OKI Advanced Array Processor (AAP) Hardware Description

RLE Technical Report No. 562

Bruce R. Musicus and Srinivasa Prasanna

February 1991

Research Laboratory of Electronics
Massachusetts Institute of Technology
Cambridge, MA 02139 USA

OKI Advanced Array Processor (AAP) Hardware Description

RLE Technical Report No. 562

Bruce R. Musicus and Srinivasa Prasanna

February 1991

Research Laboratory of Electronics
Massachusetts Institute of Technology
Cambridge, MA 02139 USA

Table of Contents

OKI AAP Array

Hardware Manual

Bruce R. Musicus, MIT

Section 1 – OVERVIEW OF OKI AAP ARRAY PROCESSOR

The AAP Array Processor is a 2 board, VME-bus array processor containing a 16 by 16 array of bit-serial processors, 128-512 Kbytes of data memory, and 88 Kbytes of microcontrol store memory. The intent of this project was to provide a prototype massively parallel image processor system with which we can experiment with massively parallel image processing algorithms. The array is packaged with a 68000 single board host computer and a single board frame grabber/video display, thus packing a complete image processing system into a rather small package.

1.1. Processor Array

- 16 by 16 array of 256 bit-serial processors, using 4 OKI AAP chips.
- ≈ 6 MHz clock rate; 1.5 Giga bit operations/sec.
- Support both bit-serial and bit-parallel computation.
    - 256 bit-serial processors
    - 16 rowwise or 16 columnwise 16-bit processors
    - many other combinations
- Also use array in bit-parallel mode for address generation.
    - 32K byte pages, with 4 bit page register.

  - For image processing, typically calculate 16 addresses simultaneously, then read/write one bit plane of a 16x16 subimage, or two columns of 16 pixel bytes each.

- Conditional jump, call, return based on AND, OR, SIGN of Memory Data Register, on flag in command register from host, or on internal sequencer flags.

- Pipeline registers on memory address (MAR) and memory data (MDR) for greater throughput.

- Extensive connectivity provided via two general busses and third additional special bus.

  - simultaneous read/write of data on data paths 1 and 2.

  - pipelined transfer of addresses and data to/from memory.

  - non-pipelined, direct connections of DC and SD pins to each other.

  - use µconst field as argument to memory address, data, or DC, SD pins.

- Some OKI features not supported completely:

  - individually programmable direction select possible via SL register bits, but data transfer not supported at array boundary.

  - only 3 sixteen bit busses available, limiting transfer capabilities, particularly on diagonal data shifts.

1.2. Data Memory

- 128 to 512 KByte, nibble-addressable data RAM

- 20 bit address

- Read/Write any 16 consecutive bits starting at any nibble address

  - uses data alignment, rotation circuitry

  - supports transfer of overlapping subimages into array.

- ≈ 12MByte/sec transfer rate

1.3. Microcontrol Unit

- Primarily Horizontal Microcode - 88 bits wide
- Uses ADSP 1401 microsequencer
  - 4 loop counters, conditional jump or call or return, with 64 level subroutine/counter stack and register bank.
  - constant or computed relative or absolute jump, subroutine addresses.
- Uses finite state machine for clock control
  - 12 MHz crystal, 4 different clock periods, Run/Halt, Single Clock Advance.
- Breakpoint bit in control RAM forces breakpoint subroutine call via ADSP 1401 interrupt.
- Reset on power on, or on host command.
- Finite State Machines allow host to read/write microntrol store RAM, or read/write pipeline register via serial shift register, using 29818 SSR registers.
- Host can monitor next microcode address bus.

1.4. Host Interface

- VME interface: 24 or 32 address lines, 16 data lines, interrupt capability, reset.

- Only acts as VME bus listener: rely on DMA chip or 68000 on single board controller to move blocks of data to or from the array.
- Memory mapped I/O within selectable VME address window.
- Control register: Run/Halt, Single Step, Reset, Breakpoint enable, µinterrupt request.
- Status register: Host and Array status flags.

1.5. Software Tools

- Microcode compiler: *aapcompile*
  - Compiles an expression-oriented language into 88-bit microcode, using detailed knowledge of microcode field multiplexing and hardware dependencies to construct correct, efficient microcode from user instructions.
  - Allows the programmer to use a simplified model of the array processor hardware, using rules to map the instructions into actual hardware capabilities, or generating detailed error messages explaining why the code cannot be compiled as specified.
  - Produces an object file with microcode, data, and labels.
- Object code linking: *aaplink*
  - Links together multiple object files from the compiler, resolving global label references.
  - Resolves forward referencing inside object files.
  - Checks consistency of microcode.
  - Produces a link file suitable for simulation.
- Object code loader: *aapload*

- Links together multiple object files from the compiler, resolving global label references.
- Resolves forward referencing inside object files.
- Checks consistency of microcode.
- Produces a load file suitable for downloading into the hardware via the AOS operating system.

• Load file decoder: *aapdecode*

- Converts load files back into link files to allow examining the contents of the file more conveniently.

• AAP Array Processor Simulator: *aapsim*

- Completely simulates behavior of the array processor, data memory, data busses, microcode memory, sequencer, and a portion of the host interface.
    - Allows simulated running with breakpoints, single stepping of the array.
    - Display and change any AAP register, AAP boundary register, MDR, MAR, PGreg, Data memory, Microcode, or Sequencer register or stack value.
    - Display and change selected intermediate temporary values inside the array, or on the busses.
- Tracks both defined and undefined data values microcode fields, allowing detailed checking for potential programming errors.
- Uses an iterative data propagation algorithm to properly simulate programming modes involving asynchronous data propagation paths through the array and the busses.

• Array Operating System: *AOS*

- Executes on the single board computer, communicating with the user via the terminal.

- Coordinates overall array activity, frame grabber status, and DMA transfers between the array and the frame grabber.
- Supplies many of the same features as found in the simulator.
    - Load files from aapload into the microcode and data memories, saving labels for use by user.
    - Run microcode with breakpoints, single step microcode.
    - Display or change registers in the AAP array, MAR, MDR, PGreg, or in the sequencer or stack.
    - Display or change data memory values.
    - Control the frame grabber, initiate DMA data transfers.
    - Cooperate with AOS microcode kernel to handle message passing between the array and the 68000.

• AOS microcode kernel: *Kernel*

- Initializes the sequencer and micro-interrupt system.
- Catches breakpoint, host micro-interrupt, and stack overflow/underflow interrupts, sending message back to AOS.
- Maintains input and output circular message buffers to communicate with host.
- Uses signalling procedure to get quick service by host for DMA transfers and other services.
- Saves and restores state of array and sequencer from data memory on any interruption of user program.. Allows AOS to examine and modify the copy in data memory, thereby modifying the contents of the internal registers next time the user program is executed.

Section 2 - SUMMARY OF AAP PROCESSOR ARCHITECTURE

In this section we review the overall structure of the architecture of the AAP Array Processor. Several figures are included to illustrate the overall board architecture, the wiring of the data busses and memory, the internal processor architecture, and the internal sequencer architecture. These diagrams are critical to understanding the behavior of this system and the structure of its language. More details may be found in later sections of this manual.

2.1. AAP Processor Array

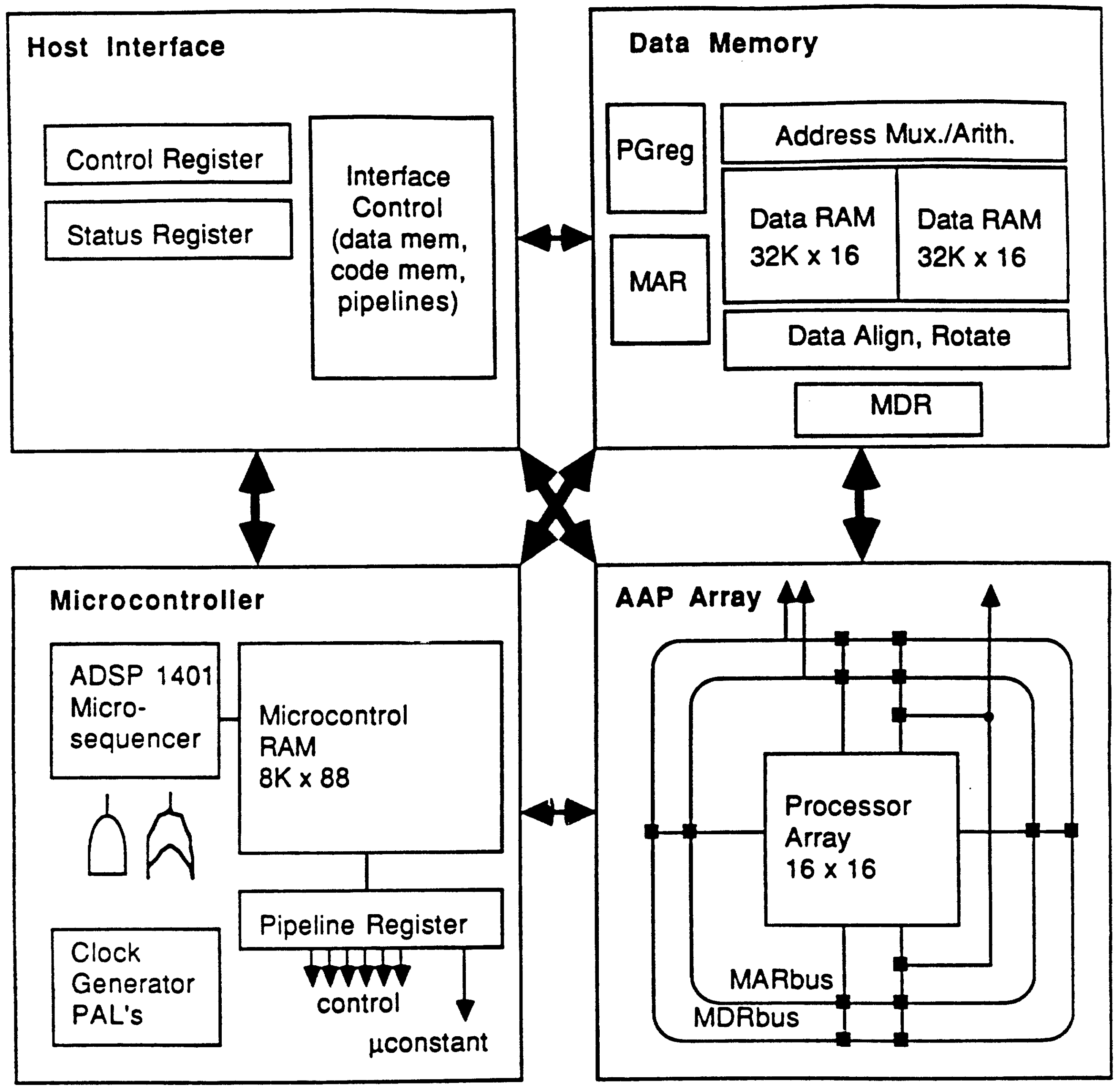
The AAP array is a 2 board system built around a 2 by 2 array of OKI AAP chips, providing a 16 by 16 square grid of 256 single-bit processors. The clock rate is about 6 MHz. Within each clock period, all the processors will execute the same micro-instruction, though internal registers can slightly modify the behavior of that instruction inside each processor. The architecture is thus in the class of Single Instruction, Multiple Data (SIMD) machines. In each clock period a processing element (PE) can read up to 2 bit values from 2 register banks, compute an arbitrary function of 2 bits with carry, can save the result in various registers, and can exchange 2 bits of data along 2 separate data path systems linking adjacent processors. The first data path system provides an interconnection with all 8 neighboring processors; the second only connects processors vertically and is intended primarily for shifting data in and out of the array. Fastest operation is achieved with bit-serial arithmetic on data available either locally, or in a neighboring processor. However, bit-parallel arithmetic is possible by ripple propagating carries from one processor to the left or down neighbor. Also various broadcast modes are available to feed data to all processors in parallel. The array is even capable of asynchronous ripple computation modes, where values passed from an adjacent processor are computed on by the ALU, then the result passed immediately to another next processor. To accommodate these more complex operations, the clock period is programmable. (Ordinarily, the appropriate clock period is chosen by the compiler.)

The basic PE architecture is straightforward. Two register banks may be accessed simultaneously: the A[] bank contains 32 bits, while the B[] bank contains 64 bits. One input to the single-bit ALU always comes from the B[] bank. The other input can come from the A[] bank, from the output of a neighboring processor via data path 1, from the shift register DIO on data path 2, or from the RS routing selector flag attached to data path 1. The output of the ALU may be stored in either or both register banks, in the RS or DIO flags, and/or in a status flag called LF. It can also be routed through the output multiplexor to neighboring chips.

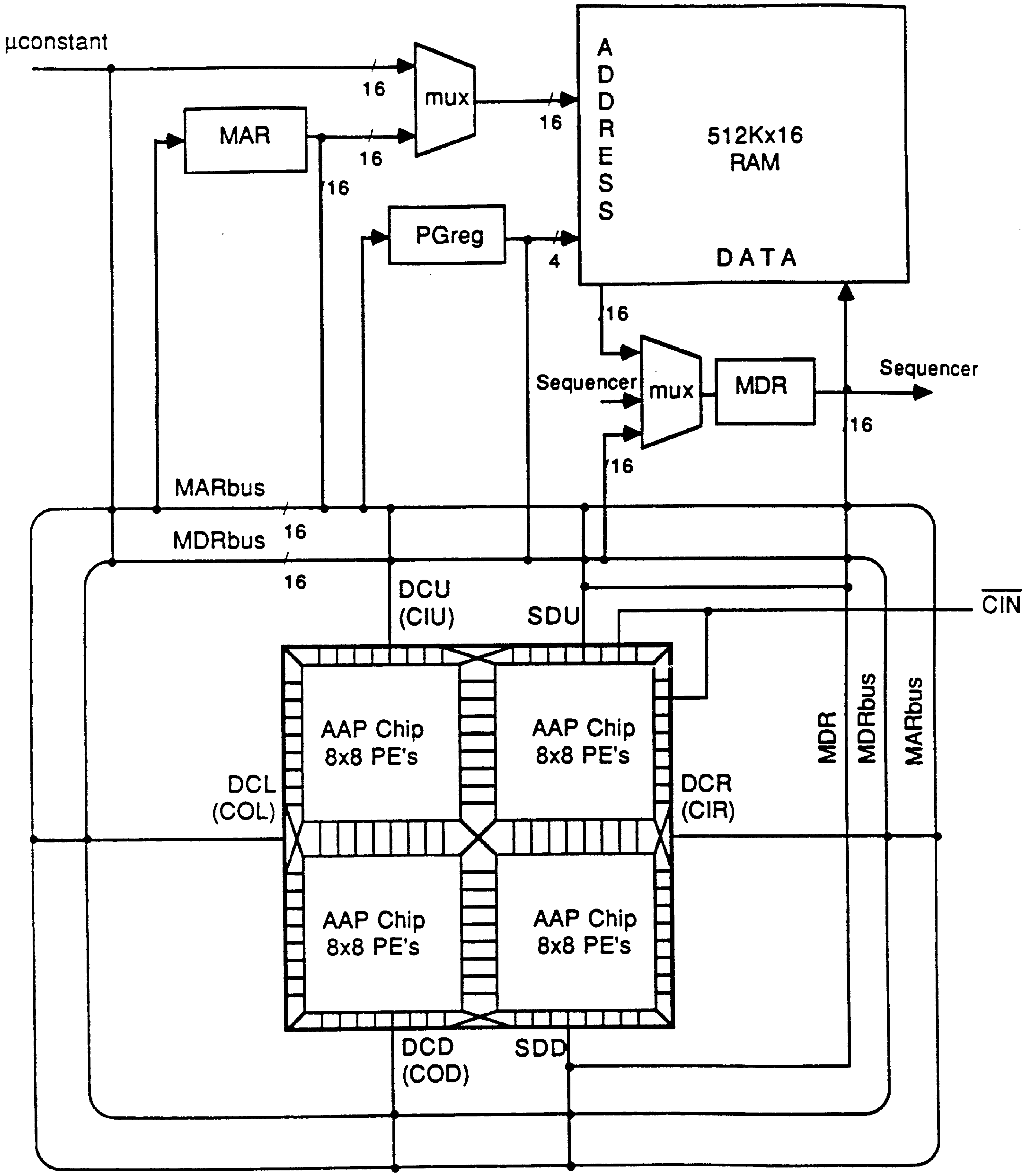
Input from any of 8 neighboring processors may be routed from data path 1 into the ALU, saved directly in RS, or it may be directly forwarded to the PE output to propagate directly to the next processor. This latter mode allows a single bit value to ripple through a series of PE's in the specified direction. Special bypass paths installed in the boundaries of the AAP chips can speed propagation of data broadcast through the array from the external data path 1 pins. The slowest propagation mode has each PE route data input from a neighbor through the ALU, to the output multiplexor, and then to the next processor in the chain. This mode can be used to have each processor add a value into a sum, then pass the new sum to the next processor which in turn adds another value into it, and so on, producing a sum bit of many values in a single (long) clock cycle.

Data path 2 operation is more straightforward. Data in the DIO registers may be shifted up or down along this path by a distance of 1 PE per clock. These registers may be loaded from the ALU output, or can used as ALU input. This data path system is intended for loading data into the array, 16 bits at a time, from the external data memory.

The carry input to every ALU may be forced to 0 or to 1, it may come from the C register bit, or it may come from the carry output from the PE to the right or above. For processors on the top and right edges of the array, the carry input may be specified instead by a one-bit field in the micro-code. Alternatively, 16 individual carry input bits along the top or the right side of the array can be



OKI Array Processor - Block Diagram



AAP Processor Array, Data Busses, Data Memory, MAR, MDR, PGreg (Programming Model)

specified from either external bus, and the 16 carry outputs along the bottom or the left side can be written to either bus.

The simplest ALU operations use a fixed carry input value of 0 or 1 in all the PE's; this is suitable for logical operations on bits, or for simply moving data through the ALU from one register to another. In order to perform operations on integers composed of multiple bits, the array must either use several clock cycles or several PE's. The simplest mode involves bit-serial computation. Each PE holds all bits of each integer in successive storage locations in A[] or B[]. The low order bits are combined in the ALU, together with a fixed carry input or 0 or 1. The carry output is saved in C. Next, successive bits of the integers are processed through the ALU, and combined with the value of C, producing a new sum bit and a new carry-out bit which is saved in C. To add 256 pairs of n-bit integers, giving 256 n+1 bit sums, would require n+1 cycles.

A more complicated arithmetic mode, called bit-parallel, distributes successive bits of the integers in the same register address of adjacent PE's in rows or columns of the array. To add sixteen pairs of 16-bit integers, stored for example in bit B[20] and bit A[31] of the 16 rows of the processors, each ALU adds the corresponding bits together with a carry-in bit from the PE to the right. It stores the sum bit locally, and ships the carry-out bit to the neighbor to the left. The sum of these sixteen 16 integers is generated in a single (longer) clock period.

The direction of data flow on data path 1 is usually determined for all processors by a field in the micro-code word. However, 2 flags SL0 and SL1 forming the SL register, can be loaded from the ALU to individually control which of 4 neighbor output PE values will be read by this PE. Boundary flags TRU, TRL, TRD, TRR in each chip must also be loaded to transfer data in the appropriate direction across each data pin of the chip. This capability allows the user to configure arbitrary data paths through the array. (This capability is also quite dangerous, since it allows the user to construct oscillating asynchronous loops, or propagation paths whose delay is substantially greater than any clock period the hardware supports.)

Normally, every PE does the same operation on the same registers, and the only difference is the data they process. However, operation of the PE can be modified by the value of a selected status (S) bit, either flag LF or register bits B[2] through B[7]. Whether or not this bit is a 0 or a 1 may be used to control whether or not a B[] register bit or the RS flag is loaded. It can also cause the PE to modify what register may be combined with a B[] bit in the ALU. It also conditions which value will be output by the processor on data path 1, and whether the carry input will be taken from a neighbor or from the C register. Normally, the compiler outputs a two instruction preamble to all object files which initializes register bits B[2] to 0, and register bits B[3] to 1. This allows the compiler to choose one of these bits as the S bit, and thereby force certain operations to execute unconditionally. Normally, the compiler prevents the user from modifying bits B[2] or B[3].

2.2. Chip Boundaries, Data Busses, Pipeline Registers, Data Memory

Surrounding each 8 by 8 processor subarray on each OKI chip are four boundary logic systems which interface the PE data paths and carries to the chip's pin drivers. The DCU, DCL, DCD, and DCR pins on the Up, Left, Down, and Right sides usually carry PE output values, or provide inputs to data path 1 input multiplexors in the PE's. The SDU and SDD pins on the Up and Down sides connect to the DIO registers in adjacent processors on data path 2. Carry input, CIR and CIU, and carry output, COL and COD, pins support the ripple carry modes. This boundary logic can be programmed to support particular data transfer modes, and can also be used to optionally multiplex the ripple carry values on unused data pins. An important feature of these boundary systems is that they can quickly forward the output of the top or bottom row, or left or right column to the DC data output pins on the opposite side of the chip. This greatly speeds broadcast of data through the array of chips. Flags SPU, SPL, SPD, SPR in the boundary can conditionally modify this broadcast bypass operation on a pin by pin basis.

The four OKI AAP chips are wired together to form a 16 by 16 grid of processors. Surrounding this array are two major 16-bit bus systems, called MARbus and MDRbus. These busses are used to connect together all sides of the two data paths in the array, as well as pipeline registers in data memory, and 16-bit constants supplied by the sequencer. These busses can be used for a variety of purposes: carrying addresses and data between the processors and data memory, wrapping data shifted out of one side of the array into the other side, moving constants from microcode memory into the processor array, and so forth.

Each PE in the array has 32 bits of storage in an A[] register bank, and 64 bits of storage in a B[] register bank. External data storage is also available in a separate data memory system. To support bit-serial arithmetic on overlapping portions of an image or data array, the data memory is nibble addressable - it can read or write 16 bits starting at any even 4 bit boundary. The top 4 bits of the 20 bit address are taken from a page register called PGreg, while the bottom 16 bits can be specified either by the Memory Address Register (MAR), or by the constant in micro-code memory. A Memory Data Register (MDR) is used to hold data to be written into memory, or that has been read from memory. The MAR and PGreg registers may be loaded from the MARbus and MAR can drive that bus. The MDR register may be loaded from or may drive the MDRbus. PGreg can drive MDRbus.

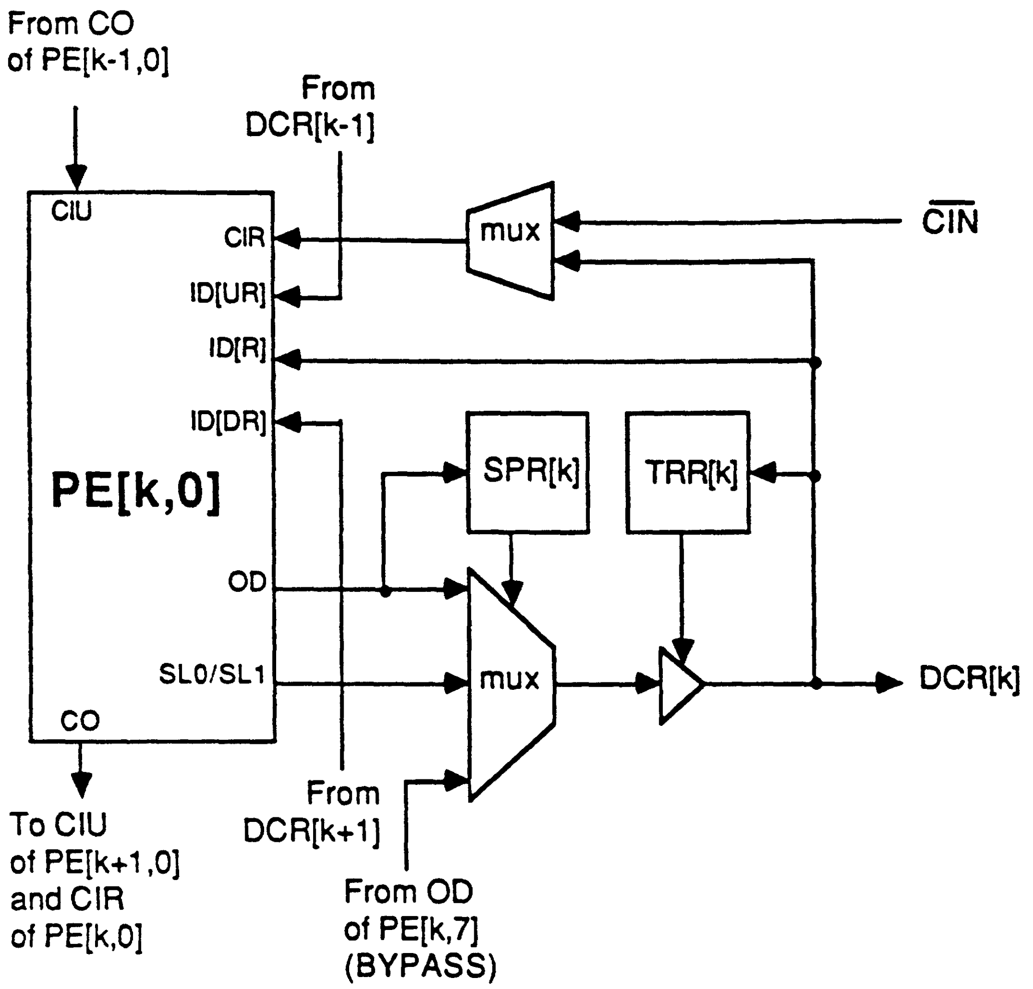
Three clock cycles are needed to read a single data memory value: first load MAR with the address, then read the value into MDR, then move the result into the processor array. Best performance is achieved by treating the MAR, MDR, and PGreg registers as *pipeline* registers. Typically, the processor array will use bit-parallel arithmetic to generate sixteen different 16-bit addresses in the sixteen rows or columns of the array. On each clock tick, an address will be shifted out of the array over the MARbus to MAR, the memory value at the previous address will be read into MDR, and the data element fetched previously will be transferred over MDRbus back into the array. In this way 16 values can be read from memory in only 18 clock periods. Similarly, 16 values can be written in 17 clock periods.

2.3. Microcode, ADSP 1401 Sequencer, Clock Generator

Horizontal micro-code is used to program this machine. The micro-code memory is 88 bits wide. In the simulator and in the object files, each code word is broken down into a smaller number of fields, each of which is represented by a mnemonic code. Different fields control different aspects of the system behavior on each clock period - some control the sequencer, some control the processor array, some control the data busses, some control the data memory, and the last 16 bits are a micro-constant field which can be used to drive either data bus, to provide the low 16 bits of the memory address, or to drive the sequencer data input bus. User instructions in the source files are translated by the compiler into appropriate field definitions in the microcode words. Fields controlling capabilities that are not used in a given instruction are deliberately left undefined by the compiler, but are filled in with default values by the loader `aapload`.

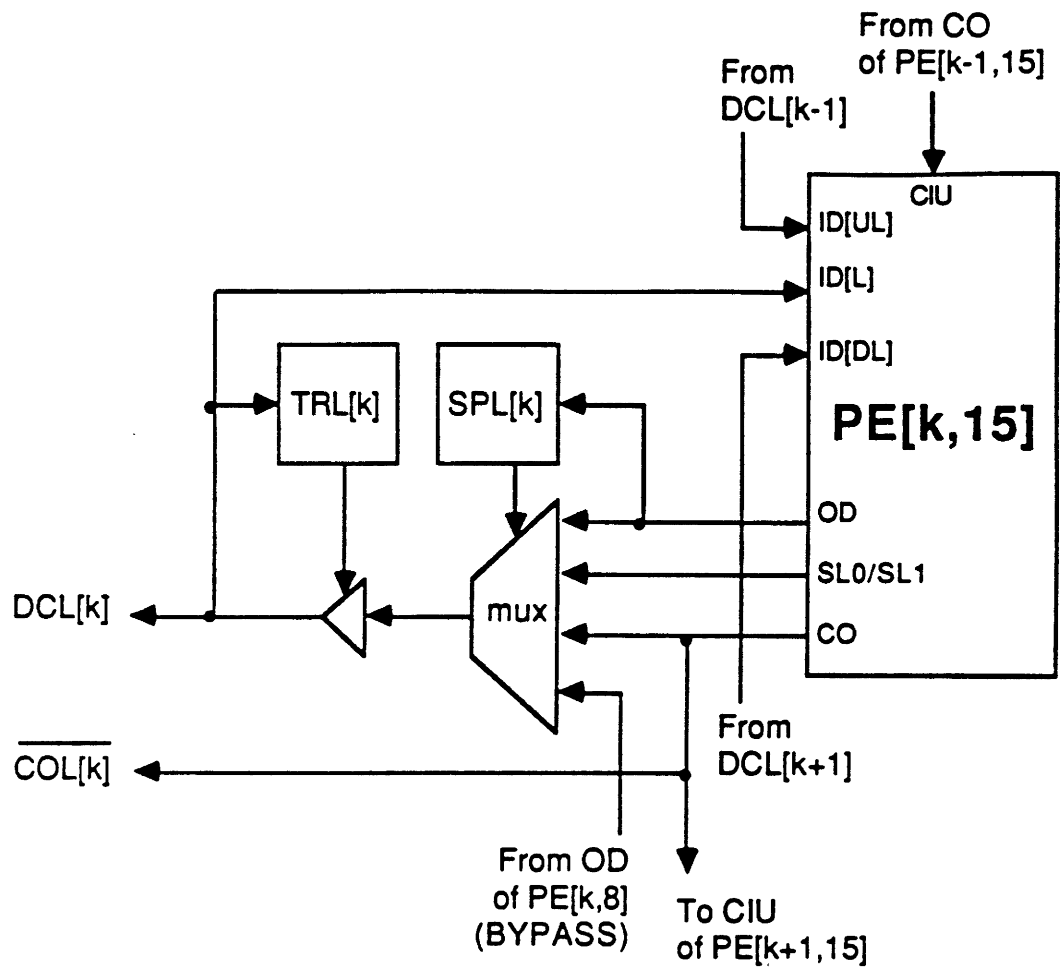
An ADSP 1401 micro-sequencer is used to control the sequence of micro-code instructions to be executed. This sequencer supports 4 internal counters for loop control, and has a 64 element stack. Part of this stack is used for subroutine nesting, and the rest is used as a "register stack" holding jump addresses and count values. Two register stack pointers are available: *gsp* (global register stack pointer), and *lsp* (local register stack pointer). This chip is also capable of extensive interrupt support, with 8 possible external interrupts, a stack overflow interrupt (level 9), and a counter overflow interrupt (level 0). Of these, the hardware supports a breakpoint interrupt triggered by a bit in the micro-code word (level 3), and a host micro-interrupt (level 8) triggered by the *HuIntR* flag in the host interface. The compiler supports the entire instruction set of this chip, although features associated with the interrupt system can only be accessed through the *sequencer()* call. Microinstructions *wcs*, *ihc*, and *idle* are not supported by the hardware, and should never be used. (Note that the compiler's "`if() idle`" command can be used because it is compiled into a conditional goto() instruction.)

Call, jump, return, and several other sequencer instructions can be conditioned on several different status bits in the architecture. Execution can be conditioned on the AND, OR, or SIGN of the 16-bit data in the MDR register. This allows testing values read from data memory (just read it into MDR), or testing values computed in the array (output 16 bits through any side of the processor array over MDRbus into MDR). Special pipeline bypass registers allow conditionally executing a sequencer instruction based on the value



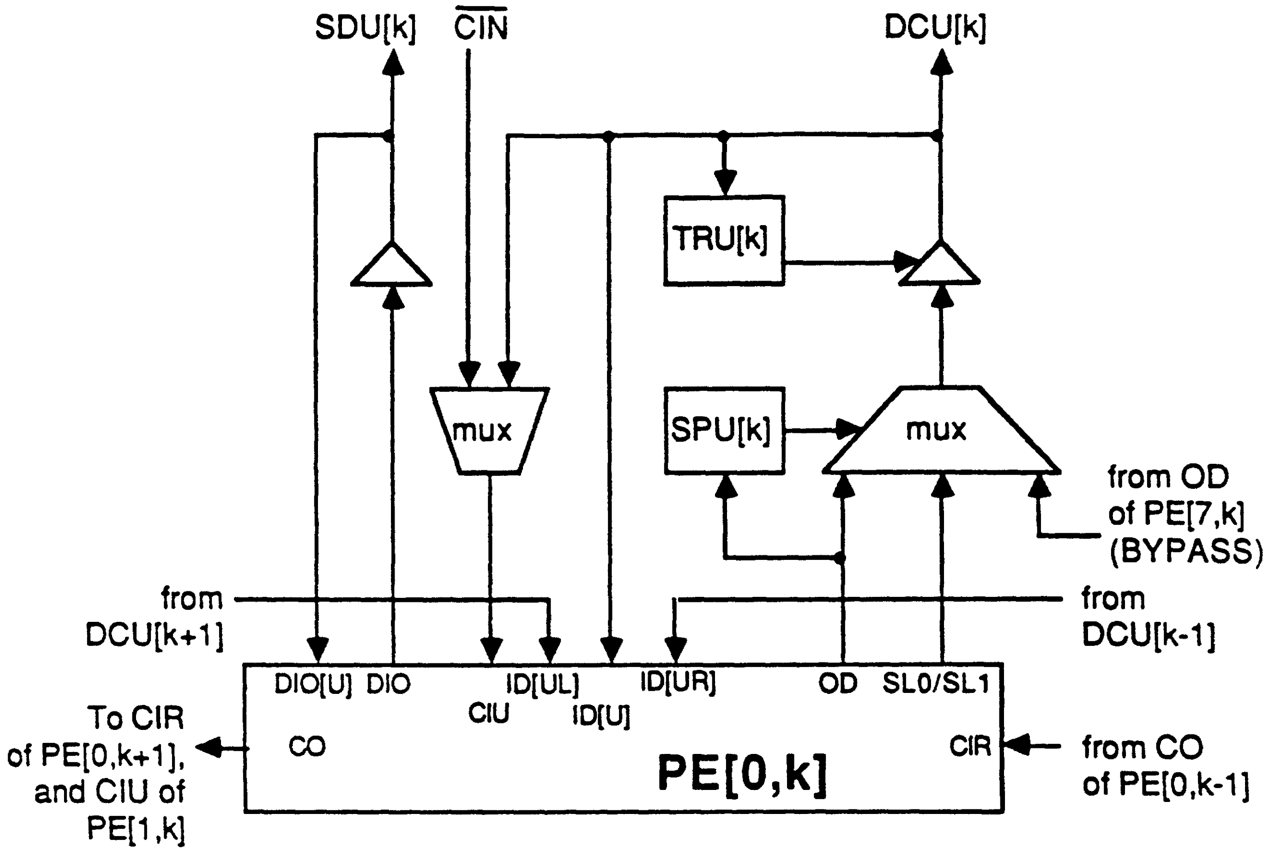
Right Side (Row k, Column 0) of Peripheral Circuitry (Programming Model)

(Control lines omitted to simplify the diagram)
(Corner processors, k=0 or k=15, connect to diagonal input pins DCUR or DCDR)



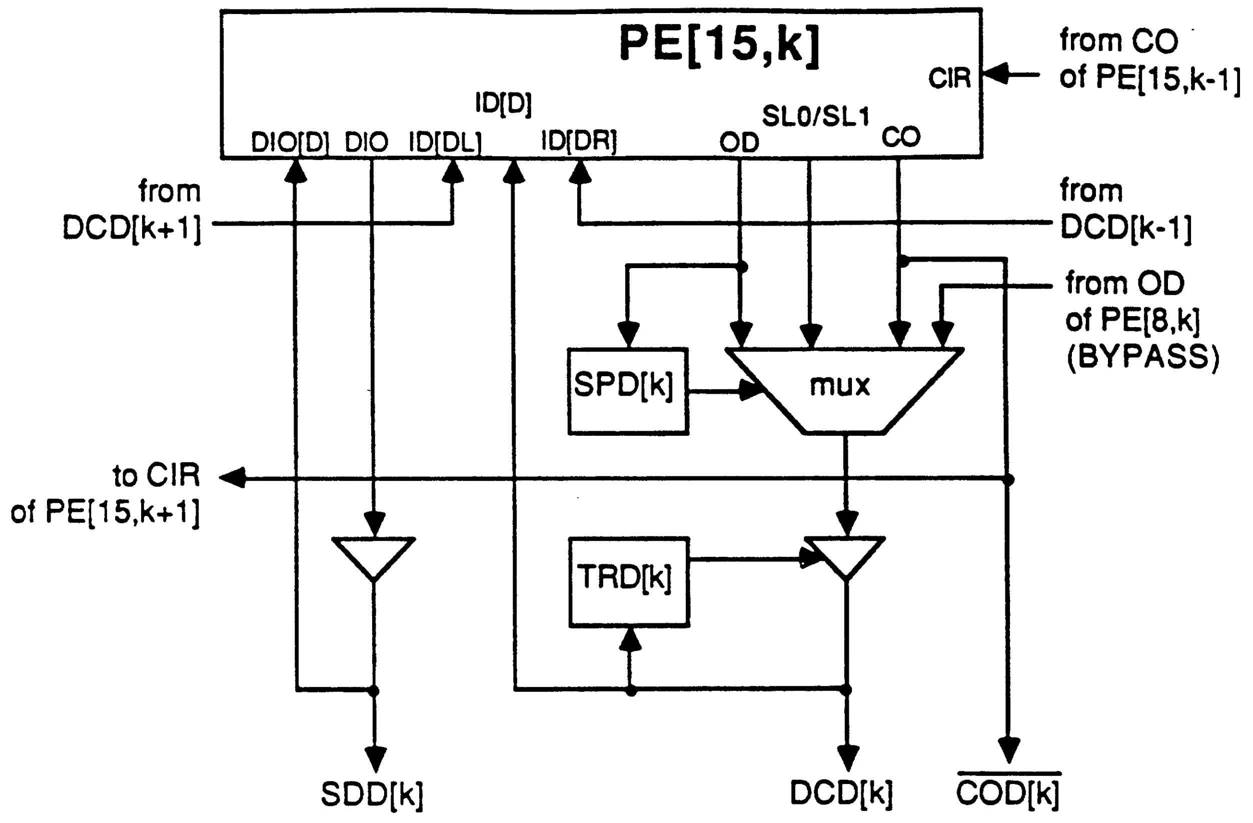
Left Side (Row k, Column 15) of Peripheral Circuitry (Programming Model)

(Control lines omitted to simplify the diagram)
(Corner processors,.k=0 or k=15, connect to diagonal input pins DCUL or DCDL)



Upper Side (Row 0, Column k) of Peripheral Circuit (Programming Model)

(Control lines omitted to simplify the diagram)
(Corner processors, k=0 or k=15, connect to diagonal input pins DCUR or DCUL)



Down Side (Row 15, Column k) of Peripheral Circuit (Programming Model)

(Control lines omitted to simplify the diagram)
(Corner processors, k=0 or k=15, connect to diagonal pins DCDL or DCDR)

loaded into MDR in the previous instruction. Execution can also be conditioned on bit *HFlag0* in the host command register. This is used by the AOS operating system to help synchronize transfer of data between the single board host and the array. Finally, execution can be conditioned on the internal sign bit in the sequencer, thus allowing use of counters to control iteration loops.

Four different clock period lengths are supported by the hardware. The second fastest clock period should be selected for operations using bypass paths to broadcast data through the array and for bit-parallel arithmetic. The third fastest period is intended for asynchronous ripple arithmetic. The fourth fastest clock period is intended for operations using asynchronous propagation of values through multiple PE's ALU's, possibly combined with bit-parallel arithmetic.

2.4. Host Interface

The host interface allows substantial control over the array by the 68000 host. By accessing I/O registers in the top 16 bytes of the data memory address space, the host can read the next microcode address, examine and change the microcode pipeline, MAR, MDR, and PGreg registers, and it can read and write microcode memory. Bytes or 16-bit words can be read or written to data memory directly from the VME bus. A command/status register allows control over array functioning. The low byte of this register contains an interrupt vector for use when the array interrupts the 68000. Bits in the upper byte allow the host to start or stop the array clock, to reset the sequencer forcing execution to start from location 0, and to single step the clock when it is halted. The host can enable host interrupts from the array with the *HIntEn* bit (currently the AOS system leaves host interrupts disabled, and relies on software polling). Bit *HIntR* in the status register is used by the array to request the host interrupt. The host can request a micro-interrupt of the array by setting *HuIntR*. Flag *HFlag0* can also be set or cleared by the host to help synchronize communication with the array, which can condition instructions on this flag value. (AOS uses *HIntR* and *HFlag0* in tandem to implement a full handshake between host and array.)

Section 3 – USING THE HOST INTERFACE

3.1. Overview of Host Capabilities

The array processor is intended to be plugged into a VME bus system, with a conventional single board processor to control it. The array acts like a slave and an interrupter for the controlling processor. Any 512K byte address segment can be mapped into the array. The top 16 bytes of this segment are mapped into special registers in the array and into special functions. The remainder is mapped into the data memory. An 8 bit command/status register allows the host to start and stop the array, single step it, and reset it. The host can enable interrupts from the array, and can in turn interrupt the array. An arbitrary 8-bit host interrupt vector can be programmed. Most registers inside the array are backed by a circular serial shift register (SSR). The host can read and write into this shift register, and can transfer the contents of the serial register into the normal registers or vice versa. The contents of the serial shift register can be written into any location in microcode memory, or any location in microcode can be loaded into the serial shift register.

3.2. VME Bus Addresses

The VME bus interface supports a variety of address and data modes. The AM0-AM5 lines on the backplane determine which of these modes is being used at any time. At present, the array processor will respond to either 24 or 32 bit addressing in user or supervisor data or program modes (AM0-AM5=09, 0A, 0D, 0E, 39, 3A, 3D, 3E). The array processor will respond only to addresses within a selected 512 Kbyte (19 bit) range. Eight switches select the top 8 address line values A24-A31 to which the board will respond (these are ignored during 24 bit addressing cycles). Another 5 switches select the address bits A19-A23 to which the board will respond. If less data memory is installed, and this much address range cannot be dedicated to this board, then jumpers can be connected to force response only in the upper 256 Kbyte of this range (A18=1) or in the upper 128 Kbyte of this range (A17=A18=1).

The upper 16 bytes of this range are dedicated to special internal registers and special actions. The remainder of this address range is mapped into the data memory. (An unfortunate side effect is that the upper 16 bytes of memory cannot be accessed directly by the host.)

(The current version of the board has 128 Kbytes of data memory, and the switches are set so that the memory appears at addresses `0xee0000` through `0xefffff`, with the I/O registers at addresses `0xeffff0` through `0xefffff`.)

The following table lists the address ranges that can be selected for the data memory and I/O registers. In this table, `base` is the first address recognized by the board, `base=` $N*2^{18}$ with $0 \le N < 2^5$.

If 128 Kbytes memory installed:

`base` through `base+0x1ffef`	Read/Write data memory
`IOreg = base+0x1fff0`	

If 256 Kbytes memory installed:

`base` through `base+0x3ffef`	Read/Write data memory
`IOreg = base+0x3fff0`	

If 512 Kbytes memory installed:

`base` through `base+0x7ffef`	Read/Write data memory
`IOreg = base+0x7fff0`	

The following table summarizes the behavior of the board in response to read and write accesses from the host. In this table, `base` is the first address recognized by the board, `base=` $N*2^{18}$ with $0 \le N < 2^5$, and `IOreg` is the address given by the table above.

Address Range	**Read/Write Action**
`base` through `IOreg-1`	Read/Write data memory Byte or word access allowed
`IOreg`	Read/Write upper byte: Command/Status Register Read/Write lower byte: Host Interrupt Vector

`IOreg+2`	Read word: Shift out the upper 16 bits of the SSR and return their value. Write word: Capture the 16 bits, and shift them into the SSR.
`IOreg+4`	Read word: Transfer parallel registers into SSR Write word: Transfer SSR into parallel registers
`IOreg+6`	Read word: Return current address on the Y-bus. Write word: Use HD0-HD15 data as the µcode address on the Y-bus. Read the 88 bit µcode word into the parallel pipeline register.
`IOreg+8`	Read word: No operation performed. Write word: Copy 88 bits from the SSR into the microcode word at the address specified by the data.
`IOreg+0xA:` `IOreg+0xC:` `IOreg+0xE:`	Read/Write byte or word: No operation performed.

The host may access any of these I/O or data memory locations even while the array is running; cycle-steal code in the PAL's satisfies the host access immediately after the present array instruction completes. (Beware that certain array operations can be quite long, up to 32 periods of the oscillator clock). If faster data memory access is required, the array could be halted first. It is also advisable to halt the array before attempting any of the more complicated actions initiated by addresses `IOreg+2` and above. In particular, always halt the array before writing to `IOreg+6` and reading some microcode word into the pipeline register. Otherwise, this microcode instruction will be executed on the next clock tick, causing random behavior in the array.

Note that many of these I/O registers should only be accessed by words, not by bytes.

The following sections explain in detail how to use the behaviors invoked through the array's I/O registers.

3.3. Accessing Data Memory

Internally, from the array point of view, the data memory appears to be a nibble addressable, 16-bit wide memory with a 20 bit address bus. From the host point of view, however, data memory is only byte addressable, and can handle either 8 or 16 bits of data on each access. (16-bit fetches must start on even byte boundaries only). Given host address `base+offset`, the base is stripped off and an extra least significant zero bit is padded onto `offset`, giving the data memory address: `2*offset`. Either or both the even and odd bytes may be read or written in any cycle. Finally, note that only host addresses `base` through `IOreg-1` refer to data memory; the top 16 bytes of data memory are inaccessible from the host.

An important issue is that numbering of the bytes within a word is not consistent between a 68000 host and the array. This can cause problems with transferring byte data between the VME bus and the array. Because the array must be able to access any 16 consecutive bits in data memory starting at any 4 bit boundary, it assumes that the nibbles in each word are numbered as follows. If `addr` is the 20 bit address of the word, then `addr` is the address of the least significant 4 bits, `addr+1` is the address of the next 4 bits, `addr+2` is the address of the next 4 bits, and `addr+3` is the address of the most significant 4 bits. From a byte point of view, `addr` is the address of the least significant 8 bits, while `addr+2` is the address of the most significant 8 bits. Unfortunately, the 68000 numbers the bytes in a word in the opposite order. Let `addr2=base+addr/2` be the VME bus address of the same word. Then the upper 8 bits are treated as byte `addr2`, and the lower 8 bits as byte `addr2+1`. If 16-bit data is transferred between the 68000 and the array, then this numbering difference can be ignored. However, if 8-bit data is transferred, the order of the bytes may have to be interchanged either in the array or in the 68000. (The AOS operating system software automatically performs the necessary byte swaps, but the DMA channel does *not.*)

3.4. Host Command/Status/Interrupt Register

The word at `IOreg` contains the command/status register in the upper byte, and the interrupt register in the lower byte. The meaning of some of these bits differs depending on whether the host reads or writes the word:

Bit	Read	Write	Description
15	`Run`	`Run`	Let Array Run
14	`HIntEn`	`HIntEn`	Enable host interrupts
13	`Hflag0`	`Hflag0`	Host Flag 0
12	---	---	---
11	`HµIntR`	`HµIntR`	Request micro-interrupt
10	---	`HReset`	Force Reset cycle
9	---	`SingleStep`	Force Single Step if halted
8	`HIntR`	---	Host interrupt request

7-0	HIntReg	HIntReg	Host Interrupt Register

Setting the Run flag will allow the clock generator to run; clearing it will force the clock to complete its present cycle and then stop. Setting HIntEn will enable the board to interrupt the host; clearing it will disable any interrupts and will also clear the pending Host Interrupt Request flag HIntR. When HIntEn=0, if the µsequencer attempts to interrupt the host, the HIntR flag will go high but no host interrupt will be generated. By reading the status register, the host can determine that a host interrupt had been requested. Host flag Hflag0 can be directly tested by the micro-sequencer, and thus is convenient to use for synchronizing activity between the host and the array. (The AOS software uses HIntR and HFlag0 as read/acknowledge signals for synchronizing data transfers between the array and the host.)

Setting the HµIntR flag requests a micro-interrupt. (This interrupt will only occur if this feature has been enabled by the 1401 sequencer.) Setting the HReset bit will force a Reset cycle in the array when the array runs. (This bit cannot be read, and it will be immediately cleared once the Reset operation is complete.) Setting the SingleStep bit will force a single clock cycle in the array if the array is halted. (This bit can not be read, and it will be immediately cleared once the clock cycle is under way.)

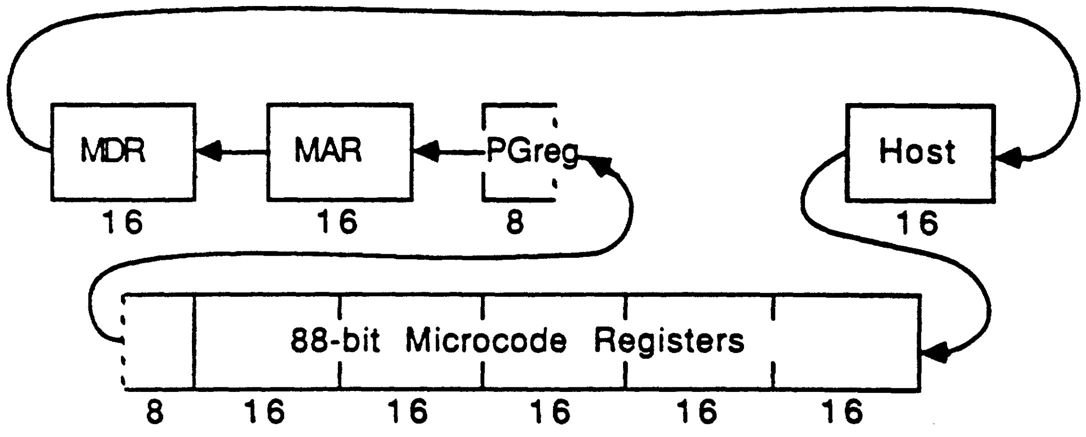
When reading the command register, bits 10, 9 are always high, while bit 8 is HIntR, which indicates whether the array has a pending host interrupt request.

The contents of the host interrupt register will be returned when the VME bus acknowledges a host interrupt request from the array. This will cause a vectored interrupt to the specified location in host memory.

Bits 12-15 of the command register will be cleared only during a VME bus system reset. HReset will clear during any array reset cycle.when the clock generator begins a Reset cycle. The host or the array will have to explicitly clear the HµIntR flag. SingleStep will clear when the clock µclk drops low during the next clock cycle.

3.5. The Serial Shift Register (SSR) System

The MDR, MAR, PGreg, and µcode pipeline registers are all built from AM29818 Serial Shift Register (SSR) Diagnostic Pipeline registers. These chips normally behave like fast parallel load, parallel read octal registers. However, they also contain an independent octal left-shift register which can be loaded into or from the parallel register. All these shift registers are connected into a single SSR shift register chain. The most significant word in this chain is the 16-bit MDR register, followed by the 16-bit MAR register, the 8-bit PGreg (only the bottom 4 bits are used), and then the 88 bit microcode pipeline register. Various PAL's implement various finite state machines allowing the host to load, unload, and shift the SSR, exchange the contents of the shift register and the parallel registers, and load or read µcode memory into the shift register.



Serial Shift Register (SSR) Chain

A technical difficulty with the present implementation is that three separate copies of the MDR register are maintained, only one of which is in a 29818 chip. The other two copies are associated with the micro-sequencer, and cannot be loaded or read by the host. Therefore, although the host is allowed to change the contents of the MDR register, beware that the sequencer will not see the change, and it may therefore act in an unpredictable fashion.

3.6. Reading and Writing Data to the Serial Shift Register

16 bits of data may be shifted in and out of the SSR registers by reading or writing into address `IOreg+2`. Writing a word of data to this address causes the shift registers to shift 16 bits left, with the new data being shifted into the least significant word of the serial register in the microcode pipeline registers. The parallel registers will be unaffected, and the array will not see this activity.

Reading a word from address `IOreg+2` will first cause the SSR to shift left 16 bits, then the value shifted out of the most significant word (the MDR shift registers) will be returned. The value shifted into the least significant word of the SSR will be unpredictable.

Note that reading and writing the SSR register is a relatively slow operation (about 20 oscillator periods).

3.7. Transferring Data Between Parallel and Serial Registers

Reading a word from address `IOreg+4` will cause all the parallel registers to be loaded into the SSR registers. The data returned from this read operation should be ignored. Writing a word to address `IOreg+4` will cause all the SSR register contents to be loaded into the parallel registers.

This feature allows the host to setup all the array registers, force a specified number of clock ticks, then read back all the register contents to analyze what the array did. To do this, first stop the array clock by writing a 0 into the Run bit in the Command/Status register, `IOreg`. Now fill the SSR chain with 8 words of data by writing into `IOreg+2`. First, write the desired contents of MDR and then MAR into `IOreg+2`. Then form a word with the PGreg page register in the top byte, and the most significant microcode byte in the low byte, and write that into `IOreg+2`. Finish by writing 5 more words of microcode bits into `IOreg+2`, with the least significant word coming last. The SSR registers are now full; copy them into the parallel registers by writing anything into `IOreg+4`. The microcode pipeline register now supplies the specified micro-instruction to the array, while MDR, MAR, and PGreg supply appropriate data. Force one or more clock ticks by writing a 1-bit into the `SingleStep` bit of the Command/Status register for each desired cycle. (Be sure to continue to set `Run=0` and `HReset=0`).

To read back the register contents, read from address `IOreg+4` to copy the parallel registers into the SSR. Now read 8 successive words from `IOreg+2`. The first word is the contents of MDR, the next is MAR, the next has PGreg in the top byte and the most siginificant byte of the next microcode instruction in the low byte. The next 5 words will contain the remaining microcode bits.

Note that the micro-sequencer circuitry maintains independent copies of MDR which cannot be loaded via the SSR registers. Therefore, unpredictable results will occur if the host loads and runs a micro-instruction for which the sequencer needs to use the value of MDR. Note also that strange things may happen if the SSR is copied into the parallel registers while the array is running.

3.8. Reading the Current Microcode Address

Reading address `IOreg+6` will freeze the array clock after the current micro-instruction completes, and then return the micro-address which the sequencer is presently asserting on the Y-bus. This address will be the micro-instruction that will be executed on the *next* cycle.

3.9. Reading Microcode Memory

To read the contents of a specified microcode address, write the desired address to `IOreg+6`. The host interface will force the address onto the . microcode address bus, and read this location into the parallel microcode pipeline registers. Be sure that the array is halted before you attempt this host action.

Next, the parallel registers should be copied into the SSR by reading `IOreg+4` (discard the data). Finally, read successive words from `IOreg+2`. The first and second words are the contents of MDR and MAR, and may be discarded. The third word contains the most significant byte of microcode in the lower byte. The next five words contain the remaining microcode words from the given location.

3.10. Writing Into Microcode Memory

The microcode memory in the array is formed entirely of RAM. Therefore, after system reset, the array is automatically halted, and the host must load micro-programs into the microcode RAM. To do this, the 88 bits of data for a single microcode location are shifted into the SSR chain by writing them into `IOreg+2`. The most significant byte is in the low byte of the first word (the upper byte is ignored). The next 5 words written into `IOreg+2` should contain the rest of the microcode data, with the least significant word coming last. Now, write the microcode address to `IOreg+8`. The host interface will force this address onto the microcode address bus, then copy the SSR contents into this microcode location.

Beware that writing microcode to RAM is a rather slow operation, requiring approximately 36 oscillator periods. Also beware that the SSR contents will be destroyed after they have been written into microcode memory.

Section 4 – HOST INTERFACE – CIRCUIT DETAILS

4.1. Host Interface Structure

The host interface consists of two interacting subsystems. The first system is responsible for coordinating the array response to VME bus cycles, and for coordinating host access to the internal state of the array. It consists of 8 PAL's, some comparators, latches, demultiplexors, and miscellaneous circuitry. The second system extends the VME data bus into an internal host data bus, which in turn connects to the command and status register, to the SSR (serial shift register) system, to the interrupt vector register, and to the microcode address bus. This section is composed of various transceivers, registers, buffers, and resistor pullups. In the following we will present how each component in the host interface contributes to the functionality of the interface.

A comment on notation: a signal such as `LBerr` is positive true, while a signal with a '–' in front, such as `-Berr`, is negative true.

4.2. VME Bus Cycles

The VME bus is a modern 16-32 bit micro-computer backplane bus system. Data transfer is asynchronous, with a full handshake required on every cycle. Arbitration for control over the bus, asserting the address, and exchanging data can all occur in pipelined fashion. In the following, we will describe how the array participates in VME bus cycles. In later sections, we will describe the circuitry in more detail.

The array acts solely as a slave for data transfers, and as an interrupt requestor. The array supports both 24 and 32 bit addressing modes (but not 16 bit addressing mode). It responds to both user and supervisor transfers in either data or program mode, and it responds to either 8 bit or 16 bit data transfers (but not 24 or 32 bit data transfers). The array is configured to respond to addresses within a selectable 512Kbyte range, although jumpers may be set to restrict response to the upper 256 or 128 Kbytes of this range.

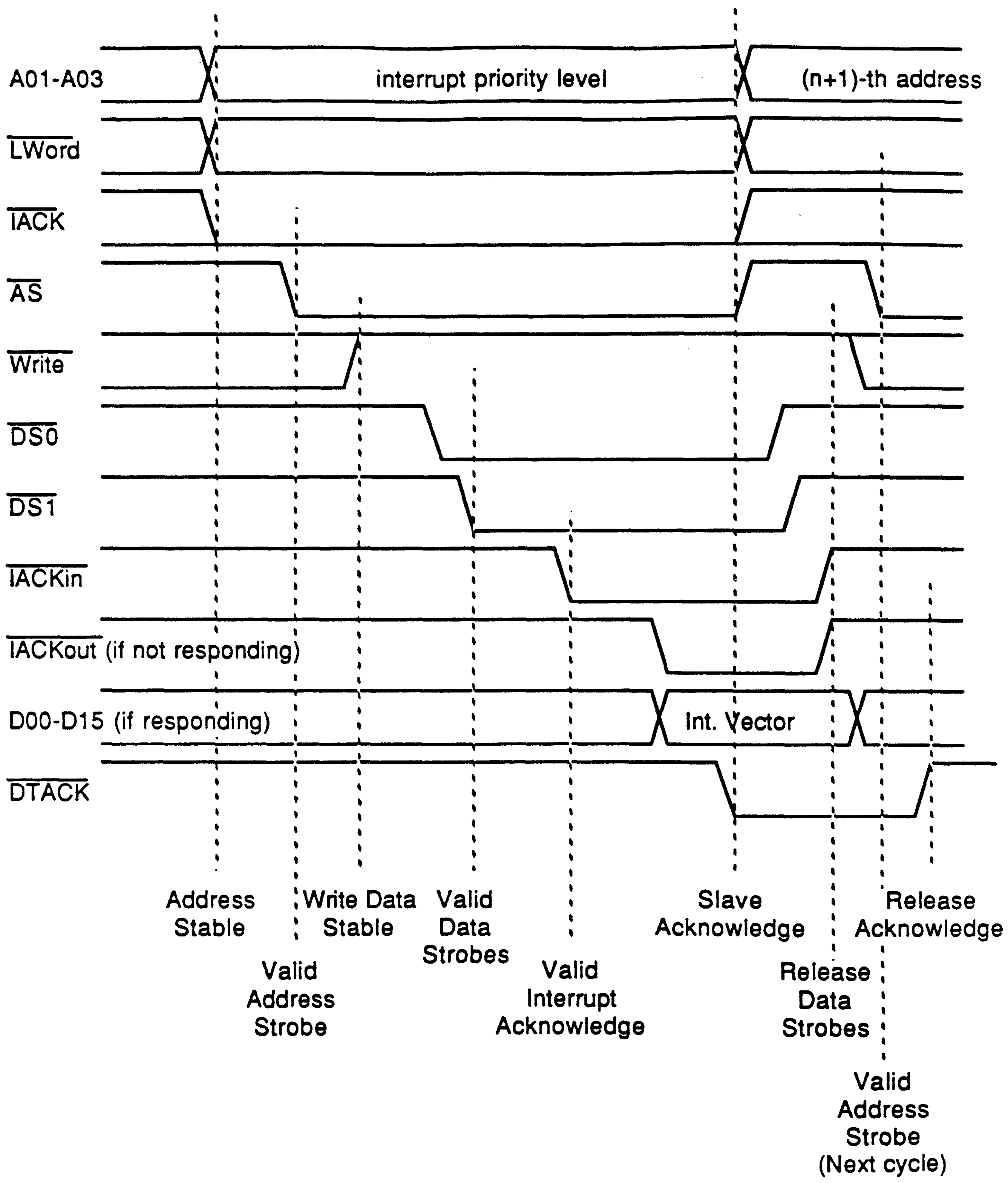
When a board wishes to access the array, it must first gain control of the VME bus via the arbitration procedure. The cycle starts when the bus master asserts a slave address A01-A31, the address modifier code AM0-AM5, a long word indicator -LWord, and flags that this is not an interrupt acknowledge cycle via -IACK. After a short delay, it asserts -AS to indicate that the address information is stable. The slave is guaranteed that -AS will arrive at least 10 nsec after the other lines have stabilized.

The master waits for the data bus lines to be completely clear from the previous transaction, with both data strobes -DS0 and -DS1 deasserted, and both slave acknowledge lines -DTACK and -Berr deasserted. It then drives -Write to indicate whether the cycle is read or write. If it is a write, then it also puts data on D0-D15 (note that the array only reads the lower 16 data lines). After waiting for the signals to stabilize, the master then asserts -DS0 if the low byte is involved in the transaction and/or -DS1 if the high byte is involved. These data strobes are guaranteed to arrive at the slave at least 10 nsec after the address lines and -Write have stabilized, and they are guaranteed to arrive within 20 nsec of each other.

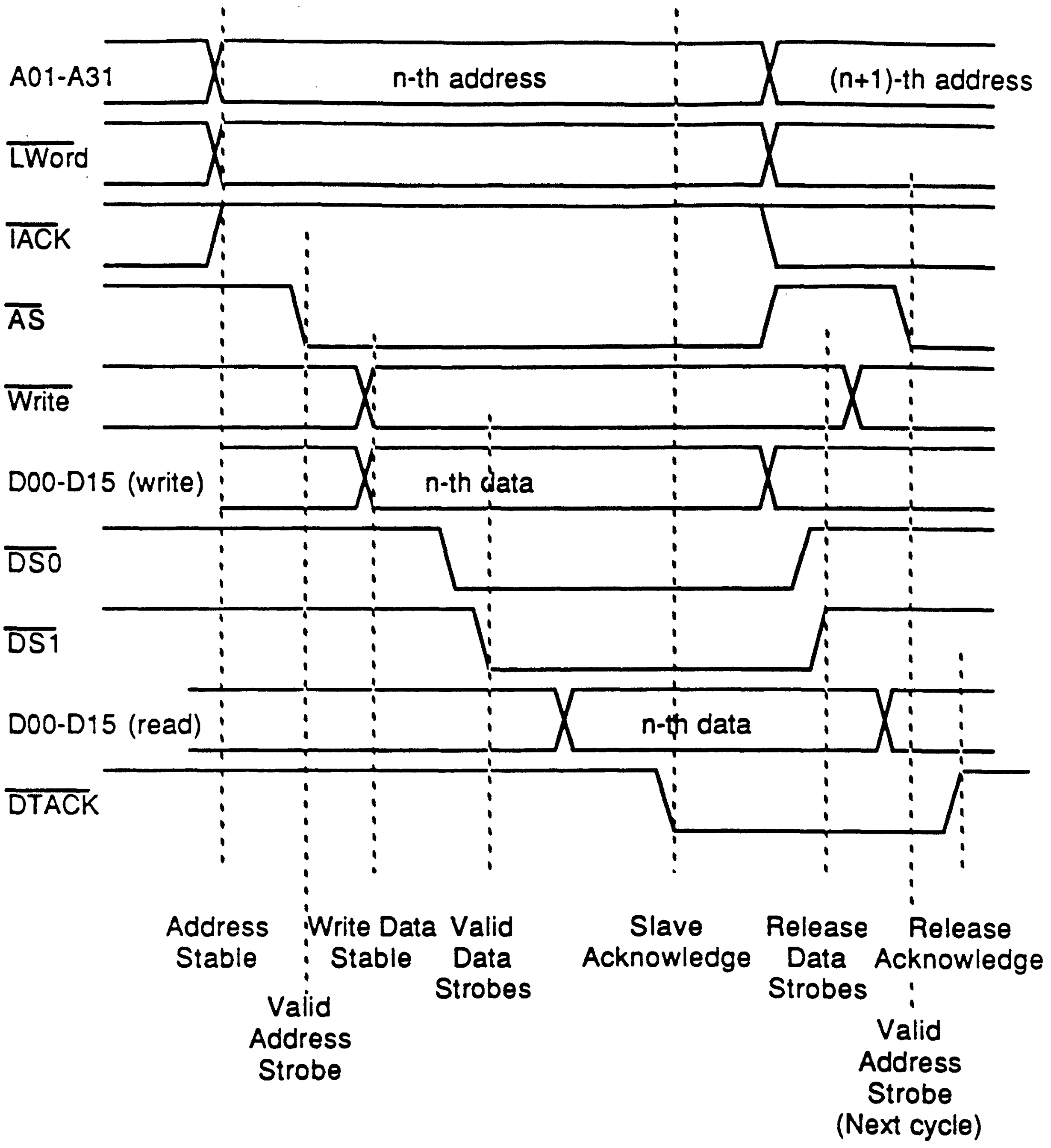
When the slave recognizes its address, with valid address and data strobes and no -DTACK or -Berr signals, then it must take action. If this is a write operation, it must capture the data and use it appropriately. If this is a read, it must retrieve the requested data and drive it onto the data lines D0-D15. When finished, the slave should assert -DTACK. If this is a 24 or 32 bit data operation, the array will assert -Berr instead.

When the master receives the acknowledge from the slave, either -DTACK or -Berr, it may remove all its signals from the bus. Note that the order of removal is unspecified. Thus, the bus master may change the address lines or data lines (in a write operation) before removing the -AS, -DS0, or -DS1 strobes. (The -Write signal, however, will remain asserted until after both data strobes are removed). On a read operation, the slave is required to continue driving the data bus until the data strobes are removed. Careful hardware design is necessary, since the master is free to assert a new address and a new -AS strobe even before it removes the data strobes, or before the slave has removed its acknowledge.

Typical Interrupt Acknowledge Cycle On VME Bus



Typical Read/Write Data Cycle On VME Bus



When the slave detects both data strobes deasserted, then it must stop driving the data bus (on a read operation), and then remove -DTACK or -Berr. At this point, the master may begin the data transfer portion of the next bus cycle. Note that the slave is guaranteed that the -AS, -DS0, and -DS1 strobes are each guaranteed to be deasserted for at least 30 nsec between cycles.

The VME bus definition also supports block transfers for read and write. The array processor, however, does not properly respond to such transfers.

Interrupt cycles are handled in a similar fashion. The array can be enabled to interrupt the host under sequencer control. Jumpers select one of 7 interrupt priority request levels, -IRQ1-7, which the array may use. The VME bus interrupt handler is responsible for arbitrating among all pending interrupts, and if interrupts are enabled at the selected level, then the interrupt handler arbitrates for access to the bus for an interrupt acknowledge cycle. When granted access, it asserts the selected interrupt priority level on address lines A01-A03, asserts -IACK, asserts a read operation via -Write, asserts a long word operation if necessary via -LWord, waits for the signals to stabilize, and then asserts -AS. The slave is guaranteed that -AS arrives at least 10 nsec after these signals are stable. The interrupt handler will also assert -DS0 and/or -DS1 to request an 8, 16, 24, or 32 bit interrupt vector. The data strobes are guaranteed to arrive at least 10 nsec after the address information is stable.

Arbitration for which interrupter responds to an interrupt acknowledge is handled through two mechanisms. First of all, an interrupter will respond to the interrupt handler only if it is requesting an interrupt, and its priority level matches that specified on A01-A03. To arbitrate between interrupt requestors at the same priority level, a daisy chain grant system is used. When the board plugged into slot 1 detects -IACK combined with -DS0 or -DS1, it waits at least 40 nsec, then drives its -IACKout line low. -IACKout on each board is connected to -IACKin on the next board plugged into the backplane. When a board receives -IACKin, but it is not requesting an interrupt, or the interrupt priority level does not match, then it must forward the interrupt acknowledge by driving -IACKout. If the board is requesting an interrupt, and the priority level matches, however, then this indicates that the board should respond to the interrupt acknowledge cycle. It must leave -IACKout deasserted. It then drives the data lines with an interrupt vector code - the array will return a programmable 8 bit code. When the lines are stable, it then asserts -DTACK.

In response, the interrupt handler reads the interrupt vector code, then removes all the addressing and data information and strobes. The array must deassert -IACKout within 30 nsec of -AS being deasserted. When -DS0 is deasserted, the array must stop driving the data lines, then deassert -DTACK. Note that the array must continue to drive the interrupt vector code as long as -DS0 is asserted, even though the master may modify the addressing information as soon as -DTACK is asserted.

4.3. VME Bus Electrical Drive Characteristics

In general, loading and capacitance restrictions on the VME backplane require that each line be connected to only 1 or 2 LS or ALS TTL inputs, and be driven by only one high current S, F, or ALS TTL driver. The toughest specification is that the total capactive loading on any pin must be below 20pF. (It is doubtful that the current array processor design meets this spec.)

Different lines require different drive capabilities. The D00-D31 drivers must supply up to 48 mA; we use 74ALS245A-1 transceivers for this. Receivers on all the standard three-state lines, A01-A31, D00-D31, AM0-AM5, -IACK, -LWord, -Write , may source no more than 700 µA at logic 0, and sink no more than 150 µA at logic 1. (Note that ALS draws 100µA at logic 0, 20µA at logic 1, and our PAL's draw 200µA at logic 0, 20µA to logic 1.) The -IACKout driver must supply 8mA at logic 0 (any TTL driver will do), and -IACKin and -IACKout must draw under 600µA at logic 0, 50µA at logic 1. Open collector lines -IRQ1-7, -DTACK, and -Berr must have drivers capable of sinking 48mA at logic 0; we use a 74S38. Recevers on these lines and on -IACK must source no more than 600µA at logic 0 (400µA for -DTACK or -Berr), and sink no more than 50µA at logic 1.

Conservative design caused us to limit bus loading on each wire to 2 receiver inputs and 1 driver per board.

4.4. VME Address Bus Decoding Hardware

The address decoding circuitry starts by buffering the strobes -AS, -DS0, -DS1, -IACK, and -SysReset through a 74ALS244A. The upper address strobes A19-A23 go directly to a 74ALS520 eight-bit identity comparator, which compares these 5 upper address lines against 5 switches. In addition, jumpers allow checking that A18 or both A18 and A17 equal 1. (Inserting these jumpers restricts the address range to 256K or 128K respectively). The comparator also checks that -IACK is not asserted. This comparator is gated by the -AS strobe. A match, -Valid, indicates that the host is attempting a valid read or write to an address mapped to the array address space. (Note that loose timing restrictions on the release of -AS relative to the release of the address may allow glitches to occur on -Valid at the end of any bus cycle while -DTACK or -Berr are asserted.)

To handle 32 bit addressing modes, another 74ALS520 octal comparator, gated by -AS, compares the highest host address lines A24-A31 against a set of octal switches. If they match, it generates a signal -ExtValid. (This signal may also glitch at the end of any bus cycle.)

PAL #HP1 partially checks for an address hit in the upper 16 bytes of the range. This generates a signal -DoIOMatch if A04-A18 are all equal to 1. This signal is not gated; it is therefore particularly glitchy, and can only be interpreted in conjunction with other strobes.

4.5. Synchronizing the VME Bus to osc

PAL #HP2 is responsible for detecting a VME bus cycle accessing the array address range, and for synchronizing the cycle to the array clock. First, a signal -DS is generated when either -DS0 or -DS1 are asserted. (This is only necessary to reduce the number of product terms per output in the PAL below 8). Action begins when -Valid is asserted, and the VME AM0-AM5 lines indicate that this is a 24 bit user or supervisor data or program access, or when both -Valid and -ExtValid drop and AM0-AM5 indicate that this is a 32 bit user or supervisor data or program access. If this is combined with -DS, and

-LWord is deasserted, and both -DTACK and -Berr are deasserted, then this implies that a new bus cycle is beginning which is accessing the array. On the next rising edge of osc, PAL #HP2 will assert -VMEHit. If all the above is valid, except that -LWord indicates that a 3 or 4 byte data operation is requested, then on the next osc edge, PAL #HP2 will assert LBerr to indicate that a bus error should be generated. An open-collector 74S38 inverts this signal to drive the -Berr line on the VME bus.

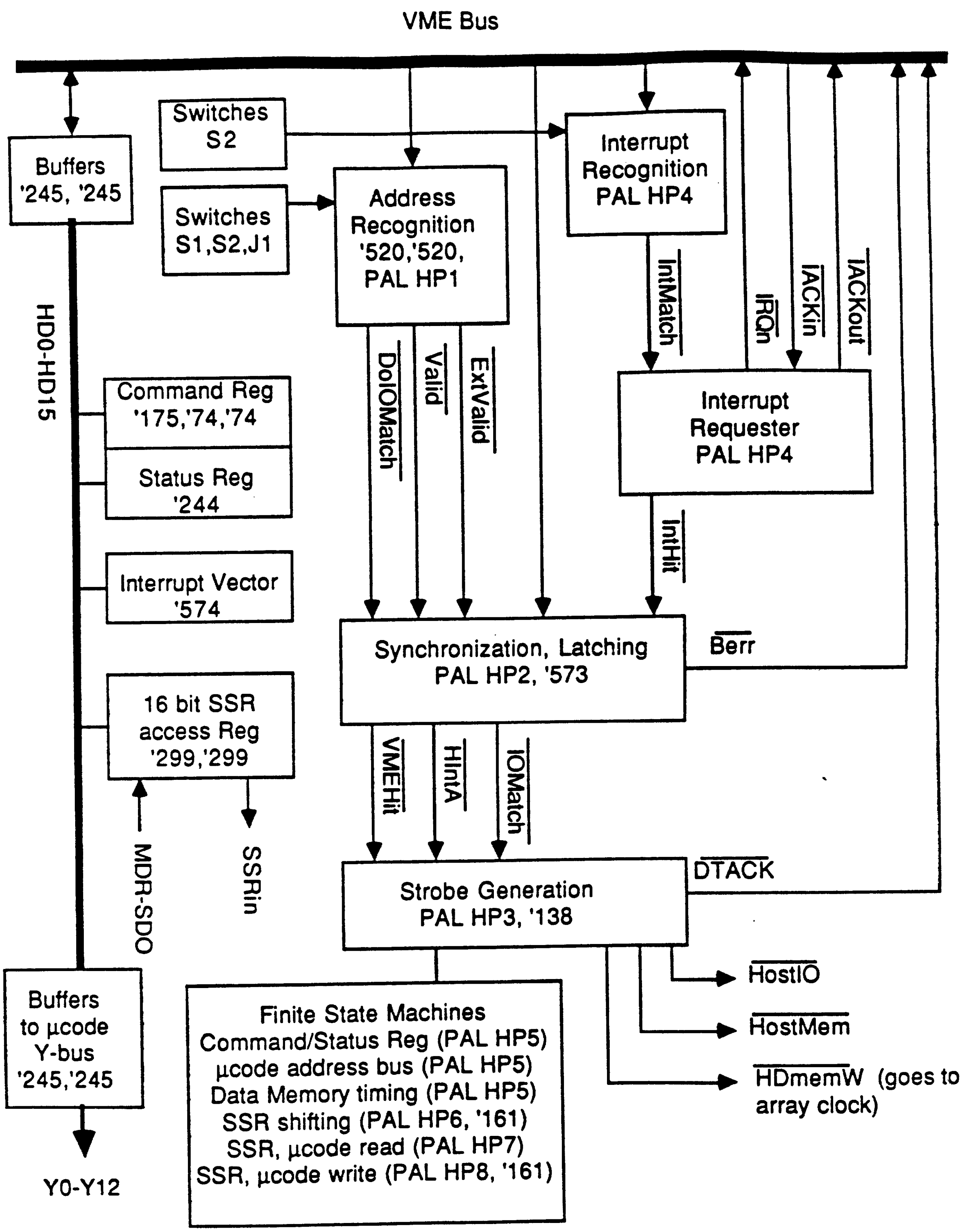
Both -VMEHit and LBerr will be asserted by PAL #HP2 until the next osc edge after both data strobes -DS0 and -DS1 are removed. Once asserted, therefore, both of these signals will remain even if the master changes the addressing information when it receives -DTACK.

When any slave on the VME bus asserts -DTACK or -Berr, the master is allowed to change the address lines and even assert a new -AS strobe on the VME bus. The slave, however, must maintain the proper data on the data bus until both data strobes -DS0 and -DS1 are deasserted. This bus pipelining feature implies that the host interface cannot rely on any address strobes being valid following -DTACK. To assist in proper operation of the interface, therefore, a 74ALS573 octal transparent latch is used to latch the values of certain lines that might change after -DTACK. Address lines A01-A03 are latched, together with the -Write and -LWord strobes, and the -DoIOMatch strobe (which is called -IOMatch following the latch). The register is latched by the signal -VMEHit which is asserted by PAL #HP2 throughout a bus cycle involving the array. Note that there is sufficient time for these signals to stabilize before -VMEHit appears.

Note that -VMEHit is the basic synchronizing strobe for the host interface. Any metastable state difficulties will show up only in this signal. All other FSM's controlling the host interface rely strictly on -VMEHit to begin operation, and all will quit operation when this signal is deasserted. Also note that except where explicitly noted, all chips in the host interface which rely on -VMEHit require that their inputs be synchronized to the osc clock.

-VMEHit also goes to the clock generator PAL #μCP3, causing it to halt the array clock PAL #μCP4 after the current cycle is complete. When the clock stops, line Halt will be asserted. This feature allows the host interface to steal

Host Interface - Block Diagram


VME Bus
Buffers '245, '245
Switches S2
Switches S1,S2,J1
Address Recognition '520,'520, PAL HP1
Interrupt Recognition PAL HP4
HD0-HD15
Command Reg '175,'74,'74
Status Reg '244
Interrupt Vector '574
16 bit SSR access Reg '299,'299
MDR-SDO
SSRin
DoIOMatch
Valid
ExtValid
IntMatch
IRQn
IACKin
IACKout
Interrupt Requester PAL HP4
IntHit
Synchronization, Latching PAL HP2, '573
Berr
VMEHit
HIntA
IOMatch
Strobe Generation PAL HP3, '138
DTACK
HostIO
HostMem
HDmemW (goes to array clock)
Buffers to µcode Y-bus '245,'245
Y0-Y12
Finite State Machines
Command/Status Reg (PAL HP5)
µcode address bus (PAL HP5)
Data Memory timing (PAL HP5)
SSR shifting (PAL HP6, '161)
SSR, µcode read (PAL HP7)
SSR, µcode write (PAL HP8, '161)

time from the array without conflicting with array activity. The clock will remain halted until -VMEHit is deasserted.

PAL #HP2 is also responsible for coordinating the board's response to an interrupt acknowledge cycle. If -IntHit is asserted by PAL #HP4, then an interrupt acknowledge cycle is in progress which involves this board. When -DS0 is asserted, and both -DTACK and -Berr are deasserted, PAL #HP2 will assert -HIntA at the next osc edge, indicating that the board should respond with the interrupt vector. This strobe will be held until -DS0 is deasserted. (This protects against the fact that the -IntHit line may be deasserted after the board returns -DTACK.)

4.6. Developing the Basic Host Read/Write Strobes

PAL #HP3 is responsible for decoding certain strobes controlling the overall behavior of the various FSM's forming the host interface. When this PAL detects -VMEHit, and the Halt line is asserted (indicating that the array clock is stopped), and -IOMatch is asserted, then it outputs the strobe -HostIO indicating that the bus cycle is accessing one of the upper 16 bytes, and therefore involves special handling. The -HostIO strobe enables a 74ALS138 octal decoder, which asserts one of the strobes -HitCmd, -HitPipe, -HitSSR, -HitμmemR, or -HitμmemW depending on whether the low three address bits A1-A3 are 0, 1, 2, 3 or 4 respectively. A hit on the top 6 bytes, address bits A1-A3=5, 6, 7, will cause -Hitnone5, -Hitnone6, -Hitnone7 to be asserted, indicating a hit on a non-existent I/O register in the address range. Because A1-A3, and -IOMatch are latched, these strobes will remain valid without glitching until -VMEHit is deasserted. The -HostIO strobe also enables a tristate transceiver connecting the upper byte of the host data bus to an internal host data bus HD8-HD15.

If PAL #HP3 detects -VMEHit and Halt, but -IOMatch is deasserted, then a data bus cycle is in progress accessing a data memory byte or word location. The PAL will assert -HostMem, which signals the data memory control PAL's that a host cycle is under way. If the -Write line is also asserted, then PAL #HP3 will assert -HDmemW, which signals the clock generator PAL #μCP4 that a host write cycle is in progress. PAL #µCP4 (with #µCP3) is responsible for generating the data memory write strobe -DmemW when -HDmemW is detected together with -VMEHit.

PAL #HP3 is also responsible for generating the strobe -EnLowDbus, which enables a tristate transceiver connecting the low byte of the host data bus to an internal host data bus HD0-HD7. This strobe is generated during any host access to the upper 16 bytes of the address range (-VMEHit and Halt and -IOMatch asserted). It is also generated in response to -HIntA, allowing the output of the interrupt vector register to be forwarded to the VME data bus during a host interrupt acknowledge cycle.

PAL #HP3 also generates a -ClrHIntR signal to clear the host interrupt request flip flop. This clear signal is generated whenever -HIntA is asserted, indicating that the requested host interrupt cycle is in progress. It is also asserted when the host writes a 0 into the HIntEn flip flop in the command register (this happens when LdCmd and HD14=0). Thus a pending host interrupt request is cleared when HIntEn is cleared; this feature may be useful when host interrupts are not desired, but the host would still like to use the HIntR signal as a flag to synchronize its activities with the array.

Finally, PAL #HP3 is responsible for generating the acknowledge for every correct data or interrupt cycle. The various FSM's in the interface generate local acknowledges -CmdDTACK, -PipeDTACK, -SSRDTACK, and -µmemDTACK. When any of these signals is present, PAL #HP3 will assert LDTACK, which is inverted by an open collector buffer to drive the VME -DTACK line.

Note that if metastable states do not occur on -VMEHit, then the outputs of PAL #HP3 and of the octal decoder will be stable well before the next osc edge.

4.7. Interrupt Acknowledge Cycle Synchronization

PAL #HP4 is responsible for coordinating the response to VME bus interrupt acknowledge cycles. The low three address lines A01-A03 are compared with three switches setting the interrupt priority level for the board. (A resistor pack provides pullups for the switches). The PAL also checks that -AS is asserted together with -IACK. If all these are true, then an interrupt acknowledge cycle is in progress at the board's priority level, and the PAL asserts -IntMatch. (This signal is only used internally by this PAL). Note that -IntMatch may glitch at the end of any bus cycle when -DTACK or -Berr are asserted, since the timing of the release of the strobes on the VME bus is not specified.

PAL #HP4 also implements a three state asynchronous finite state machine which is responsible for tracking all interrupt acknowledge cycles. First of all, if the sequencer is requesting a host interrupt (flag HIntR asserted), and host interrupts have been enabled (command register bit HIntEn asserted), then an interupt request IRQ is generated. This is inverted by an open collector buffer, then goes through a set of jumpers onto the appropriate interrupt request priority line. The interrupt request may be connected to any of the VME interrupt priority lines -IRQ1 through -IRQ7, but the line chosen must match the 3 bit code in the switches attached to the interrupt priority level comparator.

If -IACKin is asserted, then some interrupt cycle is in progress and the board must make an immediate decision as to whether this interrupt cycle should be handled by the board. In its default state, PAL #HP4 deasserts both -IACKout and -IntHit. If -IACKin is asserted while -IRQ and -IntMatch are not both asserted, then an interrupt acknowledge cycle is in progress which has either not been requested by this board, or is not at this board's priority level. In either case, PAL #HP4 immediately changes state and asserts -IACKout. The PAL directly drives this VME bus line, since the line simply connects to the next board in the interrupt acknowledge daisy chain, and thus the drive capability is sufficient. -IACKout will remain asserted until either -AS or -IACKin are deasserted. Including -AS in this release process accelerates the board's release of -IACKout at the end of an interrupt acknowledge cycle in order to meet the 30 nsec required release time.

If -IACKin arrives when -IntMatch is valid, and an interrupt request is pending and enabled, -IRQ asserted, then PAL #HP4 changes to its third state and asserts -IntHit. This signal goes to PAL #HP2, which waits until -DS0 is valid (it should be valid before -IACKin) and -DTACK and -Berr are released, and which then asserts -HintA on the next osc edge to gate the interrupt response vector onto the host data bus. -IntHit will remain valid until -IntMatch is deasserted.

To help eliminate difficulties caused by glitches on -IntMatch and by race problems when IRQ is asserted simultaneously with -IACKin, PAL #HP4 gives preference to handing the interrupt to the next board. Thus if both -IACKout and -IntHit are asserted, the FSM will remove -IntHit and leave -IACKout asserted.

4.8. Host Data Bus Buffering Hardware

The 16 VME bus data lines are buffered by a pair of 74LS245A-1 high current transceivers. Ordinarily, these transceivers are disabled and a pair of 4.7K resistor packs pulls the internal data bus lines HD0-HD15 high. The direction of these transceivers is determined by the -Write strobe. The upper byte is enabled ouring host reads or writes to the upper 16 bytes of the address space by -HostIO. The lower byte is enabled during host reads or writes to the upper 16 bytes or during interrupt acknowledge cycles by -EnLowDbus.

A 74ALS175 and a pair of 74ALS74 dual flip flops form the command register, and are loaded by a LdCmd signal from PAL #HP5. These are gated back onto the host data bus by a -EnCmd signal from PAL #HP5 through a 74ALS244A octal driver. The upper 4 bits of the command register are cleared by a VME System Reset, -SysReset. HµIntR is cleared by -ClrHµIntR which is generated by PAL #µCP2 in the µcontrol section (this bit is reset whenever -Reset is asserted, or when a "HµIntR=0" compiler command is executed by the sequencer). HReset is cleared by the internal -Reset signal, which is generated during a Reset clock cycle in the array. SingleStep is cleared by a low value on the clk line.

The interrupt register is formed from a 74ALS574 octal register with tristate outputs. It is loaded by a LdHIntReg signal, and its output is gated back onto the HD0-HD7 bus by an -EnHIntReg signal. Both strobes are generated by PAL #HP5.

The connection between the SSR registers and the host is made through a pair of 74ALS299 octal shift registers. The I/O ports of these registers sit on the HD0-HD15 bus. The register is clocked by a Hclk signal, the tristate outputs are enabled by -HOE, and the -Hshift signal allows the register to broadside load if high, or shift left if low. Data shifted out of the register, SSRin, shifts into PAL #HP8 and #µCP3, which in turn usually shifts it into the least significant microcode pipeline register via DoµSDI and µSDI. Data is shifted into these 74ALS299 shift registers from the MDR register shift output MDR-SDO.

The host data bus also connects to the microcode address bus Y0-Y12 through a pair of 74ALS245 transceivers. These transceivers are enabled by the control signal -EnµAdr generated by PAL #HP5, and their direction is controlled by -Write. Note that if the host is writing to the microcode address bus, then PAL #HP5 always tristates the 1401 sequencer bus with µHold.

4.9. Command/Status Register, µRAM Address Bus, Data Memory

PAL #HP5 is responsible for coordinating host access to the command/status register, the µcode RAM address bus, and the data memory. All the actions this PAL controls last for 2 ticks of osc. On the first tick a strobe is asserted. For almost all the actions, on the second tick -CmdDTACK is asserted to signal the end of the cycle. This acknowledge will be held until the bus cycle is over.

Input -HitCmd is asserted by the octal decoder when the host reads or writes either a byte or a word to the command/status register address. In response to -HitCmd, PAL #HP5 will assert one or two of four different strobes on the next osc rising edge, depending on whether a read or write is in progress (indicated by -Write), and on which byte is being accessed (-DS0 and/or -DS1). Writing the upper byte (-HitCmd and -Write and -DS1) clocks LdCmd, which will load HD8-HD15 into the command register. If bit HIntEn is set to 0, then PAL #HP3 will also assert -ClrHIntR to clear the host interrupt flag

HIntR. Writing the lower byte (-HitCmd and -Write and -DS0) clocks LdIntReg, which will load HD0-HD7 into the interrupt vector register. For both of these strobes, note that delaying the load pulse until one osc period following -HostIO and -EnLowDbus allows sufficient time for the host data bus to stabilize after the data bus transceivers are enabled.

Reading the upper byte (-HitCmd and not -Write and -DS1) enables -EnCmd which will gate the command register onto the host data bus HD8-HD15. Reading the lower byte (-HitCmd and not -Write and -DS0) enables -EnHIntReg, which will gate the interrupt vector register onto HD0-HD7. This strobe is also asserted during a host interrupt acknowledge cycle in response to -HIntA. Note that the delay of one osc period following -HostIO and -EnLowDbus allows time for the host data bus to settle into output mode.

In all of the above cases, -CmdDTACK will be asserted on the next osc edge following the strobe. Again, the delay of one osc period ensures sufficient setup time before the acknowledge. PAL #HP3 will use -CmdDTACK to cause LDTACK, which in turn is driven through an open collector inverter onto the VME -DTACK bus line.

The -HitμmemR line is asserted when the host accesses the "read μmemory" address. The -HitμmemW line is asserted when the host access the "write μmemory" address. Reading from either of these addresses ((-HitμmemR or -HitμmemW) and not -Write) will assert -DoEnHμAdr on the next osc edge. This causes PAL #μCP2 to enable -EnHμAdr to drive the address on the μcode address lines Y0-Y12 to be gated onto the internal host data bus HD0-HD15, and thence onto the VME data bus. (The direction of the tristate buffers on Y0-Y12 is governed by -Write). On the next osc pulse, -CmdDTACK is asserted. The net effect is to allow the host to read the μcode address being generated by the 1401 sequencer. (Note that the sequencer is halted throughout the host access).

Writing into the "read μmemory" address (-HitμmemR and -Write) or writing into the "write μmemory" address (-HitμmemW and -Write) will assert both -DoEnHμAdr and μHold strobes. The latter tristates the sequencer, while the former prompts PAL #μCP2 to assert -EnHμAdr to connect the HD0-HD15 bus to the μcode memory address bus (the direction is given by -Write), allowing the host to drive the µcode address lines. This is just the beginning of the µmemory read/write cycle - the remainder of the action is handled by PALs #HP7 and #HP8. A `-CmdDTACK` signal is *not* generated by PAL #HP5 for this type of cycle (PALs #HP7 and #HP8 worry about the acknowledge).

Finally, PAL #HP5 is responsible for timing out a data memory access. When the host accesses any address in the board's address space except for the upper 16 bytes, PAL #HP3 will assert `-HostMem`. This signal is used by the data memory to immediately switch control of the data memory address and data lines to the host interface. If `-Write` is asserted, PAL #HP3 also asserts `-HDmemW` to force the #µCP4 clock generator PAL to pulse the data memory write line `-DmemW`. This line will pulse low on the `osc` period after `-HostMem`, then will pulse high again on the `osc` period following that. The one cycle delay before `-DmemW` allows time for the memory address lines to settle. On a read data cycle `-HDmemW` and `-DmemW` will remain high. For either a read or a write cycle, PAL #HP5 will assert `-CmdDTACK` two osc periods following `-HostMem`. This should allow enough time for the cycle to complete.

PAL #HP5 will remain in its final state until the strobe which caused it to react disappears. (For read/write cycles, this will occur when the bus master removes `-DS0` and `-DS1`, and PAL #HP2 deasserts `-VMEHit`. For interrupt cycles, this will occur when the bus master removes `-AS` and `-DS0`, and PAL #HP2 deasserts `-HIntA`.) On the next `osc` tick, PAL #HP5 will deassert all strobe outputs, and also release `-CmdDTACK`. This will allow the VME bus to become free, so that the next bus cycle may begin. Note that if the array clock had been halted by `-VMEHit`, it will be restarted on the same `osc` clock tick that removes `-CmdDTACK`. Thus if the array is running, at least one array clock cycle will be interposed between successive host cycles. For maximum host-array transfer rates, the array clock should be stopped by setting the `Run` bit in the command register to 0.

4.10. Reading and Writing the SSR Register

PAL #HP6 is responsible for coordinating reading and writing the Serial Shift Register system. Two 74ALS299 parallel/load octal shift registers sit on the host data bus, HD0-HD15. The shift-out bit SSRin is connected to PAL #HP8, which usually forwards it to D0µSDI, which PAL #µCP3 forwards to the shift-in bit of the microcode register, µSDI. The serial shift register chain continues through all 11 microcode registers, then through PGreg, MAR, and MDR. The shift-out bit of MDR, MDR-SDO, finally returns as the shift-in bit of the 74ALS299's, thus forming a circular shift register. To allow the host to write another 16 bits into this register, PAL #HP6 will load the 74ALS299's with the new data, and then shift the registers by 16 bits. To read the next 16 bits from the register, PAL #HP6 will first shift the registers by 16 bits, and then enable the tristate drivers of the 74ALS299's onto the host data bus. A 4 bit binary counter is used to count to 16. The FSM relies on PALs #HP8 and #µCP3 to forward the serial-out bit from the 74ALS299's to the µcode registers, and on PAL's #HP7 and #HP8 to keep the mode control lines on the SSR registers low.

When PAL #HP6 detects -HitPipe, then the host is attempting a read/write to address IOreg+2. For a write operation (-Write asserted), Hclk is first brought low, then high. This writes the host data bus contents into the 74ALS299's. The count line is also pulled low to reset the 74LS161A binary counter. Next -HShift is asserted to prepare for shifting. A 2 cycle loop then pulses both Hclk and Dodclk low then high, with count held high. Each rising edge shifts both the 74ALS299's as well as the 29818 registers. Dodclk also clocks the binary counter. After 15 rising clock edges, the ripple carry output, Cout, of the counter will go high. PAL #HP6 completes one last Hclk and Dodclk cycle, then returns -PipeDTACK to signal that the operation is complete. PAL #HP3 will assert -DTACK. PAL #HP6 will remain in this state until -VMEHit disappears, causing -HostIO and -HitPipe to disappear. On the next osc edge, -PipeDTACK will be deasserted, and PAL #HP6 will reset.

A read operation is similar, except the registers are shifted 16 places first. When the ripple carry output Cout goes high, the last clock period is completed, and -HOE is asserted, enabling the tristate drivers on the 74ALS299's onto the HD0-HD15 data bus. On the next osc edge, -PipeDTACK is asserted, and both -HOE and -PipeDTACK will be held as long as -HitPipe is asserted. When the bus master releases the data strobes and -VMEHit is deasserted, together with -HostIO and -HitPipe, PAL #HP6 will remove -HOE and -PipeDTACK, and PAL #HP3 will remove -DTACK.

4.11. Handling Non-Existent I/O Addresses

The top 6 bytes of memory do not correspond to any particular actions by the array. It is necessary, however, to return an acknowledge if the host should access these locations, in order to avoid hanging the bus. Therefore, PAL #HP6 will generate `-PipeDTACK` in response to any of the strobes `-Hitnone5`, `-Hitnone6`, `-Hitnone7`, and will hold it until the strobes are removed at the end of the VME cycle.

4.12. Reading Microcode Memory

PAL #HP7 coordinates reading microcode memory and transferring the serial shift register contents into the parallel registers. If the host is accessing address `IOreg+6`, then `-HitµmemR` will be asserted. PAL #HP5 will use the combination of `-HitµmemR` and `-Write` to assert `µHold` and `-DoEnHµAdr` on the next `osc` edge, tristating the µsequencer and putting the microcode address to be read onto the microcode address bus. At the same time, PAL #HP7 will assert `-HµmemR` to enable the µcode RAM's to read. On the next `osc` edge, PAL #HP7 brings `Doµpclk` low, which causes PAL #µCP5 to bring `µpclk1` and `µpclk2` low. On the next `osc` edge, it brings `Doµpclk` high again, which causes PAL #µCP5 to raise `µpclk1` and `µpclk2`, thus clocking the microcode word into the microcode pipeline register. This procedure leaves two full `osc` periods for the µcode RAM outputs to settle. PAL #HP7 also asserts `-µmemDTACK`, which causes PAL #HP3 to assert `-DTACK`. PAL #HP7 will remain in this state until the next `osc` tick after `-HitµmemR` is removed.

4.13. Copying Serial Register Into Parallel Register

PAL #HP7 also handles copying the serial shift register into the parallel register. When `-HitSSR` occurs with `-Write`, then the host is trying to write into address `IOreg+4`, which indicates that the serial register is to be copied into the parallel register. On the next osc edge, PAL #HP7 will drop both Dopclk and -modeA. Dopclk goes to PAL's #µCP5 and #PP03 (on board 2), which force all the parallel register clock lines µpclk1, µpclk2, MARpclk, MDRpclk, and PGpclk low. -modeA goes to PAL #µCP3 which forces the mode lines mode1, mode2, and Mmode high. On the next osc edge, PAL #HP7 brings Dopclk high, which causes PALs #µCP5 and #PP03 to clock the serial register contents into the parallel registers. On the next osc edge, PAL #HP7 releases -modeA and asserts -µmemDTACK. PAL #HP7 then remains in this state until -HitµmemR is removed; at the next osc edge, it then removes -µmemDTACK.

4.14. Writing Microcode Memory

Normally, PAL #HP8 keeps its output -modeB high and lets its output DoµSDI track the serial output SSRin of the 74ALS299 shift registers. This value for -modeB allows the parallel registers to work normally, and causes the shift registers to shift left when clocked. This value for DoµSDI is forwarded by PAL #µCP3 to µSDI, and links the shift registers into a circular buffer.

PAL #HP8 is in charge of coordinating writing to microcode memory. When -HitµmemW occurs with -Write, then the host is writing to address IOreg+8, which indicates that it wishes to write the serial shift register into the microcode memory. On the first osc edge, PAL #HP5 will assert µHold and -DoEnHµAdr to tristate the µsequencer and drive the host data bus onto the microcode address lines. At the same time, PAL #HP8 will assert -modeB and set DoµSDI=1. This high serial value will start rippling through PAL #µCP3 and down the chain of 29818 SSR registers. Because of the slow propagation time and the large number of registers in the chain (16 in all), PAL #HP8 will use a 72ALS161B 4 bit binary counter to pause awhile for the signals to settle. It pulls Count2 low to clear the counter, and also pulls Doµdclk low in preparation. Doµdclk goes to PAL #µCP5 which in turn brings the microcode serial register clocks µdclk1 and µdclk2 low. On the next osc tick, PAL #H8 goes into waiting mode with Count2 high again. After 8 more osc ticks, bit QD on the counter will rise. (The wait therefore lasts about 640 nsec minimum, which is well in excess of the propagation delay of the SSR chain). On the next osc tick, PAL #HP8 brings Doµdclk high. This causes PAL #µCP5 to clock the µdclk1 and µdclk2 lines, thereby causing all the microcode registers to place the contents of the serial shift register onto the input lines. At the same time, PAL #HP8 asserts -HµmemW to enable the µcode memory for writing (the address lines will have been stable for quite a while). We will have to wait while the 29818 input pin drivers power up (90 nsec) and while the RAM's write the data (80 nsec). Thus PAL #HP8 brings Count2 low again to clear the counter. On the next osc edge, PAL #HP8 releases Count2 and the counter starts again. It also brings Doµdclk low again to prepare for the end of the write cycle. It also releases -modeB and lets DoµSDI follow SSRin again. After 8 osc ticks QD goes high, and on the next osc tick we enter the final state. (This wait time is extremely conservative). PAL #HP8 removes -HµmemW to end the write cycle, it clocks Doµdclk to turn off the 29818 input drivers (and also serially shift the SSR), and it asserts -SSRDTACK to signal the end of the activity. It waits in this state until -HitµmemW is released, and then on the next osc tick returns to its initial state.

Note that the operation of PAL #HP8 must not be arbitrarily interrupted in the middle of a microcode write operation. Unfortunately, once the 29818's have been instructed to drive their input lines with the contents of the serial registers, the drivers will not shut off until another dclk pulse arrives. Therefore, the PAL will continue working even if -HitµmemW disappears. In theory, premature disappearance of this strobe should only occur if a VME bus reset occurs while writing to µcode. In this case, the address on the µcode may be incorrect during the write operation. Hopefully, this circumstance should not arise too often.

4.15. Copying the Parallel Registers Into the Serial Registers

PAL #HP8 is also responsible for copying the parallel registers into the serial registers. To achieve this, the PAL waits for -HitSSR and -Write. On the next osc edge, it drops Doµdclk and DoMdclk. PAL #µCP5 and #PP04 (on the second board) will cause all the serial clocks µdclk1, µdclk2, and Mdclk on the registers to drop. PAL #HP8 also drops Count2 to clear the counter. It also asserts -Enpipe. This signal goes to all PAL's controlling the enables of the 29818's, instructing them to disable any competing tristate drivers, and to enable the output of every 29818. This is necessary because the serial registers will load whatever value is on the output pins of these chips, not necessarily the contents of the parallel register. (In the final design of board 2, this signal is not used because all 29818 chips are permanently enabled.) The `-modeB` line will be pulled low, forcing PALs #µCP3 and #PP04 to raise the `µmode1`, `µmode2`, and `Mmode` lines. The `µSDI` line will be forced to 0. In this mode, the 29818's will propagate the zero value through the entire chain.

To allow sufficient time for this to settle, on the next `osc` edge, PAL #HP8 raises `Count2` and waits until the counter counts 8 `osc` periods. (This is a very conservative delay.) Finally, when bit `QD` goes high, PAL #HP8 raises `DoµdClk` and `DoMdclk` to clock the parallel register contents into the serial register. Finally, on the next `osc` edge, PAL #HP8 releases `-Enpipe`, it allows `-modeB` to return high and `DoµSDI` to follow `SSRin`, and it asserts `-SSRDTACK`. PAL #HP3 will assert `LDTACK` in response. PAL #HP8 will remain in this terminal state until the `osc` edge after `-HitSSR` is released.

4.16. Potential Problems in the Host Interface

Timing is very tricky. This interface has been designed to maximize the speed with which the host can read or write data memory. Timing on most other actions, particularly those involving the diagnostic shift register system and accessing microcode memory, is extremely conservative. This is because the 29818's are actually relatively slow on actions different from the usual shifting or broadside parallel loading.

Timing of data memory is particularly tricky. Read operations generally have about 1.5 `osc` ticks for the output to be available. Write operations allow 1 `osc` tick for the address and chip selects to stabilize. The write pulse occurs on the next `osc` tick, and lasts for only 1 tick. The same pulse will enable the tristate drivers to put the data onto the RAM data lines. Thus the data will be available for somewhat less than 1 `osc` period. During normal array operation, the write pulse is designed to release about 1 PAL propagation delay before the clock edge arrives and the chip select and address changes. This should allow enough hold time to ensure a proper data memory write.

Section 5 – µCONTROLLER – DETAILED DESCRIPTION

5.1. Introduction to the Microcontroller

The microcontroller on board #1 is based around an Analog Devices ADSP 1401 sequencer chip. This sequencer is moderately fast, has a deep 64 word stack and register bank, four independent loop control counters, various addressing modes, extensive micro-interrupt support, and conditional testing. Data inputs are supplied from the µconst field in µcode RAM, or from the MDR register. Call, jump, return, and other microinstructions can be conditionally executed based on an internal sign bit or counter value, or based on a test bit which is generated by PAL #µCP1 from the NAND or OR of the bits in MDR, from the most significant (sign) bit of MDR, or from the host controlled flag Hflag0.

The sequencer supplies microaddresses over the Ybus to the microcontrol RAM, which is 8K words, each 88 bits wide. Bits from this µcode RAM are latched in a pipeline register. The register chips are backed by a Serial Shift Register (SSR) system which allows the host to load or read the contents of the pipeline register, or to read or write µcode RAM. Finite state machines built from PAL's in the host interface coordinate the control of this SSR system. Bits from the pipeline register are tied to control lines for the data processors, the data busses, the data memory, and the µconst bus.

A set of PAL's create the necessary control lines for the 1401, and also implement a clock generator. PAL #µCP2 decodes various microcode bits to generate some of the control lines for the 1401. PAL's #µCP3, #µCP4, and #µCP5 form the clock generator. This clock system supports 4 different clock periods, allows the host to smoothly start or stop the clock after a microinstruction completes, coordinates reset activity, and also times out the writing of the data memory. It even disables the memories to save power when the clock is halted or during long clock periods.

The host is capable of monitoring the microcode address bus Ybus, as well as forcing the sequencer to tristate this bus so that the host can directly control the address lines in order to read or write µcode into the µcode RAM.

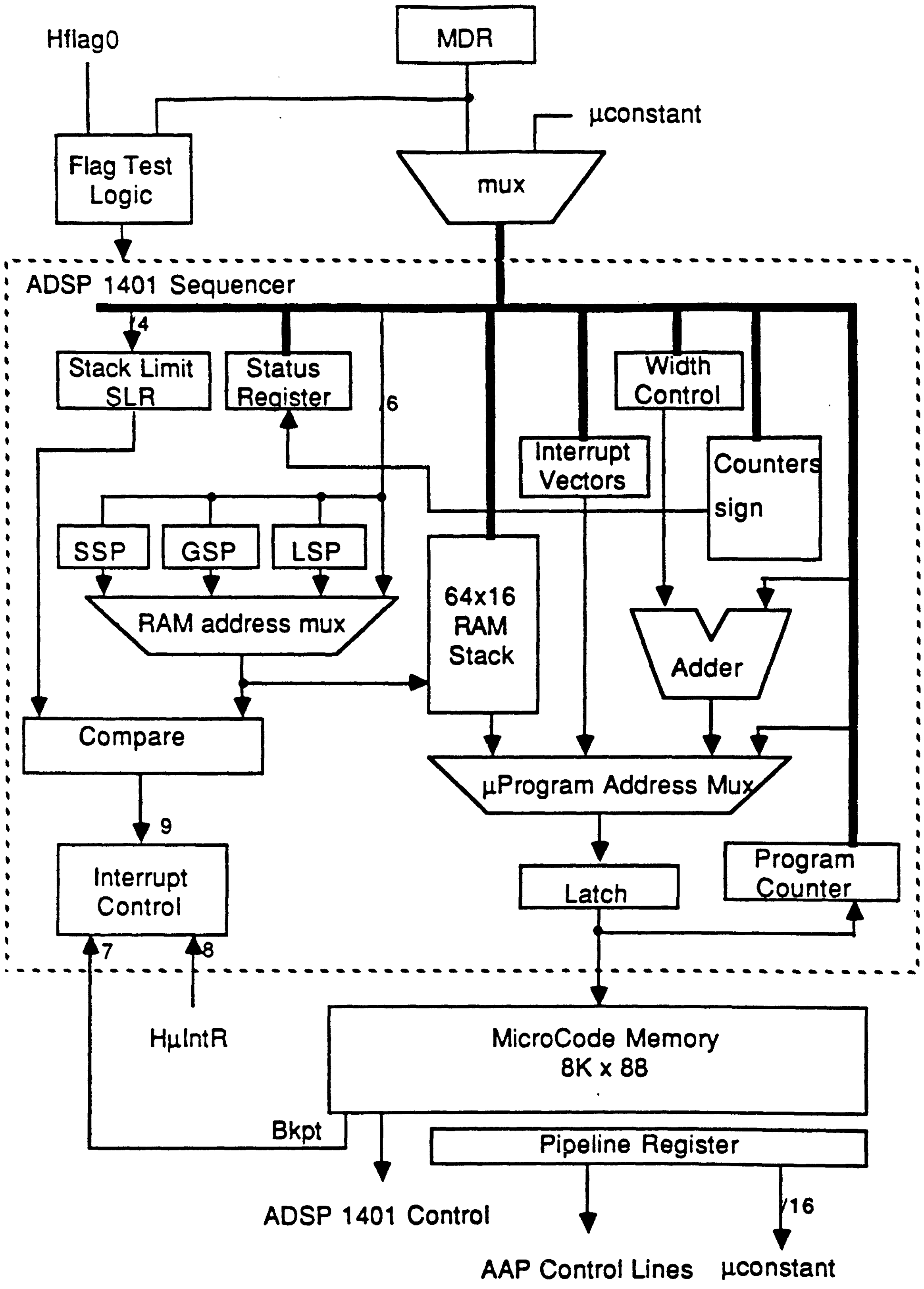
One of the complicating features of the design is that the 1401 has built-in pipeline registers on all data, test, and control line inputs. To avoid double pipeline buffering, it is necessary to bypass the pipeline registers in the µcode RAM and in the signals feeding the data and test inputs. As a result, the µinstruction lines on the 1401 are directly driven by bits from µcode RAM instead of from the pipeline register. µcode RAM bits also directly control PAL #µCP1, which generates a test bit from the *next* contents of the MDR register or from the host controlled bit *Hflag0*. A more complicated issue is the control of the 1401 D inputs, containing data to be used by the sequencer. Because the D inputs are latched, circuitry will supply either the *next* value of `µconst` or the *next* value of MDR to the 1401. This allows the software to place a constant value in the same microcode word which uses the value, or to have the sequencer use the value written into MDR in the previous instruction.

In the following, we will describe this system in more detail. A comment on notation: signals such as `Y0` are positive true logic, whereas signals such as `-EnµAdr` with a '–' in front are negative true logic. Also note that in the Mentor Graphics blueprints, some of the signal names have been abbreviated; for example, `NextMDR` is called `nmdi` ("next MDR input").

5.2. ADSP 1401 Microprogram Controller Chip

The µcontroller is organized around an Analog Devices ADSP 1401 program sequencer chip. This is a 16 bit wide, moderate speed, single chip, 48-pin sequencer built with CMOS, dissipating a measly 1/3 Watt. The chief advantage of this chip is that it is a complete sequencer on a chip with great interrupt capabilities and low power consumption. Major disadvantages are its moderate speed (90nsec cycle time in version J, 70 nsec cycle time in version K), an obscure instruction set which makes high level language implementations difficult, and the limited number of pins which forces external multiplexing of test conditions, input sources, and reset logic.

The sequencer sports a 64 deep stack, dynamically configurable into a subroutine stack, global register stack, and local register stack. Four independent counters control loops. Absolute, relative and indirect addressing



ADSP 1401 MicroSequencer, Microcode Memory, External Flag Logic, Sequencer Input Data (Programming Model)

modes are available. The next address may come from the data input (D-port), the subroutine stack, the register stack, the program counter, or any selected location in the stack RAM. The value of any of the three stack pointers, four counters, or stack contents may be read or written via the D-port. An extensive interrupt system allows 8 external interrupts, plus a counter overflow interrupt, and stack under or overflow interrupts. Interrupts are prioritized and masked on the chip, and the interrupt vectors are contained in registers on the chip. An external flag input pin supports conditional execution. Most conditional instructions can be made to depend on the flag being true or false, the internal sign bit being true, or can be unconditional.

Unusual and unpleasant features of this chip abound. The instruction set is rather irregular, and does not map neatly into a simple high level language viewpoint. Certain conditional instructions can not test the internal sign bit. Either the local or the global register pointer may be accessed at any time, but not both. Four "registers" in RAM may be indexed off the local and global stack pointers and may be used for loading the counter or as jump/call addresses. Register selection, however, is tightly coupled with counter selection, and only the top register on the stack can be read or written directly.

To run a loop N times requires loading the counter with either N-2 or 0x8000+N-2, depending on whether the loop termination instruction is at the beginning or the end of the loop. Putting the loop termination at the start works best if the jump address enters via the D-port. Putting it at the end works best if the jump address is in a register. Complications involving the use of loop counters force the `aapcompile` compiler to only use counter 0 in the "loop()" program construct. Before each loop, the previous value of counter 0 is saved, and the counter is restored after the loop exits. Setting up a loop requires about 3 microinstructions of overhead, and terminating the loop requires about 2 microinstructions of overhead; the compiler attempts to merge these extra instructions into surrounding AAP microcode in order to disguise this large amount of loop overhead.

The instruction at location 0, as well as the first instruction of any interrupt service routine, must be a continue. (Admittedly, there are excellent hardware reasons for this). A Reset pulse must last for at least 3 clock ticks. Although the

1401 directly supports writing into microcode RAM, this feature would be most useful if part of the microcode were in ROM.

5.3. Clock System for the ADSP 1401

The clock for the sequencer and the array is generated by PAL's #µCP3, #µCP4 and #µCP5. PAL #µCP3 synchronizes various `Run` and `Reset` signals from the host interface to the board's high speed `osc` clock signal. It produces signals `-Go` and `-DoReset`, which request the array to Run or to perform a Reset operation. PAL #µCP4 is a finite state machine which actually produces the fundamental clock cycles. When in Run mode, this PAL can produce 4 different length clock cycles on the `clk` line (depending on microcode bits `SelClk0`, `SelClk1`.) It also outputs an `-Enmem` strobe to enable the microcode and data memories (this saves power during long clock cycles or when the array is halted). It also outputs a data memory write strobe `-DmemW` during array or host writes to data memory. In Reset and Halt modes, it outputs appropriate `-Reset` and `Halt` signals.

The `clk` signal from PAL #µCP4 passes to PAL #µCP5, which developes a variety of clocking signals for the sequencer, the various PAL's, and the microcode pipeline registers. Signals from #µCP4 also go to board #2, where PAL's develop more clock signals for the MDR, MAR and PGreg registers. The sequencer itself is clocked by `µclk`. In a typical clock cycle, this signal goes high for one `osc` period, then goes low for 1 to 10 `osc` cycles. When the clock generator is halted, `µclk` will be kept high.

More details on the clock generation system will be found in a later section.

5.4. Y-bus, Tristate Control of the ADSP 1401

Wiring of the micro-address bus `Ybus0-Ybus15` is straightforward. The lower 13 bits drive the address lines on the 11 microcode RAM chips. The microcode address bus is shared with a pair of 74ALS245A octal transceivers connected to the host data bus HD0-HD15. This allows the host to sample the current microcode address by asserting -DoEnHµAdr, which PAL #µCP2 interprets to assert -EnHµAdr. (The direction of the transceiver is controlled by the host read/write line, -Write.)

The host can also control the microcode address when reading or writing microcode memory. Accessing locations IOreg+6 or IOreg+8 will first cause -VMEHit to be asserted, which will stop the sequencer clock. (This is handled by PAL's #µCP3 and #µCP4). µclk will be left high. On the next osc tick after the clock halts, PAL #HP5 will assert the µHold line and -DoEnHµAdr, causing PAL #µCP2 to raise the TTR line, thus causing the 1401 to tristate the Ybus. After TTR is high, PAL #µCP2 will assert -EnHµAdr, thus enabling the host data bus HD0-HD15 to drive the microcode address bus Ybus0-Ybus12. When the host is finished and the VME bus cycle completes, PAL #HP5 will release µHold and -DoEnHµAdr, and PAL #µCP2 will release -EnHµAdr and TTR. This strategy minimizes the overlap between the tristate drivers on the Ybus.

5.5. Micro-Instruction Control Lines for 1401

A major peculiarity of the 1401 is that the microinstruction control lines, together with the interrupt, flag and the data input lines, are latched into transparent latches inside the 1401 at the beginning of the µclk period, when µclk goes high. Inputs on all these lines must therefore be stable well before the clock edge.

This unusual feature has major implications for the design of this system. The micro-instruction latch functions as a pipeline register. Therefore, the microinstruction bits that feed the 1401 and associated circuitry, µDSel, µI0-6, Bkpt, and µS0-1, are taken directly from the microcode RAM, before the pipeline register.

5.6. D-port of the 1401

Wiring of the 1401 data port is greatly complicated by the latching, bidirectional behavior of this port. During the first part of the clock cycle, while µclk is high, the D-port may function as an output port. On "pop stack" instructions, for example, the value at the top of the stack will appear on the D-port at this time. On the second half of the clock cycle, while µclk is low, the D-port will act as an input port. The value read on this port at this time, however, will not be used until the *next* clock cycle. To cope with these features, our design latches the outgoing value of the D-port when µclk is high. Also, during the second half of the clock cycle, either the *next* value of the MDR register or the *next* µconst value will be fed into the D-port. This allows using either calculated values, values read from data memory, or constants when loading counters, registers, stack, or choosing a next address. Using the *next* values simplifies the programming task, since it removes an extra pipeline level from the design. Pushing the value of MDR onto the sequencer stack, therefore, will push the value that was loaded into MDR as recently as the previous instruction.

On the first half of the clock cycle, when µclk is high, the sequencer may output a value on the D-port. This value must be captured and held available during the second half of the clock cycle so that it may be written into the MDR register, if so desired. A pair of 74F543 transparent latch transceiver chips is used for this purpose. The input lines of the MDR register connect to the B side, while the D-port of the sequencer connects to the A side. Back to back transparent latches connect the two sides. During the first half of the clock cycle, PAL #µCP5 outputs both a µclk and a -µclk signal. These are exactly opposites, and their transitions occur almost simultaneously. When -µclk goes low during the first half of the clock cycle, the A-to-B latch in the 74F543 is enabled, so that it passes the value being output on the D-port. During the second half of the clock cycle, -µclk will go high, and the latch will hold this value. If the -EnSeqMDR signal is asserted during this cycle by PAL #PP08 on the second board, then this latched value will drive nextMDR into the MDR inputs, so that the MDR register will load this value on the next rising edge.

On the second half of the clock cycle, when µclk goes low, the D-port becomes an input. µDSel=0 implies that the value to be input on the D-port is the micro-constant; µDSel=1 implies that the value is the contents of MDR. If it is to be a constant, then for programming convenience, the value should be that stored in the same microcode word which holds the jump or load instruction using the constant. Since µI0-5 is taken directly from the RAM's, the constant should also be taken directly from the RAM's, not from the pipeline register. Thus, a pair of 74F244 octal drivers connect the Nextµconst field to the D-port; these are enabled by the -EnµcSeq control line from PAL #µCP2, which is pulled low when µclk is low and µDSel=0. Note that -EnµcSeq drops low one PAL delay after µclk drops low; this should allow sufficient time for -µclk to rise and latch the output value from the D-port, and also allow time for the D-port to begin disabling its own tristate drivers.

If the value to be used on the D-port is the contents of MDR, then the situation is more complex. It would be convenient to allow the programmer to calculate or retrieve a value into MDR, and then load it into a counter, register, or jump to this address in the very next microinstruction. For this to be possible, however, the D-port input latch must be loaded with the same value that is being loaded into MDR. If MDR is not being loaded on this clock cycle, then the D-port input latch must be loaded with the latest value of MDR.

To accomplish this, we use the B-to-A transparent latch in the 74F543. During clock cycles in which MDR is being loaded, the -LdMDR signal is asserted, and when clk goes low, PAL #µCP3 will assert -MDRpclk. This makes the B-to-A latch transparent, passing the MDR register inputs directly into the 1401 D-port. At the next rising clock edge -MDRpclk goes high at the same time that the MDRpclk signal generated on board #2 clocks MDR. As a result, the D-port, the 74F543 and MDR all get loaded with the same new value for MDR. During clock cycles in which MDR is not loaded, then the -LdMDR signal is not asserted, and PAL #µCP3 will not assert -MDRpclk. During these clock cycles, the 74F543 B-to-A latch holds its current value, which is also the same value held in MDR, and loads it into the D-port.

If µDSel=1, then on the second half of the clock cycle, when -µclk goes high, PAL #µCP2 will assert -EnMDRSeq. This enables the 74F543 to drive the sequencer D-port. If MDR is being loaded on this cycle, the B-to-A latch will be transparent (PAL #µCP3 is asserting -MDRpclk), and the D-port will load the same value that MDR will load. If MDR is not being loaded on this cycle, the B-to-A latch drives the current value of MDR into the D-port. Note that `-EnMDRSeq` is asserted one PAL delay after `-µclk` goes high, thus allowing time for the data output by the sequencer on the D-port to latch in the A-to-B latches of the 74F543.

In effect, the 74F543 is a local copy of MDR, and the transparent feature is used to bypass a level of pipelining when MDR is loaded in order to avoid extra pipeline delays. On the second half of the clock cycle, the latch output will always be the value that MDR will contain after the rising `µclk` edge. A potential problem is that, on power-up, the MDR register and the 74F543 latches will contain different data. This should be no problem provided that MDR is always loaded before its value is used.

Note that timing of the strobes controlling the D-port is critical. The `-µclk` signal must go back high at almost the same time as `µclk` goes low, so that the 74F543 A-to-B latches correctly grab the sequencer's output. The `-EnµcSeq` and `-EnMDRSeq` strobes must be delayed until 1 PAL propagation delay after `-µclk` goes high, so that they do not start driving the D-port until the latch has safely grabbed the data and the sequencer stops driving the bus. Thus PAL #µCP2 conditions `-EnµcSeq` and `-EnMDRSeq` on `-µclk` going high so that overlap is avoided. A drawback of this scheme is that at the start of the clock cycle, `µclk` will go high, enabling the sequencer's D-port drivers, but `-EnµcSeq` and `-EnMDRSeq` will not disable until 1 PAL delay later. This may cause some overlap of the tristate lines, although it should be minimal because the D-port drivers are relatively slow to turn on.

5.7. Flag Input for Conditional Execution

Four different conditions can be selected for an external test condition to control conditional jumps, calls, returns, and so forth. These are:

0) The NAND of all the bits in the MDR register

1) The OR of all the bits in the MDR register

2) The most significant bit in MDR, `MDR15`

3) The value of the host flag Hflag0

The microcode control bits µS0-µS1 determine which of these conditions will be chosen. Because the flag input is latched when µclk goes high, we must be careful to calculate the appropriate function of MDR using the *next* value of MDR rather than the current value. To simplify this, the NextMDR value from the inputs to the MDR register is passed through a pair of 74F573 transparent latches into PAL #µCP1. If the MDR register is being loaded on this cycle, then an inverted copy of the MDR clock, -MDRpclk, generated by PAL #µCP3, will acquire and pass NextMDR on the latter half of the clock cycle when µclk is low. If the MDR register is not being loaded on this cycle, then -MDRpclk remains high, and the last value of MDR will be trapped in this latch to feed PAL #µCP1. In either case, on the latter half of the clock cycle, the output of these latches will be the *next* value of MDR.

PAL #µCP1 is a 24 pin high speed PAL. It is fed all 16 bits of the next MDR value, together with Hflag0, and the select lines µS0-µS1. Note that these select lines are driven directly by the microcode RAM, before the pipeline register. This PAL will compute the appropriate function of the next MDR value or Hflag0, and forward the result to the flag input of the 1401.

Note that this arrangement has been carefully designed to minimize the propagation delay from NextMDR to the 1401, and from the microcode RAM to the 1401. Both sets of paths contain at most 2 chip delays. A substantial difficulty is that we have 3 independent copies of MDR (one on the array processor board, one in the 74F573's, and one in the 74F652's). This makes debugging more difficult, because the host will not be able to load all copies of MDR directly. It also means that the three copies will power-up with different values.

5.8. Micro-Interrupt System

One of the major strengths of the 1401 is its sophisticated interrupt capabilities. Unfortunately, these are mostly unnecessary in this array processor. The host HµIntR flag is connected to the EXI4 external interrupt pin, while the breakpoint bit in microcode memory, Bkpt, is connected to the EXI3 external interrupt pin. Pins EXI2 and EXI1 are simply connected to ground.

Interrupts can be independently masked and enabled inside the 1401, and the interrupt vectors are also stored inside the chip. When the host sets HµIntR, it could potentially cause both a level 8 and a level 4 interrupt. Only one of these should be enabled by microsoftware. Because there is no external micro-interrupt acknowledge pin, there is no way to clear the HµIntR flag automatically when the micro-interrupt occurs. Either the host or the microcode must explicitly clear this bit. The Bkpt bit in RAM will cause both a level 7 and a level 3 micro-interrupt. Note that this bit is taken directly from the RAM, and will not be stable until well in the second half of the clock cycle when µclk is low. Therefore, only the level 3 interrupt is legitimate and should be enabled. The interrupt handler routine for the breakpoint will be called directly after the sequencer has executed the micro-instruction containing the Bkpt bit.

Level 0 interrupts, caused by counter overflow, should not be needed in this product. Level 9 interrupts, caused by stack under or overflow, may be useful.

Note that the first instruction in an interrupt handler must be "Continue". Note that no external interrupt sources can cause level 1, 2, 5, or 6 interrupts.

5.9. Sequencer Flags

The sequencer is capable of setting a HIntR flag to request an interrupt from the host processor. This flag may be set or cleared on any micro-instruction for which the upper 3 instruction bits are µI4=µI5=µI6=0. (The continue instruction has this pattern, for example). None of these instructions test the flag line, and therefore they do not use the select lines µS0, µS1. Thus PAL #µCP2, in conjunction with half of a 74LS74 D flip flop, will adjust this Host interrupt request flag HIntR when it detects one of these instructions, together with bit µS1=1. If µS0=0 then the flag will clear; if µS0=1 then the flag will set. If host interrupts have been enabled by the host setting the HIntEn flag, then a host interrupt will be generated. When the host acknowledges the interrupt, the

-HIntA signal will clear the HIntR interrupt request. A Reset cycle, with -Reset asserted during a rising edge on µclk, will also clear this flag. HIntR will also clear if the host sets HintEn=0 in the command register.

Another special capability is that the sequencer can clear the host microinterrupt request flag HµIntR. When µI4=µI5=µI6=0, and µS1=0 and µS0=1, then PAL #µCP2 will pulse the -ClrHµIntR line on the last half of the clock cycle, thereby clearing this flag in the host command register. -ClrHµIntR will also be brought low when -Reset goes low.

Ordinarily, if neither of these flags is to be adjusted, the programmer should ensure that when µI4=µI5=µI6=0 then lines µS0=µS1=0 (the compiler handles this automatically.) Another point to notice is that PAL #µCP2 actually looks at the microinstruction control bits ThisµI4, ThisµI5, ThisµI6, ThisµS0, and ThisµS1, which are derived *after* the pipeline register (the sequencer looks at the values of these bits *before* the pipeline register. The reason is that the 1401 sequencer effectively latches these bits (or bits such as the flag which depend on these) on the rising µclk edge, and thus does not actually use them until the *next* clock cycle. So that PAL #µCP2 remains synchronized with the sequencer, it is necessary to use the control bits from the pipeline register.

5.10. Clock Generator

PAL's #µCP3, #µCP4, and #µCP5, together with PAL #PP03 on board #2, implement a finite state machine which controls the array clock, enables micro-control and data memories, and synchronizes Reset cycles and host activity to the system clock. If either Run or SingleStep are high, and the host interface is not asserting -VMEHit, then PAL #µCP3 will assert -Go. Ordinarily, if -Go is deasserted (high), PAL #µCP4 is in the "Halt" state, with the Halt output line high, and the -DmemW, -Enmem and clk lines also high. Bringing -Go low allows the clock generator to enter the "Run" state. The Halt line is first brought low on the next rising osc edge. The clk line will then be brought low and held low for a period determined by the two SelClk0 and SelClk1 lines from the microcode pipeline registers. clk will then come high for one period of osc. PAL's #µCP3 and #µCP5, and PAL #PP03 on board #2, translate this rising edge into rising clock edges on the sequencer, various PAL's, the processor chips, the microcode pipeline registers, and sometimes on the PGreg, MAR and MDR registers. SelClk0 and SelClk1 will be loaded on this edge with a new clock length, and the next period will be chosen to have the appropriate length. The clock generator is currently programmed with the following clock periods:

SelClk1,0 = 0,0	2 osc periods
SelClk1,0 = 0,1	4 osc periods
SelClk1,0 = 1,0	8 osc periods
SelClk1,0 = 1,1	11 osc periods

The µclk signal is used to reset the SingleStep flip flop. Thus, if the Run flip flop is cleared and the clock generator is halted, when the host sets SingleStep, it will force a single µclk cycle, the SingleStep flip flop will clear, and the clock generator will return to Halt state.

The -Enmem signal is used to enable micro-control and data memory. This signal will drop low 2 osc periods before the rising clk edge, and will remain low 1 osc period after the rising edge. PAL #µCP5 converts this signal into the microcode RAM chip enable strobes -E1 and -G. At the fastest clock rates, -Enmem will remain low and the RAM's will remain enabled for reading. During slow clock periods, however, or when the array is halted, -Enmem will go high, disabling the RAM's when they are not needed, and thus conserving power.

-Enmem is also used by the chip select PAL's #MEP1 and #MEP2 in the data memory. To conserve power, data memory RAM's are only enabled when -Enmem is asserted.

The -DmemW strobe controls the write enable on the data memory RAM's. It is normally held high. If -MemW is low, however, then a data memory write operation is required on this clock period. Thus PAL #µCP4 will assert -DmemW during the last osc period before the rising edge of clk. The duration of this write pulse is thus exactly one osc period, and it occurs at the end of the clock cycle. Timing of the pulse is critical. The rising edge of -DmemW and the rising edge of clk should occur nearly simultaneously. Both -DmemW and clk pass through similar buffer gates. The write strobe then connects directly to the write enable pins on the data memory RAM's. The clk signal, however, must pass through a layer of PAL's before the µclk, -µclk, µpclk1, µpclk2, MDRpclk, -MDRpclk, MARpclk, and PGpclk clocking signals occur. Thus the -DmemW write pulse should be deasserted at least 1 PAL delay before the address lines on the RAM's will change. Note that the -DmemW strobe is also used to enable tristate drivers to present the data to the data memory RAM's, and that therefore the data to be written will only be available at the RAM data pins for somewhat less than 1 osc period.

PAL #µCP4 also coordinates the data write strobe timing for host accesses to data memory. During a host write operation, -VMEHit will be asserted, thus causing the clock generator to go to Halt mode. One osc tick after reaching Halt (or state Halt2), PAL #HP3 will assert -Hostmem to give the host control over the data memories. It will simultaneously assert -HDmemW if a write operation is required. On the next osc tick, PAL #µCP4 will respond by bringing -DmemW low for one tick, then will bring it high again. This allows sufficient time for the host address to stabilize on the bus before -DmemW appears. PAL #HP5 will assert -CmdDTACK to acknowledge the bus cycle on the osc tick following -DmemW.

Clocking strobes are generated from clk by three different PAL's. To increase the drive, and also delay the signal to match -DmemW, the clk signal passes through a buffer gate. It then goes to PAL #µCP3, which generates -MDRpclk, PAL #µCP4 which generates µclk, -µclk, µpclk1, and µpclk2, and to PAL #PP03 on board #2 which generates MDRpclk, MARpclk and PGpclk. The delays through all of these PAL's must be very close so that all these clocks occur almost simultaneously, and no undesirable race conditions occur.

If the -Go strobe rises high while the clock generator is in "Run" mode, the present clock cycle will continue running until the clk and -DmemW lines go high again. When clk goes high, the Halt line will also go high. If -Go remains high on the next osc period, then PAL #µCP4 will turn off -Enmem. If -Go rises high while the clock generator has clk high, then PAL #µCP4 goes immediately to Halt mode, with clk held high, Halt asserted, and -Enmem deasserted. The timing is carefully designed to minimize the delay between -Go rising high, and Halt being asserted. This is important, because the host will ask the clock generator to stop the clock before every host access by asserting -VMEHit, and will not continue until PAL #µCP4 asserts Halt.

"Reset" mode will be entered if the VME bus signal -SysReset drops low, or if the HReset flip flop in the command register is set by the host. These two signals are combined in PAL #µCP3 to form the signal -DoReset. This line goes to the clock generator, PAL #µCP4. If the clock generator is in "Run" mode, and -Go is asserted, then the present clock period will complete, then with clk held high, the -Reset line will drop at the next osc edge. Three periods of µclk will then follow, each 2 osc periods long, with -Reset held low. (This is sufficient to completely reset the sequencer). The µclk signal is tied to the clear line on the HReset flip flop in the host command register. Therefore, if this reset were caused by the host setting this flip flop, at the conclusion of 3 Reset clock cycles, -HReset will be deasserted and -DoReset will also be deasserted. PAL #µCP4 brings -Reset high again, then resumes normal "Run" mode.

If -DoReset occurs when the -Go line is deasserted, then PAL #µCP4 will finish the current clock cycle, assert Halt, and then on the next osc edge, will assert -Reset and deassert -Enmem. This is "Reset/Halt" mode. (Note that a VME System Reset will clear all host command register flip flops, including the Run bit, and thus a VME System Reset will place the array into "Reset/Halt" mode). When -Go is asserted again, then PAL #µCP4 will deassert Halt, assert -Enmem, and go to the usual "Reset" mode. (Three clk periods will follow with the -Reset line held low.)

If the clock generator is in Halt state, with the host interface flag Run deasserted, then if the host sets both HReset and SingleStep, the clock generator will produce three µclk periods with -Reset held low throughout, and then it will return to Halt state with both HReset and SingleStep cleared.

One major purpose of the `-VMEHit` signal is to allow the host interface PAL's to request the clock generator to halt before allowing the host further access into the internal state of the array.

5.11. Writeable Micro-Control Store RAM

The writeable micro-control store contains 8K words of 88 bits each. It is formed from 11 MB8464 static CMOS RAM chips, each 8Kx8. Each of these chips can consume up to .33 Watts in active mode; keeping them in standby mode by deasserting the `-µE1` line will reduce power consumption to .05 Watts. The 13 address lines are driven by the low 13 bits of the sequencer's `Y-bus`, or by the 74F245A transceivers from the host data bus.

Enable strobe `E2` is permanently tied high. Enable strobe `-µE1` and the output enable strobe `-µG` are driven by PAL #µCP5. `-µE1` will be asserted if `-Enmem` is asserted, or if the host is accessing microcode memory with the `-HµmemR` or `-HµmemW` strobes. The output enable strobe `-µG` is asserted if `-Enmem` is asserted or if `-HµmemR` is asserted. The write enable strobe is driven directly by the `-HµmemW` strobe. (11 CMOS loads should not overtax the PAL or the sequencer pin drivers).

The data lines from the micro-code memory drive the inputs of AM29818 diagnostic pipeline register chips. To reduce the drive load, duplicate control lines for the parallel clock `µpclk1`, `µpclk2`, the serial clock `µdclk1`, `µdclk2`, and the mode `µmode1`, `µmode2` are generated by PAL's #µCP3 and #µCP5, and each line drives only half the pipeline register chips. Normally, PAL #µCP5 outputs parallel register clocks `µpclk1` and `µpclk2` to follow `clk`. These parallel register clock lines also pulse if the host interface pulses the `Doµpclk` or `Dopclk` control lines.

The tristate outputs of these register chips drive the control lines for the rest of the array. These registers are always enabled. Control lines affecting the sequencer or its associated PAL's, drivers and latches are drawn directly from the microcode RAM, bypassing the pipeline register, since the 1401 sequencer has pipeline registers built in.

Each AM29818 register dissipates .5 Watt. The advantage of these chips, however, is that behind each parallel register is a shadow serial register. These serial registers are organized into a 288 bit circular shift register which starts in a pair of 74LS299 shift registers in the host interface, passes through logic in PALs #HP8 and #µCP3, , then runs through the 11 microcode pipeline registers (LSB to MSB), then PGreg, MAR, MDR and finally back into the 74LS299's. The host can load data into this serial shift register or read data from it under control of PAL #HP6. It can transfer data between the serial and parallel registers via PAL's #HP7 and #HP8. It can write the serial register into a selected micro-code location via PAL #HP8, or can read a selected micro-code location into the parallel pipeline register via PAL #HP7. Serial data enters the microcode pipeline via the `µSDI` line, and exits via the `µSDO` line. The `mode1`, `mode2` and `µdclk1`, `µdclk2` lines are used together with the usual parallel clock lines `µpclk1`, `µpclk2` to control these registers. PAL's #µCP3 and #µCP5 combine signals from various host interface PAL's with `clk` from the clock generator PAL #µCP4 to produce these signals. PAL #PP03 on board 2 similarly produces the control lines for PGreg, MAR and MDR.

5.12. Microcode Bit Assignment (88 total)

Name	Function
Sequencer Control Lines (13)	
`Bkpt`	Breakpoint interrupt request
`µI6`	Microcontroller instruction bus
`µI5`	...
`µI4`	...
`µI3`	...
`µI2`	...
`µI1`	...
`µI0`	...

µDSel	Select constant or MDR input to sequencer
µS1	Flag Select/HIntR flag control
µS0	...
SelClk1	Clock length selection
SelClk0	...

AAP Chip Control Lines (41)

I17	DC pin control
I16	...
I15	Internal/External direction control
I14	Carry mux control
I13	...
I12	OD1 mux selector
I11	...
I10	Bypass and ripple data control
I9	...
I8	Select ALU arg A
I7	...
I6	RS latch control
I5	RS source control
I4	Direction Control data path 2
I3	...
I2	Direction Control data path 1
I1	...
I0	...
WERS	RS latch enable
-WEC	Carry reg enable
SWEB	Conditional B write enable if S=1
WEB	Conditional B write enable if S=0
WEA	A write enable
F3	ALU Function Select
F2	...
F1	...
F0	...

S2	Select flag address
S1	...
S0	...
A4	A address
A3	...
A2	...
A1	...
A0	...
B5	B address
B4	...
B3	...
B2	...
B1	...
B0	...

Peripheral PE Array Control Lines (1)

-CarryIn	Carry Input

Memory Control Lines (2)

-MemW	Read/Write Control
SelAdr	Select data memory address source

Data Path Routing Control (14)

-LdPGreg	Load page register
-LdMDR	Load MDR
-LdMAR	Load MAR
D.Out2	MDR bus driver select
D.Out1	...
D.Out0	...
D.In2	MDR bus receiver select
D.In1	...
D.In0	...

`A.Out2`	MAR bus driver select
`A.Out1`	...
`A.Out0`	...
`A.In2`	MAR bus receiver select
`A.In1`	...
`A.In0`	...

Constant Field (16)

`µconst`	16 bit constant field

Section 6 - OVERVIEW OF PROCESSORS, DATA MEMORY, DATA BUSSES

6.1. Processor Array

Board #2 contains a 16 by 16 array of single-bit processors, two major and one minor 16-bit bus systems for shuffling data around, and a 128K to 512K byte data memory. The processor array is built from four OKI AAP chips, each organized as an 8 by 8 array, and arranged in two rows of two chips each. 16-bit data can be fed into or out of any side of the array on the DC pins (carrying data path 1 information), and/or from the top and bottom of the array on the SD pins (carrying data path 2 information). Data can be transferred up, down, right, left or in any diagonal direction through the array on data path 1 using either synchronous or asynchronous data transfer. Data path 2 supports only synchronous data shift in an up or down direction.

6.2. Data Memory, MAR, MDR, PGreg

The data memory contains between 128K and 512K bytes of fast (55-70 nsec) static CMOS RAM. 20 address lines access the memory, and 16 bits are fetched on every clock cycle. The low 16 bits of addresses are supplied from either the `µconst` field in the microcode word, or from the `MAR` register. The upper 4 bits come from `PGreg`, a page register, which can be loaded by a microcode constant, or by a computed or stored value. The paging constraint implies that only 32K bytes can be accessed at a time without changing the page register. This matches well to 128x128 images with up to 16 bits per pixel, since each such image occupies 32K bytes. Data fetched or written into data memory by the array always passes through the `MDR` register. (The host computer can access data memory directly, however.)

The most unusual feature of the data memory system is that the memory is nibble (4 bit) addressable, fetching or writing 16 consecutive bits starting at any 4 bit boundary in the memory. This allows fetching words from memory which overlap by 25%, 50%, or 75%, and is highly useful for fetching and writing overlapping subimages stored by bit-planes or by pixels. A substantial amount of rotation, alignment, and address computation circuitry is dedicated to supporting this feature. (Propagation time through this circuitry is about equal to the propagation delay through the RAM chips.)

The purpose of the MAR and MDR registers is to facilitate pipelined transfers of address and data information between the processor array and the data memory. They allow the memory and the processor array to work in parallel, thus approximately doubling the speed of the system. The PGreg register is required because the array and the microcode can only support computation of 16-bit addresses.

6.3. MDBS, MABS, PFBS Data Busses

A major difficulty in the design of the array is that a large amount of data needs to flow in every clock cycle among all four sides of the array (both data paths), and the data memory pipeline registers. Data output on one side of the array may need to be transferred not only to memory, but also to another side of the array, or even may need to be input on the other data path on the same side of the array. Micro-constant data from the microcode may also need to be input into any side of the array, or loaded into MDR, MAR, or PGreg.

Two major and one minor bus system are used to transport data between the processor array and the data memory. The two major busses are called mabs and mdbs, because they respectively feed the MAR and MDR registers in the data memory. Each bus connects through transceivers to the DC pins on all four sides of the array, and to the SD pins on the top and bottom. Up, down, right, left, and diagonal data transfers are supported between chips and at the array boundaries. The µconst field also can drive either bus. The MAR and PGreg registers may be loaded from mabs, while MDR may be loaded form mdbs. MAR can drive mabs, while MDR can drive either bus. Data output from the 1401 sequencer can be routed into the MDR register. The control over the bus transceivers allows each bus to be driven by only one bus driver, and to drive only one bus listener. The extra minor bus system implements a special bypass path, allowing data from MDR to feed the top or bottom SD pins directly, while addresses shifted out of the SD pins on the other side are routed over the mabs into the MAR register. This allows the next input bit plane to be fetched from data memory, while the mdbs bus is free to support simultaneous computation in the processor array using data path 1.

6.4. Using the Data Busses and Pipeline Registers Effectively

The design of the memory system, busses, and processors is intended to support computation on 16x16 blocks of numbers. In typical use, the processor array will read successive 16x16 blocks of bits, bringing in all the bits of all the numbers that form the complete 16x16 subimage. It then performs the computation, and finally writes out the result, one 16x16 bit plane at a time. To read each bit plane, the processor starts by computing the 16 starting addresses of each word of 16-bit data, placing one address in each row of the array. Using bit-parallel arithmetic, this computation typically requires 2 long clock cycles. PGreg must be loaded with the page number. Next, the processor array will shift the computed addresses out on data path 1 via the top or bottom DC pins, onto the mabs bus, and into the MAR register. The data memory will read data at the address specified by MAR and PGreg, putting the 16-bit value into MDR. The MDR register in turn will drive the mdbs bus, and data from there will enter either the top or bottom SD pins, and will be shifted into the array on data path 2. The fastest clock period can be used for this operation. After 18 clock ticks, an entire 16x16 block of bits can be input, and the 16 addresses can be updated to point to the next block of bits. For a write operation, data path 1 could be used to supply addresses to MAR over the mabs bus, while data path 2 supplies corresponding data to MDR via the SD pins and the mdbs bus. 17 fast clock ticks would be required to write a 16x16 block of bits and to update the 16 address pointers to point to the next output block.

Note that numerous variations on this theme exist. Addresses may be computed in rows of processors, or in columns; in the latter case they would have to be shifted on data path 1 left or right out the DC pins and onto the mabs bus. Data path 1 could be used for addresses and data path 2 for data, or vice versa. On read operations, by using the bypass path, data path 2 could be used both for shifting out addresses onto the mabs bus, and also for shifting in data via the SD pins directly from MDR. (Note that in this mode, addresses must be stored by rows in the array).

The wiring of the transceivers on the two busses allows a wide variety of circular shift, rotate, and transpose operations. For example, data clocked out of the DC pins on the left side can drive either the mabs or mdbs busses, and the DC pins on the opposite side may input the values from these busses. This connection could be used, for example, to circularly rotate bits in all rows of the array one position to the left. Similar connections could be used with the top and bottom DC pins to rotate columns up or down one bit, or with the top and bottom SD pins to rotate bits in data path 2 by one bit. "Circulant" rotations of the data can be accomplished by using both data paths. Connect the left side DC pins to the right side DC pins via one bus, and connect the top side DC pins to the bottom side DC pins via the other bus. Now do a diagonal shift. In a single clock tick, data in each row will be circularly rotated by 1 bit, and the rows themselves will be circularly rotated by one position.

16 by 16 arrays of bits can be easily transposed or rotated with this arrangement, using only 16 fast clock ticks. First, note that bus line 0 connects to the rightmost pins of the top and bottom DC and SD pins, and to the topmost pins of the left and right DC pins. To transpose a bit array about the main anti-diagonal, shift the data out the rightmost DC pins, onto mabs or mdbs, and into the bottom SD pins. The top row will become the right column, and the bottom row will become the left column, with the bits ordered properly. To rotate the array clockwise 90°, shift right out the DC pins onto mabs or mdbs, and into the top SD pins. The top row will become the right column, the bottom row will become the left column, and the order of the bits will be such that the matrix has been rotated. The effect of other combinations of shifts is listed below: (For each desired effect, we list several different shift patterns that will achieve the effect.)

Transpose on anti-diagonal:

Shift out DC right, shift in SD up

Shift out DC left, shift in SD down

Shift out SD up, shift in DC right

Shift out SD down, shift in DC left

Rotate Clockwise by 90°:

Shift out DC right, shift in SD down

Shift out DC left, shift in SD up

Rotate Counterclockwise by 90°:

Shift out SD up, shift in DC left

Shift out SD down, shift in DC right

Reverse all columns:

Shift out SD (DC) up, shift in DC (SD) down

Shift out SD (DC) down, shift in DC (SD) up

More complicated transpose, reversal, and rotation operations need to use multiple operations, or use the data memory as intermediate storage, and will take approximately twice as long or more. For example, to reverse the bits in the all the rows, transpose about the main anti-diagonal (shift out DC right, shift in SD up), then rotate counterclockwise 90° (shift out SD up, shift in DC left). To transpose a 16x16 array about the main diagonal, first reverse all the bits in each column (shift out SD up, shift in DC down), then rotate clockwise 90° (shift out DC right, shift in SD down).

The "prefetch" bus is useful in certain situations to expand the data transfer bandwidth of the array. The OKI AAP processor chips support data shifts along data path 2 in parallel with computation using data path 1. The prefetch bus `pfbs` is intended to further support this potential overlap of computation and I/O. Put 16 addresses to read from data memory into the 16 rows of DIO registers on data path 2. Now shift these addresses one by one over the SD pins onto `mabs` bus and into MAR, and read the corresponding data elements into MDR. The old MDR contents can transferred in parallel into the opposite SD pins over the `pfbs` bus, and shifted into the DIO registers. This leaves the `mdbs` bus free. Therefore, while this I/O operation goes on, the processor array can be computing using the ALU's, A[], B[], RS, C, LF registers, and shifting data via data path 1 and the `mdbs` bus.

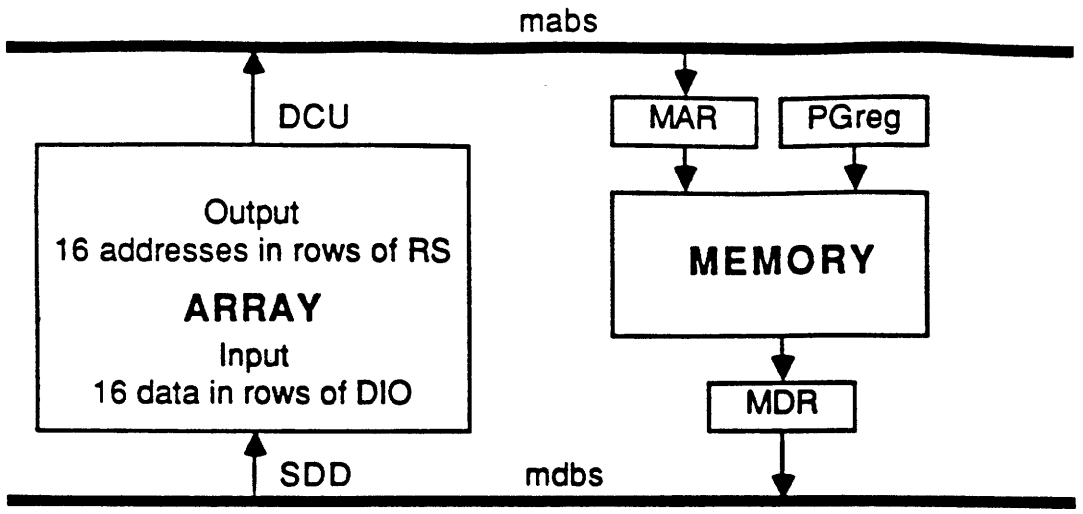
6.5. Other Data Bus Data Sources

Other input sources for the `mabs` and `mdbs` busses are the MAR, MDR, and PGreg registers themselves. MAR may drive `mabs`, PGreg may drive `mdbs` (n.b. the current prototype does not allow reading PGreg), and MDR may drive either `mabs` or `mdbs` or both. This helps save the state of the array during a micro-interrupt, and also allows pointers to be retrieved from data memory and then used as addresses.

16-bit constants from the microcode, `μconst`, can be read into the array or into the pipeline registers by driving `mabs` or `mdbs` with `μconst`, and then reading into any of the DC or SD pins, or into MAR, MDR, or PGreg. Often constants need to be broadcast to all the processors in the array; simply input the constant via the DC pins, and configure the internal AAP multiplexors for the appropriate broadcast operation.

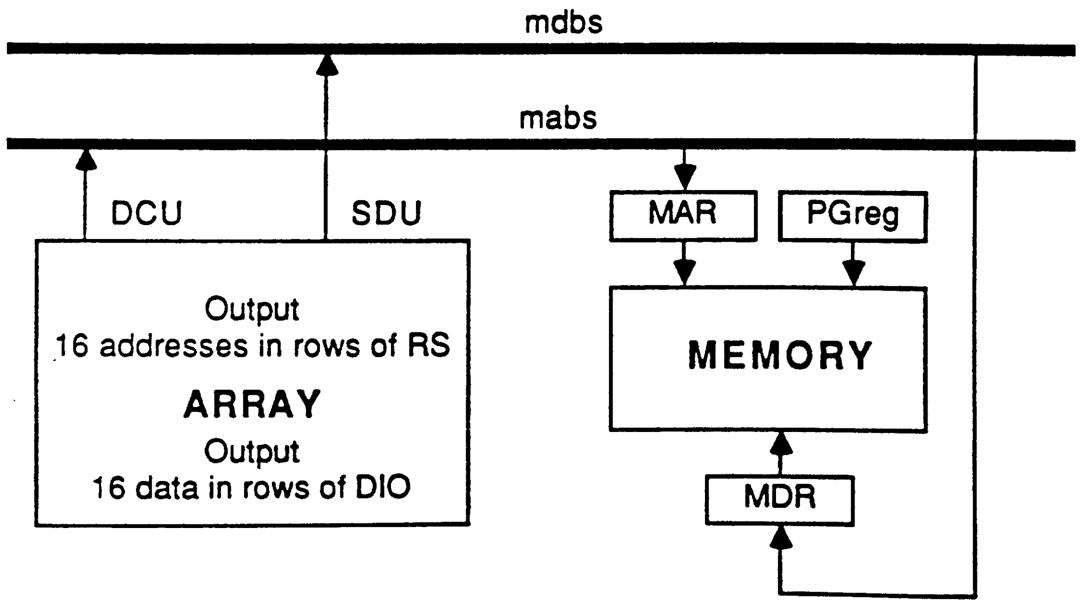
6.6. Connections with the 1401 Sequencer

If program execution must depend on a computed result, simply shift data out of the array over the `mdbs` and into MDR, then conditionally execute a jump, call, return, or other instruction on the OR or NAND of the bits in MDR, or on its most significant bit. Values read from memory or transferred from the array into MDR can also be loaded into the sequencer via its D-port. This is useful, for example, to calculate jump addresses or loop iteration counts, or to save computed values temporarily on the sequencer stack. Values from the sequencer may also be written into MDR via the D-port. This allows saving and restoring the state of the sequencer on a micro-interrupt. Note that when loading MDR from the sequencer, the MDR bus may not be used for data transfer, even though it is not actually used during this operation.



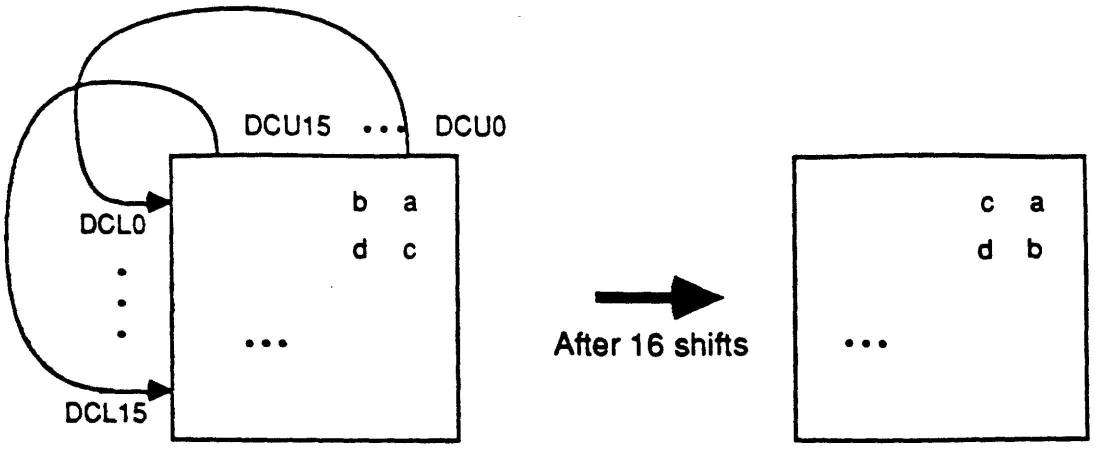
Typical Configuration to Read 16x16 Block of Data

(Shift addresses in rows of RS up through DCU into MAR,
read data at address MAR,PGreg into MDR,
shift data from MDR through SDD up into rows of DIO)

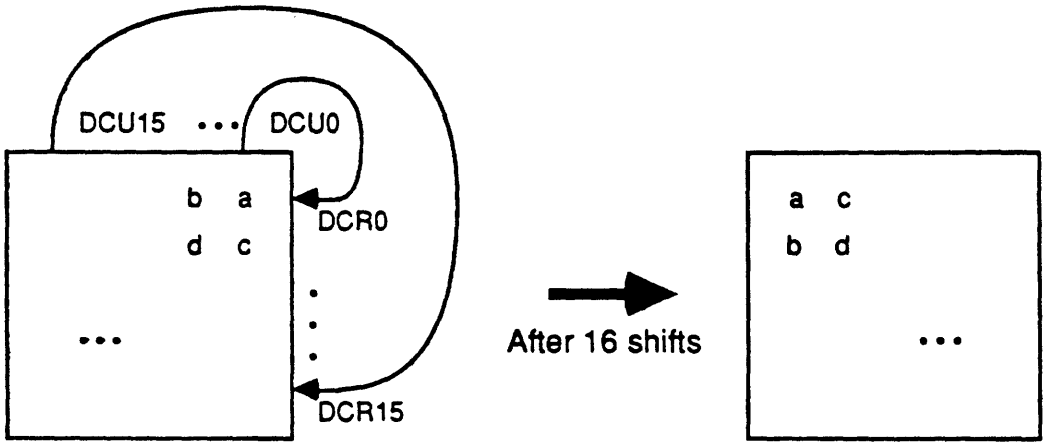


Typical Configuration to Write 16x16 Block of Data

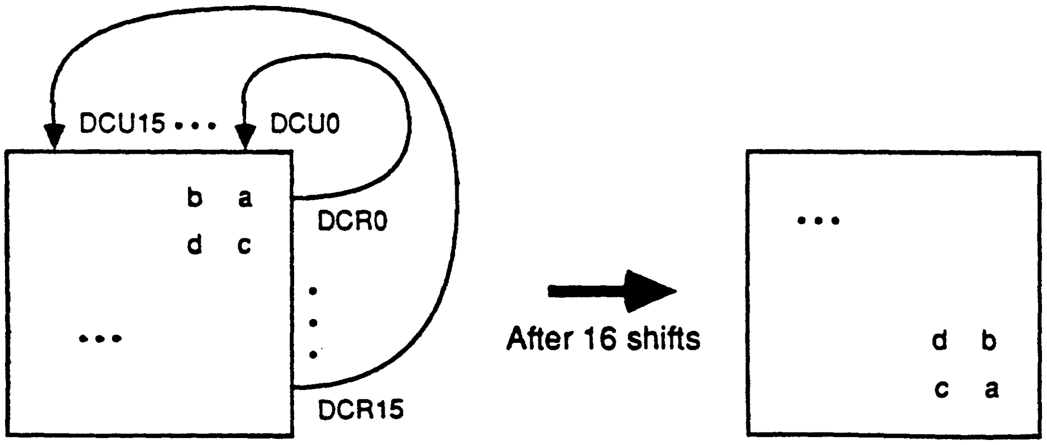
(Shift address in rows of RS up through DCU into MAR,
shift data in rows of DIO up through SDU into MDR,
write MDR at address MAR,PGreg)



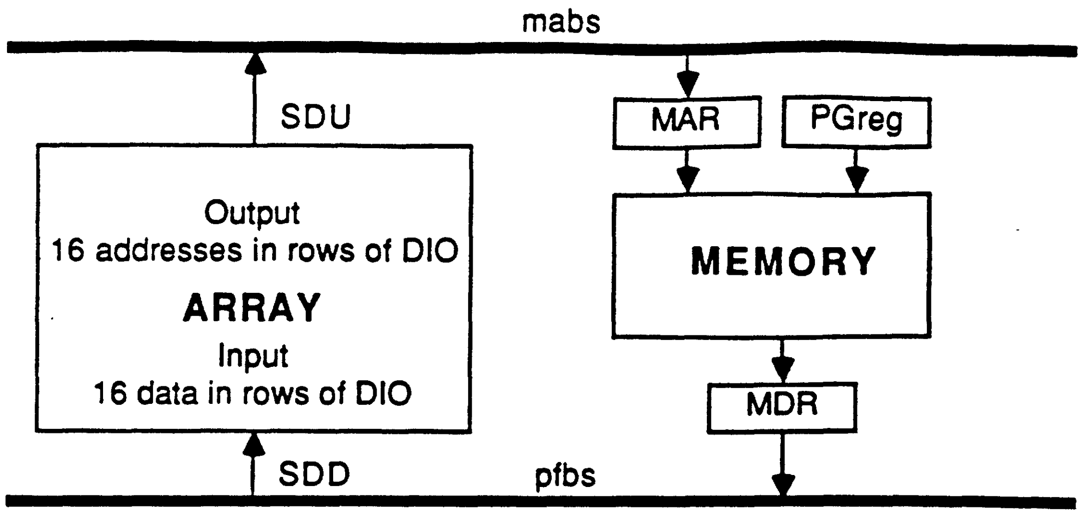
Transpose About Main Anti-Diagonal (16 shifts)



Rotate Data Counter-Clockwise (16 shifts)



Rotate Data Clockwise (16 shifts)



Read 16x16 Block of Data Using Prefetch Bus

(Shift addresses in rows of DIO up through SDU into MAR,
read data at address MAR,PGreg into MDR,
shift data from MDR on prefetch bus through SDD up into rows of DIO)

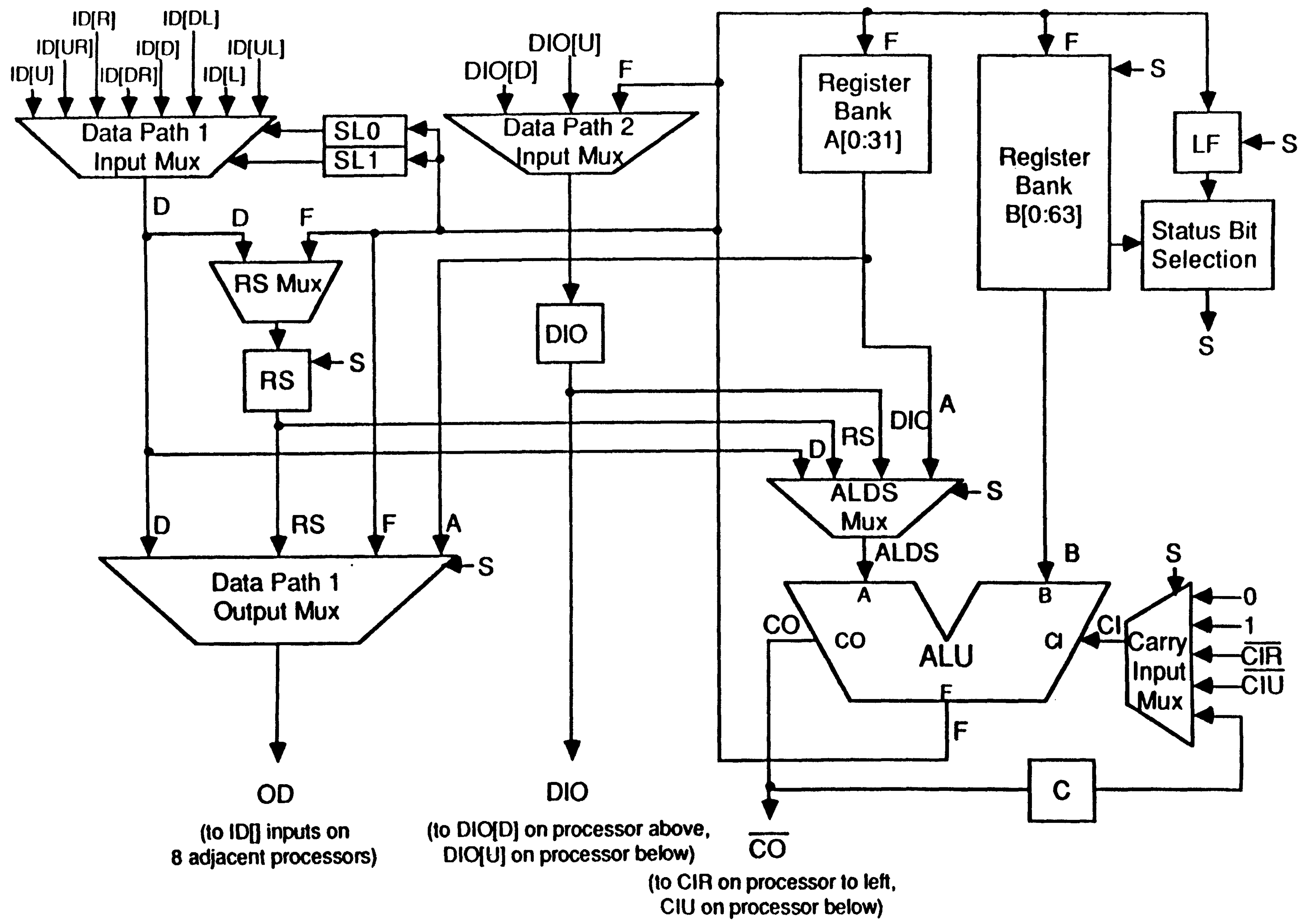
Section 7 – WIRING OF THE MSM6956AS AAP CHIPS

The wiring of the MSM6956AS AAP chips is straightforward except for the diagonal input pins and carry input and output pins on each chip. The microcode instruction control lines are bussed directly from the microcode pipeline register on board #1 to all four chips. Between chips, the DC, SD, and the input and output carry pins CI and CO are connected to their nearest neighbors, allowing data flow both horizontally and vertically. Tristate control lines are wired so that all tristate buffers are always enabled. The clock lines `aapclk` are fed a copy of the system `clk` powered by PAL #PP03.

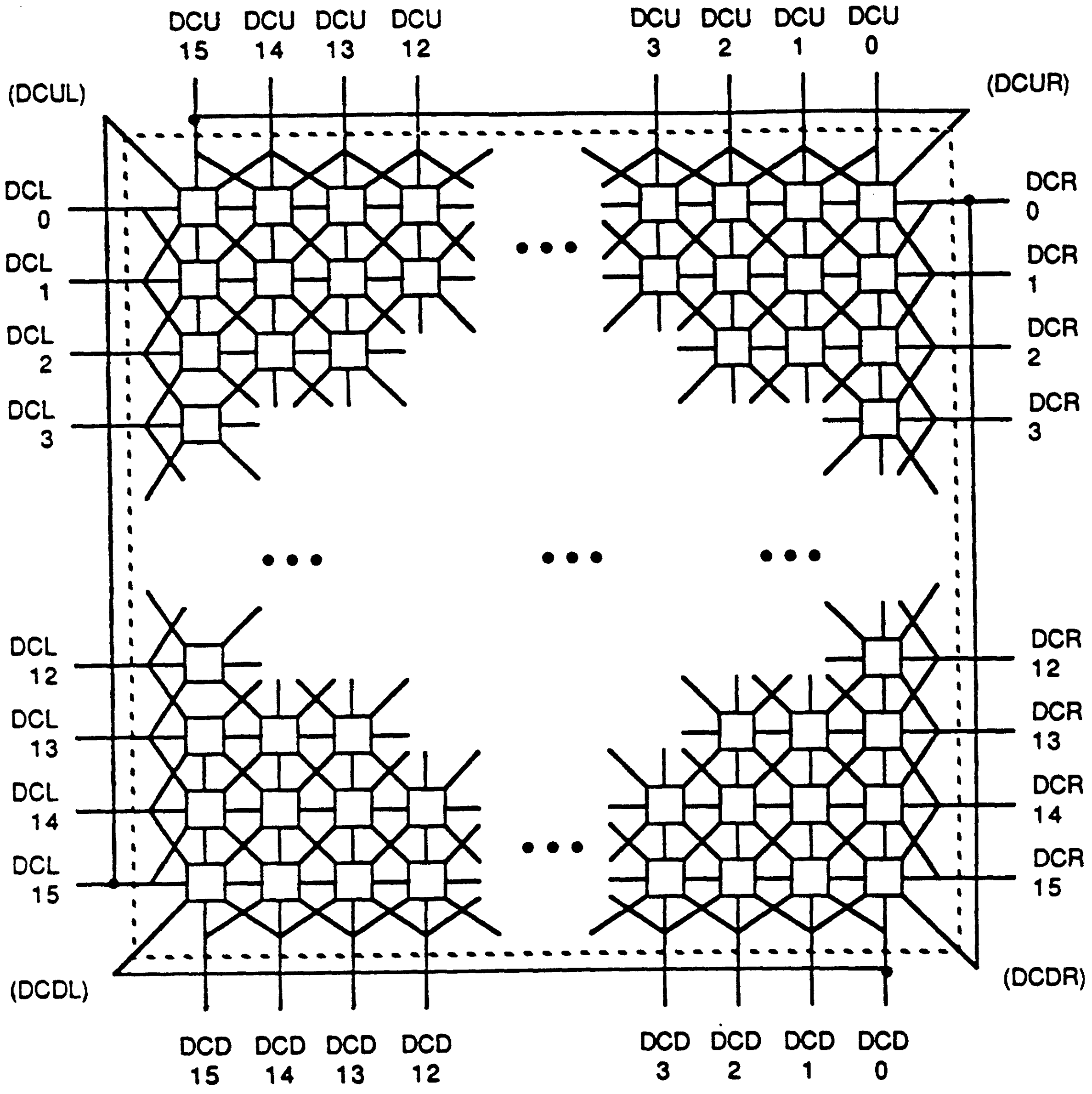
The 16 carry input pins on the right, CIR, and the 16 pins on the up side, CIU, are all driven by the CIN carry input bit in the microcode pipeline register. The 16 carry output pins on the left, COL, and on the down side, COD, are not connected to anything. With the standard configuration of the AAP chip boundaries, therefore, all the carry inputs to all rows or to all columns must be uniformly forced either high or low. To achieve bit-by-bit control over carry inputs, or to capture the carry outputs, use the multiplexed DC pin mode. In this mode (selected by microcode bits `I16` and `I17`), input values on the DC pins on the right or up sides are diverted to the carry inputs, and carry outputs on the left or down sides are diverted to the DC pins. This allows supplying arbitrary carry inputs via the `mabs` or `mdbs` busses, and capturing the carry outputs via the same busses.

Supporting diagonal transfers is tricky. The problem is that when data in the processor array is shifted diagonally, 31 bits of new data must be shifted into the array on two adjacent sides, and 31 bits of old data are shifted out on the opposite sides. Unfortunately, this structure does not mesh well with the 16-bit data bus structure of the array. The boundaries of the AAP chips are configured so that during a diagonal shift, each processor along the boundary of the chip drives its output onto its horizontally and/or vertically adjacent DC output pin(s). Processors in the corners of each chip drive two output pins. Input into each processor along the chip boundary is taken from the pin in the appropriate diagonal direction, with one corner processor pulling its input from the adjacent diagonal input pin. All diagonal pins on the chips are used solely for input.

The problem is where to connect the diagonal input pins. Inside the array of 4 chips, the diagonal pins are connected to an appropriate output DC pin on the diagonally adjacent chip. In the center of the four outside edges of the array, the diagonal pins are wired to the adjacent DC pin on the adjacent chip. These connections are all straightforward. At the four outside diagonal corners, unfortunately, good arguments could be made for wiring the diagonal pins to the DC pin at the other end of either the horizontal or the vertical row of DC pins. We have chosen, somewhat arbitrarily, to connect the diagonal pins in a counter-clockwise fashion. Thus the upper left DCUL diagonal input pin is connected to bit 15 of the left DCL bus, the upper right DCUR diagonal input pin is connected to bit 15 of the upper DCU bus, and so forth. When we connect the DCU pins to the DCD pins via one bus, and connect DCL pins to DCR pins via the other bus, then perform a diagonal shift, this configuration causes the rows of bits to be rotated by 1 bit, and the rows themselves to be rotated by one position. If we perform a diagonal shift, say in the up-right direction, and feed the left DCL inputs with MAR and feed the down DCD inputs with MDR, then this connection causes the MDR value to end up in the bottom row of the processors, rotated right one bit, and bits 0 through 14 of MAR end up in the left column of processors, shifted up one bit. Bit 15 of MAR is lost.



**PE Architecture (256 of these in AAP Array)
(Programming Model)**



Diagonal DC Input Pin Wiring, AAP Array

Section 8 - WIRING OF THE MABS, MDBS, AND PFBS BUSSES

8.1. MABS - The MAR bus

The MAR bus, `mabs`, connects the DCU (up), DCD (down), DCL (left), DCR (right), SDU (up), and SDD (down) pins from the array, feeding the MAR (chips MAR0 and MAR1) and PG (chip MAR2) registers. These registers are all formed of AMD29818 Serial Shift Registers, so that the host computer can directly read and write the values of MAR and PG. The MAR output (the lower 16 bits of the 20-bit `maad` bus) can be output back onto the `mabs`. The µconst field from microcode can be enabled to drive this bus. This allows reading the value of MAR, and allows MAR to be used for temporary storage. Several codes for the fields controlling mabs in microcode are unused, thus allowing opportunity for expanding the set of sources and sinks for `mabs` data.

Note that in the schematics, the buffers/transceivers in any pair are drawn to be mirror reflections of each other (about a vertical axis) - but the numbering of lines has been done consistently - the LSB (least significant bit) line is given to A0/B0 in both, etc.

Six microcode bits are used for controlling the MAR data path, three each for the bus driver (AOUT) and the listener (AIN). The encoding of the `mabs` driver and listener microcode fields is shown in the following tables:

MABS Input Coding

AIN code	Listener
0	DCU, DCD
1	DCL, DCR
2	SDU, SDD
3	PSU, PSD
4	Unused
5	Unused
6	Unused
7	Unused

MABS Output Coding

AOUT code	Driver
0	DCU, DCD
1	DCL, DCR
2	SDU, SDD
3	MAR
4	µconst
5	MDR
6	Unused
7	Unused

Note that opposite pairs of array pins (DCU & DCD, DCL & DCR, SDU & SDD) are never both inputs or both outputs, since we are not supporting the individually programmable data shift mode in which microcode bit `I15=1`. Hence, we specify only which group is to drive the bus or listen to the bus, and then use microcode bits `I15`, `I0`, `I1`, `I2` which determine the shift direction on data path 1, or bits `I3`, `I4` which determine the shift direction on data path 2, to infer exactly which of the pair is to be used. The decoding is done by PAL's #PP05 and #PP06, and is based on the following tables:

DC Paths
(I = Input, 0 = Output)

I15	I012 code	DCU	DCD	DCL	DCR
0	0	I	0	0	I
0	1	I	0	0	I
0	2	I	0	0	I
0	3	0	I	0	I
0	4	0	I	0	I
0	5	0	I	I	0
0	6	I	0	I	0
0	7	I	0	I	0
1	x	I	I	I	I

SD Paths
(I = Input, 0 = Output)

I34 code	SDD	SDU
0	I	0
1	0	I
2	I	I
3	I	I

Note that code value 3 in the `mabs` `AIN` microcode field is used to activate the prefetch bus `pfbs`. PSU and PSD refer to the buffer-pairs (MAB20, MAB21) and (MAB30, MAB31) directly connecting the MDR output to the SDU and SDD pins respectively. Also note that some of the eight `mabs` driver and listener codes are unused. These could be used for expansion. At present, all listeners or all drivers are disabled when one of these codes is specified. All DC drivers and listeners are also disabled when the microcode bit `I15=1`, specifying individually programmed data transfer mode inside the AAP array. Note that separate microcode field bits `-LdMAR` and `-LdPGreg` are used to enable loading MAR and PG from the `mabs` bus. Thus it is possible to use the value driven onto the `mabs` bus to simultaneously load MAR, PG, and also to drive a side of the array.

The MDR output (`mdo`) is fed to the `mabs` through buffer pair (MAB00, MAB01) as well as to each SD port, through buffer pairs (MAB20, MAB21) and (MAB30, MAB31). These latter buffers provide the `pfbs` prefetch bus path described earlier.

This prefetch is performed by selecting one of the SD ports as output (`AOUT`=2), and selecting the PSU, PSD pair as input (`AIN`=3). Simultaneously we program the DIO data path 2 registers to shift up or down (set `I34`=0 or 1 respectively), and we assert `-LdMAR` to clock the addresses shifted out of DIO into MAR. Suppose we choose to shift the addresses in DIO up (`I34`=0) through SDU into MAR, and the data in from MDR through SDD. Then the transceiver-pair (MAT10, MAT11) will be turned on, and addresses will be delivered to the MAR through the `mabs`. Simultaneously, the buffer-pair connecting the MDR directly to SDD (MAB20, MAB21) will turn on, thus delivering the data to the SDD port. Note that the DC paths and the `mdbs` are not involved in this operation, and remain free for computation.

The 74ALS245 transceivers and 74ALS244 buffers which drive and listen to mabs are controlled by setting the transceiver direction (these control lines are shared with those used for the mdbs) using the microcode data direction fields `I012` and `I34` combined with the `AIN` and `AOUT` microcode field values. The naming convention for the control lines is as follows:

```
Busses (eg DCU, DCD, etc.)   <BusName>(<bus width>)

Output Enables:

        MARbus               a._oe.<name of associated bus>

        MDRbus               d._oe.<name of associated bus>

Direction Control            t._r.<name of associated bus>
```

8.2. Detailed mabs Control Example

Suppose that we wanted to bit-reverse the columns of a 16×16 array of bits. We choose to shift out data from the SDU port, and immediately shift it back into the DCU port. We can use the mabs to do this transfer by selecting the SDU port as a bus driver (output) and the DCU port as the bus listener (input). This requires the following set of microcode fields:

`I34 = 0`	This sets up the SD datapath to shift up, outputting data on the SDU port.
`I012 = 0`	This sets up the DC datapath to shift down, inputting data from the DCU port.
`I15 = 0`	This selects conventional,uniform data propagation.
`AIN = 0`	This selects the DCU port for input from `mabs`.
`AOUT = 2`	This selects the SDU port to drive `mabs`.

The detailed decoding of the signals controlling the DCU transceivers (MAT50, MAT51) is as follows. PAL #PP05 sets "`u`" high. Note that this is an internal term, not a PAL output. This causes the `t._r.dcu` line to go low, which sets up the 74ALS245 transceivers MAT50 and MAT51 to function as B to A buffers. Also, if I15 = 0, `a._oe.dcu` goes low, thus turning on the B to A transceivers. So the datum on `mabs` gets routed to the DCU pins. The SDU transceiver control signals can be derived similarly.

The current PAL #PP05 and #PP06 programming does not force tristate enables to turn off before others turn on. Unfortunately, this can allow transient bus clashes at each clock edge when the next microcode instruction begins. Also, no interlocks have been provided to prevent programming the same pins to listen to both busses. For example, it is possible to select the DCU/DCD pins to listen to both `mabs` and `mdbs`, which could cause a bus conflict at the input DC pins. Currently, it is the programmer's responsibility to never allow this (the AOS operating system and the `aapcompile` compiler both help to avoid producing microcode with bus conflicts.)

8.3. Timing, Loading, Power Details for the mabs Bus

Timing:

T1 = CK to pipeline + 2*PAL's + Chip Enable time ('ALS245)

= 13 + 2*25 + 20 = 83 nsec.

T2 = CK to pipeline + F,I codes to DC, CO stable + Delay ('ALS245)

= 13 + 85 + 10 = 108 nsec. > T1

T3 = T2 + Propagation Delay ('ALS245) + Setup Time(DC)

= 108 + 10 + 30 = 150 nsec.

CLOCK PERIOD ≥ 150 nsec.

Total Bus Loading:

Inputs	Capacitance	High Current	Low Current
9*Tristate	9*12 = 108 pf	9*20 = 180 μA	9*0.1 = 0.9 mA
3*AM29818	3*5 = 15 pf	3*100 = 300 μA	3*0.45 = 1.35 mA
TOTAL:	125 pf	0.5 mA	2.3 mA

Can be handled by 'ALS245

Power Consumption: (excluding MAR, PGreg)

12*'ALS245	12*60 = 0.72 A = 3.6W
10*'ALS244	10*30 = 0.3 A = 1.5W
Total	5.1W

8.4. MDBS – The MDR Bus

The mdbs bus connects the DC and SD pins and the µconst field to the MDR register. The coding of the microcode fields for the mdbs driver DOUT (output) and the mdbs listener DIN (input) are similar to those for the mabs bus, and are given in the table below:

MDBS Input Coding

DIN code	Listener
0	DCU, DCD
1	DCL, DCR
2	SDU, SDD
3	MDR
4	Sequencer
5	Unused
6	Unused
7	Unused

MDBS Output Coding

DOUT code	Driver
0	DCU, DCD
1	DCL, DCR
2	SDU, SDD
3	MDR
4	µconst
5	Unused
6	Unused
7	Unused

Note that there is an explicit MDR input selection code, DIN=3, as well as a separate -LdMDR load enable strobe. This is necessary because the MDR register can be loaded from three distinct sources, and we must differentiate between these. The three possible inputs are the mdbs bus, the left rotator (rotleft) output (i.e. data memory output), and the 1401 sequencer D-port output. The -LdMDR field in microcode controls whether the MDR register will be loaded at all. Note that if the MDR register is to be loaded from the mdbs bus, then no other bus listener can be selected.

Another compromise is that code DIN=4 is used to force MDR to load from the sequencer D-port output. Pal #PP08 decodes the DIN microcode field and generates the -EnSeqMDR control line, which is transmitted back to board #1 to turn on the F543 B outputs. At the same time, an appropriate sequencer micro-instruction should be selected which will output a value on the D-port.

(The `aapcompile` compiler automatically selectes the appropriate `DIN` code whenever it detects a sequencer instruction which will output on the D-port.)

The MDR output (`mdo`) connects to the `mabs` through the buffer pair (MAB00, MAB01), and to the SDD and SDU ports through buffer pairs (MAB20, MAB21) and (MAB30, MAB31) (these form the `pfbs` prefetch bus). This output can also be routed back to `mdbs`, or to the memory via the right rotator. The enables and controls for the right rotator are discussed in the section on the data memory. These enables also depend on the signals `-VMEHIt`, `-IOMatch`, and `Halt` which are generated by the host interface on board 1.

8.5. Timing, Loading, Power Details for MDBS

Timing Details:

T1 = CK to pipeline + PAL's + Chip Enable time ('ALS245)
= 13 + 25 + 20 = 60 nsec

T2 = CK to pipeline + F,I microcode to DC, CO stable + Delay ('ALS245)
= 13 + 85 + 10 = 108 nsec > T1

T3 = T2 + Propagation Delay ('ALS245) + Setup Time (DC)
= 108 + 10 + 30 = 150 nsec.

T4 = T2 + (Path to Flag input of µsequencer)
= T2 + Delay ('ALS244) + Delay ('F573) + Delay (PAL)
= 108 + 10 + 8 + 15 = 141 nsec.

Thus CLOCK PERIOD > 150 nsec.

Total Bus Loading:

Less than MAR bus. Can be handled by 'ALS245.

Power Consumption (Excluding MDR and Rotators)

12*'ALS245	12*60 = 0.7 A = 3.5 W
6*'ALS244	6*30 = 0.2 A = 1W
TOTAL	<5W

8.6. Datapath Control PALS (PAL's PP05 to PP08)

PALs #PP05 and #PP06 are primarily responsible for controlling the mabs, while PALs #PP07 and #PP08 are responsible for mdbs. The transceiver direction control lines t._r.xxx are shared between both busses. Each PAL drives at most four t._r.xxx lines, which needs 0.4 mA sinking current, which can be handled by these TI PALs. The total power consumption is 4W, and the delay 25 nsec max.

The convention for the naming of the select lines is:

mabs:	a.<in0-2> for input, a.<out0-2> for output
mdbs:	d.<in0-2> for input, d.<out0-2> for output

PAL #PP08, in addition to producing the d._oe.md signal to turn on the buffer from mdbs to the MDR register, drives -EnSeqMDR, turning on the 'F543 latches driving the sequencer D-port output into the MDR input, as described in the previous section. PAL #PP04 decodes DIN to generate the signal enabling the left rotator, selecting whether MDR will load from data memory or from the mdbs bus.

PAL's #PP03 and #PP04 generate the signals for controlling the MAR and MDR registers, the clocks to the array, enables for rotators, etc. These are documented in detail in the data memory section

Section 9 – DATA MEMORY SYSTEM

9.1. Overview

The data memory is organized to allow reading and writing 16-bit words on any nibble (4-bit) addressable boundary. This unusual memory structure is required to support many image processing and related problems. For example, suppose we wish to apply a 3x3 low pass filter to an image stored in data memory. The image may be stored in bit plane format (consecutive bits in memory come from the same bit position in consecutive pixels) or in bit-parallel format (consecutive bits in memory are consecutive bits from a single pixel). We read in one 16x16 block of pixel data at a time, do the convolution with the filter coefficients, write out the result, then bring in the next block and repeat the operation. Unfortunately, the edges of each output block contain incorrect results, and only the center 14x14 block of data is correct. Thus the next 16x16 block that is input must overlap the previous block by two pixels. In bit-plane storage, this means that successive blocks overlap by 2 bits. As a hardware compromise, we allow succesive reads to overlap by 4 bits. This leads to a minor level of inefficiency, since we will have to overlap input blocks by 4 columns instead of 2.

Using conventional static RAM's in this unusual memory leads to substantial addressing difficulties. The scheme we use is to break the memory into two banks, each bank at least 16 bits wide. We supply appropriate addresses to each bank of memory, and fetch at least 32 bits. Data alignment and rotation circuitry selects the 16 bits we want from these. On a write operation, the 16 bits will have to be rotated appropriately into the correct position within the two banks, then written into a subset of the memory chips in each bank. Depending on the position of the desired 16 bits in the banks, the two addresses fed to the two banks may either be the same, or may differ by one.

uPD4362 16Kx4 55nsec MIX-MOS static RAMs are used to fabricate the data memory. Sixteen memory chips are read out in parallel, with eight in each bank. Of these, only two or four will be affected by the read or write operation.

In all, 128Kbytes of memory are installed. The delay (40 nsec) through the data rotation circuitry is almost equal to the access time of the memories. The address incrementer attached to one bank adds another 29 nsec to the access time. Total power consumption for the data memory system is 18 Watts.

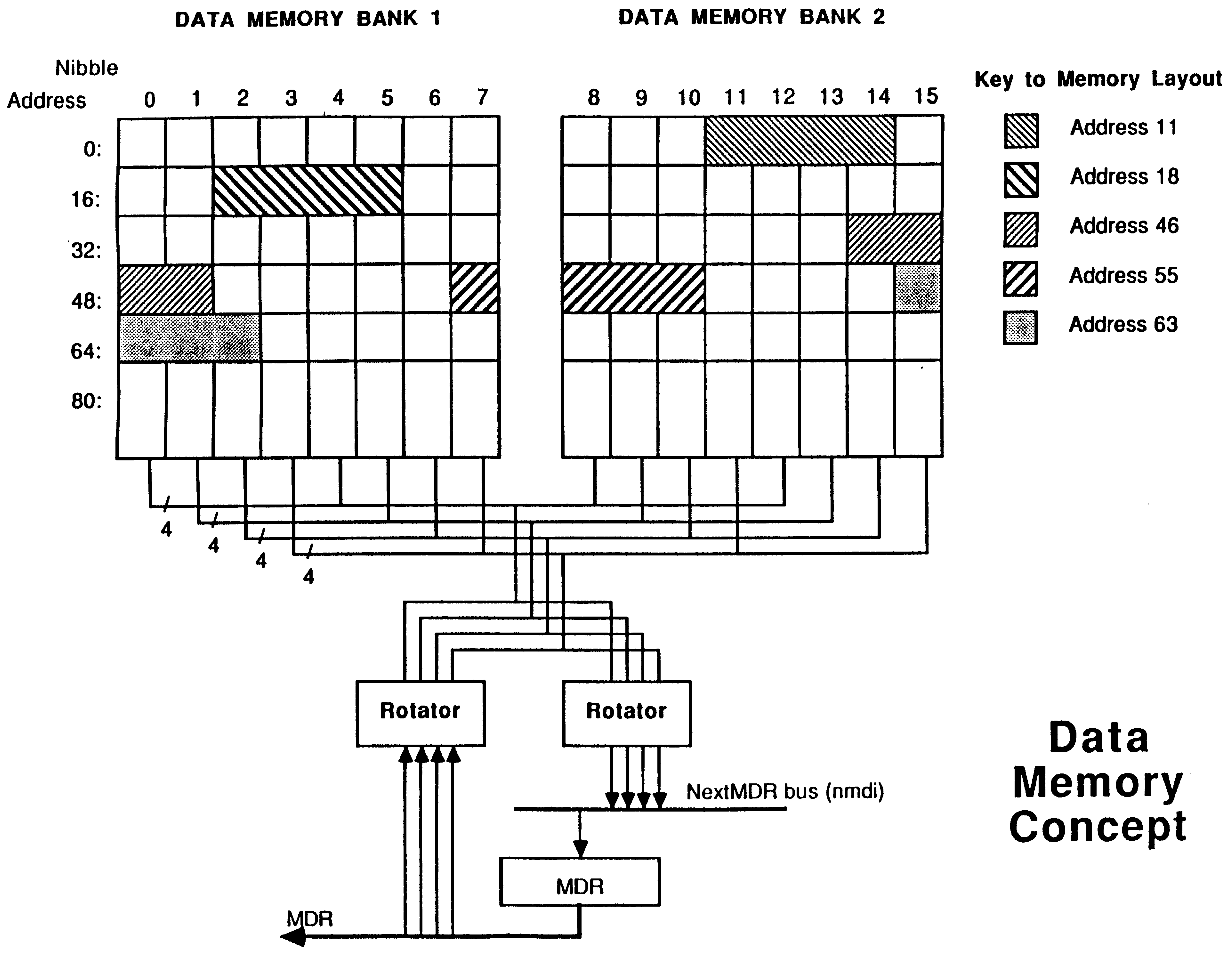
9.2. Memory Address

When the array accesses data memory, the top four bits of the memory address (mead - 20 bits) always come from the PG register. The low 16 bits can come from the MAR register or from the µconst field in microcode. On host accesses, the address is derived directly from the VME bus address supplied by the host. The maximal address space decoded by the PALs is 512 Kbytes. In the current version of the system, only 128 Kbytes has been installed, and thus the two higher order bits mead18 and mead19 are unused.

Because the top 4 bits of the address are always driven by PGreg when the array accesses memory, the array is limited to operating within a single 32 Kbyte page unless it modifies PGreg. Both PGreg and MAR are loaded from the mabs bus; PGreg is loaded from the lower 4 bits of this bus.

The host always addresses data aligned on word (16-bit) boundaries. Hence the two least significant memory address lines mead0 and mead1 are always zero during host access. During host access, we connect VME address lines ha1 through ha18 to memory address lines mead2 through mead19. The uppermost host address lines (and perhaps also ha17 and ha18) are decoded by the host interface on board 1 to determine whether this VME bus cycle is aimed at the data memory or at some other address block. VME bus signals -DS0 and -DS1 are used to determine which byte(s) are to be accessed.

Array access to data memory is more complicated. Each 16-bit access to data memory affects four consecutive 4-bit wide RAM chips. If the low nibble we want is located in the first 13 chips, MEM0 through MEM12, then all 4 nibbles will be in the same address in successive chips. However, if the low nibble is located in chips MEM13, MEM14, or MEM15, then one or more of the upper nibbles in the word may be located in address+1 in chips MEM0, MEM1, MEM2.



Data Memory Concept

To accomodate this, we break the chips into two banks. The address for chips MEM0 through MEM7 comes from an adder circuit `add1`, while the address for chips MEM8 through MEM15 comes from `mead` (this arrangement balances the loading of the address drivers). The adder circuit conditionally increments the address on `mead` if the low nibble falls in chips MEM8 through MEM15. This arrangement ensures that the desired 16 consecutive bits will always be located within the 16*4=64 bits that are potentially available at one time from the data memory. Only bits `mead4` through `mead19` are used to compute the addresses for the chips. The low order bits `mead0` through `mead3` are used to derive chip enables to select the appropriate 4 chips. Bits `mead0` and `mead1` are used to control data rotation and alignment circuitry. This is all described in a later section.

The selection of who drives the `mead` bus is determined as follows. When the array is driving data memory, the microcode control line `SelAdr` chooses between using the MAR or the `µconst` field for the low 16 bits. The top 4 bits always come from PGreg. However, if the host is accessing data memory, then the `mead` values come from the VME bus. The host interface on board 1 is responsible for deciding when a host data memory access is taking place. Three signals, `-VMEHit`, `-IOmatch`, and `Halt` are send from board 1 to board 2, and help determine whether a host memory access is in progress. PAL #PP03 determines the source for `mead` as follows:

-VMEHit	-IOMatch	Halt	SelAdr	Address source
0	1	1	x	Host
...Other combinations...			0	MAR
			1	µconst

The address lines from the VME bus are latched (chips MEL0, MEL1, MEL2) using `-VMEHit`. This is necessary because the VME bus address might change after a `-DTACK` acknowledge has been asserted, but before the current bus cycle has ended. The `-VMEHit` signal is deliberately delayed by board 1 so that it disappears only after `-DS1` and `-DS0` strobes have been deasserted and the VME bus cycleis finished. PAL #PP03 decodes the signals from board 1 to produce the latching signal `_oe.HAD` from the signals sent by board 1.

Timing of a write operation is particularly tricky. When the array writes to data memory, it must first preload the datum to be written into MDR. Next it issues a write command. The `SelAdr` microcode bit from the microcode pipeline register selects the appropriate address source (MAR or µ`const`), and at the end of the cycle the `-DMemW` signal is asserted, writing the datum at the MDR output into the specified location in the enabled chips. Board 1 ensures that `-DMemW` is deasserted 1 PAL delay before the next clock rising edge, thus preventing any glitches at the end of the write cycle. If `-LdMDR` is asserted in this same clock cycle (loading MDR, for example, from the `mdbs` bus), then the value of MDR will only change after the next clock edge, well after the write cycle is complete.

When the host accesses data memory, `-VMEHit` and `Halt` are asserted and `-IOMatch` is deasserted at least one oscillator cycle before `-DMemW` is asserted. This allows both the address and chip selects to stabilize before any write takes place.

Reading data into MDR is complicated by the fact that three separate copies of MDR are maintained. The version on board 2 is used to drive data memory and the data busses. A copy is kept in the sequencer circuitry for use when the sequencer inputs data through its D-port. Yet another copy is effectively kept in the circuitry which develops the flag test bit for the sequencer. All three copies must be loaded whenever a data memory read is performed, or when MDR is loaded from the `mdbs` or from the sequencer. To do this, PAL #PP03 uses the `SelAdr` bit from the microcode pipeline, together with the host interface signals from board 1, to generate the `_oe.maad`, `_oe.ucad`, `_oe.had`, and the `_oe.pgr` signals. These are asserted (pipeline register delay + cable delay + PAL delay) = 13+15+25 = 55 nsec after the rising clock edge. This selects the appropriate address bus driver for this cycle, and the memory contents will be output for loading into MDR on the next clock rising edge. Simultaneously, because `-LdMDR` is asserted, the `-MDRpclk` signal goes high in the second half of the cycle, causing the 'F573 latches on the sequencer D-port to become transparent, directly passing this memory value into the D-port input latch. A computed flag value based on the memory value will also be latched for use as a test bit. At the next rising clock edge, the sequencer D-port, the 'F573, the flag, and the MDR register all latch this copy of the memory contents. This strategy maintains synchronism between the multiple copies of MDR.

9.3. Memory Rotator, Alignment Circuitry

The data pins on the RAM chips are connected through a pair of rotators to MDR, and through a bidirectional port to the VME bus. During host reads, no data alignment is necessary since access is always on word boundaries. Sixteen nibbles are read out in parallel, of which four nibbles are used for a word read operation, or two nibbles for a byte read operation. On a write operation, either two or four chips are enabled to write the given byte or word data into the specified address.

When the array accesses data memory, data alignment is ensured by using 74F350 barrel rotator chips, one set for reads and the other set for writes. The amount of rotation is selected from the low three memory address bits, and the rotator outputs are selectively enabled by PAL #PP04 on the basis of the values of the `DIN` microcode field, `-DMemW`, and the `-VMEHit`, `Halt`, and `-IOMatch` signals. During host access, both rotator outputs are disabled.

The "read" rotator outputs are enabled whenever the data memory is configured for a read operation, and MDR is not being loaded from the `mdbs` bus or from the sequencer D-port.

The "write" rotator outputs are enabled when the `-DMemW` signal is asserted, and no host access is in progress. The timing here is quite critical. Note that the `-DMemW` signal enables the RAM's to write through two 'ALS244 buffers. This delays this strobe by about 10 nsec. However, the enable signal for the "write" rotator is derived from `-DMemW` by PAL #PP04, which has a delay of about 25 nsec. Hence at the end of the write cycle, the RAM chips should be disabled from writing before the rotator outputs return to tristate. Also note that the `-DMemW` signal is a strobe which goes low only in the last `osc` period of a clock cycle. Thus the "write" rotator outputs are valid for somewhat less than a single `osc` period.

PAL #PP04 is responsible for enabling the rotator outputs. The amount of rotation is controlled by the low two address bits mead0 and mead1.

During host access, the transceivers between the VME data bus and the data RAM's are controlled by the -Write and -DMemW signals. If the host is reading, the transceivers are turned on as soon as -VMEHit and Halt are asserted, and -IOMatch and -Write are deasserted. If the host is writing, however, then we have to wait until the RAM chip outputs have been disabled and the address and chip selects stabilize. Thus the transceivers during a host write are not enabled until -DMemW is asserted in the final osc period of the host access. This is handled by PAL #PP04.

9.4. Loading MAR, PG, and MDR Registers

During normal operation, loading of the MAR, PGreg, and MDR registers is enabled by the micricode fields -LdMAR, -LdPG, and -LdMDR in the microcode pipeline register. These pipeline bits are gated with the clk signal from board 1 in PAL #PP03 to form the clocking signals MARpclk, PGpclk, and MDRpclk. These signals latch new values into the corresponding registers at the next rising edge of the clock, thus loading the register with whatever values have been computed during this clock cycle. The 85 nsec oscillator period is enough to accomodate any delays (clock to pipeline output + cable delay) in the appearance of these signals.

The MDR, MAR, and PG register are all built from AM29818 serial shift registers, and are connected with the microcode pipeline registers and the host interface into an 18-byte circular shift register. The µSDO output from the microcode pipeline register (UCR0) on board 1 connects to the SDI pin of the PGreg. This in turn connects to the MAR register, which in turn connects to the MDR. Finally, the MDR-SDO output goes back to the 'ALS299 register HR0 in the host interface on board 1. If the host wishes to load the MAR, MDR, and PGreg registers, it first shifts in the new values via the shadow serial shift registers, and clocks the values into the parallel registers. To examine the contents of these registers, the host can transfer their values into the shadow serial shift register, then shift the values out.

PALs #HP7 and #HP8 in the host interface coodinate copying the shadow shift registers into the parallel registers, and the parallel registers into the shadow registers respectively. PAL #HP7 uses signals `-ModeA`, `Dopclk` and PAL #HP8 uses signals `-ModeB`, `DoMdclk` for this purpose. Both `-ModeA` and `-ModeB` are normally deasserted. PAL #µCP3 on board 1 combines these signals, driving signal `MMode` low, thus enabling all the shadow shift registers for shifting, and enabling the parallel registers to load normally from their D-ports. `Dodclk` and `Dopclk` are clock signals generated to drive the shadow register pipeline and the parallel registers during host access. `MARpclk`, `PGpclk`, and `MDRpclk` are unconditionally generated by PAL #PP03 whenever `Dopclk` on board 1 pulses low. This only occurs during certain host accesses, and when `clk` is high. Similarly, PAL #PP04 unconditionally generates an `Mdclk` signal to clock the shadow serial shift registers on board 2 whenever board 1 pulses `Dodclk`.

When the shadow serial shift registers are to be copied to the parallel registers, PAL #HP7 assets `-ModeA`, which in turn causes `Mmode` to be asserted by PAL #µCP3. This sets up the AM29818's to load the parallel registers from th shadow registers when `Dopclk` is brought low and then high by PAL #HP7. Note that #HP7 also handles reading µcode RAM into the parallel registers. When reading microcode, however, a separate `Dopclk` signal is generated only for the pipeline registers, and not for the MAR, MDR, or PG registers.

When the parallel registers are to be copied into the shadow shift registers, PAL #HP8 asserts `-ModeB`, which causes `MMode` to be asserted by PAL #µCP3. This sets up the AM29818's to load the shadow registers from the respective Y ports when `Dodclk` is brought low and then high by PAL #HP8. Note that before `Dodclk` appears, PAL #HP8 has waited long enough for a low value to propagate through the `SDI/SDO` pins of all the AM29818's. Also note that the `-EnPipe` signal is not used on board 2, since the outputs of all 29818's area always enabled. Finally note that when PAL #HP8 writes the contents of the shadow shift registers into µcode RAM, `Dodclk` does not appear on board 2 and so the MDR, MAR, and PG registers do not participate in this action.

In order for timing to be correct, PALs #PP03 and #PP04 must be matched in speed with PAL #µCP5 (25 nsec), so that clocks `µpclk1`, `µpclk2`, `µdclk1`, `µdclk2`, and so forth are clocked simultaneously. Note also that the clock for the array, `aapclk`, is delayed through PAL #PP03 to synchronize its timing with the other pclk's.

9.5. Chip Selection, Loading, and Timing

The lower four bits of the memory address, `mead0` through `mead3`, the `-EnMem` from the clock generator PAL's on board 1, `-VMEHit`, `Halt`, and `-IOMatch` from the host interface, and `-DS1` and `-DS0` strobes from the VME bus, are all combined by PAL's #MEP1 and #MEP2 to form the RAM chip enables. All chip selects are turned off if `-EnMem` is high, unless `-VMEHit` and `Halt` are asserted, `-IOMatch` is deasserted, and either `-DS0` or `-DS1` are asserted (the clock generator is in a halt state during host access, and cannot conflict). Note that during host access, `-VMEHit` is asserted one `osc` cycle before `-DMemW` arrives, thus allowing both the address and chip selects to stabilize before any write takes place. The PAL's #MEP1 and #MEP2 used for the logic are fast 15nsec PAL's. Tables describing the selection logic used by these PAL's are shown below:

Chip Selection Logic

Normal Operation (`VMEHit` & `Halt` & `!IOMatch` = 0, `EnMEM` = 1)

	Chip selection (1 = selected)															
Addr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	1	1												
1		1	1	1	1											
2			1	1	1	1										
3				1	1	1	1									
4					1	1	1	1								
5						1	1	1	1							
6							1	1	1	1						
7								1	1	1	1					
8									1	1	1	1				

9										1	1	1	1			
10											1	1	1	1		
11												1	1	1	1	
12													1	1	1	1
13	1													1	1	1
14	1	1													1	1
15	1	1	1													1

Chip Selection Logic

Host Access (VMEHit & Halt & !IOMatch = 1)

				Chip Selection (1 = selected)															
DS1	DS0	A3	A2	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	x	0	0	1	1														
1	x	0	1					1	1										
1	x	1	0									1	1						
1	x	1	1													1	1		
x	1	0	0			1	1												
x	1	0	1							1	1								
x	1	1	0											1	1				
x	1	1	1															1	1

The -DMemW strobe is fed to all 16 chips. Two buffers provide enough drive capability to handle the heavy 80 pf capacitance of these lines. Timing on the write cycle is compensated by delaying the -DMemW signal through a 25 nsec delay in PAL #PP03 to delay the enables on the rotators.

The critical propagation delay path is from the output of the pipeline register, through the address section PAL #PP03, then through the memory address bus buffers (or the enable time of the 'F573), through the adder, the memory chips, the rotator, and finally through the setup time for the MDR register. The time taken is surprisingly large, as high as 160 nsec, close to the maximum delay anywhere in the entire system. Of this, almost 40 nsec is due to the data rotation circuitry, and another 29 nsec to the adder. (In a future design we could improve things somewhat by using a faster 15 nsec PAL. *At the same time, however, we should drive AAPclk, MARpclk, PGpclk, and MDRpclk with a slower PAL* to keep the time allowed for `-DMemW` skew to 25 nsec, and thus preserve a correct write cycle.)

The RAM's consume much of the power on board 2 (about 7.5 W peak). By using `-EnMem` to disable the memories when they are not used, however, the average power consumption should be well below the peak levels.

9.6. Timing, Loading, Power Details

Timing Details:

Adder = 'AS30 + 'AS02 + 2*'F283
= 5 + 4.5 + 8.5 + 10.5 nsec = 29 nsec.

T1 = (Clock to MAR) + 'ALS244 + Adder + uPD4362 + 'F350 + Setup(MDR)
= 13 + 10 + 29 + 55 + 11 + 8 nsec = 126 nsec.

T2 = (Clock to pipeline) + PP03 + 'ALS244/'F573 + Adder + uPD4362 + 'F350 + Setup(MDR)
= 13 + 25 + 20 + 29 + 55 + 11 + 8 nsec = 160 nsec.

Minimum CLOCK PERIOD ≥ 160 nsec.

Total Bus Loading

A3 drivers:

2 * 'ALS244 outputs	24 pf	
1 * 'F283 input	5 pf	0.6 mA
2 * 'AS02 input	10 pf	0.2 mA

2 * TIBPAL16L8	10 pf	0.4 mA
TOTAL	50 pf	1.2 mA

an 'ALS244 can easily handle this.

A4-A11 drivers:

2 * 'ALS244 outputs	24 pf	
1 * 'F283 input	5 pf	0.6 mA
1 * 'AS30 input	5 pf	0.1 mA
8 * uPD4362	40 pf	
TOTAL	75 pf	0.7 mA

an 'ALS244 can easily handle this.

-DMemW:

16 * uPD4362	80 pf	(shared between two drivers)

Adder Outputs:

8 * uPD4362	40 pf

uPD4362 Outputs:

3 other uPD4362 outputs	21 pf
1 'F350 output	12 pf
2 'F350 inputs	10 pf
1 'ALS245 I/O	12 pf
TOTAL:	55 pf.

Specs are rated at 30 pf!!! Good thing we got 55 nsec chips!

Power Consumption:

Host Addr	3 * 'F573	3*30 = 90 mA	0.5 W

µconst Addr	3 * 'ALS244	3*30 = 90 mA	0.5 W
PG, MAR Addr	3 * 'ALS244	3*30 = 90 mA	0.5 W
	3 * 29818	3*155 = 465 mA	2.4 W
Adder:	4 * 'F283	4*55 = 220 mA	1.1 W
	1 * 'AS02	20 mA	0.1 W
	1 * 'AS30	5 mA	-
2 * TIBPAL16L8		2*180 = 360 mA	1.8 W
16 * uPD4362		16*90 = 1.44 A	7.5 W
8 * 'F350		8*42 = 336 mA	1.6 W
MDR	2 * 29818	2*155 = 310 mA	1.5 W
Host Data	2 * 'ALS245	2*60 = 120 mA	0.6 W
TOTAL:			18 W