS low w/valid col
"       !OEa  := !PixBankMR; !OEb  := PixBankMR;  Drive the VRAM data out
        FifoDisable := 1;
        goto Read4;
  State Read4:
"       !RASa := !PixBankMR; !RASb := PixBankMR;
"       !CASa := !PixBankMR; !CASb := PixBankMR;
"       !OEa  := !PixBankMR; !OEb  := PixBankMR;
        FifoDisable := 1;
        PixLoad := 1;
        goto Read5;
  State Read5:
        DataACK := 1;
        if (DataREQ) then Read5;
          else Init;

 " Eliminate trap states...
```

vc0.abl

```
  State Trap0: goto Init;
  State Trap1: goto Init;
  State Trap2: goto Init;
  State XferTrap: goto Init;
  State ReadTrap: goto Init;
  State WriteTrap: goto Init;

end VC0
```

vc1_0.abl

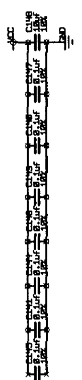
```
module VC1_0

@const MDP = 0;

title 'VRAM Control PAL #1

  Eric McDonald    July 14, 1992
  Last revised:    April 28, 1994'

VC1_0 device 'P22V10';

"*** Inputs
FClk               Pin 1;
Q0,Q1,Q2,Q3,Q4     Pin 2,3,4,5,6; " State bits generated by VCtrl0
FifoOR             Pin 7;         " FIFO has data ready
PixBank            Pin 9;         " The FIFO data should go into this bank
DataREQ            Pin 8;         " MDP wants to read data back from VRAM
Transfer           Pin 10;        " VC wants us to transfer a row of data
PixBankMR_0        Pin 11;  " The MDP0 readback data should come from this bank
PixBankMR_1        Pin 13;  " The MDP1 readback data should come from this bank

"*** Outputs
MuxRAS           Pin 23 istype 'reg,buffer';  "Mux RAS or CAS select
FifoSO           Pin 22 istype 'reg,buffer';  "Shift FIFO one word
RASa,RASb        Pin 21,17 istype 'reg,invert';  "VRAM /RAS signals
CASa,CASb        Pin 20,16 istype 'reg,invert';  "VRAM /CAS signals
WEa,WEb          Pin 19,15 istype 'reg,invert';  "VRAM /WBWE signals
OEa,OEb          Pin 18,14 istype 'reg,invert';  "VRAM /DTOE signal

"*** Aliases
@if (MDP == 0) { PixBankMR = PixBankMR_0; }
@if (MDP != 0) { PixBankMR = PixBankMR_1; }

"*** State declarations
@include 'vc.sta'

"*** Equations

equations

[MuxRAS,FifoSO,RASa,RASb,CASa,CASb,WEa,WEb,OEa,OEb].clk = FClk;

FifoSO := In_State(Write5);

MuxRAS := In_State(Xfer0)
        # In_State(Xfer1)
        # In_State(Write0)
        # In_State(Write1)
        # In_State(Read0)
        # In_State(Read1);

!RASa := In_State(Refresh1)
      # In_State(Refresh2)
      # In_State(Refresh3)
      # In_State(Refresh4)
      # In_State(Xfer1)
      # In_State(Xfer2)
```

vc1_0.abl

```
      # In_State(Xfer3)
      # In_State(Xfer4)
      # (In_State(Write1) & !PixBank)
      # (In_State(Write2) & !PixBank)
      # (In_State(Write3) & !PixBank)
      # (In_State(Write4) & !PixBank)
      # (In_State(Write5) & !PixBank)
      # (In_State(Read1) & !PixBankMR)
      # (In_State(Read2) & !PixBankMR)
      # (In_State(Read3) & !PixBankMR)
      # (In_State(Read4) & !PixBankMR);

!RASb := In_State(Refresh1)
      # In_State(Refresh2)
      # In_State(Refresh3)
      # In_State(Refresh4)
      # In_State(Xfer1)
      # In_State(Xfer2)
      # In_State(Xfer3)
      # In_State(Xfer4)
      # (In_State(Write1) & PixBank)
      # (In_State(Write2) & PixBank)
      # (In_State(Write3) & PixBank)
      # (In_State(Write4) & PixBank)
      # (In_State(Write5) & PixBank)
      # (In_State(Read1) & PixBankMR)
      # (In_State(Read2) & PixBankMR)
      # (In_State(Read3) & PixBankMR)
      # (In_State(Read4) & PixBankMR);

!CASa := In_State(Refresh0)
      # In_State(Refresh1)
      # In_State(Refresh2)
      # In_State(Refresh3)
      # In_State(Refresh4)
      # In_State(Xfer3)
      # In_State(Xfer4)
      # (In_State(Write3) & !PixBank)
      # (In_State(Write4) & !PixBank)
      # (In_State(Write5) & !PixBank)
      # (In_State(Read3) & !PixBankMR)
      # (In_State(Read4) & !PixBankMR);

!CASb := In_State(Refresh0)
      # In_State(Refresh1)
      # In_State(Refresh2)
      # In_State(Refresh3)
      # In_State(Refresh4)
      # In_State(Xfer3)
      # In_State(Xfer4)
      # (In_State(Write3) & PixBank)
      # (In_State(Write4) & PixBank)
      # (In_State(Write5) & PixBank)
      # (In_State(Read3) & PixBankMR)
      # (In_State(Read4) & PixBankMR);

!WEa := (In_State(Write4) & !PixBank);
```

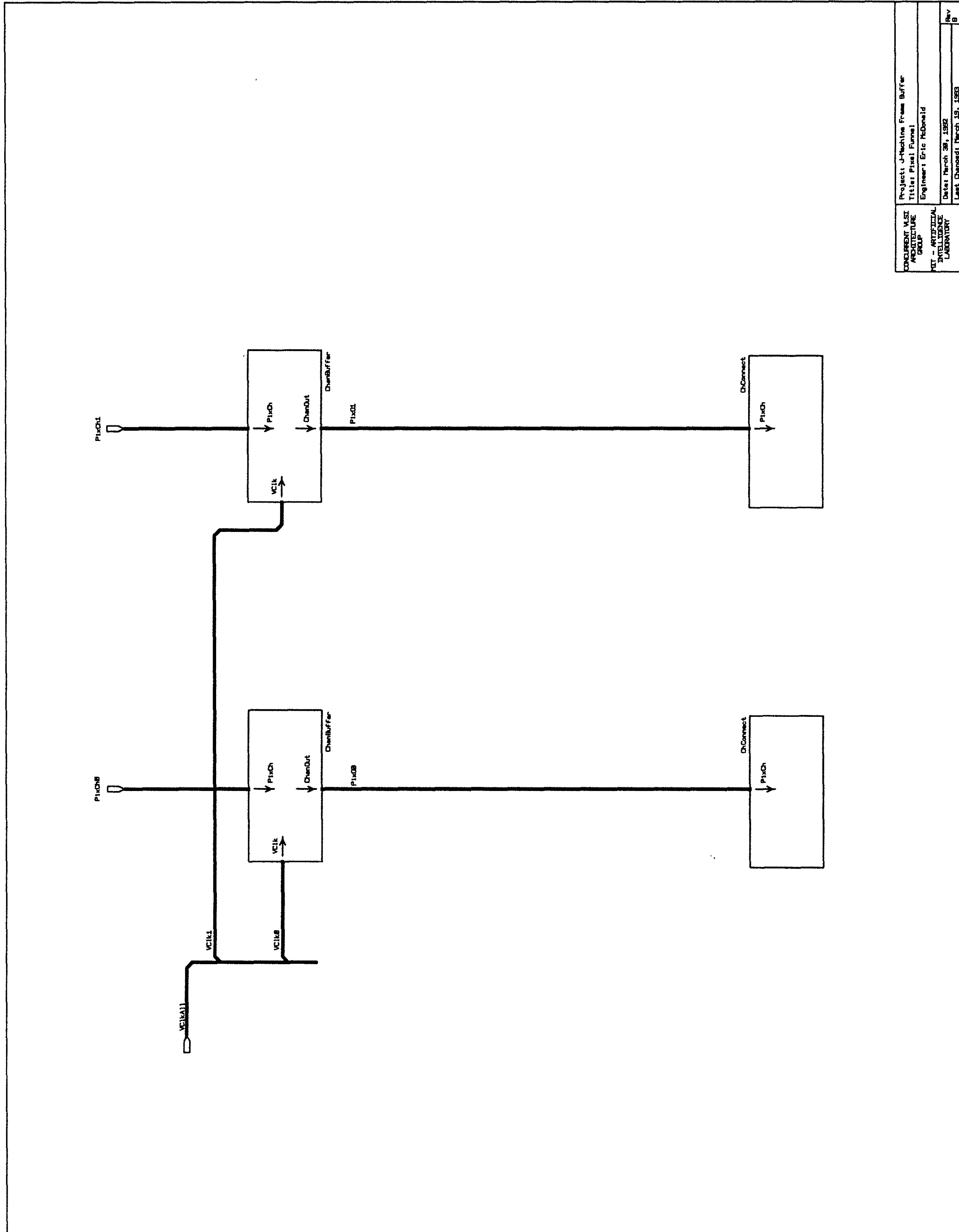
vc1_0.abl

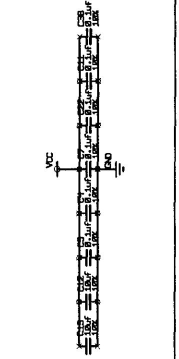
```
!WEb := (In_State(Write4) &  PixBank);

!OEa := In_State(Xfer0)
      # In_State(Xfer1)
      # In_State(Xfer2)
      # In_State(Xfer3)
      # In_State(Xfer4)
      # (In_State(Read2) & !PixBankMR)
      # (In_State(Read3) & !PixBankMR)
      # (In_State(Read4) & !PixBankMR);

!OEb := In_State(Xfer0)
      # In_State(Xfer1)
      # In_State(Xfer2)
      # In_State(Xfer3)
      # In_State(Xfer4)
      # (In_State(Read2) & PixBankMR)
      # (In_State(Read3) & PixBankMR)
      # (In_State(Read4) & PixBankMR);

end VC1_0
```

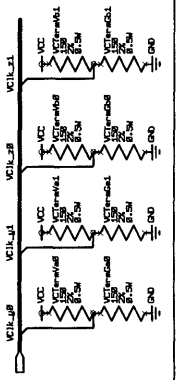


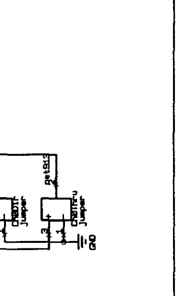
CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY

Project: J-Machine Frame Buffer
Title: VRAM Cell
Engineer: Eric McDonald
Date: June 15, 1991
Last Changed: March 19, 1993
Rev B

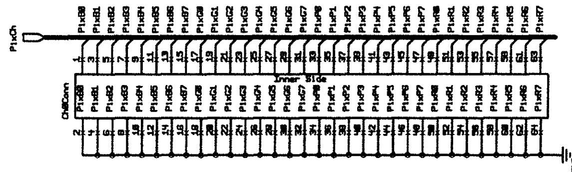

Project: J-Machine Frame Buffer
Title: Pixel Funnel
Engineer: Eric McDonald
Date: March 30, 1992
Last Changed: March 19, 1993
Rev
B
CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY
PixCh1
PixCh
ChanOut
ChanBuffer
PixO1
ChConnect
PixCh
VClk
PixCh0
PixCh
ChanOut
ChanBuffer
PixO0
ChConnect
PixCh
VClk
VClk1
VClk0
VClkAll





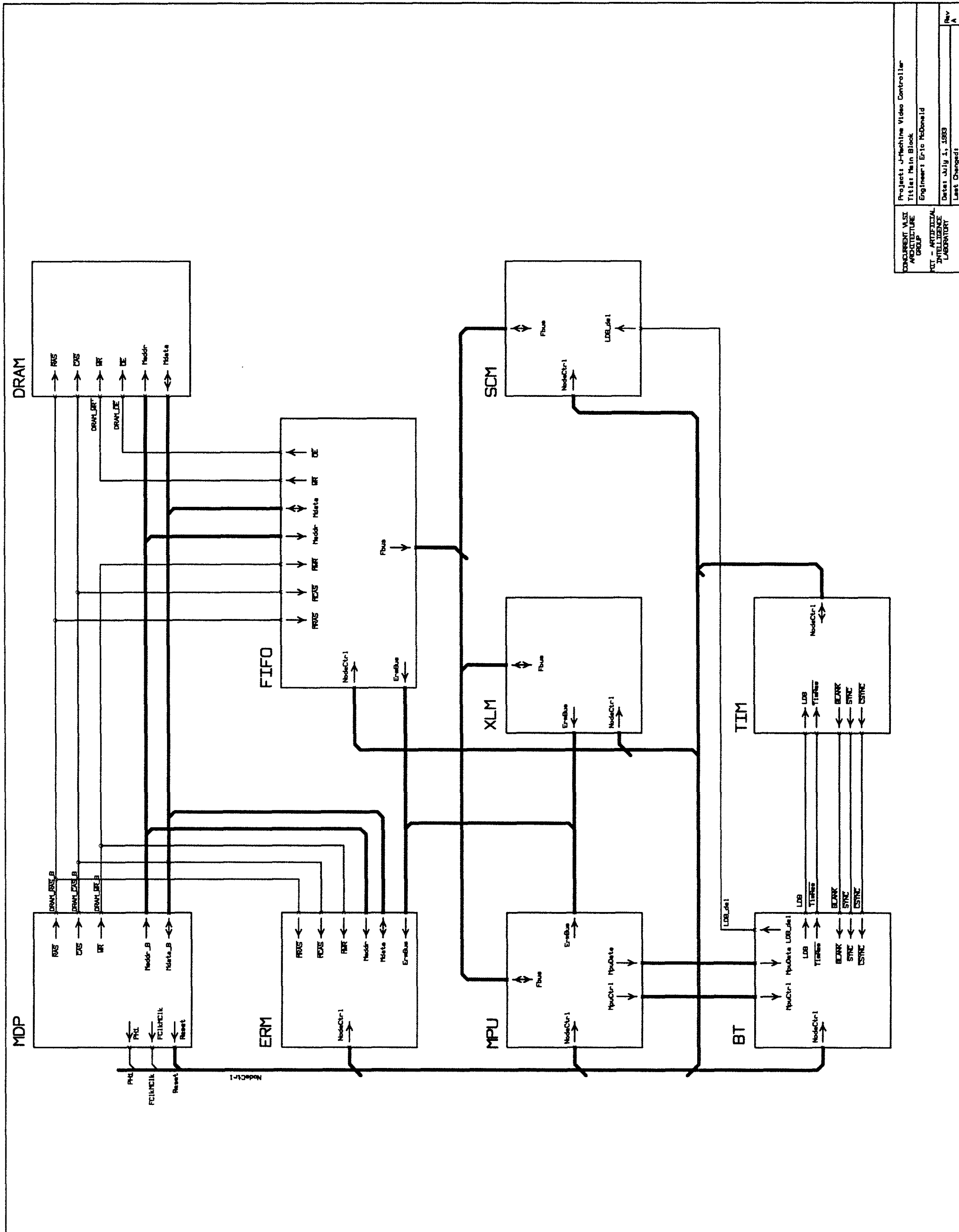


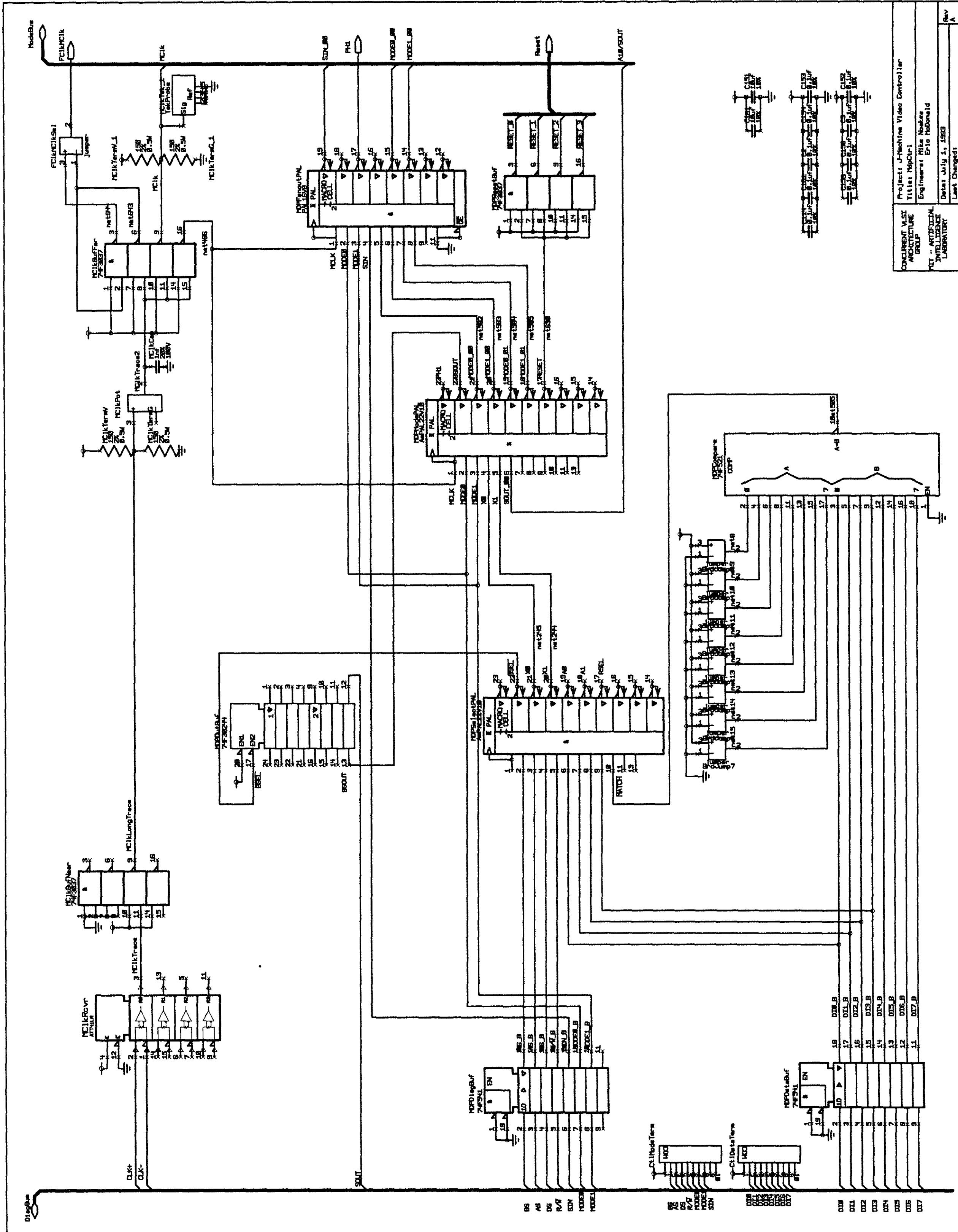
CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY
Project: J-Machine Frame Buffer
Title: Pixel Funnel (Channel Connector)
Engineer: Eric McDonald
Date: March 30, 1992
Last Changed: March 19, 1993
Rev B



Appendix B

VC Schematics and PAL Files


DRAM
FIFO
SCM
XLM
TIM
MDP
ERM
MPU
BT
CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY
Project: J-Machine Video Controller
Title: Main Block
Engineer: Eric McDonald
Date: July 1, 1993
Last Changed:
Rev
A

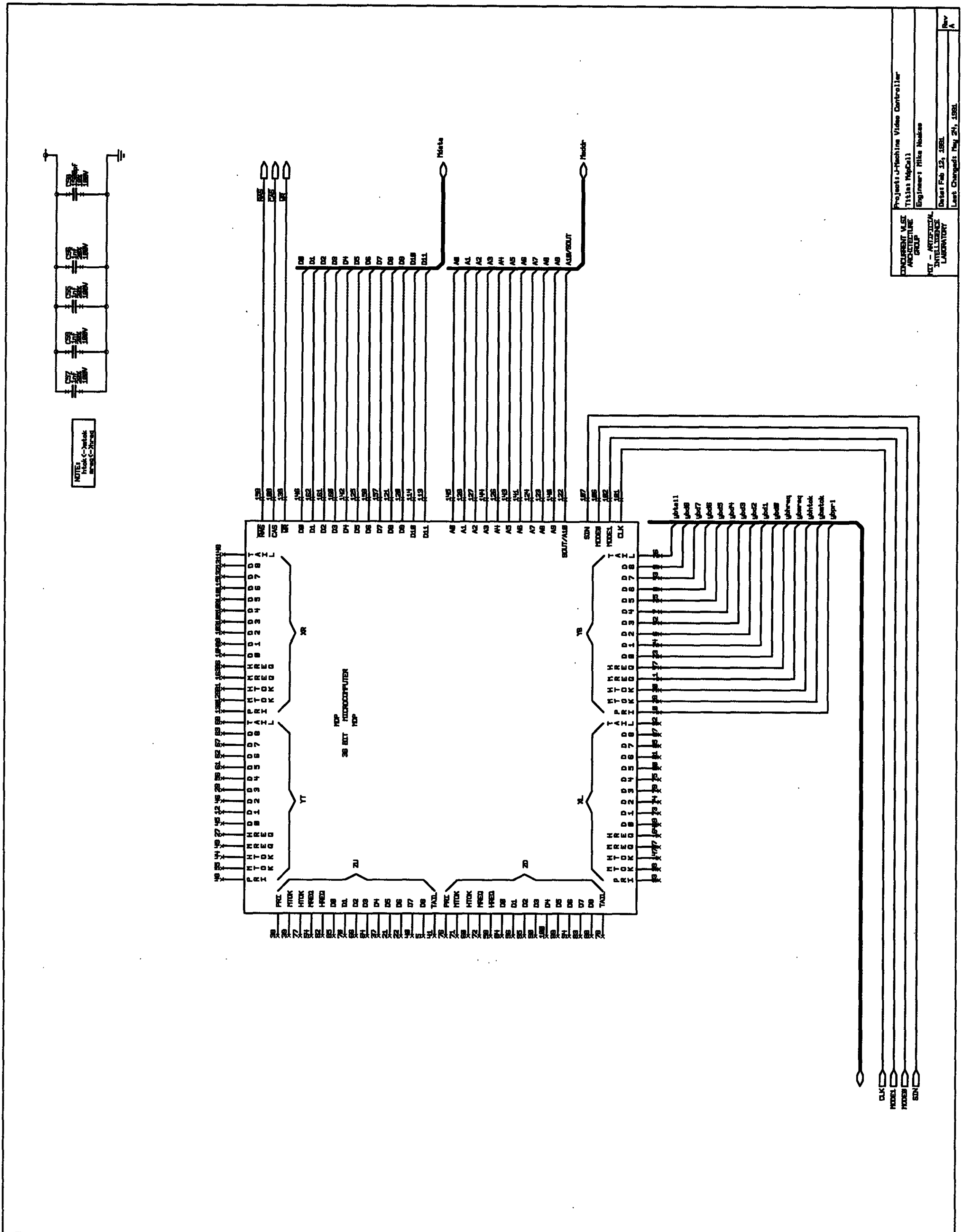


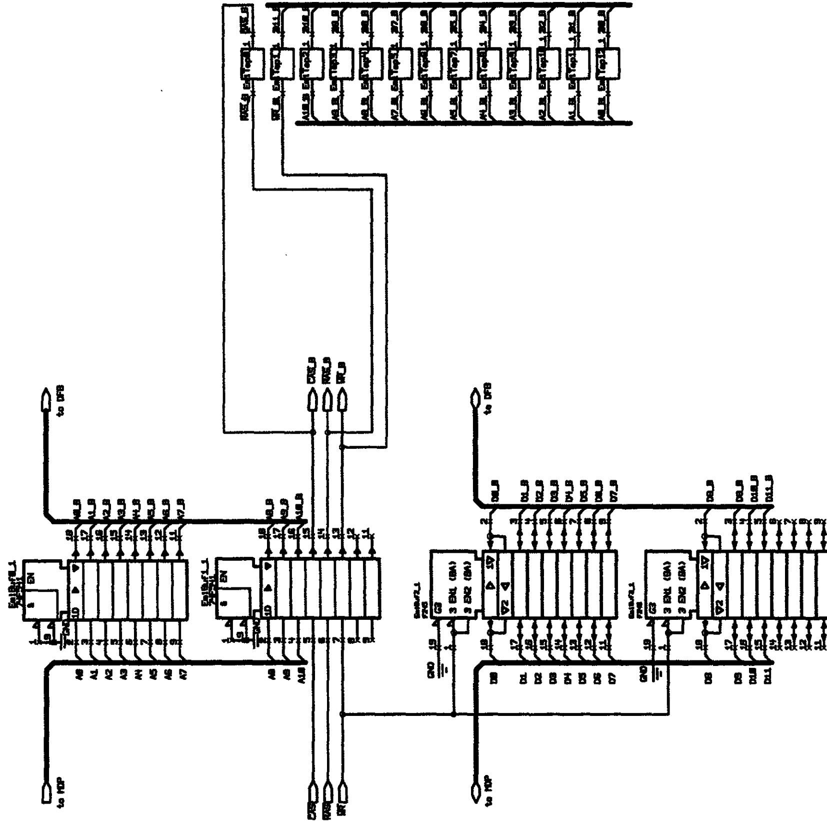
mdpsel.abl

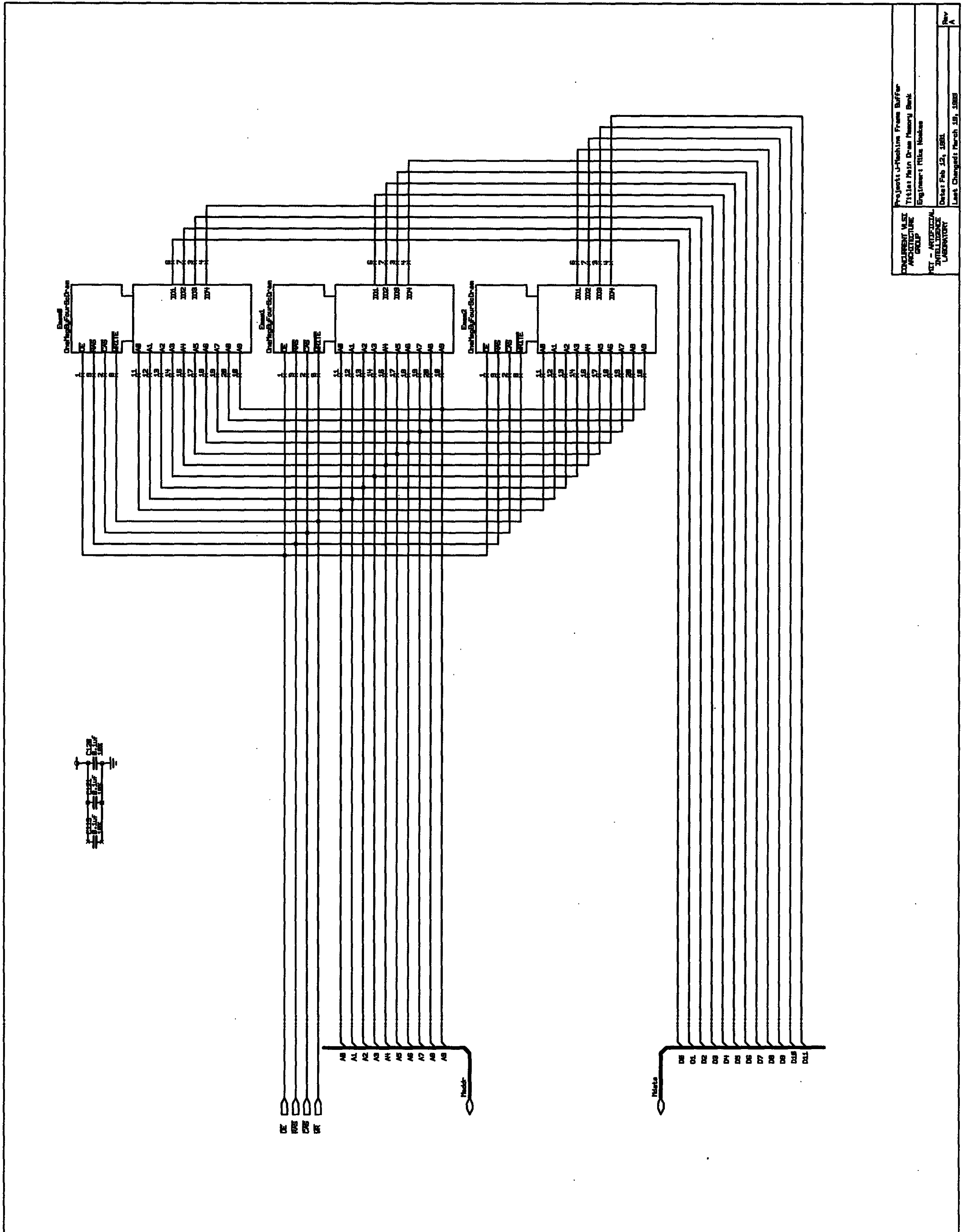
```
module mdpsel

title 'MDP Diag Interface - MDPSelectPal    Eric McDonald    January 15, 1994'

mdpsel device 'P22V10';

"*** Inputs

" These 5 signals are asserted LOW
BS, AS, DS, RW                                  Pin 2, 3, 4, 5;
MATCH                                           Pin 10;

D0, D1, D2, D3                                  Pin 6, 7, 8, 9;


"*** Outputs

" BSel is asserted LOW
BSel                                            Pin 22;
X0, X1                                          Pin 21, 20;
A0, A1                                          Pin 19, 18;
Xen                                             Pin 17;

"*** Bus definitions

ADDR                   = [D1, D0];
ADDRB                  = [A1, A0];
DATA                   = [D1, D0];
XR                     = [X1, X0];

"*** Alias equates

XE_REG                 = [0, 0];
YE_REG                 = [0, 1];
A_TO_D_REG             = [1, 0];
H, L, X                = 1, 0, .X.;

"*** Output equations

equations

!BSel = (!BS & !MATCH) # (BS & !BSel);

"Store the register address selection.
ADDRB = (!BSel & !AS & ADDR) # ( !(!BSel & !AS) & ADDRB);

"Latch the X register with the appropriate value.
"
"Note that this value is not encoded base 2...instead, each 1/0 bit
"corresponds to one MDP node.
Xen = !BSel & !DS & RW & (ADDRB == XE_REG);
XR = (DATA & Xen) # (XR & !Xen);

end mdpsel
```

mdpfan.abl

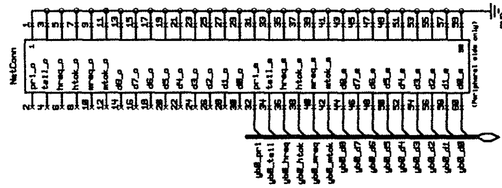
```
module mdpfan

title 'MDP Diag Interface - MDPFanoutPal    Eric McDonald    January 15, 1994'

mdpfan device 'P16V8';

"*** Inputs

Mclk                                    Pin 1;
Mmode0, Mmode1                          Pin 2, 3;
Sin                                     Pin 4;
ModeIn0_0, ModeIn1_0                    Pin 5, 6;
ModeIn0_1, ModeIn1_1                    Pin 7, 8;

"*** Outputs

Sin_0, Sin_1                            Pin 19, 18;
Ph1_0, Ph1_1                            Pin 17, 16 istype 'reg,invert';
Mode0_0, Mode1_0                        Pin 15, 14 istype 'reg';
Mode0_1, Mode1_1                        Pin 13, 12 istype 'reg';

"*** Buses

Mmode   = [Mmode1 , Mmode0];

"*** Aliases

NOP     = [  0 ,  0  ];
EX      = [  0 ,  1  ];
SH      = [  1 ,  0  ];
RESET   = [  1 ,  1  ];
H, L, X = 1, 0, .X.;

"*** Equations

equations

[Ph1_0, Ph1_1].clk = Mclk;

"Just buffer and distribute Sin:

Sin_0 = Sin;
Sin_1 = Sin;

"Just buffer and distribute Mode:
Mode0_0 := ModeIn0_0;
Mode1_0 := ModeIn1_0;
Mode0_1 := ModeIn0_1;
Mode1_1 := ModeIn1_1;

"Calculate and distribute ph1

Ph1_0.d   := (Ph1_0.q  == L) # (Mmode == RESET);
Ph1_1.d   := (Ph1_1.q  == L) # (Mmode == RESET);

end mdpfan
```

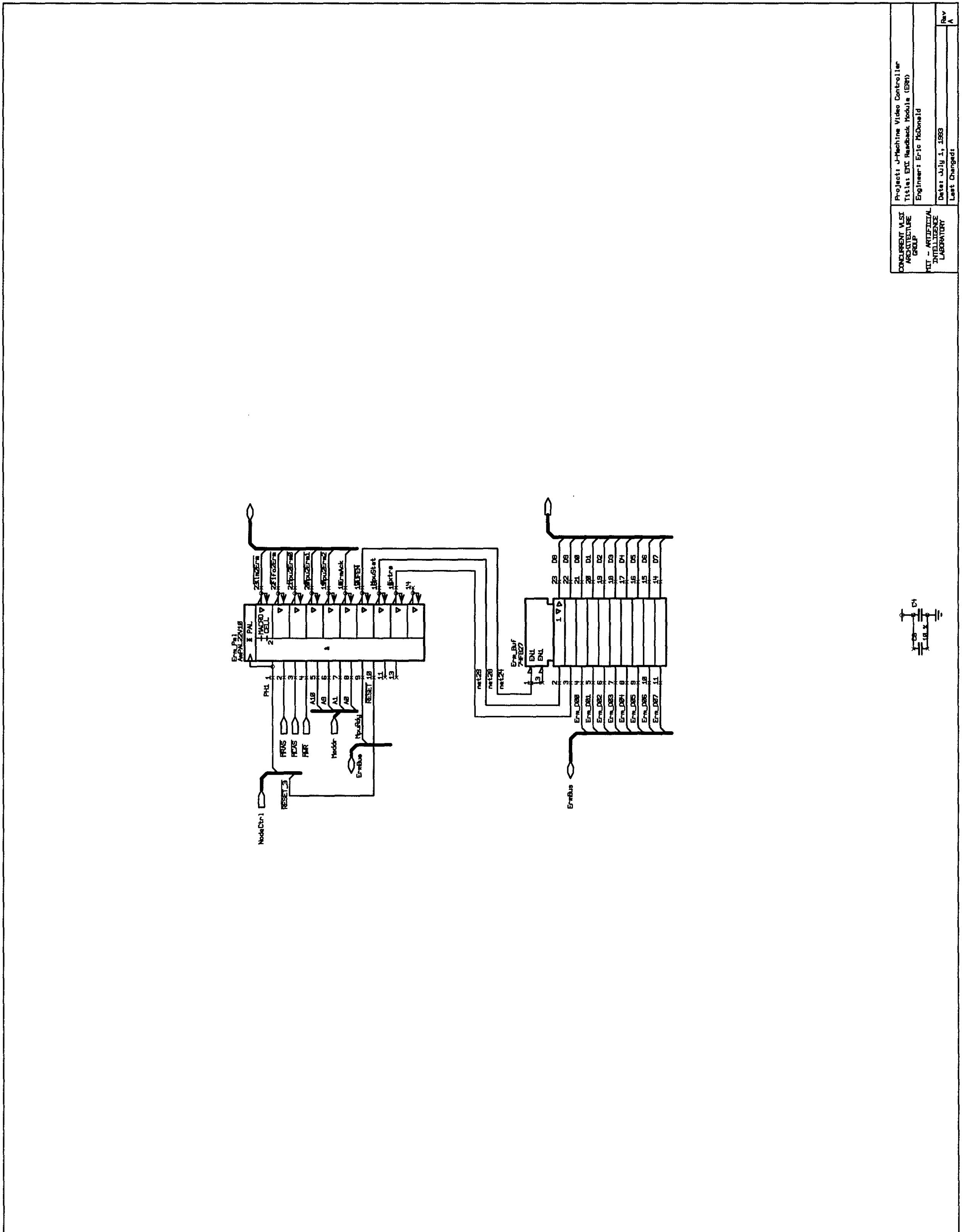






CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY
Project: J-Machine Frame Buffer
Title: y-Channel Network Connection
Engineer: Eric McDonald
Date: March 30, 1992
Last Changed: March 30, 1992
Rev A



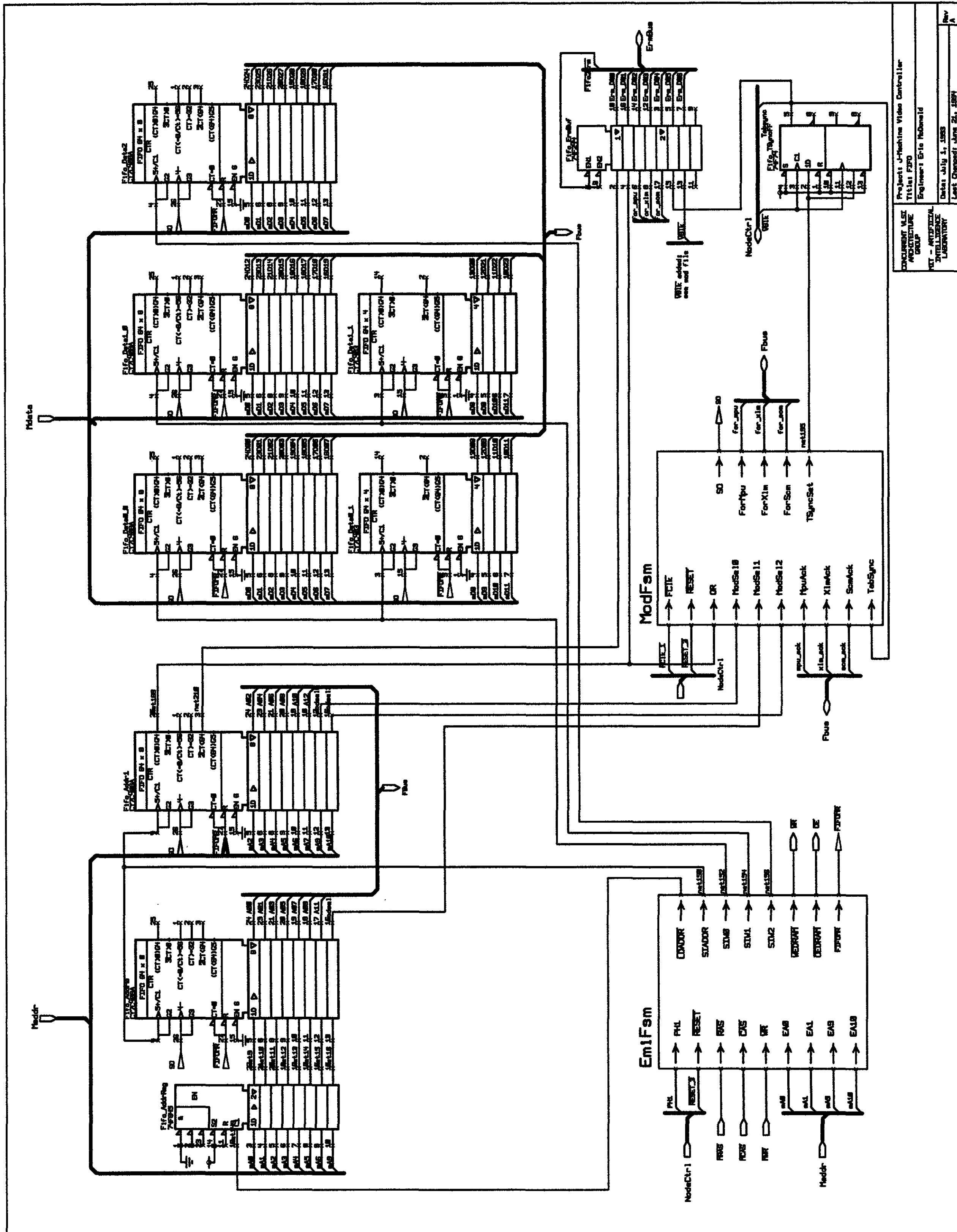

CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY
Project: J-Machine Video Controller
Title: EMI Readback Module (ERM)
Engineer: Eric McDonald
Date: July 1, 1993
Last Changed:
Rev A

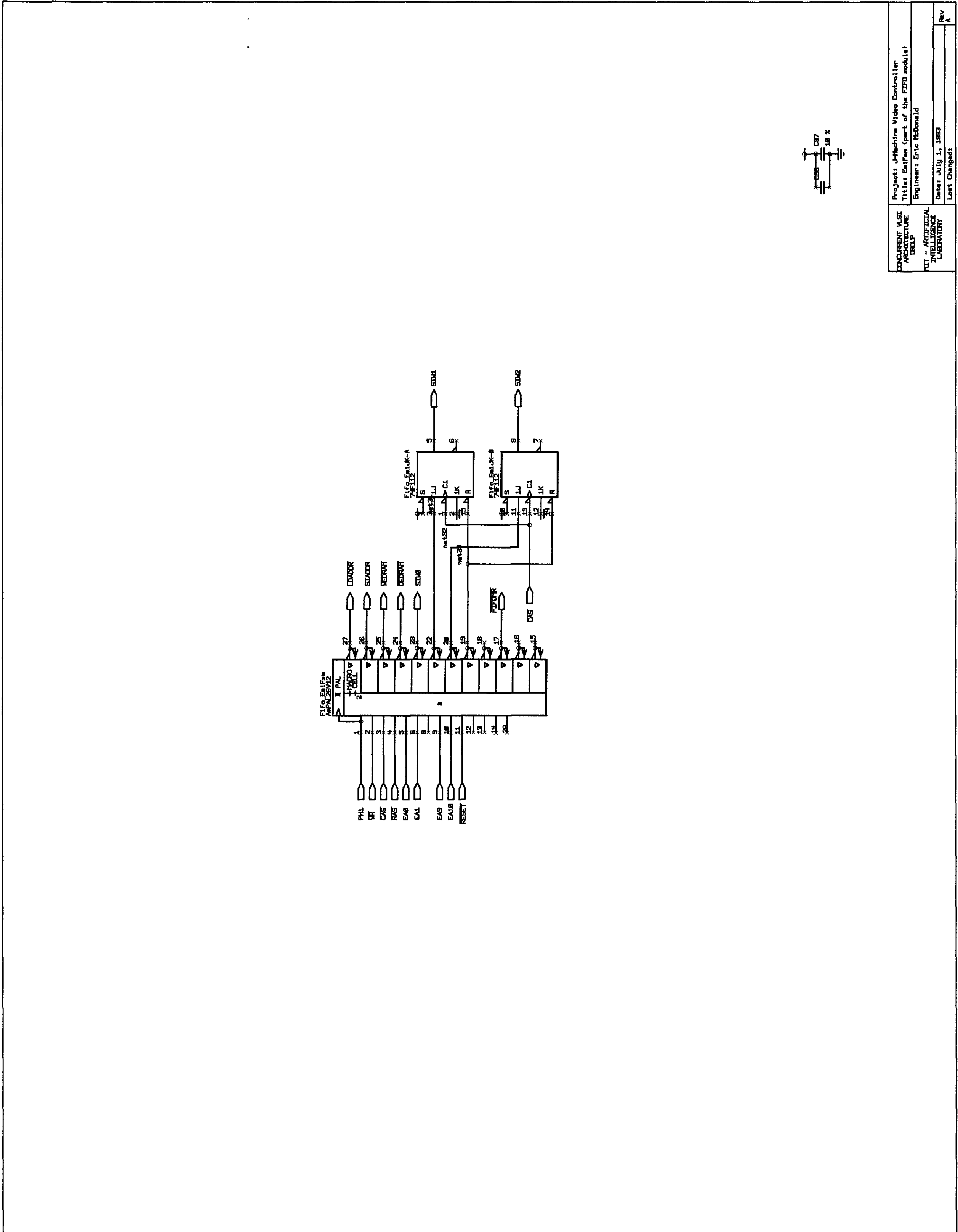
erm.abl

```
module Erm

title 'EMI Readback Module PAL

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------
" Decodes EMI reads from address >= $80000 according to this table:
"  A[19:16]     Sample address         Output signal
"  --------     -------------          -------------
"    10X1          $90000                /MPU2ERM0  (cycle 0)
"                                        /MPU2ERM1  (cycle 1)
"                                        /MPU2ERM2  (cycle 2)
"                                        (access here will also set ErmAck)
"    10X0          $80000                same as above w/o setting ErmAck
"    11X0          $c0000                /XLM2ERM
"    11X1          $d0000                /FIFO2ERM

Erm device 'P22V10';

"*** Inputs

PH1                                      Pin 1;
RAS, CAS, WR                             Pin 2, 3, 4;
EA10, EA9, CSB1, CSB0                    Pin 5, 6, 7, 8;
MpuStat                                  Pin 9;
RESET                                    Pin 10;

"*** Outputs

Xlm2Erm                                  Pin 23 istype 'invert';
Fifo2Erm                                 Pin 22 istype 'invert';
Mpu2Erm0, Mpu2Erm1, Mpu2Erm2             Pin 21, 20, 19 istype 'invert';
ErmAck                                   Pin 18 istype 'reg,buffer';
BufEn                                    Pin 17;
MpuReady                                 Pin 16 istype 'reg,buffer';
Extra                                    Pin 15;

"*** For internal use
DoAck                                    Pin 14 istype 'reg,buffer';

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;
ZERO = [0, 0];
ONE  = [0, 1];
TWO  = [1, 0];
CSB = [CSB1, CSB0];

"*** Equations

equations

[DoAck,ErmAck,MpuReady].clk = PH1;
```

erm.abl

```
Extra = 0;

" A19 appears on EA10 during the zeroth cycle (when RAS first drops), and
" A18,A16 appear on EA10,EA9 during the next memory cycle (when CAS 1st drops).
!BufEn = (WR & EA10 & !RAS & CAS) # (!BufEn & !RAS);

!Fifo2Erm = (!BufEn & !CAS & EA10 & EA9 & (CSB == ZERO))
           # (!Fifo2Erm & !RAS);

!Xlm2Erm  = (!BufEn & !CAS & EA10 & !EA9 & (CSB == ZERO))
           # (!Xlm2Erm & !RAS);

!Mpu2Erm0 = (!BufEn & !CAS & !EA10 & (CSB == ZERO));

!Mpu2Erm1 = (!BufEn & !CAS & !EA10 & (CSB == ONE));

!Mpu2Erm2 = (!BufEn & !CAS & !EA10 & (CSB == TWO));

DoAck     := (!BufEn & !CAS & !EA10 & EA9 & (CSB == ZERO))
           # (DoAck & !ErmAck);

ErmAck    := (MpuReady & !Mpu2Erm2 & DoAck)
           # (ErmAck & MpuReady);

MpuReady := MpuStat;

end Erm
```


ModFsm
EmlFsm
Mdata
Maddr
Fbus


CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY
Project: J-Machine Video Controller
Engineer: Eric McDonald
Date: July 1, 1993
Last Changed:
Rev
A

fifopal.abl

```
module FifoPal
title 'FIFO Controller Pal

  Eric McDonald     February 15, 1994
  Last revised:     March 1, 1994

  DESCRIPTION:
  ------------
  Determines whether an MDP emem access is meant for the DRAM (< $80000)
  or the FIFO (>= $80000).  It provides appropriate signals to DRAMs and PALs.'

FifoPal  device 'P26V12';

"*** Inputs

PH1                             Pin 1;
WR,CAS,RAS                      Pin 2, 3, 4;  "EMI control signals
CSB0, CSB1                      Pin 5, 6;     "Indicate which word is on bus.
EA9,EA10                        Pin 9, 10;
RESET                           Pin 11;

"*** Outputs

"FIFO control lines...
LdAddr                          Pin 27;    "Latch 10 bits of address
SIaddr                          Pin 26;    "Shift in all 20 bits of address
                                           "(current 10 + latched 10)
SIword0                         Pin 23;    "Shift in first 12 data bits
SIword1set                      Pin 22;    "...next 12 data bits     (to JK FF)
SIword2set                      Pin 20;    "...and last 12 data bits (to JK FF)
SIwordReset                     Pin 19;    "Clear SIword{1,2} from JK FF

"EMI DRAM control lines...
WE                              Pin 25;    "/WE line for DRAMs
OE                              Pin 24;    "/OE line for DRAMs

"FIFO reset line
FifoMR                          Pin 17;

"Internal register bits
A19                             Pin 18;
A19Read                         Pin 16;

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;
ZERO = [0, 0];
ONE  = [0, 1];
TWO  = [1, 0];
CSB = [CSB1, CSB0];

DRAM_ACCESS = !A19;          "DRAM takes the lower half of address space
FIFO_ACCESS =  A19;          "..and the Video RAM takes the upper half

Word0 = (CSB == ZERO);
Word1 = (CSB == ONE);
Word2 = (CSB == TWO);
```

fifopal.abl

```
"*** Equations

equations

"All MDP external WRITES to locations >= $80000 should go to the FIFO.
"Therefore, if EA10 is 1 when RAS first goes low (i.e. A19 = 1) during
"  a write access, we are writing into the FIFO.
A19 = (!WR & RAS & EA10) # (!RAS & A19);
A19Read = (RAS & WR & EA10) # (!RAS & A19Read);

"***
"DRAM control lines...
"***
"We should de-assert the DRAMs /OE and /WE inputs for the duration of
"  FIFO or MDPReadBack access cycles.
"
OE = A19Read # !WR;
WE = FIFO_ACCESS # WR;

"***
"FIFO control lines...
"***
"The high-to-low transition of /LdAddr latches the row of the external
"  memory address.  Keep it low for the entire ememaccess cycle.
!LdAddr = (EA10 & !WR & !RAS & CAS) # (!RAS & !LdAddr);

"The low-to-high transition of SIaddr shifts the latched addr row and the
"  current addr column into the FIFOs.  The subsequent high-to-low return
"  of SIaddr propagates the address through the FIFO.
SIaddr = (!WR & !RAS & FIFO_ACCESS & !CAS & Word0) # (SIaddr & !RAS);
" WARNING: Watch out for a race condition here.  The 'output ready' from
"   the FIFO's comes only from a DATA FIFO, so it is assumed that the
"   ADDR FIFO is in sync with it.  However, SIaddr goes low at 'about'
"   the same time as the SIword's do, so the ADDR FIFO may not have
"   propagated the address all the way through yet.
"
"SIword0 goes right to its FIFO.  SIword{1,2} go to the J inputs of a
" JK Flip Flop whic is clocked off the falling edge of /CAS.
SIword0 = (!WR & !RAS & FIFO_ACCESS & !CAS & Word0) # (SIword0 & !RAS);
SIword1set = (!WR & !RAS & FIFO_ACCESS & Word1);
SIword2set = (!WR & !RAS & FIFO_ACCESS & Word2);
!SIwordReset = RAS;

FifoMR = RESET;

end FifoPal
```

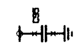
CONCURRENT VLSI ARCHITECTURE GROUP

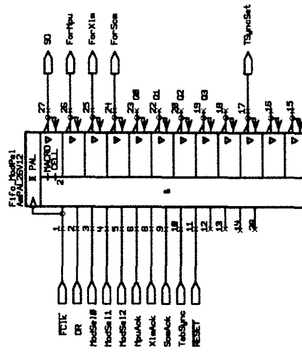
MIT - ARTIFICIAL INTELLIGENCE LABORATORY

Project: J-Machine Video Controller
Title: ModFsm (part of the FIFO module)
Engineer: Eric McDonald

Date: July 1, 1993
Last Changed:

Rev A





mod.abl

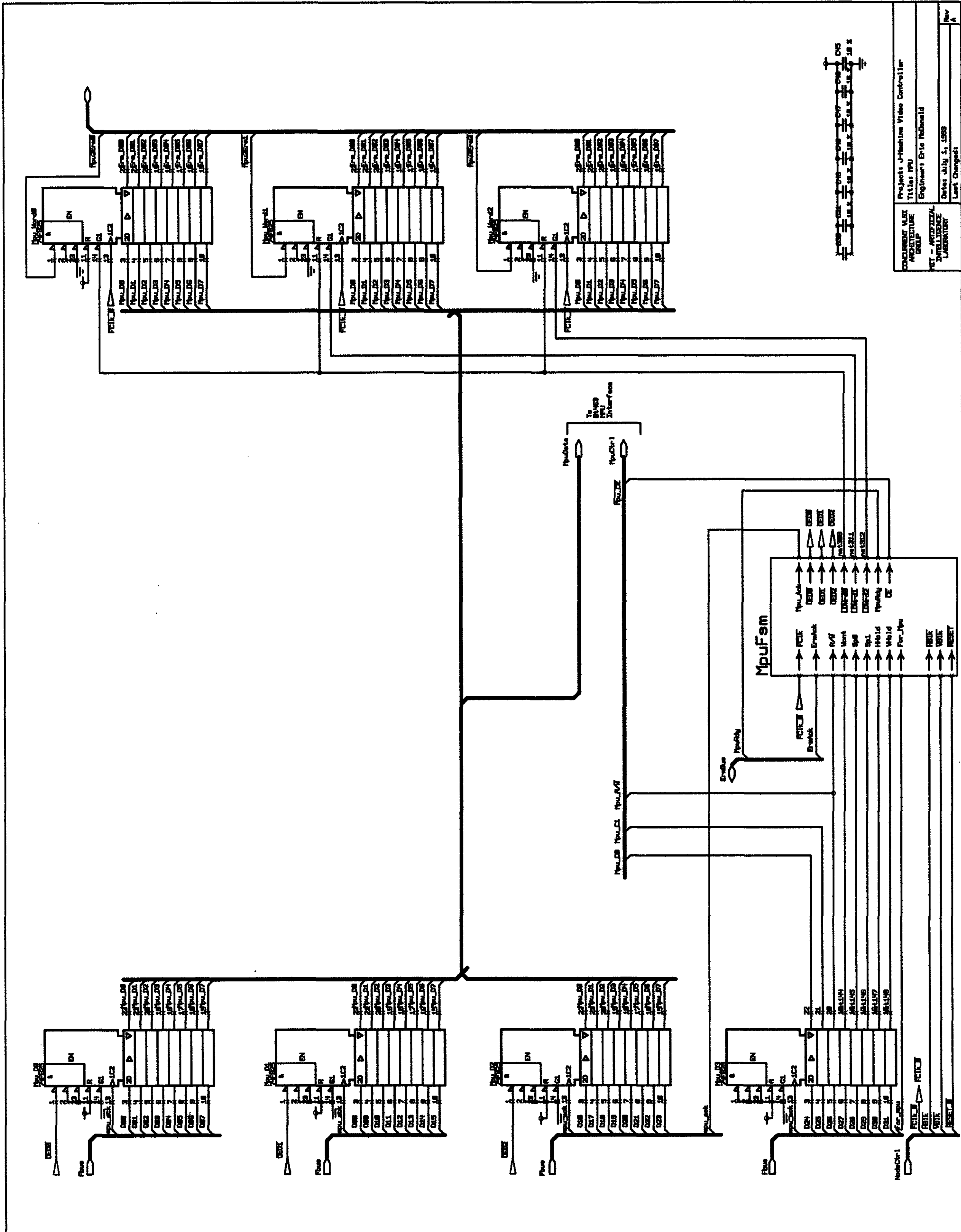
```
module Mod

title 'FIFO Mod FSM

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------
"  Awaits OR condition from FIFOs and tries to deliver data to
"  appropriate module, as determined by the table:
"
"    Modsel     Beg. MDP Addr.          Module
"    ------     --------------          ---------
"     000         $80000                MPU
"     001         $90000                XLM
"     100         $c0000                SCM
"     101         $d0000                Tabsync

Mod device 'P26V12';

"*** Inputs

FClk                          Pin 1;
OR                            Pin 2;          " FIFO Output Ready
ModSel0, ModSel1, ModSel2     Pin 3, 4, 5;    " Addr[18:16]
MpuAck, XlmAck, ScmAck        Pin 6, 8, 9;    " ACKs from modules
TabSync                       Pin 10;
RESET                         Pin 11;

"*** Outputs

SO                    Pin 27 istype 'reg,buffer';          " FIFO Shift Out
ForMpu, ForXlm, ForScm  Pin 26, 25, 24 istype 'reg,buffer';   " RDYs to mods
Q0, Q1, Q2, Q3        Pin 23, 22, 20, 19 istype 'reg,buffer'; " state bits
TSyncSet              Pin 17;


"*** Bits used internally
TSet                  Pin 16 istype 'reg,buffer';

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;
ModSel = [ModSel2..ModSel0];
MPUSEL =   [0, 0, 0];
XLMSEL =   [0, 0, 1];
SCMSEL =   [1, 0, 0];
TSYNCSEL = [1, 0, 1];


"*** State declarations

current_state  = [Q2..Q0];

" Requires single-bit-only differences between:
"   Init0,Idle   (coming from Init1)
```

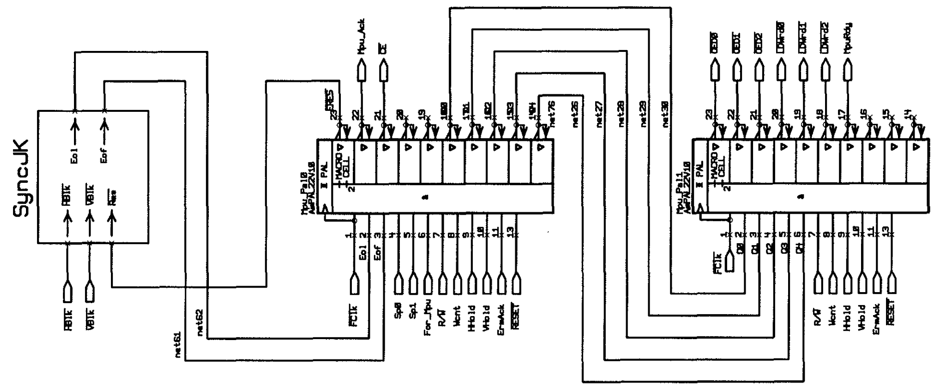
mod.abl

```
"   Idle,NewData (coming from Idle)

Init0    = [ 0, 0, 0 ];   "Q= 0
Init1    = [ 0, 0, 1 ];   "Q= 1
Idle     = [ 0, 1, 0 ];   "Q= 2
NewData  = [ 1, 1, 0 ];   "Q= 6
BadData  = [ 1, 1, 1 ];   "Q= 7
Handoff0 = [ 0, 1, 1 ];   "Q= 3
Handoff1 = [ 1, 0, 0 ];   "Q= 4
TabSync0 = [ 1, 0, 1 ];   "Q= 5


"*** Equations

equations

[current_state,TSet].clk = FClk;
[current_state,TSet].ar  = !RESET;
[SO,ForMpu,ForXlm,ForScm].clk = FClk;

TSyncSet = TSet # (TSyncSet & !TabSync);

State_Diagram current_state
        state Init0:
          SO := 1;                      " Upon RESET, flush FIFO
          goto Init1;
        state Init1:
          if (OR) then Init0;
            else Idle;

        state Idle:                     " Wait here for OR from FIFO
          if (OR) then NewData
            else Idle;

        state NewData:
          if (ModSel == MPUSEL) then Handoff0
                        with ForMpu := 1 endwith;
            else if (ModSel == XLMSEL) then Handoff0
                        with ForXlm := 1 endwith;
              else if (ModSel == SCMSEL) then Handoff0
                        with ForScm := 1 endwith;
                else if (ModSel == TSYNCSEL) then TabSync0
                        with SO := 1; TSet := 1 endwith;
                  else BadData
                        with SO := 1 endwith; "Ignore unrecognized MODSEL
        state BadData:
          goto Idle;    " This state allows SO to be asserted after bad data

        state Handoff0:                 " Tell module valid data is waiting
          ForMpu := ForMpu;  ForScm := ForScm;  ForXlm := ForXlm;
          if ((ForMpu & MpuAck) # (ForXlm & XlmAck) # (ForScm & ScmAck))
             then Handoff1;
               else Handoff0;
        state Handoff1:                 " Drop our DR signal and wait
          SO := 1;                      " for module to drop its ACK...
          if (MpuAck # XlmAck # ScmAck)
             then Handoff1;             " Note that ALL ACKs must be clear
```

mod.abl

```
             else Idle;                  " (can fix this with more states)

        state TabSync0:
          goto Idle;  " This state allows TSet and SO to be asserted

end Mod
```




SyncJK
Project: J-Machine Video Controller
Title: Mpu FSM
Engineer: Eric McDonald
Date: July 1, 1993
Last Changed:
Rev
A
CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY

mpu0.abl

```
module Mpu0

title 'Bt463 Mpu FSM PAL#0

  Eric McDonald     January   5, 1994
  Last revised:     February 15, 1994'

" DESCRIPTION:
" ------------

Mpu0 device 'P22V10';

"*** Inputs

FClk                            Pin 1;
Eol, Eof                        Pin 2, 3;       " Horiz and Vert syncs
Sp0, Sp1                        Pin 4, 5;       " Spare inputs (unused)
For_Mpu                         Pin 6;          " FIFO has data for us
Read                            Pin 7;          " Is access read or /write?
Wcnt                            Pin 8;          " # of 8-bit wrds? 0->1 1->3
HHold, VHold                    Pin 9, 10;      " Should we sync up?
ErmAck                          Pin 11;         " ERM rcvd our data
RESET                           Pin 13;         " Global /RESET line

"*** Outputs

ERES                            Pin 23 istype 'reg,invert'; " Reset Eol, Eof
Mpu_Ack                         Pin 22 istype 'reg,buffer'; " Latch & ACK data
CE                              Pin 21 istype 'reg,invert'; " Bt463 /CE line
Q0,Q1,Q2,Q3,Q4,Q5,Q6            Pin 20,19,18,17,16,15,14 istype 'reg,buffer';

"*** Bits used internally

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;
LongWord  = (Wcnt == 1);
ShortWord = (Wcnt == 0);
Write = !Read;

current_state  = [Q6..Q0];
@include 'mpu.sta'

"*** Equations

equations

[current_state, ERES, Mpu_Ack, CE].clk = FClk;
[current_state].ar  = !RESET;

State_Diagram current_state

        state Idle:                        " Loop here...
           if (For_Mpu) then NewData0 " If new incoming data, grab it
                         with Mpu_Ack := 1; !ERES := 1; endwith;
           else Idle;
```

mpu0.abl

```
        state NewData0:            " Loop until we know FIFO has seen our ACK
          if (For_Mpu) then NewData0 with Mpu_Ack := 1 endwith;
            else NewData1;
        state NewData1:
          if (VHold # HHold) then LoopTilEnd;
              else case Read : ReadData0a;
                         Write : WriteData0a;
                    endcase;

        state LoopTilEnd:
          if ((VHold & !Eof) # (HHold & !Eol)) then LoopTilEnd;
            else Dispatch;
        state Dispatch:
          !ERES := 1;
          case Read : ReadData0a;
              Write : WriteData0a;
          endcase;

        state WriteData0a:
"         !DriveW0 = 1;
          !CE := 1;
          goto WriteData0b;
        state WriteData0b:
"         !DriveW0 = 1;
          !CE := 1;
          goto WriteData0c;
        state WriteData0c:
"         !DriveW0 = 1;
          if (ShortWord) then Idle;
            else WriteData1a;

        state WriteData1a:
"         !DriveW1 = 1;
          !CE := 1;
          goto WriteData1b;
        state WriteData1b:
"         !DriveW1 = 1;
          !CE := 1;
          goto WriteData1c;
        state WriteData1c:
"         !DriveW1 = 1;
          goto WriteData2a;

        state WriteData2a:
"         !DriveW2 = 1;
          !CE := 1;
          goto WriteData2b;
        state WriteData2b:
"         !DriveW2 = 1;
          !CE := 1;
          goto WriteData2c;
        state WriteData2c:
"         !DriveW1 = 1;
          goto Idle;

        state ReadData0a:
          !CE := 1;
```

mpu0.abl

```
          goto ReadData0b;
        state ReadData0b:
          !CE := 1;
"         !LdMR0 := 1;
          goto ReadData0c;
        state ReadData0c:
          !CE := 1;
          if (ShortWord) then SendToErm0;
            else ReadData0d;
        state ReadData0d:
          goto ReadData1a;  " Meets minimum !CE high requirement

        state ReadData1a:
          !CE := 1;
          goto ReadData1b;
        state ReadData1b:
          !CE := 1;
"         !LdMR1 := 1;
          goto ReadData1c;
        state ReadData1c:
          !CE := 1;
          goto ReadData1d;
        state ReadData1d:
          goto ReadData2a;  " Meets minimum !CE high requirement

        state ReadData2a:
          !CE := 1;
          goto ReadData2b;
        state ReadData2b:
          !CE := 1;
"         !LdMR2 := 1;
          goto ReadData2c;
        state ReadData2c:
          !CE := 1;
          goto ReadData2d;
        state ReadData2d:
          goto SendToErm0;  " Meets minimum !CE high requirement

        state SendToErm0:
"         MpuRdy := 1;
          goto SendToErm1;
        state SendToErm1:
"         MpuRdy := 1;
          if (ErmAck) then SendToErm3;
            else SendToErm2;
        state SendToErm2:
"         MpuRdy := 1;
           if (For_Mpu) then NewData0 " If new incoming data, grab it
                         with Mpu_Ack := 1 endwith;
           else SendToErm1;
        state SendToErm3:
          if (ErmAck) then SendToErm3;    " Wait for ErmAck to drop
            else SendToErm4;
        state SendToErm4:
          goto Idle;

end Mpu0
```

mpu1.abl

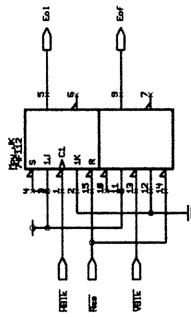
```
module Mpu1

title 'Bt463 Mpu FSM PAL#1

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------

Mpu1 device 'P22V10';

"*** Inputs

FClk                           Pin 1;
Q0,Q1,Q2,Q3,Q4                 Pin 2,3,4,5,6;
Read                           Pin 7;          " Is access read or /write?
Wcnt                           Pin 8;          " # of 8-bit wrds? 0->1 1->3
HHold, VHold                   Pin 9, 10;      " Should we sync up?
ErmAck                         Pin 11;         " ERM rcvd our data
RESET                          Pin 13;         " Global /RESET line

"*** Outputs

DriveW0,DriveW1,DriveW2        Pin 23,22,21 istype 'reg,invert';  " /OE
LdMR0,LdMR1,LdMR2              Pin 20,19,18 istype 'reg,invert';  " /Latch
MpuRdy                         Pin 17 istype 'reg,buffer';  " To ERM module

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

current_state  = [Q4..Q0];
Idle         = [ 0, 0, 0, 0, 0 ];
NewData0     = [ 0, 0, 0, 1, 0 ];
NewData1     = [ 0, 0, 0, 1, 1 ];
LoopTilEnd   = [ 0, 0, 0, 0, 1 ];
Dispatch     = [ 0, 0, 0, 0, 1 ];

WriteData0   = [ 0, 0, 1, 0, 0 ];
WriteData1   = [ 0, 0, 1, 0, 1 ];
WriteData2   = [ 0, 0, 1, 1, 1 ];

ReadData0b   = [ 0, 1, 0, 0, 1 ];
ReadData1b   = [ 0, 1, 0, 1, 1 ];
ReadData2b   = [ 0, 1, 1, 1, 1 ];

SendToErm0   = [ 1, 0, 0, 0, 0 ];
SendToErm1   = [ 1, 0, 0, 0, 1 ];
SendToErm2   = [ 1, 0, 0, 1, 1 ];
SendToErm3   = [ 1, 0, 1, 1, 1 ];
SendToErm4   = [ 1, 0, 1, 1, 1 ];

In_State MACRO (st) { (current_state == ?st) };

"*** Equations
```

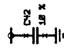
mpu1.abl

```
equations

!DriveW0  := In_State(WriteData0);
!DriveW1  := In_State(WriteData1);
!DriveW2  := In_State(WriteData2);

!LdMR0 := In_State(ReadData0b);
!LdMR1 := In_State(ReadData1b);
!LdMR2 := In_State(ReadData2b);

MpuRdy := In_State(SendToErm0)
        # In_State(SendToErm1)
        # In_State(SendToErm2);

end Mpu1
```

mpu.sta

```
"*** State declarations
" These pairs require one-bit state bit differences:
"    Idle->NewData0
"    NewData0 -> NewData1
"    LoopTilEnd -> Dispatch
"    SendErm2, SendErm3 (from SendErm1)
"    SendErm3 -> SendErm4

Idle        = [ 0, 0, 0, 0, 0,    0, 0 ]; "00-0 (00)
NewData0    = [ 0, 0, 0, 1, 0,    0, 0 ]; "02-0 (08)
NewData1    = [ 0, 0, 0, 1, 1,    0, 0 ]; "03-0 (0c)
LoopTilEnd  = [ 0, 0, 0, 0, 1,    0, 0 ]; "01-0 (04)
Dispatch    = [ 0, 0, 0, 0, 1,    0, 1 ]; "01-1 (05)

WriteData0a = [ 0, 0, 1, 0, 0,    0, 0 ]; "04-0 (10)
WriteData0b = [ 0, 0, 1, 0, 0,    0, 1 ]; "04-1 (11)
WriteData0c = [ 0, 0, 1, 0, 0,    1, 1 ]; "04-3 (13)

WriteData1a = [ 0, 0, 1, 0, 1,    0, 0 ]; "05-0 (14)
WriteData1b = [ 0, 0, 1, 0, 1,    0, 1 ]; "05-1 (15)
WriteData1c = [ 0, 0, 1, 0, 1,    1, 1 ]; "05-3 (17)

WriteData2a = [ 0, 0, 1, 1, 1,    0, 0 ]; "07-0 (1c)
WriteData2b = [ 0, 0, 1, 1, 1,    0, 1 ]; "07-1 (1d)
WriteData2c = [ 0, 0, 1, 1, 1,    1, 1 ]; "07-3 (1f)

ReadData0a  = [ 0, 1, 0, 0, 0,    0, 0 ]; "08-0 (20)
ReadData0b  = [ 0, 1, 0, 0, 1,    0, 0 ]; "09-0 (24)
ReadData0c  = [ 0, 1, 0, 0, 0,    0, 1 ]; "08-1 (21)
ReadData0d  = [ 0, 1, 0, 0, 0,    1, 1 ]; "08-3 (23)

ReadData1a  = [ 0, 1, 0, 1, 0,    0, 0 ]; "0a-0 (28)
ReadData1b  = [ 0, 1, 0, 1, 1,    0, 0 ]; "0b-0 (2c)
ReadData1c  = [ 0, 1, 0, 1, 0,    0, 1 ]; "0a-1 (29)
ReadData1d  = [ 0, 1, 0, 1, 0,    1, 1 ]; "0a-3 (2b)

ReadData2a  = [ 0, 1, 1, 1, 0,    0, 0 ]; "0e-0 (38)
ReadData2b  = [ 0, 1, 1, 1, 1,    0, 0 ]; "0f-0 (3c)
ReadData2c  = [ 0, 1, 1, 1, 0,    0, 1 ]; "0e-1 (39)
ReadData2d  = [ 0, 1, 1, 1, 0,    1, 1 ]; "0e-3 (3b)

SendToErm0  = [ 1, 0, 0, 0, 0,    0, 0 ]; "10-0 (40)
SendToErm1  = [ 1, 0, 0, 0, 1,    0, 0 ]; "11-0 (44)
SendToErm2  = [ 1, 0, 0, 1, 1,    0, 0 ]; "13-0 (4c)
SendToErm3  = [ 1, 0, 1, 1, 1,    0, 0 ]; "17-0 (5c)
SendToErm4  = [ 1, 0, 1, 1, 1,    0, 1 ]; "17-1 (5d)
```

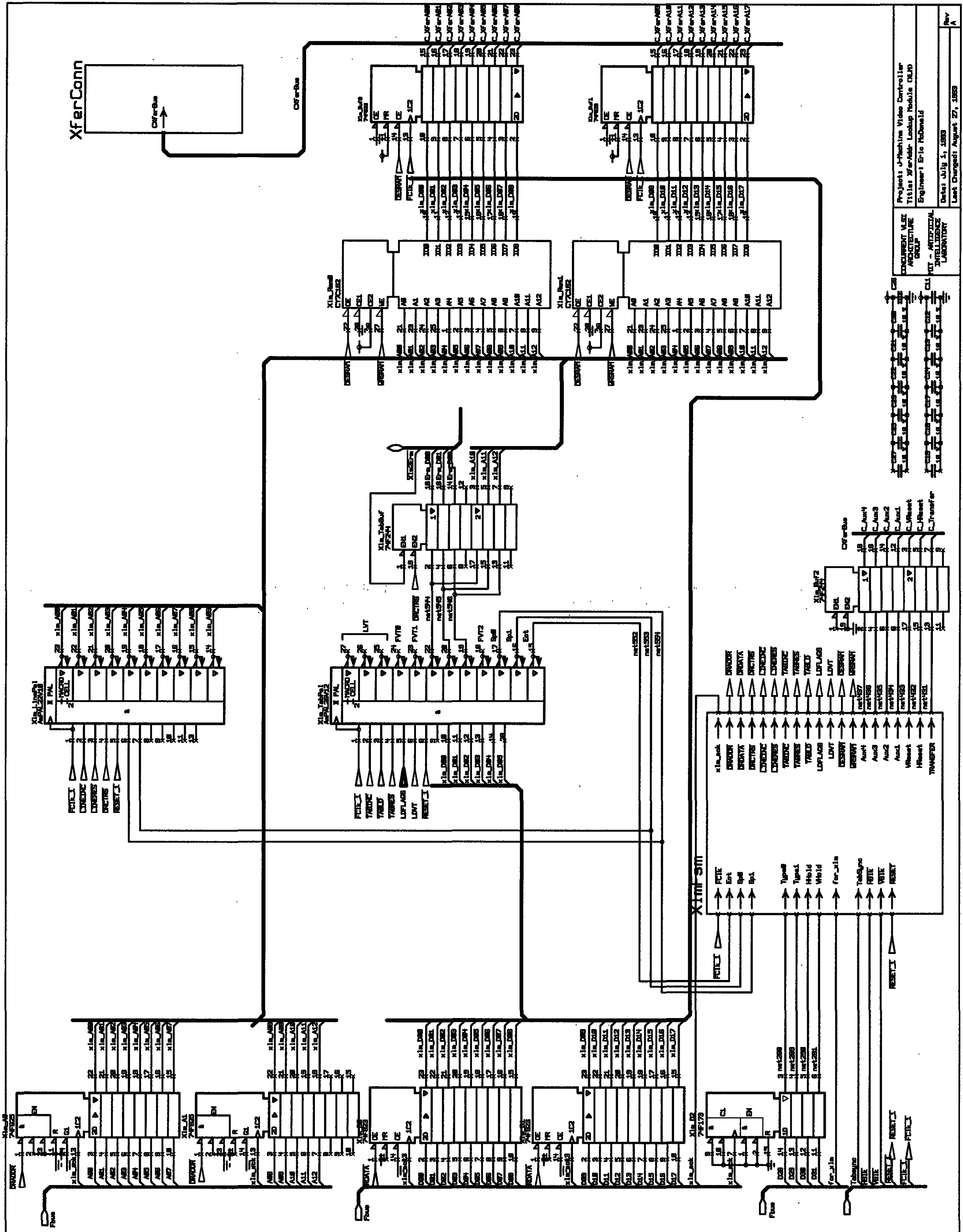


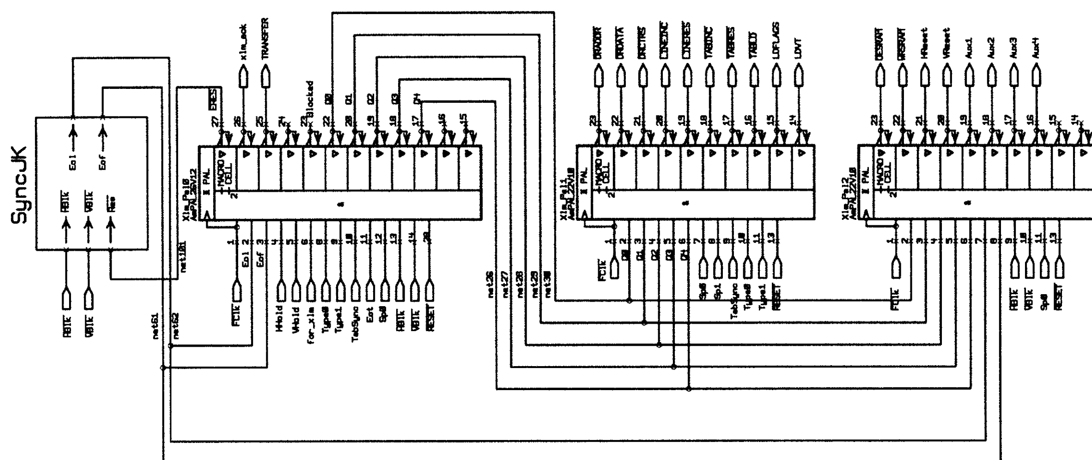


CONCURRENT VLSI ARCHITECTURE GROUP

MIT - ARTIFICIAL INTELLIGENCE LABORATORY

Project: J-Machine Video Controller
Title: Video Sync JK
Engineer: Eric McDonald
Date: July 1, 1993
Last Changed:
Rev A


XferConn



xlm0.abl

```
module Xlm0

title 'XLM FSM PAL#0

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------

Xlm0 device 'P26V12';

"*** Inputs

FClk                            Pin 1;
Eol_Async, Eof_Async            Pin 2, 3;        " Horiz and Vert syncs
HHold, VHold                    Pin 4, 5;        " Should we sync up?
For_Xlm_Async                   Pin 6;           " FIFO has data for us
Type0, Type1                    Pin 8, 9;        " Type of data
TabSync                         Pin 10;          "
Eot                             Pin 11;          " Reached end of tables?
Sp0                             Pin 12;          " Unused spare
HBlk, VBlk                      Pin 13, 14;      " Blanking signals
RESET                           Pin 28;          " Global /RESET line

"*** Outputs

ERES                            Pin 27 istype 'reg,invert'; " Reset Eol, Eof
Xlm_Ack                         Pin 26 istype 'reg,buffer'; " Latch & ACK data
TRANSFER                        Pin 25 istype 'reg,buffer'; " XLM addr valid
Q0, Q1, Q2, Q3, Q4              Pin 22,20,19,18,17 istype 'reg,buffer';  "State
Eol,Eof,For_Xlm                 Pin 24,16,15 istype 'reg,buffer'; "Sync up

"*** Bits used internally
Blocked                         Pin 23 istype 'reg,buffer'; " Waiting for blk?

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

Type =          [Type1, Type0];
WRITE_XFERADDR = [0, 0];
LOAD_TABLES    = [0, 1];
LOAD_FLAGS     = [1, 1];
UNDEFINED      = [1, 0];

"*** State declarations
@include 'xlm.sta'

"*** Equations

equations

current_state.clk          = FClk;
[TRANSFER,ERES,Xlm_Ack,Blocked,Eof,Eol,For_Xlm].clk = FClk;
current_state.ar           = !RESET;
```

xlm0.abl

```
Eol := Eol_Async;
Eof := Eof_Async;
For_Xlm := For_Xlm_Async;

State_Diagram current_state

        state Idle:
          if (Eof&ERES) then UpdateCtrs;      "End of frame, update counters
            else if (Eol&ERES) then Broadcast0; "End of line, send address
              else if (For_Xlm) then NewData0     "New table data, record it
                else Idle;

        state NewData0:             " Loop until we know FIFO has seen our ACK
          if (For_Xlm) then NewData0
                with Xlm_Ack := 1 endwith;
          else NewData1;
        state NewData1:
          if (HHold # VHold) then LoopTilEnd
            else goto Dispatch;

        state Dispatch:
          if (Type == WRITE_XFERADDR) then WriteData0;
            else if (Type == LOAD_TABLES) then LoadTables;
            else if (Type == LOAD_FLAGS) then LoadFlags;
            else goto Idle;                 " Discard unknown data type

        state LoopTilEnd:
          Blocked := 1;
          if (Eof # Eol) then Broadcast0;
            else LoopTilEnd;

        state WriteData0:
"         !DRADDR := 1;         Enable address and data lines coming
"         !DRDATA := 1;         from FIFO registers...
          goto WriteData1;
        state WriteData1:
"         !DRADDR := 1;         Write into SRAM one cycle later
"         !DRDATA := 1;
"         !WRSRAM := 1;
          goto WriteData2;
        state WriteData2:
"         !DRADDR := 1;         And keep address/data valid for 1 more cycle
"         !DRDATA := 1;
          goto Idle;

        state LoadTables:
"         !DRDATA := 1;         Drive FIFO data lines and ask TABPAL to
"         LDVT := 1;            latch in new tables
          goto Idle;

        state LoadFlags:
"         !DRDATA := 1;         Drive FIFO data lines and ask TABPAL to
"         LDFLAGS := 1;         latch in new flags
          goto Idle;

        state UpdateCtrs:
"         !LINERES := 1;    Reset low SRAM address bits to zero
```

xlm0.abl

```
          if (TabSync # Eot) then ResetTabCtr; " Reset high addr bits if
                                               " last table or TabSync
           else IncTabCtr;                     " Otherwise, just increment them
        state ResetTabCtr:
"         !TABLD := 1;
          goto Broadcast0;
        state IncTabCtr:
"         !TABINC := 1;
          goto Broadcast0;

        state Broadcast0:
          Blocked := Blocked;    " Remember if blocked
"         !DRCTRS := 1;            Drive address lines to SRAM...
"         !OESRAM := 1;            and drive SRAM into output latches
          goto Broadcast1;
        state Broadcast1:
          Blocked := Blocked;    " Remember if blocked
"         !DRCTRS := 1;            Drive address lines to SRAM...
"         !OESRAM := 1;            and drive SRAM into output latches
          goto Wait0;

        state Wait0:
          Blocked := Blocked;
          goto Wait1;

        state Wait1:
          Blocked := Blocked;
          goto Wait2;

        state Wait2:
          Blocked := Blocked;
          goto Wait3;

        state Wait3:
          Blocked := Blocked;
          if (!VBlk)
             then BrdcstWithoutInc;
          else
             BrdcstAndInc;

        state BrdcstAndInc:
          Blocked := Blocked;    " Remember if blocked
          TRANSFER := 1;
"         !LINEINC := VBlk;
          if (!Blocked) then Idle
                         with !ERES := 1 endwith;
          else if (HHold # (VHold & Eof)) then Dispatch
                         with !ERES := 1 endwith;
          else LoopTilEnd
                         with Blocked := 1; !ERES := 1 endwith;
        state BrdcstWithoutInc:
          Blocked := Blocked;    " Remember if blocked
          TRANSFER := 1;
          if (!Blocked) then Idle
                         with !ERES := 1 endwith;
          else if (HHold # (VHold & Eof)) then Dispatch
                         with !ERES := 1 endwith;
```

xlm0.abl

```
          else LoopTilEnd
                         with Blocked := 1; !ERES := 1 endwith;
end Xlm0
```

xlm1.abl

```
module Xlm1

title 'XLM FSM PAL#1

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------

Xlm1 device 'P22V10';

"*** Inputs

FClk                    Pin 1;
Q0, Q1, Q2, Q3, Q4      Pin 2,3,4,5,6;   " State
Sp0, Sp1                Pin 7, 8;        " Unused spares
TabSync                 Pin 9;           "
Type0, Type1            Pin 10, 11;      " Type of data
RESET                   Pin 13;          " Global /RESET line

"*** Outputs

DRADDR                  Pin 23 istype 'reg,invert';  " Ctrl sgnl to ADDR0,1
DRDATA                  Pin 22 istype 'reg,invert';  " Ctrl sgnl to DATA0,1
DRCTRS                  Pin 21 istype 'reg,invert';  " Ctrl sgnl to LINE/TAB

LINEINC                 Pin 20 istype 'reg,invert';  " Ctrl sgnl to LINEPAL
LINERES                 Pin 19 istype 'reg,invert';  " Ctrl sgnl to LINEPAL
TABINC                  Pin 18 istype 'reg,invert';  " Ctrl sgnl to TABPAL
TABRES                  Pin 17 istype 'reg,invert';  " Ctrl sgnl to TABPAL
TABLD                   Pin 16 istype 'reg,invert';  " Ctrl sgnl to TABPAL

LDFLAGS                 Pin 15 istype 'reg,buffer';  " Ctrl sgnl to TABPAL
LDVT                    Pin 14 istype 'reg,buffer';  " Ctrl sgnl to TABPAL


"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

Type =         [Type1, Type0];
WRITE_XFERADDR = [0, 0];
LOAD_TABLES    = [0, 1];
LOAD_FLAGS     = [1, 1];
UNDEFINED      = [1, 0];

"*** State declarations
@include 'xlm.sta'

"*** Equations

equations

[DRADDR,DRDATA,DRCTRS].clk = FClk;

!DRADDR := In_State(WriteData0) #
```

xlm1.abl

```
            In_State(WriteData1) #
            In_State(WriteData2);

!DRDATA := In_State(WriteData0) #
           In_State(WriteData1) #
           In_State(WriteData2) #
           In_State(LoadTables) #
           In_State(LoadFlags);

!DRCTRS := In_State(Broadcast0) #
           In_State(Broadcast1);

!LINEINC := In_State(BrdcstAndInc);

!LINERES := In_State(UpdateCtrs);

!TABINC  := In_State(IncTabCtr);

!TABLD   := In_State(ResetTabCtr);

!TABRES  := 0;

LDFLAGS := In_State(LoadFlags);

LDVT    := In_State(LoadTables);

end Xlm1
```

xlm2.abl

```
module Xlm2

title 'XLM FSM PAL#2

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------

Xlm2 device 'P22V10';

"*** Inputs

FClk                     Pin 1;
Q0, Q1, Q2, Q3, Q4       Pin 2,3,4,5,6;  " State
Eol, Eof                 Pin 7, 8;       " Horiz and Vert syncs
HBlk,VBlk                Pin 9, 10;      " Blanking signals
Sp0                      Pin 11;         " Unused spare
RESET                    Pin 13;         " Global /RESET line

"*** Outputs

OESRAM                   Pin 23 istype 'reg,invert';  " Ctrl sgnl to SRAM
WRSRAM                   Pin 22 istype 'reg,invert';  " Ctrl sgnl to SRAM
HReset,VReset            Pin 21,20;
Aux1,Aux2,Aux3,Aux4      Pin 19,18,17,16;


"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

"*** State declarations
@include 'xlm.sta'

"*** Equations

equations

[OESRAM,WRSRAM,Aux1,Aux2,Aux3,Aux4].clk = FClk;

!OESRAM := In_State(Broadcast0) #
           In_State(Broadcast1);

!WRSRAM := In_State(WriteData1);

HReset = Eol;

VReset = Eof;

Aux1 = !HBlk;

Aux2 = !VBlk;

end Xlm2
```

xlm.sta

```
"*** State declarations

current_state = [Q4, Q3, Q2, Q1, Q0];

Idle         = [ 0, 0, 0, 0, 0 ];   "Q= 00

NewData0     = [ 0, 0, 0, 0, 1 ];   "Q= 01
NewData1     = [ 0, 0, 0, 1, 0 ];   "Q= 02
Dispatch     = [ 1, 0, 0, 1, 1 ];   "Q= 13

LoopTilEnd   = [ 0, 0, 0, 1, 1 ];   "Q= 03

WriteData0   = [ 0, 1, 0, 0, 0 ];   "Q= 08
WriteData1   = [ 0, 1, 0, 0, 1 ];   "Q= 09
WriteData2   = [ 0, 1, 0, 1, 0 ];   "Q= 0a

LoadTables   = [ 0, 1, 1, 0, 0 ];   "Q= 0c
LoadFlags    = [ 0, 1, 1, 1, 0 ];   "Q= 0e

Broadcast0   = [ 1, 0, 0, 0, 0 ];   "Q= 10
Broadcast1   = [ 1, 0, 0, 0, 1 ];   "Q= 11
BrdcstAndInc = [ 1, 0, 0, 1, 0 ];   "Q= 12
BrdcstWithoutInc = [ 1, 0, 1, 1, 0 ]; "Q=16

Wait0        = [ 0, 0, 1, 0, 0 ];    "Q=04
Wait1        = [ 0, 0, 1, 0, 1 ];    "Q=05
Wait2        = [ 0, 0, 1, 1, 1 ];    "Q=07
Wait3        = [ 0, 0, 1, 1, 0 ];    "Q=06

UpdateCtrs   = [ 1, 1, 0, 0, 0 ];   "Q= 18
ResetTabCtr  = [ 1, 1, 1, 0, 0 ];   "Q= 1c
IncTabCtr    = [ 1, 1, 1, 1, 0 ];   "Q= 1e

In_State MACRO (st) { (current_state == ?st) };
```

xlmline.abl

```
module XlmLine

title 'XLM Line Counter FSM

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------
" This PAL is basically just a 10-bit counter with /INC, /RES, and /OE lines

XlmLine device 'P22V10';

"*** Inputs

FClk                    Pin 1;
LineInc                 Pin 2;          " Increment counter
LineRes                 Pin 3;          " Reset counter to zero
OE                      Pin 4;          " Output enable
GRESET                  Pin 5;          " Global /RESET line

"*** Outputs

A9,A8,A7,A6,A5,A4,A3,A2,A1,A0
                        Pin 14,15,16,17,18,19,20,21,22,23 istype 'reg,buffer';

"*** Equations

HOLD = LineInc & LineRes;
INC  = !LineInc;
RESET = LineInc & !LineRes;

equations

[A9..A0].clk = FClk;
[A9..A0].oe  = !OE;
[A9..A0].ar  = !GRESET;

!A0 := RESET
     # (HOLD & !A0.fb)
     # (INC  &  A0.fb);

!A1 := RESET
     # (HOLD & !A1.fb)
     # (INC  & !A0.fb & !A1.fb)
     # (INC  &  A0.fb &  A1.fb);

!A2 := RESET
     # (HOLD & !A2.fb)
     # (INC  & !A0.fb & !A2.fb)
     # (INC  & !A1.fb & !A2.fb)
     # (INC  &  A0.fb &  A1.fb & A2.fb);

!A3 := RESET
     # (HOLD & !A3.fb)
     # (INC  & !A0.fb & !A3.fb)
     # (INC  & !A1.fb & !A3.fb)
```

xlmline.abl

```
     # (INC  & !A2.fb & !A3.fb)
     # (INC  &  A0.fb &  A1.fb & A2.fb & A3.fb);

!A4 := RESET
     # (HOLD & !A4.fb)
     # (INC  & !A0.fb & !A4.fb)
     # (INC  & !A1.fb & !A4.fb)
     # (INC  & !A2.fb & !A4.fb)
     # (INC  & !A3.fb & !A4.fb)
     # (INC  &  A0.fb &  A1.fb  & A2.fb & A3.fb & A4.fb);


!A5 := RESET
     # (HOLD & !A5.fb)
     # (INC  & !A0.fb & !A5.fb)
     # (INC  & !A1.fb & !A5.fb)
     # (INC  & !A2.fb & !A5.fb)
     # (INC  & !A3.fb & !A5.fb)
     # (INC  & !A4.fb & !A5.fb)
     # (INC  &  A0.fb &  A1.fb  & A2.fb & A3.fb & A4.fb & A5.fb);

!A6 := RESET
     # (HOLD & !A6.fb)
     # (INC  & !A0.fb & !A6.fb)
     # (INC  & !A1.fb & !A6.fb)
     # (INC  & !A2.fb & !A6.fb)
     # (INC  & !A3.fb & !A6.fb)
     # (INC  & !A4.fb & !A6.fb)
     # (INC  & !A5.fb & !A6.fb)
     # (INC  &  A0.fb &  A1.fb  & A2.fb & A3.fb & A4.fb & A5.fb & A6.fb);

!A7 := RESET
     # (HOLD & !A7.fb)
     # (INC  & !A0.fb & !A7.fb)
     # (INC  & !A1.fb & !A7.fb)
     # (INC  & !A2.fb & !A7.fb)
     # (INC  & !A3.fb & !A7.fb)
     # (INC  & !A4.fb & !A7.fb)
     # (INC  & !A5.fb & !A7.fb)
     # (INC  & !A6.fb & !A7.fb)
     # (INC  &  A0.fb &  A1.fb  & A2.fb & A3.fb & A4.fb & A5.fb & A6.fb
         & A7.fb);

!A8 := RESET
     # (HOLD & !A8.fb)
     # (INC  & !A0.fb & !A8.fb)
     # (INC  & !A1.fb & !A8.fb)
     # (INC  & !A2.fb & !A8.fb)
     # (INC  & !A3.fb & !A8.fb)
     # (INC  & !A4.fb & !A8.fb)
     # (INC  & !A5.fb & !A8.fb)
     # (INC  & !A6.fb & !A8.fb)
     # (INC  & !A7.fb & !A8.fb)
     # (INC  &  A0.fb &  A1.fb  & A2.fb & A3.fb & A4.fb & A5.fb & A6.fb
         & A7.fb & A8.fb);

" A9.fb is programmed differently because it was stuck on a pin without
```

xlmline.abl

```
" enough product terms (we need 11 but have only 8).
" It doesn't toggle back to 0 once it's been set.
" But we shouldn't be trying to write beyond the 10-bit address space anyway.
A9 :=  (HOLD &  A9.fb)
     # (INC  &  A9.fb)
     # (INC  &  A0.fb &  A1.fb  & A2.fb & A3.fb & A4.fb & A5.fb & A6.fb
         & A7.fb & A8.fb & !A9.fb);

end XlmLine
```

xlmtab.abl

```
module XlmTab

title 'XLM Table Counter FSM

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------

XlmTab device 'P26V12';

"*** Inputs

FClk                    Pin 1;
TabInc                  Pin 2;          " Increment table counter
TabLd                   Pin 3;          " Load FVT into table counter
TabRes                  Pin 4;          " Reset table counter to 0
LdFlags                 Pin 5;          " Load into FLAGS register
LdVT                    Pin 6;          " Load FVT and LVT registers
RESET                   Pin 8;          " Global /RESET line
LvtD0,LvtD1,LvtD2       Pin 10,11,12;   " Inputs for LVT
FvtD0,FvtD1,FvtD2       Pin 13,14,28;   " Inputs for FVT

"*** Outputs

LVT0,LVT1,LVT2          Pin 27,26,25 istype 'reg,buffer';  " Last Valid Table
FVT0,FVT1,FVT2          Pin 24,23,18 istype 'reg,buffer';  " First Valid Table
Tab0,Tab1,Tab2          Pin 22,20,19 istype 'reg,buffer';  " Current table
Sp0,Sp1                 Pin 17,16 istype 'reg,buffer';     " Spare flags
Eot                     Pin 15;   " Current table == Last Valid Table

"*** Bits used internally

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

LVT = [LVT2, LVT1, LVT0];
FVT = [FVT2, FVT1, FVT0];
Tab = [Tab2, Tab1, Tab0];
LvtData = [LvtD2, LvtD1, LvtD0];
FvtData = [FvtD2, FvtD1, FvtD0];

HOLD =  TabInc & TabLd;
INC  =  !TabInc;
LOAD =  TabInc & !TabLd;

"*** Equations

equations

LVT.clk = FClk;  FVT.clk = FClk;  Tab.clk = FClk;
[Sp1,Sp0].clk = FClk;

" Reload FVT and LVT registers upon receipt of LdVT signal from XLM0 Pal
FVT0 := (!LdVT & FVT0) # (LdVT & FvtD0);
```

xlmtab.abl

```
FVT1 := (!LdVT & FVT1) # (LdVT & FvtD1);
FVT2 := (!LdVT & FVT2) # (LdVT & FvtD2);

LVT0 := (!LdVT & LVT0) # (LdVT & LvtD0);
LVT1 := (!LdVT & LVT1) # (LdVT & LvtD1);
LVT2 := (!LdVT & LVT2) # (LdVT & LvtD2);

" Reload FLAGS register upon receipt of LdFlags signal from XLM0 Pal
Sp0 := (!LdFlags & Sp0) # (LdFlags & LvtD0);
Sp1 := (!LdFlags & Sp1) # (LdFlags & LvtD1);

" The table counter can either hold, reload, or increment by one

Tab.ar = !RESET;

!Tab0 := (HOLD & !Tab0)
       # (LOAD & !FVT0)
       # (INC & Tab0);

!Tab1 := (HOLD & !Tab1)
       # (LOAD & !FVT1)
       # (INC & !Tab0 & !Tab1)
       # (INC &  Tab0 &  Tab1);

!Tab2 := (HOLD & !Tab2)
       # (LOAD & !FVT2)
       # (INC & !Tab0 & !Tab2)
       # (INC & !Tab1 & !Tab2)
       # (INC &  Tab0 & Tab1 & Tab2);


" Raise Eot output when table counter == LVT

Eot = Tab == LVT;

end XlmTab
```

CONCURRENT VLSI ARCHITECTURE GROUP

MIT - ARTIFICIAL INTELLIGENCE LABORATORY

Project: J-Machine Video Controller

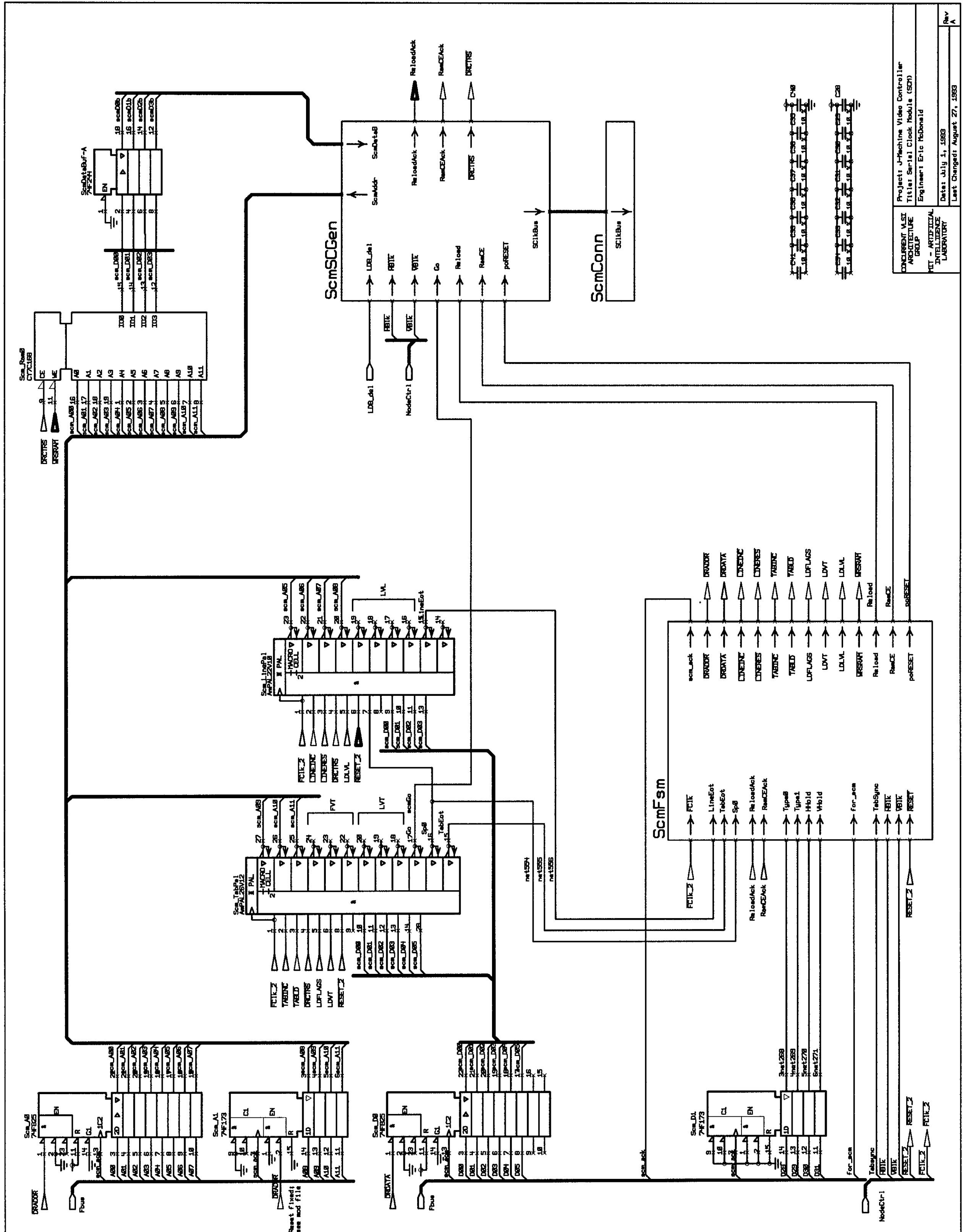
Title: XferConn (XferAddr Connector)

Engineer: Eric McDonald

Date: July 1, 1993

Last Changed:

Rev A


ScmSCGen
ScmConn
ScmFsm
Project: J-Machine Video Controller
Title: Serial Clock Module (SCM)
Engineer: Eric McDonald
Date: July 1, 1993
Last Changed: August 27, 1993
CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY

scmline.abl

```
module ScmLine

title 'SCM Line Counter FSM

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------
" This PAL implements a 4-bit counter, a 4-bit register, and a comparator
" between the pair.

ScmLine device 'P22V10';

"*** Inputs

FClk                    Pin 1;
LineInc                 Pin 2;          " Increment counter
LineRes                 Pin 3;          " Reset counter to zero
OE                      Pin 4;          " Output enable
LdLVL                   Pin 5;          " Load LVL register
GRESET                  Pin 6;          " Global /RESET line
Sp0                     Pin 7;          " Spare; unused
LvlD0,LvlD1,LvlD2,LvlD3 Pin 9,10,11,13; " Data inputs for LVL register


"*** Outputs

L0,L1,L2,L3             Pin 23,22,21,20 istype 'reg,buffer';
LVL0,LVL1,LVL2,LVL3     Pin 19,18,17,16 istype 'reg,buffer'; " Last Valid Line
NotAtEot                Pin 15 istype 'invert';          " Current cntr != LVL

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

Line = [L3..L0];
LVL = [LVL3..LVL0];

HOLD =  !LineInc & !LineRes;
INC  =   LineInc & !LineRes;
RESET = !LineInc &  LineRes;
IGNORE = LineInc &  LineRes;

"*** Equations

equations

[Line,LVL].clk = FClk;
Line.oe   = !OE & !LineInc & !LineRes;
Line.ar   = !GRESET;

" Reload LVL registers upon receipt of LdLVL signal from SCM0 Pal
LVL0 := (!LdLVL & LVL0) # (LdLVL & LvlD0);
LVL1 := (!LdLVL & LVL1) # (LdLVL & LvlD1);
LVL2 := (!LdLVL & LVL2) # (LdLVL & LvlD2);
LVL3 := (!LdLVL & LVL3) # (LdLVL & LvlD3);
```

scmline.abl

```
!L0 := RESET
     # (HOLD & !L0.fb)
     # (IGNORE & !L0.fb)
     # (INC   &  L0.fb);

!L1 := RESET
     # (HOLD & !L1.fb)
     # (IGNORE & !L1.fb)
     # (INC   & !L0.fb & !L1.fb)
     # (INC   &  L0.fb &  L1.fb);

!L2 := RESET
     # (HOLD & !L2.fb)
     # (IGNORE & !L2.fb)
     # (INC   & !L0.fb & !L2.fb)
     # (INC   & !L1.fb & !L2.fb)
     # (INC   &  L0.fb &  L1.fb & L2.fb);

!L3 := RESET
     # (HOLD & !L3.fb)
     # (IGNORE & !L3.fb)
     # (INC   & !L0.fb & !L3.fb)
     # (INC   & !L1.fb & !L3.fb)
     # (INC   & !L2.fb & !L3.fb)
     # (INC   &  L0.fb &  L1.fb & L2.fb & L3.fb);

" Raise Eot output when line counter == LVL
!NotAtEot = (Line.fb != LVL);

end ScmLine
```

scmtab.abl

```
module ScmTab

title 'SCM Table Counter FSM

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------

ScmTab device 'P26V12';

"*** Inputs

FClk                    Pin 1;
TabInc                  Pin 2;          " Increment table counter
TabLd                   Pin 3;          " Load FVT into table counter
OE                      Pin 4;          " Output enable for Tab[2:0]
LdFlags                 Pin 5;          " Load into FLAGS register
LdVT                    Pin 6;          " Load FVT and LVT registers
RESET                   Pin 8;          " Global /RESET line
LvtD0,LvtD1,LvtD2       Pin 10,11,12;   " Inputs for LVT
FvtD0,FvtD1,FvtD2       Pin 13,14,28;   " Inputs for FVT

"*** Outputs

Tab0,Tab1,Tab2          Pin 27,26,25 istype 'reg,buffer';  " Current table
FVT0,FVT1,FVT2          Pin 24,23,22 istype 'reg,buffer';  " First Valid Table
LVT0,LVT1,LVT2          Pin 20,19,18 istype 'reg,buffer';  " Last Valid Table
Go                      Pin 17 istype 'reg,buffer';        " ScmGo signal
Sp0                     Pin 16 istype 'reg,buffer';        " Spare flags
NotAtEot                Pin 15 istype 'invert';

"*** Bits used internally

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

LVT = [LVT2, LVT1, LVT0];
FVT = [FVT2, FVT1, FVT0];
Tab = [Tab2, Tab1, Tab0];
LvtData = [LvtD2, LvtD1, LvtD0];
FvtData = [FvtD2, FvtD1, FvtD0];

HOLD =  !TabInc & !TabLd;
INC  =   TabInc & !TabLd;
LOAD =  !TabInc &  TabLd;
IGNORE = TabInc &  TabLd;

"*** Equations

equations

[LVT,FVT,Tab,Go,Sp0].clk = FClk;

" Reload FVT and LVT registers upon receipt of LdVT signal from SCM0 Pal
```

scmtab.abl

```
FVT0 := (!LdVT & FVT0) # (LdVT & FvtD0);
FVT1 := (!LdVT & FVT1) # (LdVT & FvtD1);
FVT2 := (!LdVT & FVT2) # (LdVT & FvtD2);

LVT0 := (!LdVT & LVT0) # (LdVT & LvtD0);
LVT1 := (!LdVT & LVT1) # (LdVT & LvtD1);
LVT2 := (!LdVT & LVT2) # (LdVT & LvtD2);

" Reload FLAGS register upon receipt of LdFlags signal from SCM0 Pal
Go.ar = !RESET;
Go  := (!LdFlags & Go) # (LdFlags & LvtD0);
Sp0 := (!LdFlags & Sp0) # (LdFlags & LvtD1);

" The table counter can either hold, reload, or increment by one

Tab.ar = !RESET;
Tab.oe = !OE & !TabInc & !TabLd;

!Tab0 := (HOLD & !Tab0.fb)
       # (IGNORE & !Tab0.fb)
       # (LOAD & !FVT0)
       # (INC & Tab0.fb);

!Tab1 := (HOLD & !Tab1.fb)
       # (IGNORE & !Tab1.fb)
       # (LOAD & !FVT1)
       # (INC & !Tab0.fb & !Tab1.fb)
       # (INC &  Tab0.fb &  Tab1.fb);

!Tab2 := (HOLD & !Tab2.fb)
       # (IGNORE & !Tab2.fb)
       # (LOAD & !FVT2)
       # (INC & !Tab0.fb & !Tab2.fb)
       # (INC & !Tab1.fb & !Tab2.fb)
       # (INC &  Tab0.fb & Tab1.fb & Tab2.fb);


" Raise Eot output when table counter == LVT

!NotAtEot = (Tab.fb != LVT);

end ScmTab
```

scm0.abl

```
module Scm0

title 'Scm FSM PAL#0

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------

Scm0 device 'P26V12';

"*** Inputs

FClk                    Pin 1;
Eol, Eof                Pin 2, 3;       " Horiz and Vert syncs
HHold, VHold            Pin 4, 5;       " Should we sync up?
For_Scm                 Pin 6;          " FIFO has data for us
Type0, Type1            Pin 8, 9;       " Type of data
TabSync                 Pin 10;         "
LVT,LVL                 Pin 11, 12;     " Reached end of tables?
RamCEAck                Pin 13;         " ACK from SCCtl for RamCE request
ReloadAck               Pin 14;         " ACK from SCCtl for Reload request
poRESET                 Pin 28;         " Power-on/global RESET line
Sp0                     Pin 15;         " Unused spare

"*** Outputs

ERES                    Pin 27 istype 'reg,invert';         " Reset Eol, Eof
Q0,Q1,Q2,Q3,Q4          Pin 25,24,23,22,20 istype 'reg,buffer';  "State bits
Reload                  Pin 19 istype 'reg,buffer';  " Reload SCBlocks request
RamCE                   Pin 18 istype 'reg,buffer';  " Assert /RAMCE request
Scm_Ack                 Pin 17 istype 'reg,buffer';  " Latch & ACK FIFO data

"*** Bits used internally
Blocked                 Pin 26 istype 'reg,buffer'; " Data ready to be written?

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

Type =          [Type1,Type0];
WRITE_DATA       = [0, 0];
LOAD_TABLES      = [0, 1];
LOAD_LVL         = [1, 0];
LOAD_FLAGS       = [1, 1];

"*** State declarations
@include 'scm.sta'

"*** Equations

equations

[current_state,ERES,Reload,RamCE,Scm_Ack,Blocked].clk = FClk;
current_state.ar  = poRESET;
```

scm0.abl

```
State_Diagram current_state
        state Idle0:  " If end of line, reload SCBlocks
          if (Eol) then DoReload;
            else Idle1;
        state Idle1:  " If new data from FIFO, ack and process it
          if (For_Scm) then NewData0 with Scm_Ack := 1 endwith;
            else Idle0;

        state NewData0:             " Loop until we know FIFO has seen our ACK
          if (For_Scm) then NewData0 with Scm_Ack := 1 endwith;
             else NewData1;
        state NewData1:
          if (HHold # VHold # (Type == LOAD_FLAGS)) then LoopTilEol;
" We require an implicit HHold on LOAD_FLAGS to avoid race conditions
" with the Go flag in ScmCtl0
             else Dispatch;

        state LoopTilEol:
          Blocked := 1;
          if (Eol) then DoReload;
            else LoopTilEol;

        state DoReload:
          Blocked := Blocked;   " Remember if blocked
          Reload := 1;          " Ask SCCtl to do a reload
          if (!ReloadAck) then DoReload;    " Loop here until we get ReloadAck
             else if (Eof) then UpdateTab; " TAB and LINE get updated at Eof
               else UpdateLine;             " Just LINE gets updated at Eol
        state UpdateTab:
          Blocked := Blocked;
"         TabOp =  (LVT & LOAD) # (!LVT & INC);
"         LineOp = LOAD;
          goto UpdateDone;
        state UpdateLine:
          Blocked := Blocked;
"         LineOp = (LVL & LOAD) # (!LVL & INC);
          goto UpdateDone;
        state UpdateDone:
          !ERES := 1;
          if (!Blocked) then Idle0;
            else if (!VHold # Eof) then Dispatch;
              else LoopTilEol;

        state Dispatch:
          if (Type == WRITE_DATA) then WriteData0;
            else if (Type == LOAD_TABLES) then LoadTables;
              else if (Type == LOAD_LVL) then LoadLvl;
                else LoadFlags;

        state LoadTables:
"         !DRDATA := 1;          Drive FIFO data lines and ask TABPAL to
"         LDVT := 1;             latch in new tables
          goto Idle0;

        state LoadFlags:
"         !DRDATA := 1;          Drive FIFO data lines and ask TABPAL to
"         LDFLAGS := 1;          latch in new flags
```

scm0.abl

```
          goto Idle0;

        state LoadLvl:
"         !DRDATA := 1;          Drive FIFO data lines and ask LINEPAL to
"         LDLVL := 1;            latch in new LVL
          goto Idle0;

        state WriteData0:    " Wait here til SCCtl has seen our RamCE request
          RamCE := 1;
"         TabOp = IGNORE;       LineOp = IGNORE;
          if (!RamCEAck) then WriteData0;
            else WriteData1;
        state WriteData1:
          RamCE := 1;
"         TabOp = IGNORE;       LineOp = IGNORE;
"         !DRADDR := 1;          Enable address and data lines coming
"         !DRDATA := 1;          from FIFO registers...
          goto WriteData2;
        state WriteData2:
          RamCE := 1;
"         TabOp = IGNORE;       LineOp = IGNORE;
"         !DRADDR := 1;          Write into SRAM one cycle later
"         !DRDATA := 1;
"         !WRSRAM := 1;
          goto WriteData3;
        state WriteData3:
          RamCE := 1;
"         TabOp = IGNORE;       LineOp = IGNORE;
"         !DRADDR := 1;          And keep address/data valid for 1 more cycle
"         !DRDATA := 1;
          goto WriteData4;
        state WriteData4:
"         TabOp = IGNORE;  LineOp = IGNORE;
          if (RamCEAck) then WriteData4;  " Wait for SCCtl to drop RamCEAck
            else Idle0;

end Scm0
```

scm1.abl

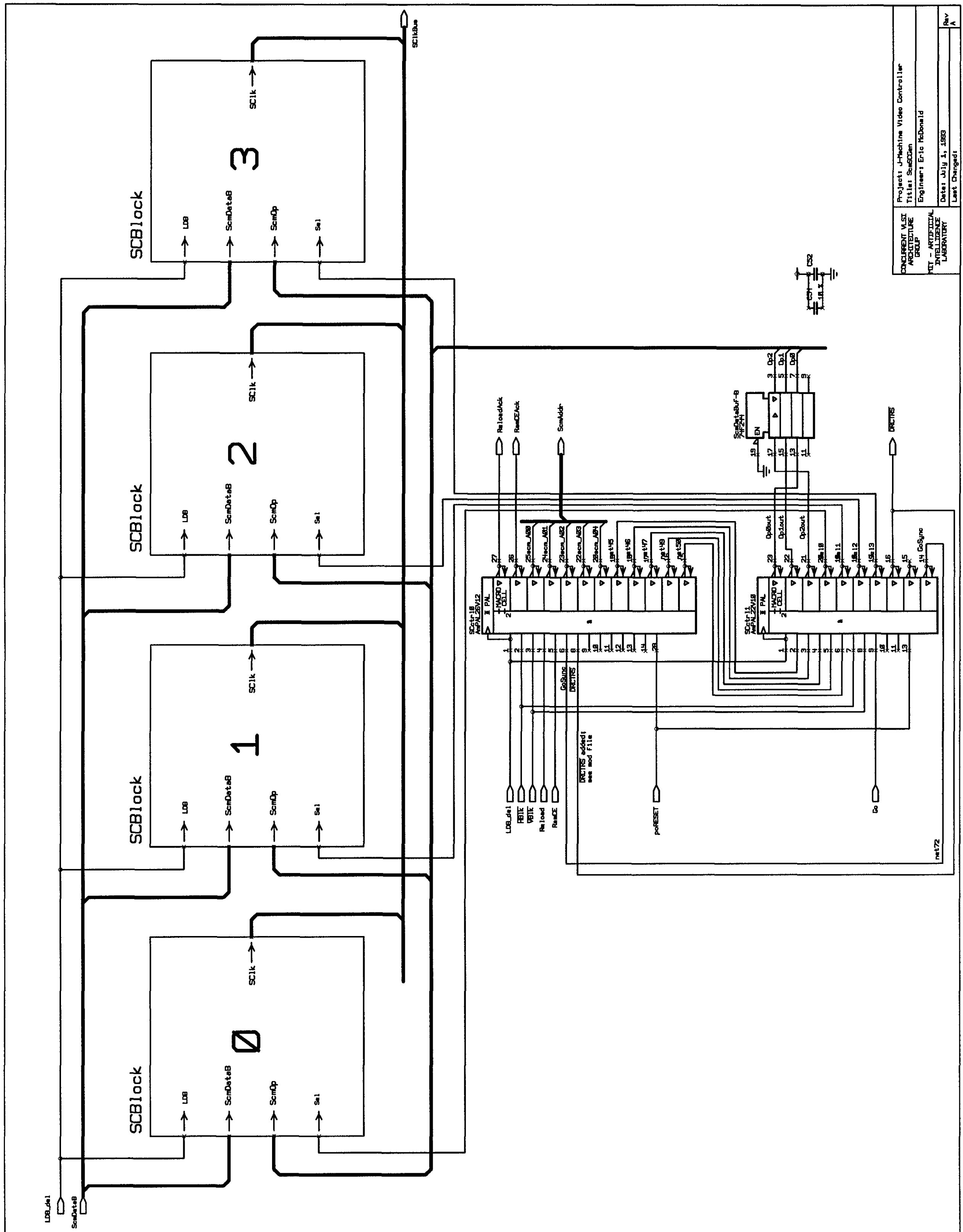
```
module Scm1

title 'Scm FSM PAL#1

  Eric McDonald     January   5, 1994
  Last revised:     February 15, 1994'

" DESCRIPTION:
" ------------

Scm1 device 'P22V10';

"*** Inputs

FClk                    Pin 1;
Q0,Q1,Q2,Q3,Q4          Pin 2,3,4,5,6;  " State bits
TabSync                 Pin 7;          "
LVT, LVL                Pin 8, 9;       " Reached end of tables?
poRESET                 Pin 10;         " Power-on/global RESET line
Sp0                     Pin 11;         " Unused spare

"*** Outputs

DRADDR,DRDATA           Pin 23,22 istype 'reg,invert';  " Drive registers
LINEINC,LINERES         Pin 21,20;                      " LINEPAL signals
TABINC,TABLD            Pin 19,18;                      " TABPAL signals
LDFLAGS,LDVT,LDLVL      Pin 17,16,15 istype 'reg,buffer';" More PAL load sigs
WRSRAM                  Pin 14 istype 'reg,invert';     " Ctrl signal to SRAM

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

TabOp =       [TABINC,  TABLD];
LineOp =      [LINEINC, LINERES];
INC =         [  1,       0  ];
LOAD  =       [  0,       1  ];
IGNORE =      [  1,       1  ];

"*** State declarations
@include 'scm.sta'

"*** Equations

equations

[DRDATA,DRADDR,LDFLAGS,LDVT,LDLVL,WRSRAM].clk = FClk;

!DRADDR := In_State(WriteData1)
         # In_State(WriteData2)
         # In_State(WriteData3);

!DRDATA := In_State(LoadTables)
         # In_State(LoadFlags)
         # In_State(LoadLvl)
         # In_State(WriteData1)
         # In_State(WriteData2)
```

scm1.abl

```
         # In_State(WriteData3);

LineOp = (In_State(UpdateTab) & LOAD)
       # ((In_State(UpdateLine) & LVL) & LOAD)
       # ((In_State(UpdateLine) & !LVL) & INC)
       # (In_State(WriteData0) & IGNORE)
       # (In_State(WriteData1) & IGNORE)
       # (In_State(WriteData2) & IGNORE)
       # (In_State(WriteData3) & IGNORE)
       # (In_State(WriteData4) & IGNORE);

TabOp = ((In_State(UpdateTab) & LVT) & LOAD)
      # ((In_State(UpdateTab) & !LVT) & INC)
      # (In_State(WriteData0) & IGNORE)
      # (In_State(WriteData1) & IGNORE)
      # (In_State(WriteData2) & IGNORE)
      # (In_State(WriteData3) & IGNORE)
      # (In_State(WriteData4) & IGNORE);

LDFLAGS := In_State(LoadFlags);

LDVT    := In_State(LoadTables);

LDLVL   := In_State(LoadLvl);

!WRSRAM := In_State(WriteData2);

end Scm1
```

scm.sta

```
"*** State declarations
" These pairs require one-bit state bit differences:
"    Idle->NewData0
"    NewData0 -> NewData1
"    LoopTilEnd -> Dispatch
"    SendErm2, SendErm3 (from SendErm1)
"    SendErm3 -> SendErm4

Idle        = [ 0, 0, 0, 0, 0,    0, 0 ]; "00-0 (00)
NewData0    = [ 0, 0, 0, 1, 0,    0, 0 ]; "02-0 (08)
NewData1    = [ 0, 0, 0, 1, 1,    0, 0 ]; "03-0 (0c)
LoopTilEnd  = [ 0, 0, 0, 0, 1,    0, 0 ]; "01-0 (04)
Dispatch    = [ 0, 0, 0, 0, 1,    0, 1 ]; "01-1 (05)

WriteData0a = [ 0, 0, 1, 0, 0,    0, 0 ]; "04-0 (10)
WriteData0b = [ 0, 0, 1, 0, 0,    0, 1 ]; "04-1 (11)
WriteData0c = [ 0, 0, 1, 0, 0,    1, 1 ]; "04-3 (13)

WriteData1a = [ 0, 1, 1, 0, 0,    0, 0 ]; "06-0 (30)
WriteData1b = [ 0, 1, 1, 0, 0,    0, 1 ]; "06-1 (31)
WriteData1c = [ 0, 1, 1, 0, 0,    1, 1 ]; "06-3 (32)

WriteData2a = [ 0, 1, 0, 0, 0,    0, 0 ]; "08-0 (20)
WriteData2b = [ 0, 1, 0, 0, 0,    0, 1 ]; "08-1 (21)
WriteData2c = [ 0, 1, 0, 0, 0,    1, 1 ]; "08-3 (23)

ReadData0a  = [ 1, 0, 0, 0, 0,    0, 0 ]; "10-0 (40)
ReadData0b  = [ 1, 0, 0, 0, 1,    0, 1 ]; "11-1 (45)
ReadData0c  = [ 1, 0, 0, 0, 0,    0, 1 ]; "10-1 (41)
ReadData0d  = [ 1, 0, 0, 0, 0,    1, 1 ]; "10-3 (43)

ReadData1a  = [ 1, 0, 0, 1, 0,    0, 0 ]; "12-0 (48)
ReadData1b  = [ 1, 0, 0, 1, 1,    0, 1 ]; "13-1 (4d)
ReadData1c  = [ 1, 0, 0, 1, 0,    0, 1 ]; "12-1 (49)
ReadData1d  = [ 1, 0, 0, 1, 0,    1, 1 ]; "12-3 (4b)

ReadData2a  = [ 1, 0, 1, 1, 0,    0, 0 ]; "16-0 (58)
ReadData2b  = [ 1, 0, 1, 1, 1,    0, 1 ]; "17-1 (5d)
ReadData2c  = [ 1, 0, 1, 1, 0,    0, 1 ]; "16-1 (59)
ReadData2d  = [ 1, 0, 1, 1, 0,    1, 1 ]; "16-3 (5b)

SendToErm0  = [ 1, 1, 0, 0, 0,    0, 0 ]; "18-0 (60)
SendToErm1  = [ 1, 1, 0, 0, 1,    0, 0 ]; "19-0 (64)
SendToErm2  = [ 1, 1, 0, 1, 1,    0, 0 ]; "1b-0 (6b)
SendToErm3  = [ 1, 1, 1, 1, 1,    0, 0 ]; "1f-0 (7b)
SendToErm4  = [ 1, 1, 1, 1, 1,    0, 1 ]; "1f-1 (7d)
```


SCBlock
SCBlock
SCBlock
SCBlock
Ø
1
2
3
SClk
LDB
ScmDataB
ScmOp
Sel
SClkBus
Project: J-Machine Video Controller
Title: ScmSCGen
Engineer: Eric McDonald
Date: July 1, 1993
Last Changed:
CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY
Rev A

scmctl0.abl

```
module ScmCtl0

title 'SCCtrl FSM PAL#0

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------

ScmCtl0 device 'P26V12';

"*** Inputs

Clk                      Pin 1;
HBlk,VBlk                Pin 2, 3;        " Horiz and Vert blanking
Reload                   Pin 4;           " Scm request to Reload
RamCE                    Pin 5;           " Scm request to enable /DRCTRS
Go                       Pin 6;           " Sync'd Go signal
DRCTRS                   Pin 8;           " Drive ADDR bits only when !DRCTRS
GReset                   Pin 28;          " Reset signal

"*** Outputs

ReloadAck                Pin 27 istype 'reg,buffer';  " ACK for Reload request
RamCEAck                 Pin 26 istype 'reg,buffer';  " ACK for RamCE request
A0,A1,A2,A3,A4           Pin 25,24,23,22,20 istype 'reg,buffer'; " RAM address
Q0,Q1,Q2,Q3,Q4           Pin 19,18,17,16,15 istype 'reg,buffer'; " State bits

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

"*** State declarations
@include 'scmctl.sta'

"*** RAM Addresses
Addr =          [A3,A2,A1,A0];
Setup_N01_N00 = [ 0, 0, 0, 0 ];
Setup_N03_N02 = [ 0, 0, 0, 1 ];
Setup_N05_N04 = [ 0, 0, 1, 0 ];
Setup_N07_N06 = [ 0, 0, 1, 1 ];
Setup_N09_N08 = [ 0, 1, 0, 0 ];
Setup_N11_N10 = [ 0, 1, 0, 1 ];
Setup_N13_N12 = [ 0, 1, 1, 0 ];
Setup_N15_N14 = [ 0, 1, 1, 1 ];
DFB8_Flag     = [ 1, 0, 0, 0 ];

"*** Equations

equations

[current_state,ReloadAck,RamCEAck,Addr].clk = Clk;
[current_state,ReloadAck,RamCEAck].ar = GReset;
A4 = 0;
Addr.oe = !DRCTRS & !RamCEAck;
```

scmctl0.abl

```
State_Diagram current_state

" NOTE: Reload, RamCE, and Go are guaranteed (by Scm0) not to go high
"     close to each other.  In addition, none of them will occur close
"     to VBlk or HBlk going high.

" If we're not in Go condition, loop in Idle looking
" for Reload and RamCE requests or a Go condition
        state Idle:
"         op = ((GReset == 1) & CLEAR_REG);
          if (Reload) then Rel0;
            else if (RamCE) then CE0;
              else if (Go) then AwaitVBlanking;
                else Idle;

        state CE0:
"         !DRCTRS := 1;
          RamCEAck := 1;
          if (RamCE) then CE0;
            else CE1;
        state CE1:
          goto HBlanking0;      " If we actually came here from Idle, we'll
                                " get back to Idle after a hop through
                                " HBlanking0 and HBlanking1

" When we first see a Go condition, we wait until Eof before doing
" anything
        state AwaitVBlanking:
          if (VBlk) then AwaitVBlanking;
            else HBlanking0;

" We wait here during HBlanking periods, servicing any Reload or RamCE
" requests while we wait for the next Active period
        state HBlanking0:
          if (!Go # Reload # RamCE) then HBlanking1;
            else if (HBlk & VBlk) then Active0;
                else HBlanking0;
" ONLY enter this state if we know !Go, Reload or RamCE is true
        state HBlanking1:
          if (Reload) then Rel0;
            else if (RamCE) then CE0;
             else Idle;

        state Active0:
"         Op = GO_INIT_STATE;
          goto Active1;
        state Active1:
"         Op = NEXT;
          if (!HBlk) then HBlanking0;
              else Active1;

        state Rel0:     " Enable /DRCTRS
"         !DRCTRS := 1;
          goto Rel0a;
        state Rel0a:    " Copy RAM [0] -> SCReg0
          Addr = Setup_N01_N00;
"         !DRCTRS := 1;
```

scmctl0.abl

```
"         Sel0 = 1;
"         Op = LOAD_SETUP;
          goto Rel0b;
        state Rel0b:     " Copy RAM [1] -> SCReg0
          Addr = Setup_N03_N02;
"         !DRCTRS := 1;
"         Sel0 = 1;
"         Op = LOAD_SETUP;
          goto Rel1a;
        state Rel1a:     " Copy RAM [2] -> SCReg1
          Addr = Setup_N05_N04;
"         !DRCTRS := 1;
"         Sel1 = 1;
"         Op = LOAD_SETUP;
          goto Rel1b;
        state Rel1b:     " Copy RAM [3] -> SCReg1
          Addr = Setup_N07_N06;
"         !DRCTRS := 1;
"         Sel1 = 1;
"         Op = LOAD_SETUP;
          goto Rel2a;
        state Rel2a:     " Copy RAM [4] -> SCReg2
          Addr = Setup_N09_N08;
"         !DRCTRS := 1;
"         Sel2 = 1;
"         Op = LOAD_SETUP;
          goto Rel2b;
        state Rel2b:     " Copy RAM [5] -> SCReg2
          Addr = Setup_N11_N10;
"         !DRCTRS := 1;
"         Sel2 = 1;
"         Op = LOAD_SETUP;
          goto Rel3a;
        state Rel3a:     " Copy RAM [6] -> SCReg3
          Addr = Setup_N13_N12;
"         !DRCTRS := 1;
"         Sel3 = 1;
"         Op = LOAD_SETUP;
          goto Rel3b;
        state Rel3b:     " Copy RAM [7] -> SCReg3
          Addr = Setup_N15_N14;
"         !DRCTRS := 1;
"         Sel3 = 1;
"         Op = LOAD_SETUP;
          goto Rel4;
        state Rel4:      " Copy RAM [8] -> SCReg[0:3]
          Addr = DFB8_Flag;
"         Sel0 = 1;  Sel1 = 1; Sel2 = 1; Sel3 = 1;
"         Op = LOAD_DFB8;
          goto Rel5 with ReloadAck := 1; endwith;
        state Rel5:
          ReloadAck := 1;
          if (Reload) then Rel5;
            else HBlanking0;

end ScmCtl0
```

scmctl1.abl

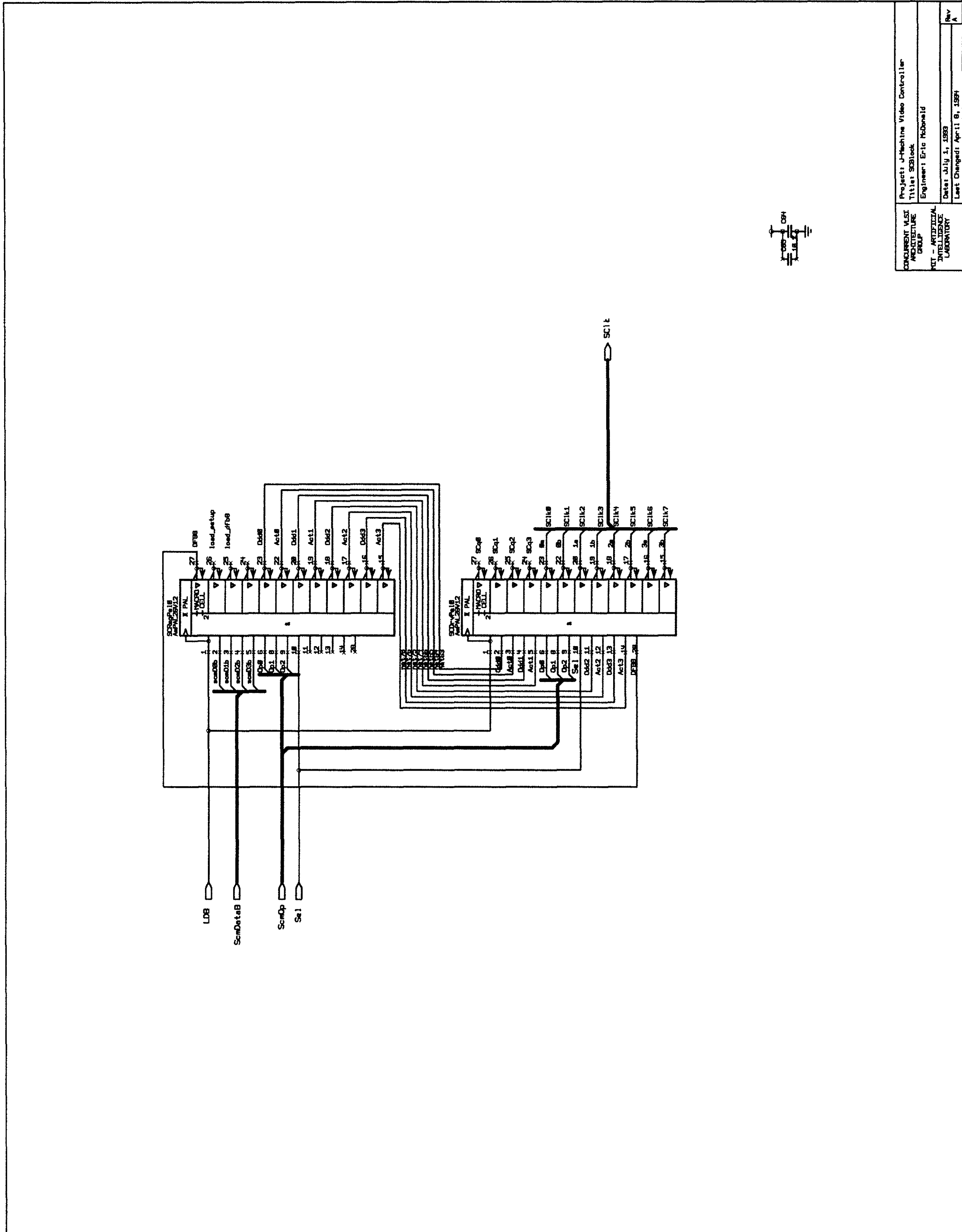
```
module ScmCtl1

title 'SCCtrl FSM PAL#1

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------

ScmCtl1 device 'P22V10';

"*** Inputs

Clk                      Pin 1;
Q0,Q1,Q2,Q3,Q4           Pin 2,3,4,5,6;  " State bits
HBlk,VBlk                Pin 7, 8;       " Horiz and Vert blanking
ASyncGo                  Pin 9;          " Asynchronous Go signal
GReset                   Pin 13;         " Reset signal

"*** Outputs

Op0,Op1,Op2              Pin 23,22,21;     " Op command
Sel0,Sel1,Sel2,Sel3      Pin 20,19,18,17; " Select SCBlock control lines
DRCTRS                   Pin 16 istype 'reg,invert';
Go                       Pin 14 istype 'reg,buffer';


"*** Aliases

H, L, C, X = 1, 0, .C., .X.;
In_State MACRO (st) { (current_state == ?st) };


"*** State declarations
@include 'scmctl.sta'

"*** Op command definitions (consistent with scmreg.abl and scmdrv.abl)
@include 'scmctl.ops'

"*** Equations

equations

[DRCTRS,Go].clk = Clk;
Go.ar           = GReset;

Go := ASyncGo;

Op = (In_State(Idle) & (GReset == 1) & CLEAR_REG)
   # (In_State(Active0) & GO_INIT_STATE)
   # (In_State(Active1) & NEXT)
   # (In_State(Rel0a) & LOAD_SETUP) # (In_State(Rel0b) & LOAD_SETUP)
   # (In_State(Rel1a) & LOAD_SETUP) # (In_State(Rel1b) & LOAD_SETUP)
   # (In_State(Rel2a) & LOAD_SETUP) # (In_State(Rel2b) & LOAD_SETUP)
   # (In_State(Rel3a) & LOAD_SETUP) # (In_State(Rel3b) & LOAD_SETUP)
   # (In_State(Rel4) & LOAD_DFB8);
```

scmctl1.abl

```
Sel0 = (In_State(Rel0a)) # (In_State(Rel0b))
     # In_State(Rel4);

Sel1 = (In_State(Rel1a)) # (In_State(Rel1b))
     # In_State(Rel4);

Sel2 = (In_State(Rel2a)) # (In_State(Rel2b))
     # In_State(Rel4);

Sel3 = (In_State(Rel3a)) # (In_State(Rel3b))
     # In_State(Rel4);

!DRCTRS := In_State(CE0)
         # In_State(Rel0)
         # In_State(Rel0a) # In_State(Rel0b)
         # In_State(Rel1a) # In_State(Rel1b)
         # In_State(Rel2a) # In_State(Rel2b)
         # In_State(Rel3a) # In_State(Rel3b);

end ScmCtl1
```

scmctl.sta

```
"*** State declarations

" States transitions that demand single-bit changes only:
"   Idle -> AwaitVBlanking, CE0, Rel0
"   CE0 -> CE1
"   AwaitVBlanking -> HBlanking0
"   HBlanking0 -> Active0, HBlanking1
"   Active1 -> HBlanking0;

current_state = [Q4,Q3,Q2,Q1,Q0];
Idle =          [ 0, 0, 0, 0, 0 ];      "00

AwaitVBlanking =[ 0, 0, 0, 1, 0 ];      "02

CE0  =          [ 0, 0, 1, 0, 0 ];      "04
CE1  =          [ 0, 0, 1, 0, 1 ];      "05

HBlanking0 =    [ 0, 0, 0, 1, 1 ];      "03
HBlanking1 =    [ 0, 0, 0, 0, 1 ];      "01

Active0 =       [ 0, 0, 1, 1, 1 ];      "07
Active1 =       [ 0, 1, 0, 1, 1 ];      "0b

Rel0 =          [ 1, 0, 0, 0, 0 ];      "10
Rel0a =         [ 1, 1, 0, 0, 0 ];      "18
Rel0b =         [ 1, 1, 0, 0, 1 ];      "19
Rel1a =         [ 1, 1, 0, 1, 0 ];      "1a
Rel1b =         [ 1, 1, 0, 1, 1 ];      "1b
Rel2a =         [ 1, 1, 1, 0, 0 ];      "1c
Rel2b =         [ 1, 1, 1, 0, 1 ];      "1d
Rel3a =         [ 1, 1, 1, 1, 0 ];      "1e
Rel3b =         [ 1, 1, 1, 1, 1 ];      "1f
Rel4 =          [ 1, 0, 0, 0, 1 ];      "11
Rel5 =          [ 1, 0, 0, 1, 0 ];      "12
```



scmreg.abl

```
module ScmReg

title 'ScmReg PAL

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------

ScmReg device 'P26CV12';

"*** Inputs

Clk                             Pin 1;
Data0,Data1,Data2,Data3         Pin 2,3,4,5;
Op0,Op1,Op2                     Pin 6,8,9;
Sel                             Pin 10;          " Select line

"*** Outputs
DFB8                           Pin 27 istype 'reg,buffer';
Odd0,Act0,Odd1,Act1,
  Odd2,Act2,Odd3,Act3          Pin 23,22,20,19,18,17,16,15 istype 'reg,buffer';

"*** For internal use
do_load_setup                  Pin 26;
do_load_dfb8                   Pin 25;
clear_reg                      Pin 24 istype 'reg,buffer';

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

"*** Op command definitions
@include 'scmctl.ops'

"*** Equations

equations

[Odd0..Odd3,Act0..Act3,DFB8,clear_reg].clk = Clk;
[Odd0..Odd3,Act0..Act3,DFB8].ar = clear_reg;

do_load_setup = Sel & (Op == LOAD_SETUP);
do_load_dfb8  = Sel & (Op == LOAD_DFB8);
clear_reg    :=       (Op == CLEAR_REG);

Odd0 := (do_load_setup & Data0) # (!do_load_setup & Odd0);
Act0 := (do_load_setup & Data1) # (!do_load_setup & Act0);
Odd1 := (do_load_setup & Data2) # (!do_load_setup & Odd1);
Act1 := (do_load_setup & Data3) # (!do_load_setup & Act1);

Odd2 := (do_load_setup & Odd0) # (!do_load_setup & Odd2);
Act2 := (do_load_setup & Act0) # (!do_load_setup & Act2);
Odd3 := (do_load_setup & Odd1) # (!do_load_setup & Odd3);
Act3 := (do_load_setup & Act1) # (!do_load_setup & Act3);
```

scmreg.abl

```
DFB8 := (do_load_dfb8 & Data0) # (!do_load_dfb8 & DFB8);

end ScmReg
```

scmdrv.abl

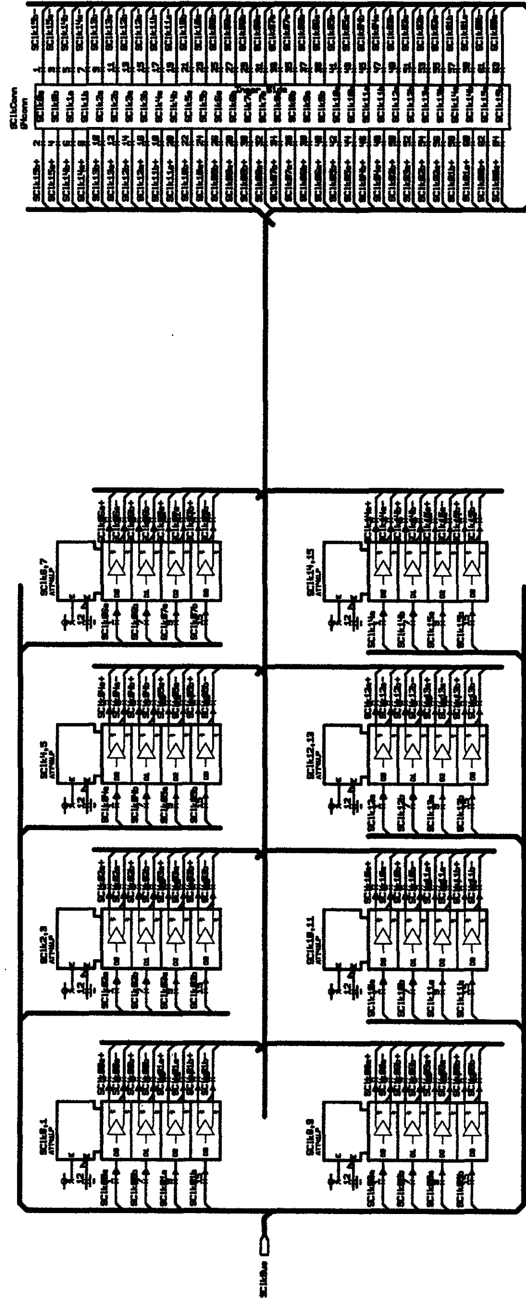
```
module ScmDrv

title 'ScmDrv PAL

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------

ScmDrv device 'P26CV12';

"*** Inputs

Clk                                Pin 1;
Odd0,Act0,Odd1,Act1,
  Odd2,Act2,Odd3,Act3              Pin 2,3,4,5,11,12,13,14;
Op0, Op1, Op2                      Pin 6,8,9;
Sel                                Pin 10;     " Select line
DFB8                               Pin 28;     " 8DFB flag

"*** Outputs

SCa0,SCb0,SCa1,SCb1,SCa2,SCb2,SCa3,SCb3 Pin 23,22,20,19,18,17,16,15
                                         istype 'reg,buffer'; " SClk signals
SCq0,SCq1,SCq2,SCq3                Pin 27,26,25,24 istype 'reg,invert';

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

"*** Op command definitions
@include 'scmctl.ops'

SClk0 = [SCq0,SCb0,SCa0];
SClk1 = [SCq1,SCb1,SCa1];
SClk2 = [SCq2,SCb2,SCa2];
SClk3 = [SCq3,SCb3,SCa3];

"*** State declarations
EnabA = [ 0, 0, 1 ];
Idle0 = [ 0, 0, 0 ];
EnabB = [ 0, 1, 0 ];
Idle1 = [ 1, 0, 0 ];
Trap4 = [ 0, 1, 1 ];
Trap5 = [ 1, 0, 1 ];
Trap6 = [ 1, 1, 0 ];
Trap7 = [ 1, 1, 1 ];

do_state_diag macro (SVar,Act,Odd) {
state_diagram ?SVar
state EnabA:
        if (?Act & (Op == NEXT)) then
          case !DFB8 : EnabB;
                DFB8 : Idle0;
          endcase;
        else if (?Act & (Op == GO_INIT_STATE)) then
```

scmdrv.abl

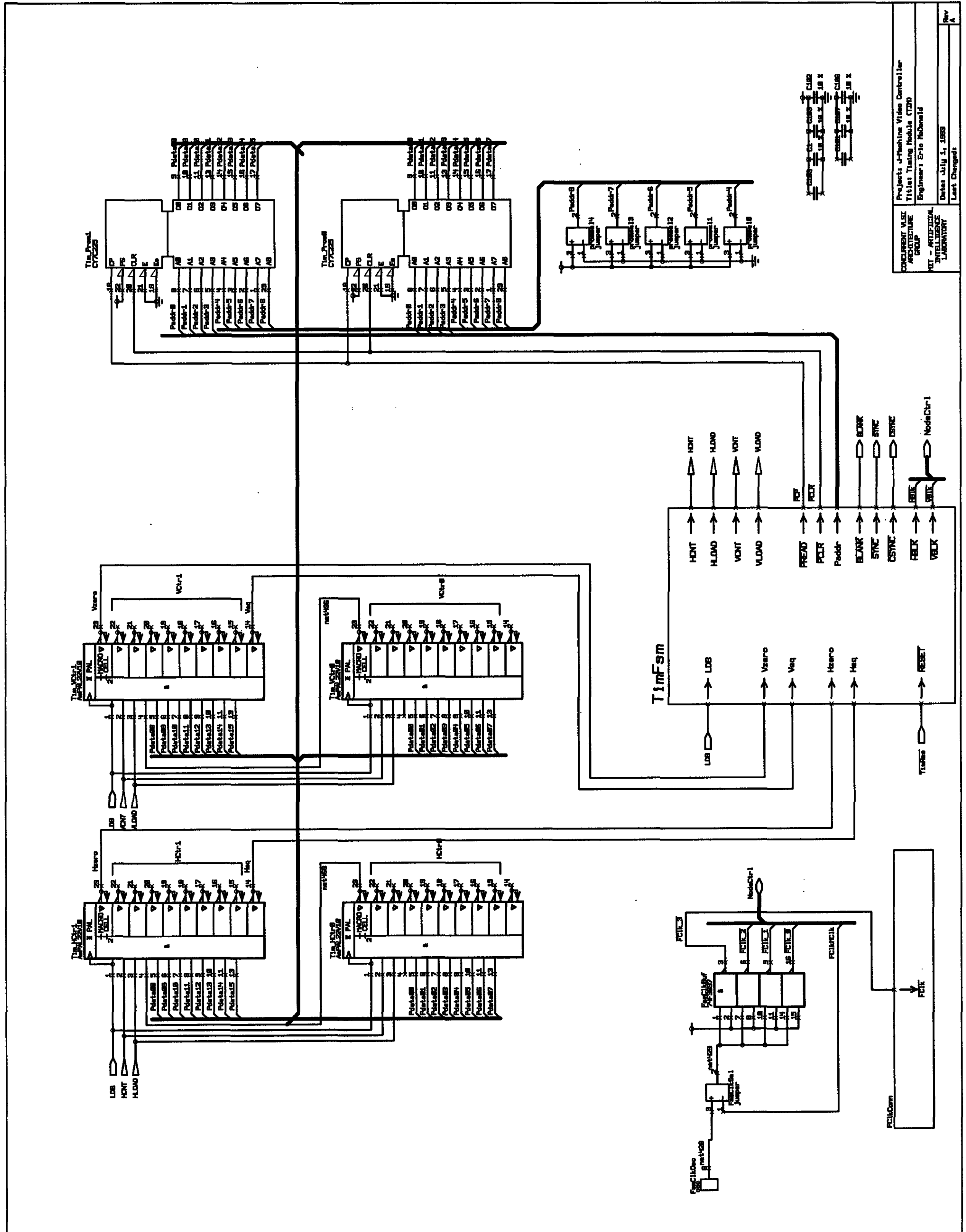
```
              case (!DFB8) : EnabA;
                   (DFB8 & !?Odd) : EnabA;
                   (DFB8 & ?Odd) : Idle0;
              endcase;
         else Idle0;
state Idle0:
         if (?Act & (Op == NEXT)) then
           case !DFB8 : EnabA;    " Should never reach here
                 DFB8 : EnabB;
           endcase;
         else if (?Act & (Op == GO_INIT_STATE)) then
              case (!DFB8) : EnabA;
                   (DFB8 & !?Odd) : EnabA;
                   (DFB8 & ?Odd) : Idle0;
              endcase;
         else Idle0;
state EnabB:
         if (?Act & (Op == NEXT)) then
           case !DFB8 : EnabA;
                 DFB8 : Idle1;
           endcase;
         else if (?Act & (Op == GO_INIT_STATE)) then
              case (!DFB8) : EnabA;
                   (DFB8 & !?Odd) : EnabA;
                   (DFB8 & ?Odd) : Idle0;
              endcase;
         else Idle0;
state Idle1:
         if (?Act & (Op == NEXT)) then
           case !DFB8 : EnabA;    " Should never reach here
                 DFB8 : EnabA;
           endcase;
         else if (?Act & (Op == GO_INIT_STATE)) then
              case (!DFB8) : EnabA;
                   (DFB8 & !?Odd) : EnabA;
                   (DFB8 & ?Odd) : Idle0;
              endcase;
         else Idle0;
state Trap4:
         goto Idle0;
state Trap5:
         goto Idle0;
state Trap6:
         goto Idle0;
state Trap7:
         goto Idle0;
}

"*** Equations

equations

[SClk0,SClk1,SClk2,SClk3].clk = Clk;

do_state_diag(SClk0,Act0,Odd0);

do_state_diag(SClk1,Act1,Odd1);
```

scmdrv.abl

```
do_state_diag(SClk2,Act2,Odd2);

do_state_diag(SClk3,Act3,Odd3);

end ScmDrv
```



Project: J-Machine Video Controller
Title: SclkConn - SClk Connector
Engineer: Eric McDonald
Date: July 1, 1993
Last Changed:
Rev A
CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY


TimFsm
HCNT
HLOAD
VCNT
VLOAD
BLANK
SYNC
CSYNC
RESET

timctr0.abl

```
module TimCtr0

title 'TIM module HCtr0/VCtr0 counter FSM

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------
" This PAL compares an 8-bit input value with an internal down counter.
" Asserts CO when counter == 0 and CI == 0,
" asserts Zero when counter == 0 regardless of CI.
" The counter can be decremented with Dec or loaded with Load.


TimCtr0 device 'P22V10';

"*** Inputs

Clock                   Pin 1;
Dec                     Pin 2;          " Decrement counter
Load                    Pin 3;          " Load counter with D inputs
@Ifdef HIGH_ORDER
{CI                     Pin 4;          " Carry bit input}
@Ifndef HIGH_ORDER
{CI = 1;                                " No carry bit input on low order ctr}

D0,D1,D2,D3,D4,D5,D6,D7 Pin 5,6,7,8,9,10,11,13; " Data inputs

"*** Outputs

Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7 Pin 22,15,21,16,20,17,19,18 istype 'reg,buffer';
CO                      Pin 23 istype 'reg,buffer';
Zero                    Pin 14;

"*** Equations

HOLD = !Dec & !Load;
DEC  =  Dec & !Load;
LOAD =  Load;

equations

[CO,Q7..Q0].clk = Clock;

!Q0 := (HOLD & !Q0)
     # (LOAD & !D0)
     # (DEC  & !CI & !Q0)
     # (DEC  &  CI &  Q0);

!Q1 := (HOLD & !Q1)
     # (LOAD & !D1)
     # (DEC  & !CI & !Q1)
     # (DEC  &  Q0 & !Q1)
     # (DEC  &  CI & !Q0 & Q1);

!Q2 := (HOLD & !Q2)
```

timctr0.abl

```
      # (LOAD & !D2)
      # (DEC  & !CI & !Q2)
      # (DEC  &  Q0 & !Q2)
      # (DEC  &  Q1 & !Q2)
      # (DEC  &  CI & !Q0 & !Q1 & Q2);

!Q3 := (HOLD & !Q3)
      # (LOAD & !D3)
      # (DEC  & !CI & !Q3)
      # (DEC  &  Q0 & !Q3)
      # (DEC  &  Q1 & !Q3)
      # (DEC  &  Q2 & !Q3)
      # (DEC  &  CI & !Q0 & !Q1 & !Q2 & Q3);

!Q4 := (HOLD & !Q4)
      # (LOAD & !D4)
      # (DEC  & !CI & !Q4)
      # (DEC  &  Q0 & !Q4)
      # (DEC  &  Q1 & !Q4)
      # (DEC  &  Q2 & !Q4)
      # (DEC  &  Q3 & !Q4)
      # (DEC  &  CI & !Q0 & !Q1 & !Q2 & !Q3 & Q4);

!Q5 := (HOLD & !Q5)
      # (LOAD & !D5)
      # (DEC  & !CI & !Q5)
      # (DEC  &  Q0 & !Q5)
      # (DEC  &  Q1 & !Q5)
      # (DEC  &  Q2 & !Q5)
      # (DEC  &  Q3 & !Q5)
      # (DEC  &  Q4 & !Q5)
      # (DEC  &  CI & !Q0 & !Q1 & !Q2 & !Q3 & !Q4 & Q5);

!Q6 := (HOLD & !Q6)
      # (LOAD & !D6)
      # (DEC  & !CI & !Q6)
      # (DEC  &  Q0 & !Q6)
      # (DEC  &  Q1 & !Q6)
      # (DEC  &  Q2 & !Q6)
      # (DEC  &  Q3 & !Q6)
      # (DEC  &  Q4 & !Q6)
      # (DEC  &  Q5 & !Q6)
      # (DEC  &  CI & !Q0 & !Q1 & !Q2 & !Q3 & !Q4 & !Q5 & Q6);

!Q7 := (HOLD & !Q7)
      # (LOAD & !D7)
      # (DEC  & !CI & !Q7)
      # (DEC  &  Q0 & !Q7)
      # (DEC  &  Q1 & !Q7)
      # (DEC  &  Q2 & !Q7)
      # (DEC  &  Q3 & !Q7)
      # (DEC  &  Q4 & !Q7)
      # (DEC  &  Q5 & !Q7)
      # (DEC  &  Q6 & !Q7)
      # (DEC  &  CI & !Q0 & !Q1 & !Q2 & !Q3 & !Q4 & !Q5 & !Q6 & Q7);

CO  :=  CI & !Q0 & !Q1 & !Q2 & !Q3 & !Q4 & !Q5 & !Q6 & !Q7;
```

timctr0.abl

```
Zero =        !Q0 & !Q1 & !Q2 & !Q3 & !Q4 & !Q5 & !Q6 & !Q7;

end TimCtr0
```

timctr1.abl

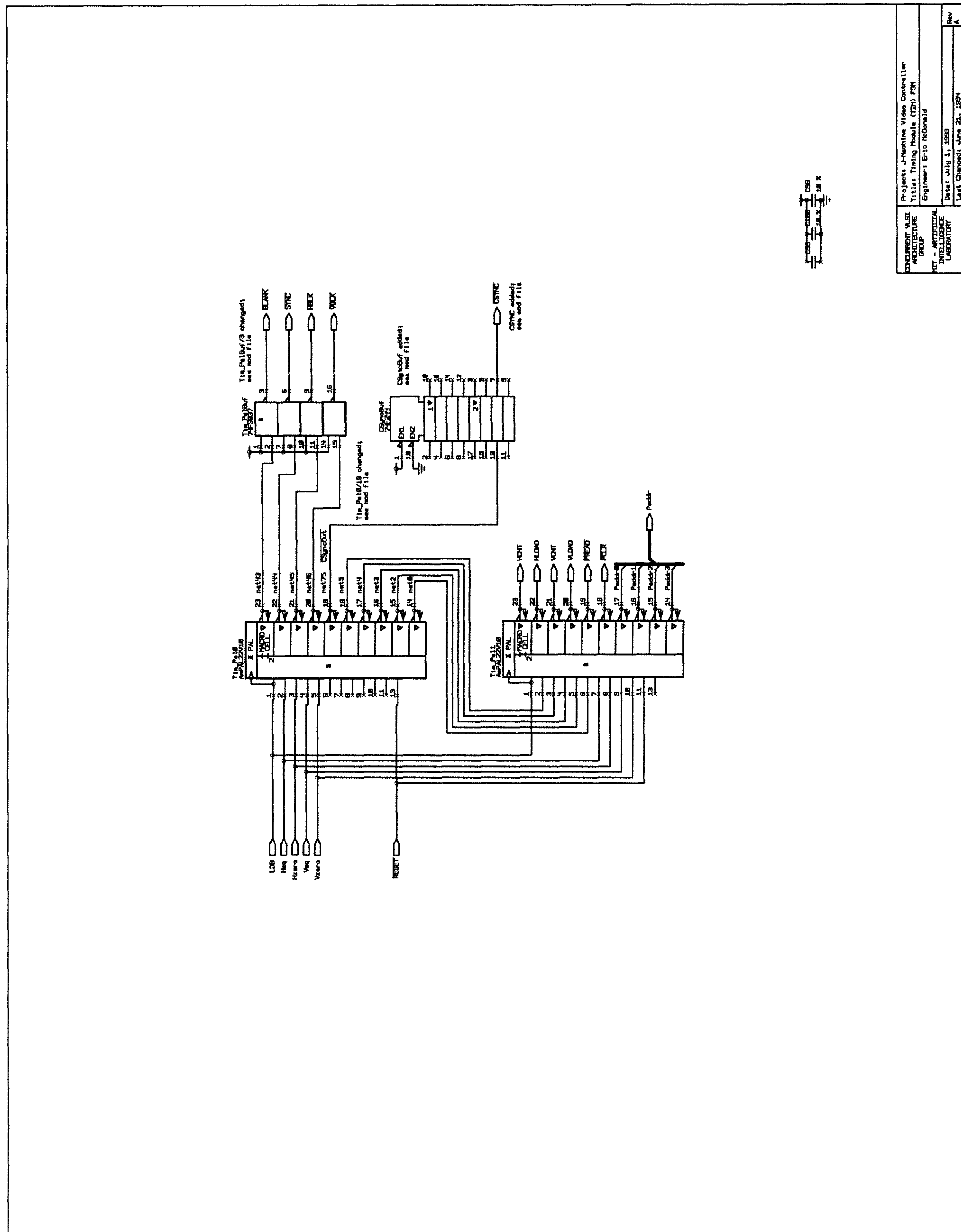
```
module TimCtr1

@const HIGH_ORDER = 1;

title 'TIM module HCtr1/VCtr1 counter FSM

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------
" This PAL compares an 8-bit input value with an internal down counter.
" Asserts CO when counter == 0 and CI == 0,
" asserts Zero when counter == 0 regardless of CI.
" The counter can be decremented with Dec or loaded with Load.


TimCtr1 device 'P22V10';

"*** Inputs

Clock                   Pin 1;
Dec                     Pin 2;          " Decrement counter
Load                    Pin 3;          " Load counter with D inputs
@Ifdef HIGH_ORDER
{CI                     Pin 4;          " Carry bit input}
@Ifndef HIGH_ORDER
{CI = 1;                                " No carry bit input on low order ctr}

D0,D1,D2,D3,D4,D5,D6,D7 Pin 5,6,7,8,9,10,11,13; " Data inputs

"*** Outputs

Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7 Pin 22,15,21,16,20,17,19,18 istype 'reg,buffer';
CO                      Pin 23 istype 'reg,buffer';
Zero                    Pin 14;

"*** Equations

HOLD = !Dec & !Load;
DEC  =  Dec & !Load;
LOAD =  Load;

equations

[CO,Q7..Q0].clk = Clock;

!Q0 := (HOLD & !Q0)
     # (LOAD & !D0)
     # (DEC  & !CI & !Q0)
     # (DEC  &  CI &  Q0);

!Q1 := (HOLD & !Q1)
     # (LOAD & !D1)
     # (DEC  & !CI & !Q1)
     # (DEC  &  Q0 & !Q1)
     # (DEC  &  CI & !Q0 & Q1);
```

timctr1.abl

```
!Q2 := (HOLD & !Q2)
     # (LOAD & !D2)
     # (DEC  & !CI & !Q2)
     # (DEC  &  Q0 & !Q2)
     # (DEC  &  Q1 & !Q2)
     # (DEC  &  CI & !Q0 & !Q1 & Q2);

!Q3 := (HOLD & !Q3)
     # (LOAD & !D3)
     # (DEC  & !CI & !Q3)
     # (DEC  &  Q0 & !Q3)
     # (DEC  &  Q1 & !Q3)
     # (DEC  &  Q2 & !Q3)
     # (DEC  &  CI & !Q0 & !Q1 & !Q2 & Q3);

!Q4 := (HOLD & !Q4)
     # (LOAD & !D4)
     # (DEC  & !CI & !Q4)
     # (DEC  &  Q0 & !Q4)
     # (DEC  &  Q1 & !Q4)
     # (DEC  &  Q2 & !Q4)
     # (DEC  &  Q3 & !Q4)
     # (DEC  &  CI & !Q0 & !Q1 & !Q2 & !Q3 & Q4);

!Q5 := (HOLD & !Q5)
     # (LOAD & !D5)
     # (DEC  & !CI & !Q5)
     # (DEC  &  Q0 & !Q5)
     # (DEC  &  Q1 & !Q5)
     # (DEC  &  Q2 & !Q5)
     # (DEC  &  Q3 & !Q5)
     # (DEC  &  Q4 & !Q5)
     # (DEC  &  CI & !Q0 & !Q1 & !Q2 & !Q3 & !Q4 & Q5);

!Q6 := (HOLD & !Q6)
     # (LOAD & !D6)
     # (DEC  & !CI & !Q6)
     # (DEC  &  Q0 & !Q6)
     # (DEC  &  Q1 & !Q6)
     # (DEC  &  Q2 & !Q6)
     # (DEC  &  Q3 & !Q6)
     # (DEC  &  Q4 & !Q6)
     # (DEC  &  Q5 & !Q6)
     # (DEC  &  CI & !Q0 & !Q1 & !Q2 & !Q3 & !Q4 & !Q5 & Q6);

!Q7 := (HOLD & !Q7)
     # (LOAD & !D7)
     # (DEC  & !CI & !Q7)
     # (DEC  &  Q0 & !Q7)
     # (DEC  &  Q1 & !Q7)
     # (DEC  &  Q2 & !Q7)
     # (DEC  &  Q3 & !Q7)
     # (DEC  &  Q4 & !Q7)
     # (DEC  &  Q5 & !Q7)
     # (DEC  &  Q6 & !Q7)
     # (DEC  &  CI & !Q0 & !Q1 & !Q2 & !Q3 & !Q4 & !Q5 & !Q6 & Q7);
```

timctr1.abl

```
CO   :=  CI & !Q0 & !Q1 & !Q2 & !Q3 & !Q4 & !Q5 & !Q6 & !Q7;
Zero =        !Q0 & !Q1 & !Q2 & !Q3 & !Q4 & !Q5 & !Q6 & !Q7;


end TimCtr1
```



tim40.abl

```
module Tim40

title 'TIM Module FSM #0

  Eric McDonald    January   5, 1994
  Last revised:    June 6, 1994'

" DESCRIPTION:
" ------------

Tim40 device 'P22V10';

"*** Inputs

FClk                    Pin 1;      " LdClk from Bt440
HZero                   Pin 3;      " HCtr is 0
VZero                   Pin 5;      " VCtr is 0
RESET                   Pin 13;     " Global /RESET line

"*** Outputs

BLANK                   Pin 23 istype 'reg,buffer';
SYNC                    Pin 22 istype 'reg,buffer';
HBLK,VBLK               Pin 21,20 istype 'reg,buffer';
" All of the above outputs are inverted outside of this PAL, so we
" are writing for the complements
CSYNC4                  Pin 19 istype 'reg,invert';
Q0,Q1,Q2,Q3,Q4          Pin 18,17,16,15,14 istype 'reg,buffer'; "State

"*** State declarations
@include 'tim4.sta'

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;

"*** Equations

equations

current_state.clk = FClk;
current_state.ar  = !RESET;

SYNC := L;

State_Diagram current_state
        state Init:
"               PAddr := HLinesInVFP;
                goto VertFrontPorch0;
"                       with VLoad = H;         HLinesInVFP -> VCtr

        state HorizActive:
"               PAddr := HFrontPorch;
"               HDec = H;
                if (HZero) then HorizFrontPorch;
"                       with HLoad = H;         HFrontPorch -> HCtr
"                            VDec = H;
```

tim40.abl

```
                else HorizActive;

        state HorizFrontPorch:
                BLANK := H;  HBLK := H;
"               HDec = H;
"               PAddr := HSync;
                if (HZero) then HorizSync;
"                       with HLoad = H;         HSync -> HCtr
                else HorizFrontPorch;


        state HorizSync:
                BLANK := H;  HBLK := H;  !CSYNC4 := H;
"               HDec = H;
                if (HZero) then
                        if (VZero) then VertFrontPorch0;
"                               with VLoad = H;     HLinesInVFP -> VCtr
"                                    PAddr := HActiveInVBlk;
                          else HorizBackPorch;
"                               with HLoad = H;     HBackPorch -> HCtr
                else HorizSync;
"                 with PAddr := (VZero & HLinesInVFP) # (!VZero & HBackPorch);

        state HorizBackPorch:
                BLANK := H;  HBLK := H;
"               HDec = H;
"               PAddr := HActive;
                if (HZero) then HorizActive;
"                       with HLoad = H;         HActive -> HCtr
                else HorizBackPorch;

        state VertFrontPorch0:
"               [wait state for registered PROM with registered addr inputs]
                BLANK := H;  VBLK := H;
                goto VertFrontPorch1;
        state VertFrontPorch1:
                BLANK := H;  VBLK := H;
                goto VertFrontPorch2;
"                       with HLoad = H;         HActiveInVBlk -> HCtr
        state VertFrontPorch2:
                BLANK := H;  VBLK := H;
"               HDec = H;
"               PAddr := HSync;
                if (HZero) then HorizSyncInVFP;
"                       with HLoad = H;         HSync -> HCtr
"                            VDec = H;
                else VertFrontPorch2;

        state HorizSyncInVFP:
                BLANK := H; !CSYNC4 := H; HBLK := H; VBLK := H;
"               HDec = H;
                if (HZero) then
                        if (VZero) then VertSync0;
"                               with HLoad = H;   HActiveInVBlk -> HCtr
"                                    PAddr := HLinesInVS;
                        else VertFrontPorch2;
"                               with HLoad = H;   HActiveInVBlk -> HCtr
```

tim40.abl

```
                else HorizSyncInVFP;
"                  with PAddr := HActiveInVBlk;

        state VertSync0:
"               [wait state for registered PROM with registered addr inputs]
                BLANK := H; !CSYNC4 := H; VBLK := H;
                goto VertSync1;
        state VertSync1:
                BLANK := H; !CSYNC4 := H; VBLK := H;
                goto VertSync2;
"                       with VLoad = H;         HLinesInVS -> VCtr
        state VertSync2:
                BLANK := H; !CSYNC4 := H; VBLK := H;
"               PAddr := HSync;
"               HDec = H;
                if (HZero) then HorizSyncInVS;
"                        with HLoad = H;        HSync -> HCtr
"                              VDec = H;
                else VertSync2;

        state HorizSyncInVS:
                BLANK := H;  VBLK := H;  HBLK := H;
"               HDec = H;
                if (HZero) then
                        if (VZero) then VertBackPorch0;
"                               with VLoad = H;   HLinesInVBP -> VCtr
"                                    PAddr := HSync;
                        else VertSync2;
"                               with HLoad = H;   HActiveInVBlk -> HCtr
                else HorizSyncInVS;
"                 with PAddr:=(VZero & HLinesInVBP) # (!VZero & HActiveInVBlk);

        state VertBackPorch0:
"               [wait state for registered PROM with registered addr inputs]
                BLANK := H;  VBLK := H;
                goto VertBackPorch1;
        state VertBackPorch1:
                BLANK := H;  VBLK := H;
                goto HorizSyncInVBP0;
"                       with HLoad = H;         HSync -> HCtr

        state VertBackPorch2:
                BLANK := H; VBLK := H;
"               PAddr := HSync;
"               HDec = H;
                if (HZero) then HorizSyncInVBP0;
"                        with HLoad = H;        HSync -> HCtr
"                             VDec = H;
                else VertBackPorch2;

        state HorizSyncInVBP0:
                BLANK := H;  !CSYNC4 := H; VBLK := H; HBLK := H;
"               HDec = H;
                if (HZero) then
                        if (VZero) then HorizSyncInVBP1;
"                               with VLoad = H;   HLinesActive -> VCtr
"                                    PAddr := HBPaVBP;
```

tim40.abl

```
                        else VertBackPorch2;
"                               with HLoad = H;   HActiveInVBlk -> HCtr
                else HorizSyncInVBP0;
"                 with PAddr:=(VZero & HLinesActive)#(!VZero & HActiveInVBlk);

        state HorizSyncInVBP1:
"               [wait state for registered PROM with registered addr inputs]
                BLANK := H;  !CSYNC4 := H; VBLK := H; HBLK := H;
                goto HorizSyncInVBP2;
        state HorizSyncInVBP2:
                BLANK := H;  !CSYNC4 := H; VBLK := H; HBLK := H;
                goto HBPAfterVBP;
"                       with HLoad = H;         HBPaVBP -> HCtr

        state HBPAfterVBP:
                BLANK := H;  VBLK := H; HBLK := H;
"               HDec = H;
"               PAddr := HActive;
                if (HZero) then HorizActive;
"                       with HLoad = H;         HActive -> HCtr
                else HBPAfterVBP;

end Tim40
```

tim41.abl

```
module Tim41

title 'TIM Module FSM #1

  Eric McDonald    January   5, 1994
  Last revised:    February 15, 1994'

" DESCRIPTION:
" ------------

Tim41 device 'P22V10';

"*** Inputs

FClk                    Pin 1;      " LdClk from Bt440
Q0,Q1,Q2,Q3,Q4          Pin 2,3,4,5,6; " Current state
HZero                   Pin 8;      " HCtr is 0
VZero                   Pin 10;     " VCtr is 0
RESET                   Pin 11;     " Global /RESET line

"*** Outputs

HDec,HLoad              Pin 23,22; " Dec and Load signals to HCtr
VDec,VLoad              Pin 21,20; " Dec and Load signals to VCtr
PromClk, PClr           Pin 19,18 istype 'reg,invert'; " Ctrl sigs to PROM
PA0,PA1,PA2,PA3         Pin 17,16,15,14; " Address inputs to PROM

"*** State declarations
@include 'tim4.sta'

"*** Aliases

H, L, C, X = 1, 0, .C., .X.;
In_State MACRO (st) { (current_state == ?st) };

PAddr = [PA3..PA0];

" PROM addresses of parameters used by timing FSM

HFrontPorch    = [0, 0, 0, 0];
HSync          = [0, 0, 0, 1];
HBackPorch     = [0, 0, 1, 0];
HActive        = [0, 0, 1, 1];

HLinesActive   = [0, 1, 0, 0];
HLinesInVFP    = [0, 1, 0, 1];
HLinesInVBP    = [0, 1, 1, 0];

HActiveInVBlk = [0, 1, 1, 1];
HBPaVBP        = [1, 0, 0, 0];
HLinesInVS     = [1, 0, 0, 1];

"*** Equations

equations

PromClk = FClk;
```

tim41.abl

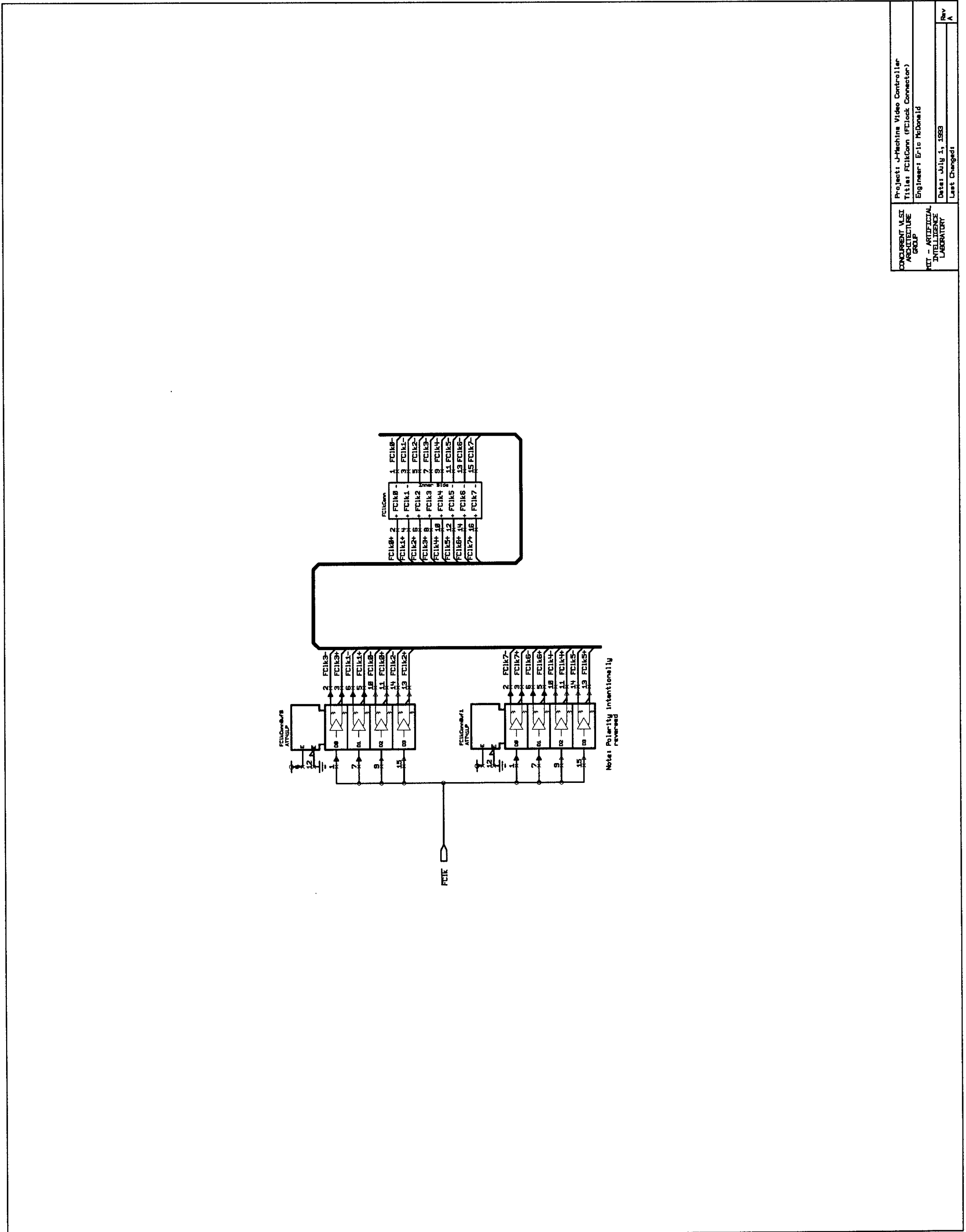
```
HDec = In_State(HorizActive)
     # In_State(HorizFrontPorch)
     # In_State(HorizSync)
     # In_State(HorizBackPorch)
     # In_State(VertFrontPorch2)
     # In_State(HorizSyncInVFP)
     # In_State(VertSync2)
     # In_State(HorizSyncInVS)
     # In_State(VertBackPorch2)
     # In_State(HorizSyncInVBP0)
     # In_State(HBPAfterVBP);

HLoad  = (In_State(HorizActive) & HZero)
       # (In_State(HorizFrontPorch) & HZero)
       # (In_State(HorizSync) & HZero & !VZero)
       # (In_State(HorizBackPorch) & HZero)
       # In_State(VertFrontPorch1)
       # (In_State(VertFrontPorch2) & HZero)
       # (In_State(HorizSyncInVFP) & HZero)
       # (In_State(VertSync2) & HZero)
       # (In_State(HorizSyncInVS) & HZero & !VZero)
       # In_State(VertBackPorch1)
       # (In_State(VertBackPorch2) & HZero)
       # (In_State(HorizSyncInVBP0) & HZero & !VZero)
       # In_State(HorizSyncInVBP2)
       # (In_State(HBPAfterVBP) & HZero);

VDec = (In_State(HorizActive) & HZero)
     # (In_State(VertFrontPorch2) & HZero)
     # (In_State(VertSync2) & HZero)
     # (In_State(VertBackPorch2) & HZero);

VLoad = In_State(Init)
      # (In_State(HorizSync) & HZero & VZero)
      # In_State(VertSync1)
      # (In_State(HorizSyncInVS) & HZero & VZero)
      # (In_State(HorizSyncInVBP0) & HZero & VZero);


!PClr := 0;

PAddr := (In_State(Init) & HLinesInVFP)
       # (In_State(HorizActive) & HFrontPorch)
       # (In_State(HorizFrontPorch) & HSync)
       # (In_State(HorizSync) & HZero & HActiveInVBlk)
       # (In_State(HorizSync) & !HZero & VZero & HLinesInVFP)
       # (In_State(HorizSync) & !HZero & !VZero & HBackPorch)
       # (In_State(HorizBackPorch) & HActive)
       # (In_State(VertFrontPorch2) & HSync)
       # (In_State(HorizSyncInVFP) & HZero & HLinesInVS)
       # (In_State(HorizSyncInVFP) & !HZero & HActiveInVBlk)
       # (In_State(VertSync2) & HSync)
       # (In_State(HorizSyncInVS) & HZero & HSync)
       # (In_State(HorizSyncInVS) & !HZero & VZero & HLinesInVBP)
       # (In_State(HorizSyncInVS) & !HZero & !VZero & HActiveInVBlk)
       # (In_State(VertBackPorch2) & HSync)
```

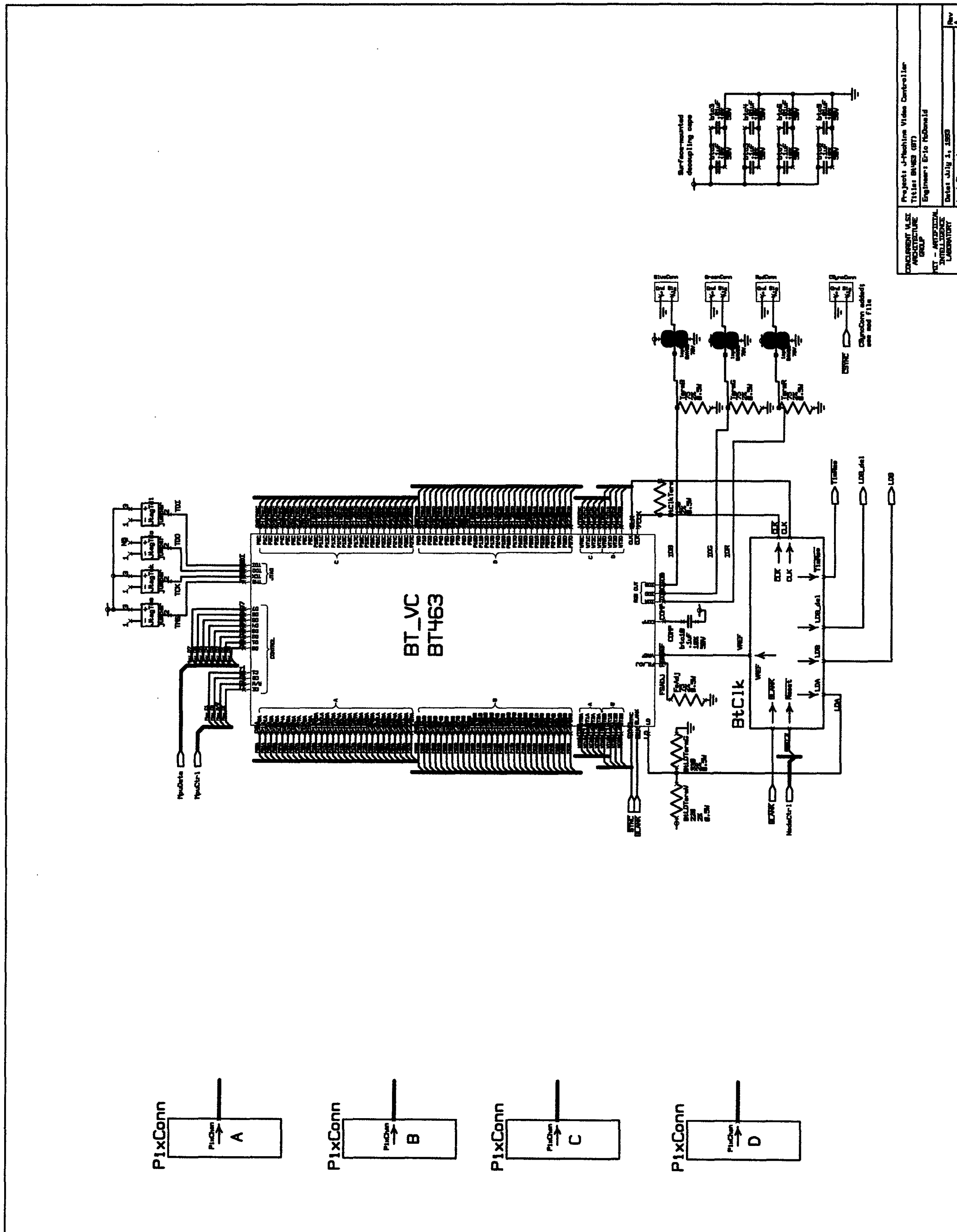
tim41.abl

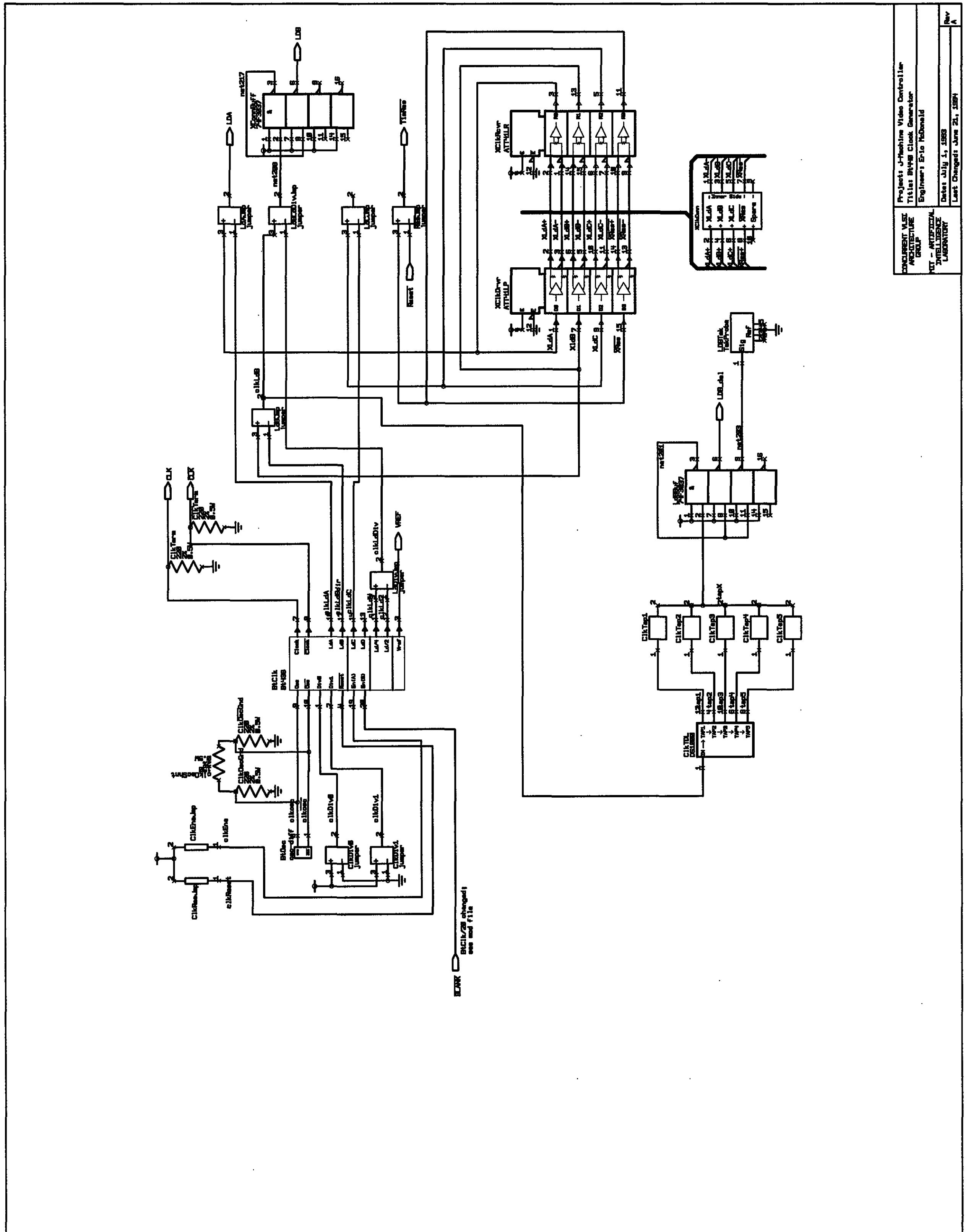
```
        # (In_State(HorizSyncInVBP0) & HZero & HBPaVBP)
        # (In_State(HorizSyncInVBP0) & !HZero & VZero & HLinesActive)
        # (In_State(HorizSyncInVBP0) & !HZero & !VZero & HActiveInVBlk)
        # (In_State(HBPAfterVBP) & HActive);

end Tim41
```

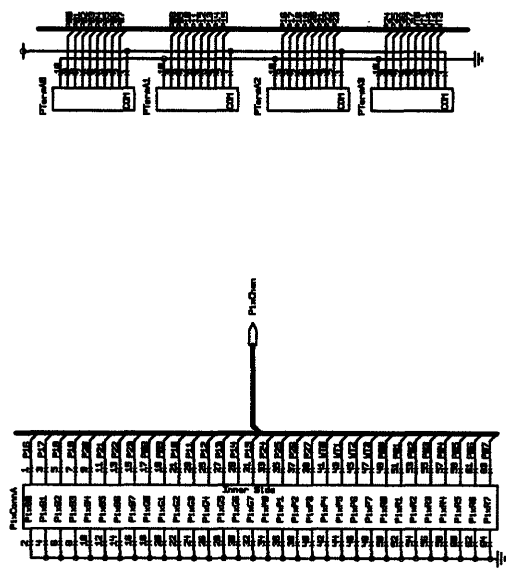
tim4.sta

```
current_state   = [Q4..Q0];

Init              = [ 0, 0, 0, 0, 0 ];   "0
HorizActive       = [ 0, 0, 0, 0, 1 ];   "1
HorizFrontPorch   = [ 0, 0, 0, 1, 1 ];   "3
HorizSync         = [ 0, 0, 0, 1, 0 ];   "2
HorizBackPorch    = [ 0, 0, 1, 1, 0 ];   "6

VertFrontPorch0   = [ 0, 1, 0, 1, 0 ];   "a
VertFrontPorch1   = [ 0, 1, 0, 0, 0 ];   "8
VertFrontPorch2   = [ 0, 1, 0, 0, 1 ];   "9
HorizSyncInVFP    = [ 0, 1, 1, 0, 1 ];   "d

VertSync0         = [ 0, 1, 1, 0, 0 ];   "c
VertSync1         = [ 1, 1, 1, 0, 0 ];   "1c
VertSync2         = [ 1, 1, 0, 0, 0 ];   "18
HorizSyncInVS     = [ 1, 1, 0, 0, 1 ];   "19

VertBackPorch0    = [ 1, 0, 1, 0, 0 ];   "14
VertBackPorch1    = [ 0, 0, 1, 0, 0 ];   "4
VertBackPorch2    = [ 0, 0, 1, 0, 1 ];   "5
HorizSyncInVBP0   = [ 0, 0, 1, 1, 1 ];   "7
HorizSyncInVBP1   = [ 1, 0, 1, 1, 1 ];   "17
HorizSyncInVBP2   = [ 1, 0, 1, 0, 1 ];   "15
HBPAfterVBP       = [ 1, 0, 1, 1, 0 ];   "16
```


CONCURRENT VLSI ARCHITECTURE GROUP
MIT - ARTIFICIAL INTELLIGENCE LABORATORY
Project: J-Machine Video Controller
Title: FClkConn (FClock Connector)
Engineer: Eric McDonald
Date: July 1, 1993
Last Changed:
Rev A
Note: Polarity intentionally reversed
FClk


BT_VC
BT463
BtClk
PixConn
A
PixConn
B
PixConn
C
PixConn
D





Bibliography

[1] K. Akeley and T. Jermoluk. High-Performance Polygon Rendering. *Computer Graphics*, 22(4):239–246, August 1988.

[2] Gary Bishop, Henry Fuchs, Leonard McMillan, and Ellen Scher Zagier. Frameless Rendering: Double Buffering Considered Harmful. Proceedings of SIGGRAPH '94. In *Computer Graphics* Proceedings, 1994, ACM SIGGRAPH. pages 175–176.

[3] Electronic Industries Association Engineering Dept. Electrical Performance Standards for High Resolution Monochrome Closed Circuit Television Camera. Technical Report EIA-343-A, Electronic Industries Association, September 1969.

[4] Jerko Fatovic. A Ray Tracer for the J-Machine. MS Thesis, Massachusetts Institute of Technology Department of Electrical Engineering, May 1992.

[5] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, 1990.

[6] H. Fuchs and B. Johnson. prepublication draft of "An Expandable Multiprocessor Architecture for Video Graphics". *Proceedings of the 6th ACM-IEEE Symposium on Computer Architecture*, pages 58–67, April 1979.

[7] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. *Computer Graphics*, 26(2):231–240, July 1992.

[8] Michael Noakes and John Cha. J-Machine Host Interface Specification. MIT VLSI Memo, Dec 1993.

[9] Michael Noakes and William J. Dally. System Design of the J-Machine. In William J. Dally, editor, *Sixth MIT Conference of Advanced Research in VLSI*, pages 179–194, Cambridge, MA 02139, 1990. The MIT Press.

[10] M. Potmesil and E. Hoffert. The Pixel Machine: A Parallel Image Computer. *Computer Graphics*, 23(3):69–78, July 1989.

[11] Thucydides Xanthopoulos. A Disk Array for the J-Machine. BS Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1992.

[12] Sasan Zamani. A Scalable Distributed Frame Buffer for the J-Machine. BS Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1991.

5239-38