2.0 Central Processor (CP)

2.1 Introduction

The Central Processor (CP) controls the high-speed I/O devices and the main memory of the Dandelion. It provides short-latency memory access and ALU service for the integral I/O controllers and can emulate the Mesa Processor as defined by the Mesa Processor Principles of Operation. It is composed of about 160 standard chips and resides entirely on one 11" by 15" printed circuit card located in slot 3.

This chapter presents the hardware structures of the Central Processor and its interfaces with the rest of the Dandelion. Another manual, the *Dandelion Microcode Reference (DMR)*, presents the assembler microcode format and is interspersed with hardware details and examples.[1]

The CP is a microprogrammed, 16-bit general-purpose computer. The microstore can hold up to 4096 48-bit microinstructions[2] and can be read or written by the low-speed Input/Output Processor (IOP). Each microinstruction is decoded and executed in 137 nanoseconds, a *cycle*.[3] All microinstruction operations are completed in one cycle; instruction execution is not pipelined over several cycles, except that while one is being executed its successor is being read from the microstore.

Cycles are grouped into *clicks*, where one click equals three successive cycles labeled c1, c2, and c3. Cycles are always enumerated in order c1, c2, c3, and then c1 again.[4] This sequence is never interrupted or altered; accordingly, both targets of a two-way branch must be specified with the same cycle number. (Strictly speaking, this is necessary only if the target microinstructions contain cycle-dependent operations.) The microcoder's task of aligning instructions so that they execute in successive cycles is a necessary outcome of the fixed-tasking, click structure. Moreover, when one desires code which is speed optimized, this structure usually requires the elimination of three microinstructions instead of one.

While the three microinstructions of a click are executing, a memory read or write can be performed: the address is sent to the memory in c1, a single data word may be sent during c2, and data is returned from memory in c3. A memory operation can *only* be initiated in cycle 2.

Clicks are grouped into *rounds*: five successive clicks (numbered 0..4) comprise a round, which is two microseconds in duration. Each click of a round is permanently allocated to one or more of the I/O controllers. If an I/O controller does not request the service of its corresponding *task* microcode, the Emulator-microcode task runs during that click instead of the device-microcode task. When there is a transition between tasks, the hardware preserves the outgoing task's microprogram counter and restores it when it runs again.

The click is a basic microcode time unit: devices and the Emulator are serviced in units of clicks and the microcode can transfer exactly one memory word in this time. For purposes of synchronization, the click is an atomic operation. Since a click is 411 nanoseconds in duration, the maximum memory bandwidth available through the CP is 40 Mbits/s (2.4 megawords/s).

The CP is implemented using four 2901 bit-slice chips plus external memories and registers. The 2901 provides 17 registers readily accessible to the microcoder, the usual logical and arithmetic functions, and single bit shifting.

Available to the microprogrammer and external to the 2901 are four register sets (U, RH, IB, and Link), a four-bit rotator, the I/O registers and memory, and four Emulator registers (stackP, ibPtr, pc16, and MInt). There are no task specific registers: all registers can be addressed by all tasks.

2.2 Microinstruction Format

The microinstruction format attempts to strike a balance between some naturally opposing constraints: control store width versus control store size, encoding schemes versus decoding hardware constraints, and coverage of all possible data operations versus exclusion of impracticable operations. The goal of the format is that frequently applied operations are encoded in the smallest number of bits. Furthermore, it was designed so that the most important Mesa Emulator and I/O operations execute in one click. The format is illustrated and summarized in Figure 2.

A 48-bit microinstruction has three major parts: 2901-control bits, miscellaneous functions, and a "goto"-address field. The field names are abbreviated as:

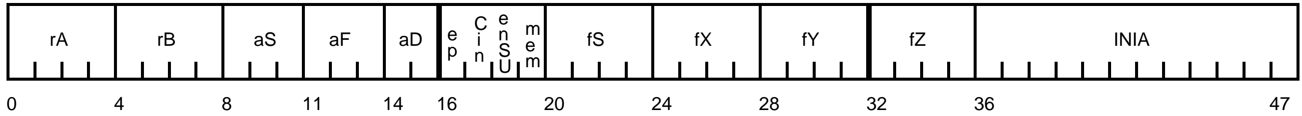
rA, rB	R registers A and B
aS, aF, aD	ALU source address, function, destination address
ep	even parity
Cin	2901 carry input
enSU	enable stack/U registers
mem	memory operation
fS	function fields selector
fX, fY, fZ	function fields X, Y, and Z
INIA	intermediate next instruction address.

The 2901-control bits occupy the first word: rA, rB, aS, aF, and aD. The "goto" address, INIA, utilizes 12 bits. INIA is a control-store-destination address unless condition bits, specified by the previous microinstruction, are *or'd* into it, resulting in a branch or dispatch. Thus, every microinstruction is a potential jump instruction.

The fS field is broken into two subfields: fS[0-1] and fS[2-3]. These control the deciphering of the fY and fZ fields, respectively. Both the fY and fZ fields have four possible enumerations as defined by fS:

The fY field can, depending on fS[0-1]: (1) name a branch or multi-way dispatch, (2) specify a miscellaneous function, (3) name an I/O register to be loaded, or (4) equal the high nibble of an 8-bit constant. These four functions are called DispBr, fYNorm, IOOut, and Byte.

The fZ field can (1) enumerate a miscellaneous function, (2) equal a 4-bit constant or the low half of an 8-bit constant, (3) be the low half of a U register address, or (4) name an I/O register to be read. These four classes are abbreviated fYNorm, Nibble, Uaddr, and IOXIn, respectively.



Field	Description
rA	2901 A reg addr, U addr [0-3]
rB	2901 B reg addr, RH addr
aS	2901 alu Source operand pair
aF	2901 alu Function
aD	2901 alu Destination/shift control
ep	Even Parity
Cin	2901 Carry In, Shift Ends, writeSU (if enSU=1)
enSU	enable SU reg file
mem	MAR_ (if c1), MDR_ (if c2), _MD (if c3)
fS	Function field Selector
fX	X Function
fY	Y Function
fZ	Z Function
INIA	Next Instruction Address

aS	R, S
0	A, Q
1	A, B
2	0, Q
3	0, B
4	0, A
5	D, A
6	D, Q
7	D, 0

aF	F
0	R + S + Cin
1	S − R − Cin'
2	R − S − Cin'
3	R or S
4	R and S
5	~R and S
6	R xor S
7	~R xor S

sh..aD	R[rB]	Q	Ybus
0	no write	F	F
1	no write	no write	F
2	F	no write	A
3	F	no write	F
4	F/2	Q/2	F
5	F/2	no write	F
6	2F	2Q	F
7	2F	no write	F

sh _ (fX=shift) OR (fX=cycle) OR (fY=cycle)

fS[0-1]	fY=
0	DispBr
1	fYNorm
2	IOOut
3	Byte

fS[2-3]	fZ=	SU addr[0-7]
0	fZNorm	0,,stackP
1	Nibble	0,,stackP
2	Uaddr[4-7]	rA,,fZ | rA,,Y[12-15]* IF fZ=AltUaddr*
3	IOXIn	rA,,fZ | rA,,Y[12-15]* IF fZ=AltUaddr*

* as executed by previous u-instr

fX	fXNorm
0	pCall/Ret0
1	pCall/Ret1
2	pCall/Ret2
3	pCall/Ret3
4	pCall/Ret4
5	pCall/Ret5
6	pCall/Ret6
7	pCall/Ret7
8	Noop
9	RH_
A	shift
B	cycle
C	Cin_pc16
D	Map_
E	pop
F	push

fY	fYNorm	DispBr	IOOut
0	ExitKern	NegBr	IOPOData_
1	EnterKernel	ZeroBr	IOPCtl_
2	ClrIntErr	NZeroBr	KOData_
3	IBDisp	MesaIntBr	KCtl_
4	MesaIntRq	PgCarryBr	EOData_
5	stackP_	CarryBr	EICtl_
6	IB_	XRefBr	DCtlFifo_
7	cycle	NibCarryBr	DCtl_
8	Noop	XDisp	DBorder_
9	Map_	YDisp	PCtl_
A	Refresh	XC2npcDisp	MCtl_
B	push	YIODisp	_TStatus
C	ClrDPRq	XwdDisp	EOCtl_
D		XHDisp	KCmd_
E	ClrRefRq	XLDisp	_TIData
F	ClrKFlags	PgCrOvDisp	POData_

fZ	fZNorm	IOXIn
0	Refresh	_EIData
1	IBPtr_1	_EStatus
2	IBPtr_0	_KIData
3	Cin_pc16	_KStatus
4	Bank_	KStrobe
5	pop	_MStatus
6	push	_KTest
7	AltUaddr	EStrobe
8	Noop	_IOPIData
9		_IOPStatus
A		_ErrnIBnStkp
B		_RH
C	LRot0	_ibNA
D	LRot12	_ib
E	LRot8	_ibLow
F	LRot4	_ibHigh

pCall when NIA[7]=0. pRet when NIA[7]=1.

Equivalent names: XDirtyDisp = XLDisp; EtherDisp = YIODisp; TAddr_ = ClrDPRq; TCtl_ = PCtl_; TOData_ = POData_

Figure 1. Dandelion CP Microinstruction Format

2.3 Registers and Data Paths

Figure 2 illustrates the registers and data paths layout for the CP. The area inside the dashed lines represents the internal components of the 2901 ALU. The Y bus corresponds to the Y output of the 2901 and the X bus is connected to the 2901 D input. Both the X and Y buses are available on the backplane.

2.3.1 R and Q Registers and 2901 Data Paths

Figure 2 shows the 16-word, two-port register file called the R registers. One of the output ports is labeled A and the other B. These are the "fast" registers of the CP and can be used to hold temporaries, memory data and addresses, and arithmetic operands.

Every cycle, the contents of the R register given by the register-A (rA) field of the microinstruction is available at the A port, and likewise for the B port. If rA=rB, then the same data appears at both ports.

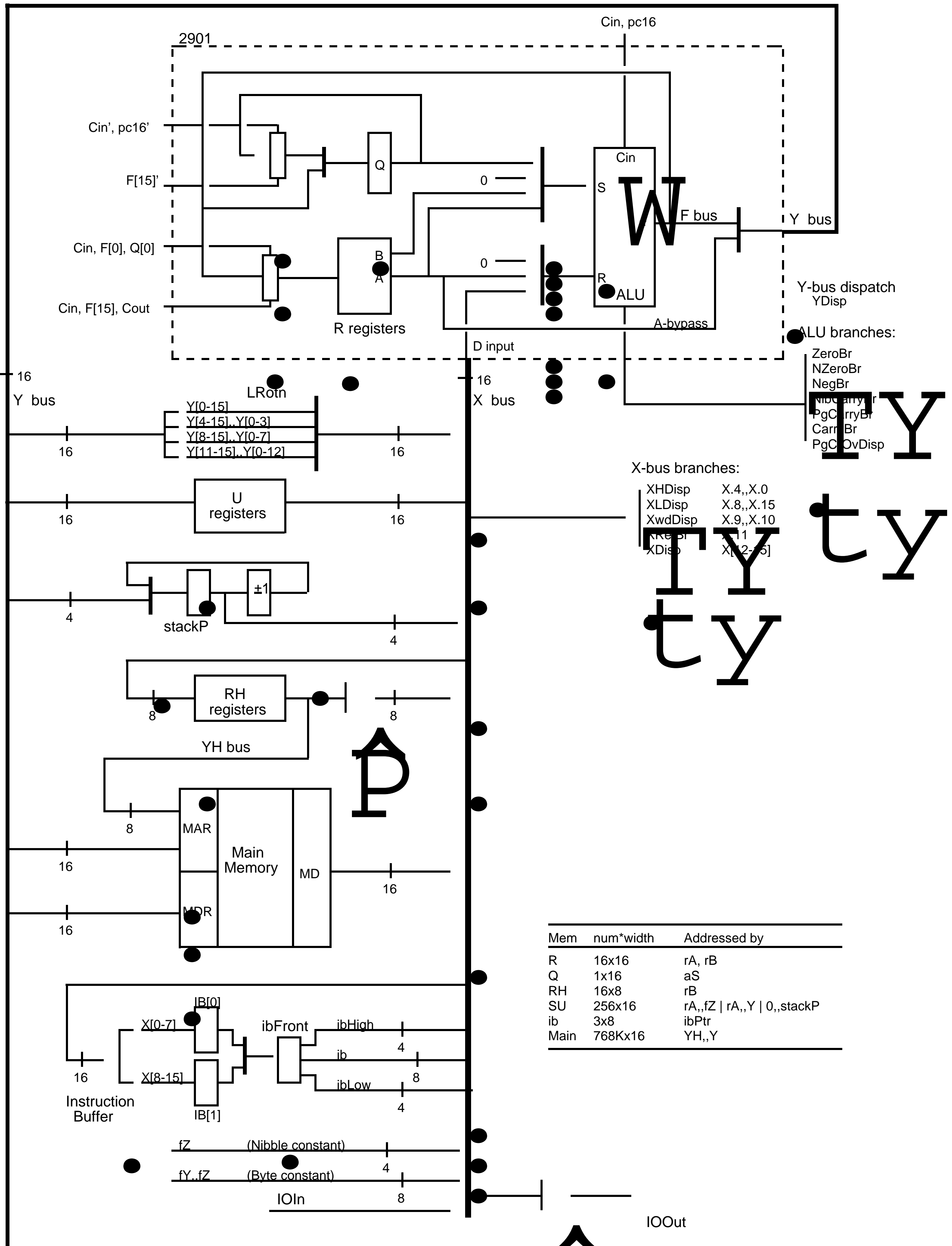
If the alu-Destination (aD) field specifies a write back into an R register, the rB field specifies which one: at the end of the cycle, register B is written with the ALU output (named F) or it is written with F shifted one bit.

The Q register holds 16 bits which can be written with the ALU output or its old value single-bit shifted left or right. It is implicitly referenced by the aS field of the microinstruction and can be used for double-word shifting.

The 2901 arithmetic unit has three inputs: R, S and Carryin (Cin). The R input can be set to the output of the A port, the value of the X bus, or zero. The S input can be driven by the output of the A or B ports, the value of the Q register, or zero. Cin can be either 0 or 1, or the value of the single-bit Emulator register pc16.

The 2901 can perform three arithmetic and five logical operations as specified by the alu-Function (aF) field. Arithmetic follows the two's-complement conventions. Three of the logical operations are symmetrical with respect to R and S: logical *or*, *and*, and *xor*. The remaining two logical operations complement R: ~R *xor* S and ~R *and* S.

Figure 3 shows a matrix of ALU computations as a function of possible aS and aF values. From the table it is clear there are many possible ways to generate zero within the ALU. All one's (0FFFF) is easily produced for some functions if rA=rB.



Mem	num*width	Addressed by
R	16x16	rA, rB
Q	1x16	aS
RH	16x8	rB
SU	256x16	rA,,fZ | rA,,Y | 0,,stackP
ib	3x8	ibPtr
Main	768Kx16	YH,,Y

Figure 2. Dandelion CP Data Paths

										rA=rB = R
aF	Cin	aS: (A,Q)	(A,B)	(0,Q)	(0,B)	(0,A)	(D,A)	(D,Q)	(D,0)	(A,B)
R+S	0	A+Q	A+B	Q	B	A	X+A	X+Q	X	2R
	1	A+Q+1	A+B+1	Q+1	B+1	A+1	X+A+1	X+Q+1	X+1	2R+1
S-R	0	Q-A-1	B-A-1	Q-1	B-1	A-1	A-X-1	Q-X-1	-X-1	-1
	1	Q-A	B-A	Q	B	A	A-X	Q-X	-X	0
R-S	0	A-Q-1	A-B-1	-Q-1	-B-1	-A-1	X-A-1	X-Q-1	X-1	-1
	1	A-Q	B-A	-Q	-B	-A	X-A	X-Q	X	0
R or S		A or Q	A or B	Q	B	A	X or A	X or Q	X	R
R and S		A and Q	A and B	0	0	0	X and A	X and Q	0	R
~R and S		~A and Q	~A and B	Q	B	A	~X and A	~X and Q	0	0
R xor S		A xor Q	A xor B	Q	B	A	X xor A	X xor Q	X	0
~R xor S		~A xor Q A xor ~Q	~A xor B A xor ~B	~Q	~B	~A	~X xor A X xor ~A	~X xor Q X xor ~Q	~X	-1

Figure 3. ALU Operations as a function of aS, aF, and Cin.

The F output of the ALU can be written into an R register, loaded into the Q register, or placed onto the Y bus. Although the F output is normally placed onto the Y bus, it is possible to route output-port A of the R register file onto the Y bus. This mode is called A-bypass or "A-pass-around."

The two-bit alu-Destination (aD) field, in combination with a one-bit value called sh, specifies whether R and/or Q is written and whether F or A-bypass is placed on the Y bus. The sh field is defined by certain functions of the microinstruction word (see Figure 1 for sh's definition). In general, when sh = 1 the F output is shifted one bit position before being written back into R or Q. This is accomplished inside the 2901 by 3-input multiplexers at the inputs to R and Q. What is shifted into the ends of R or Q determines the type of shift.

When sh concatenated with aD (sh,,aD) equals 001, neither an R register nor Q is written. This may be desired when writing an external register or when comparing two quantities. When sh,,aD = 000, Q is loaded with the ALU output. When sh,,aD is equal to 010 or 011, an R register is loaded with the ALU output.

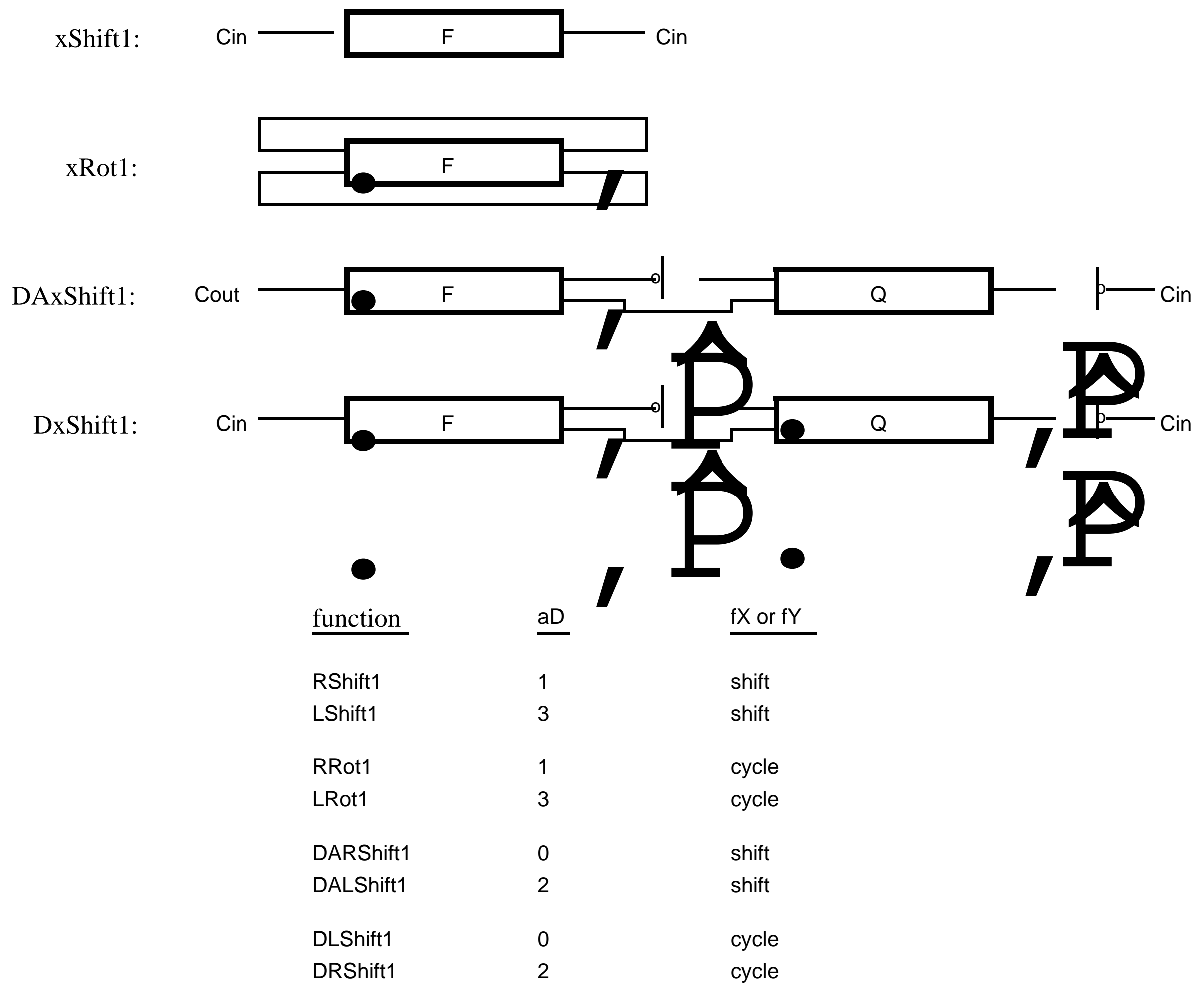
The Y bus gets the ALU output in all cases except when sh,,aD = 010, when it receives the A-bypass value. Two general rules: When A-bypass is utilized an R register must be written and it is not possible simultaneously to write R and Q with F.

When sh=1, a single-bit shifting operation is performed on the ALU output and/or Q. There are two major types of shift operations (Figure 4): a double-word shift of F,,Q and a single-word shift of F alone. These two types of shifting, combined with the two directions, are named by the four values of aD when sh=1.

For single-word shifts, the Q register is unaffected and the R register gets the ALU output shifted one bit to the left or right. The end of F which is vacated by the shift operation is replaced by Cin or the bit shifted out of the opposite side of F (a single bit rotate).

For double-word shifts, both the ALU output and the Q register are shifted together. The low-order bit of the ALU output is "connected" with the high-order Q bit to form a 32-bit quantity. The high-order bit of F which is vacated by a right double shift can be written with Cin or the Carryout (Cout) of the current ALU computation. Similarly, the low end of Q is written with the complement of Cin (~Cin) if the shift direction is left. Note that the high bit of Q is written with the complement of the low bit of F. A general rule: Shift inputs into Q are *complemented.*

In summary, the following 2901-related restrictions apply: (1) When A-bypass is utilized an R register must be written, (2) it is not possible simultaneously to load R and Q, and (3) A-bypass cannot be used with single bit shifts or when loading Q.



function	aD	fX or fY
RShift1	1	shift
LShift1	3	shift
RRot1	1	cycle
LRot1	3	cycle
DARShift1	0	shift
DALShift1	2	shift
DLShift1	0	cycle
DRShift1	2	cycle

Figure 4. CP Single-Bit Shifting

2.3.2 External 2901 Data Paths

There are two major 16-bit data buses external to the 2901: the X bus and Y bus. Both are present on the backplane; however, they are *not* general purpose, bidirectional buses. The YH bus, an 8-bit extension of the Y bus, is used for memory addressing.

The Y bus is driven only by the Y output of the 2901. It can be used to supply a memory address, memory data, U register data, or device output data.

The X bus is the major system bus and is connected to multiple drivers and multiple receivers.[5] X bus sinks are: the D input of the 2901, the RH registers, the Instruction Buffer (IB), and controller output registers. X bus sources are: the U registers, RH registers, the IB, constants, memory data, and controller input registers. The IB, RH, and controller output registers receive data from the X bus so that they can be loaded directly from memory in one cycle.

Data can be passed from the Y bus to the X bus via a 4-bit rotator, called LRotn. Data can be rotated zero, four, eight, or twelve positions to the left, as specified by the fZ field. A zero rotation allows Y bus data to be placed unaffected onto the X bus; an example is loading controller output registers from the ALU output.

Eight- or four-bit constants can be placed onto the X bus directly from the fY and/or fZ fields. The upper 8 or 12 bits of the X bus are set to zero.

The following table lists the registers which are addressable by the CP and the buses to which they are attached:

Register	inputs from	Register	outputs to	
MAR_	YH,,Y	_MD	X	Memory
Map_	YH,,Y			
IB_	X	_ib, _ibNA	X	Instruction Buffer
		_ibLow, _ibHigh		X[12-15]
		~ibPtr	X[10-11]	
RH_	X[8-15]	_RH	X[8-15]	
U_	Y	_U	X	
stackP_	Y[12-15]	~stackP	X[12-15]	
MDR_	Y	EKErr	X[8-9]	
MCtl_	Y	_MStatus	X	Memory
KOData_	X	_KIData	X	Rigid Disk
EOData_	X	_EIData	X	Ethernet
POData_/TOData_	X	_TIData	X	LSEP/MagTape
IOPOData_	X	_IOPIData	X	IOP
KCtl_	X	_KStatus	X	Rigid Disk
KCmd_	X	_KTest	X	Rigid Disk
EICtl_	X	_EStatus	X	Ethernet
EOCtl_	X			
IOPCtl_	X	_IOPStatus	X	IOP
DCtl_	X			Display
DBorder_	Y			
DCtlFifo_	Y			
PCtl_/TCtl_	X	_TStatus	X	LSEP/MagTape
TAddr_	X			

2.3.3 U Registers

A 256-word register file, called the U registers, can be written from the Y bus and read onto the X bus. These 16-bit general purpose, "slow" registers are used to hold a 16-word stack, virtual page addresses, temporaries, counters, and constants.

With respect to accessibility, U registers are situated between main memory and the R registers: they cannot be both read and written in the same cycle, nor can they be used as an operand or destination register in 16-bit ALU arithmetic.

As illustrated below, there are three ways to form an 8-bit U register address: normal, stack-pointer, and alternate.

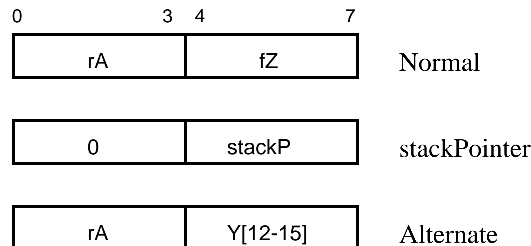


Figure 5. U Register Addressing Modes

In the normal mode, true when fS[2]=1, the U register address is defined by the concatenation of the rA and fZ microinstruction fields. This sharing of the rA field between R and U register addresses has several implications. In general, a U register can be loaded into any R register since the rB field defines the write address. However, an arbitrary U register and an arbitrary R register cannot both be ALU operands unless the upper four bits of the U register address equal the R register address. This addressing mechanism partitions the U registers into sixteen, 16-word banks such that, in one cycle, a bank's U register can only be combined with the bank's corresponding R register.

In the stack-pointer addressing mode, used when fS[2]=0, the U register is selected by the 4-bit stackPointer register (stackP) from the low bank; that is, the address is 0,,stackP. The stackP is *not* explicitly modified with this addressing mode and if an instruction uses this mode and also executes a pop or push function, the stackP before modification is used to access the U register.

The alternate mode provides indirect addressing and is used when fS[2]=1 and fZ=AltUaddr for the *previously* executed microinstruction. In this mode, the low nibble of the U address equals the least significant Y bus nibble for the *previously* executed microinstruction the same one that did the AltUaddr. Thus, instead of rA,,fZ, the U address is rA,,Y[12-15].

While reading or writing U registers, the fZ field can specify both a U register address and another function. Specifically, when fS[2-3] = 3, fZ can take on IOXIn values. This is commonly used to read an RH register or the IB while simultaneously writing a U register. When the stackPointer addressing mode is used, the fZ field is free to be interpreted as either fZNorm or a Nibble.

The U registers are also controlled by two other microinstruction fields: enSU and Cin. The enSU bit is 1 for any cycle which either reads or writes a U register. Cin must be 1 if writing, and 0 if reading. Thus, if a U register is written and the ALU function is addition or subtraction, these computations execute with Cin=1. Note that normal two's complement subtraction implies Cin=1.

2.3.4 RH Registers

Located on the X bus is the 16-by-8-bit RH register file, an extension of the R registers. The principle application of this small memory is to hold the highest-order memory address bits. Moreover, it can be utilized as general-purpose storage: for flags, counters, temporaries, and subroutine return pointers (see *DMR*).

The RH registers are addressed by the rB field, and, since this field names the R register to be written, an RH register can only be written into its corresponding R register (or the Q register).

Like the U registers, the RH registers cannot be both read and written in the same cycle. An RH register is written from the low byte of the X bus when fX = RH_ and is read onto X[8-15] when fZ = _RH. Whenever it is read onto the X bus, the high half of the bus is set to zero.

Every cycle, the 8-bit YH bus is driven with the value of the addressed RH register, thereby supplying the high order memory address bits to the Memory Control card. However, these bits are only used by the memory if a MAR_ or Map_ is specified. As a corollary to the rule that RH registers cannot simultaneously be read and written, an RH register cannot be loaded if the microinstruction also executes a MAR_ or Map_.

2.3.5 Instruction Buffer

The Instruction Buffer (IB) was designed to hold up to three Emulator macroinstructions or data bytes. It is used in a first-in, first-out manner. Data loaded into the IB from the X bus can be read back onto the X bus or be used to define a 256-way dispatch in control store. The IB is loaded by special Emulator "refill" microcode (sec. 2.6.4) while the actual control of the registers is accomplished by a hardware state machine.

The IB is maintained by the Emulator in a way that guarantees all macroinstructions will find necessary code segment operands there. Furthermore, the IB is where the 256-way dispatch is made on the next macroinstruction to be executed. This dispatch (IBDisp) occurs in c2 so that the next macroinstruction begins in c1, thereby adjoining the previous one. However, when IBDisp is executed and the buffer is not full, a microcode trap occurs and the refill microcode loads the buffer with more bytes from memory. If an IBDisp is executed and there is a pending interrupt (MInt=1), special interrupt trap (IB-Refill) microcode runs instead of the refill microcode. Since the IB is so small, IBDisp's frequently trap; however, since the IB-Refill trap runs at memory speed, this scheme of supplying operand bytes to the macroinstructions is very efficient.

This scheme is efficient from both memory bandwidth and page-fault handling perspectives. In the former case, macroinstructions would otherwise have to call an operand-fetching subroutine, which would waste time becoming cycle aligned. In the latter case, macroinstructions need not worry about a page fault from the code segment. (The occurrence of a code segment page fault can add major complications to the implementation of macroinstructions since the microcode must, before processing the fault, restore the Mesa machine state to its value at the beginning of the instruction.) The IB insures that macroinstructions can always find code segment arguments present in the IB. In this sense, the IB is more like an operand data buffer than an instruction buffer.

The minimum number of bytes in the buffer required to prevent an IB-Refill trap is three (the maximum size of a Mesa macroinstruction) and they only occur between the execution of macroinstructions. The refill code completes in one click if the buffer requires two bytes and finishes in two clicks if four are needed. Because the buffer is small, the only codebytes which do not result in an IB-Refill trap are single-byte opcodes executed from even memory locations.

The instruction buffer itself consists of three 8-bit registers, called IB[0], IB[1], and ibFront. IB[0] holds the even code segment byte and IB[1] the odd. The bytes are shuffled through ibFront in even/odd, sequential order. There are four states which enumerate the location of data bytes among the holding registers. These states are indicated by the 2-bit register ibPtr and are defined

below. The following diagram shows the four IB states (the cross-hatching indicates the position of the data bytes):

state name	bytes in IB	ibPtr
full	3	2
word	2	3
byte	1	1
empty	0	0

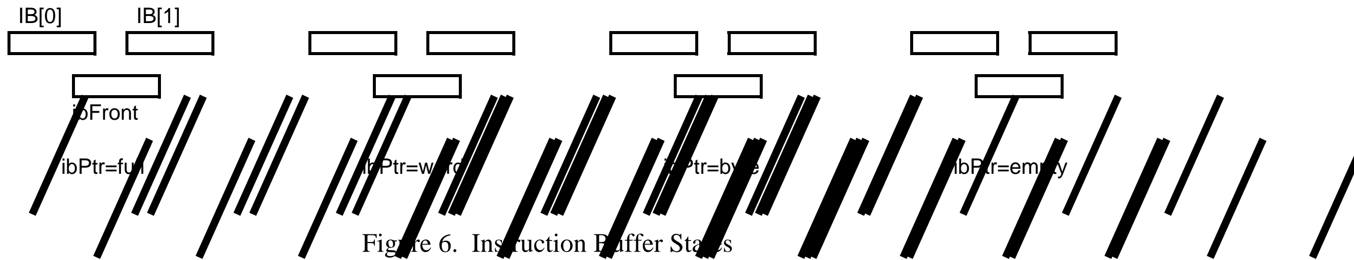


Figure 6. Instruction Buffer States

There is a total of 8 microinstruction functions which affect the IB. In general, the functions maintain the original even/odd byte ordering while updating ibPtr and ibFront. The following table lists the functions and their effect on ibPtr, ibFront, and the X bus. A discussion of the table follows, except that IB dispatches and IB-Refill traps are presented in sections 2.5.2 and 2.5.5.1.

function	new ibPtr	new ibFront	X bus _
_ib	ibPtr-1	IF ibPtr[1]=0 THEN IB[0] ELSE IB[1]	0,,ibFront
_ibNA	unchanged	unchanged	0,,ibFront
_ibHigh	unchanged	unchanged	0,,ibFront[0-3]
_ibLow	unchanged	unchanged	0,,ibFront[4-7]
IBDisp	ibPtr-1	IB[ibPtr[1]]	unaffected
AlwaysIBDisp	ibPtr-1	IB[ibPtr[1]]	unaffected
IB_	IF empty THEN word ELSE full	IF ibPtr=empty THEN X[0-7] ELSE unchanged	unaffected
IB_, IBPtr_1	IF empty THEN byte ELSE full	IF ibPtr=empty THEN X[8-15] ELSE unchanged	unaffected
IBPtr_0	word	IB[0]	unaffected
IBPtr_1	byte	IB[1]	unaffected
_ErrnIBnStkp	unchanged	unchanged	X[10-11]_~ibPtr

Figure 7. Effects of IB-related Functions

The IB is loaded from the X bus: the high-order, even byte is written into IB[0] and the low-order, odd byte into IB[1]. If the buffer is empty, then the X bus byte passes through IB[0] or IB[1] and is loaded directly into ibFront in one cycle; thus, the data can be used immediately in the cycle following the IB load.

The default IB write operation is to write ibFront with X[0-7]. However, if IBPtr_1 is coincident with IB_, then ibFront is written with X[8-15] instead, thereby throwing away the even data byte. If there are one or two bytes in the buffer, then IB[0] and IB[1] are loaded and there is no feed through into ibFront.

ibFront can be read onto the X bus: when the microcoder specifies a _ib or _ibNA, ibFront is placed onto X[8-15] and the high byte of the X bus is set to zero.

There are several variations to this basic read. With the _ibHigh function, ibFront[0-3] is placed onto X[12-15]. Analogously, _ibLow places ibFront[4-7] onto X[12-15]. In both cases the upper 12 bits of the X bus are set to zero.

When _ib is executed, a funneling process occurs: ibFront is loaded with the next byte from either IB[0] or IB[1] and ibPtr is "decremented" by one. ibPtr is gray code decremented: 2, 3, 1, and then 0. Thus, the low order bit of ibPtr divides the values of ibPtr into two classes with respect to refill: empty and not empty. (This scheme equates the empty and full states, but note that the buffer is not full when the IB-Refill trap occurs.)

Several of the microcode functions have no effect on the state of the buffer: The _ibNA function (used to read the IB without advancing ibPtr), _ibHigh, and _ibLow do not change ibPtr. Also, like the RH and U registers, it is not possible simultaneously to read and write IB; hence, the combination of IB_ and _ib in the same cycle does nothing.

The functions IBPtr_0 and IBPtr_1, when used alone, merely load ibFront from IB[0] or IB[1], respectively. They typically occur in the cycle after the IB has been loaded with a jump-target codebyte, thereby selecting the even or odd destination opcode.

The complement of ibPtr can be read onto X[12-13] with the _ErrnIBnStkp function.

2.3.6 stackP Register

The 4-bit stack pointer, stackP, is used to address one location from U register bank 0 (Sec. 2.3.3) and can be incremented or decremented independently of the 2901. The pop function decrements (modulo 16) and the push function increments (modulo 16) the stackP at the end of a cycle. Unlike the U and RH registers, the stackP can be read and written in the same cycle.

The stackP can be loaded from Y[12-15] with an fY function. However, one cycle must intercede between a stackP_ and a microinstruction which uses the stack-pointer addressing mode and expects the new value. A pop or push can be used in the intervening instruction and appropriately modifies the value loaded.

The pop and push functions have been sprinkled throughout the microinstruction function fields to ameliorate the checking of stack overflow or underflow. The push function occurs in all three function fields while pop is in fX and fZ. An outcome of this arrangement is that when push is specified in the same microinstruction as pop, the stackP does not change: it does not matter how many pop's or push's there are; as long as there are both, the stackP is unaffected. Also, multiple pops or pushs in the same instruction do not decrement or increment the stackP by more than one. Multiple pop and push functions are used to check for stack overflow or underflow (sec. 2.5.5.2).

2.3.7 pc16 Register

The pc16 register is designed to serve as a low-order, 1-bit extension of an R register; namely, the R register which holds the Emulator's macroprogram counter (PC). That is, pc16 can be used as the byte index of a PC memory address.

If fX or fZ is Cin_pc16, the pc16 bit becomes the carry input of the 2901 and pc16 is inverted at the conclusion of the cycle. Thus, Cin_pc16, in combination with ALU addition and subtraction, properly adjusts the 17-bit byte program counter PC,,pc16 (See *DMR*).

Since Cin is also the shift ends (Sec. 2.3.1), Cin_pc16 can be used to shift pc16 into the low-order bit of an R register in one cycle, thereby reconstructing a byte program counter in an R register.

Due to the hardware implementation of the carry input, when the Cin field of the microinstruction is 0, the fX version of Cin_pc16 must be used. If Cin=1, then either the fX or fZ version of Cin_pc16 can be specified.

2.3.8 Timing Limitations

The architecture of the CP allows the execution of microinstructions which will not always properly complete. This is due to either "slow" X bus operands or "slow" destination registers; that is, certain sources can not be loaded into certain destinations because the source value is not stable in time. Basically, the delay time of the source plus the setup time of the destination must be less than the cycle time, 137 nS. MASS will flag such instructions with a timing violation error.

All ALU internal register-to-register operations complete on time. All Y bus destinations can be loaded as a result of any ALU operation which does not use the X bus as an operand (except for the high 12 bits of a U register).

If the ALU operation uses an X bus operand (aS = D,A, D,Q, D,0), depending on the register, the operation may not complete in time. In general, all X bus sources can at least be loaded into an R register, which is a logical operation (aS = D,0, aF = R*or*S).

Figure 7 should answer the question: "Is a microinstruction legal with respect to X bus timing?" The table deals with all possible X bus sources and destinations: X-bus-source-to-X-bus destination, X bus ALU operands (aS = D,A, D,Q, D,0), and X bus branching and dispatching. Intersections marked with a full, half, or quarter square blob indicate legal source/destination combinations or branching phrases.

X + R represents the 3 arithmetic operations (aF = R+S, S-R, R-S) and X or R the 5 logical operations (aF = R*or*S, R*and*S, ~R*and*S, R*xor*S, ~R*xor*S). B_ implies the loading of an R register; Q_ has the same timing. pgCross refers to the automatic page cross branch with MAR_ and pageCross & OVR refer to PgCrOvDisp.

Branching and dispatching have different timing than the basic ALU operations and a potential statement must meet both conditions. In general, zero, negative, or overflow branching is not possible with any X bus operand.

The ALU performs arithmetic at three different speeds depending on which bits of the result you're looking at. Thus, figure 7 has three numbers for arithmetic operations depending on which bits of the result are of interest. ALU[0-7] are the slowest since they depend on a carry from the lookahead unit. ALU[8-11] are next as they depend on a ripple carry from the low nibble. Finally, ALU[12-15] are fastest since Cin arrivies very early relative to X bus sources. Thus, the low nibble always has the timing of a corresponding ALU logic operation.

Note that some "+1" or "-1" operations do not necessarily imply use of the X bus, but use Cin instead. Thus, R _ R + 1, NegBr is legal where R _ R + 2, NegBr is not.

All arithmetic operations with the ALU internal zero as an operand (aS = 0,Q, 0,B, 0,A, or D,0) complete on time. This obviously includes all X bus sources.

X Operation	X setup	X Source: U	MD	RH	constant, _ib ErrIBStkp	IOIn	A LRotn	(A or B) LRotn	(A+B) LRotn
X Source Time		75	97	74	59	63	91	102	131 127 105
B_X or R	47	■	■	■	■	■	■	—	—
B_X or R, ZeroBr	63	■		■	■	■		—	—
B_X or R, NZeroBr	68								
B_X or R, NegBr	62	■		■	■	■		—	—
[]_X or R, YDisp	71				■	■		—	—
B_LShift1 (X or R)	58	■		■	■	■	—	—	—
B_LRot1 X (X or R)	66	■		■	■	■	—	—	—
MAR_ X or R	78		—		■		—	—	—
Map_ X or R	78		—		■		—	—	—
MDR_ X or R	45	■	—	■	■	■	—	—	—
U _ X or R	87	—					—	—	—
IOYOut_ X or R	64	■		■	■	■	—	—	—
B_X + R	78 71 47	◤	◤	◢	■	■	◤	—	—
B_X + R, ZeroBr	97							—	—
B_X + R, NegBr	91							—	—
B_X + R, OVR	90							—	—
B_X + R, CarryBr	81				■			—	—
B_X + R, NibCarryBr	60	◤		◤	◤	◤		—	—
B_X + R, PgCarryBr	65	◢		◢	◢	◢		—	—
B_X + R, pageCross	77				◢	◢		—	—
MAR_X + R, pgCross	72	◢		◢	◢	◢			
B_X + R, YDisp	71				◢	◢		—	—
B_RShift1 (X+R)	91 87 56	◤		◤	◤	◤	—	—	—
B_RRot1 (X+R)	101 97 66	◤		◤	◤	◤	—	—	—
MAR_ X + R	78		—		■		—	—	—
Map_ X + R	110		—				—	—	—
MDR_ X + R	77 70 48	◤	—	◤	■	■	—	—	—
U _ X + R	119 114 90	—					—	—	—
IOOut _ X + R stackP_	80 73 51	◤		◤	■	◢	—	—	—
Xbus _ X, XDisp	32	■	■	■	■	■	■	■	◤
RH _ X	36	■	■	■	■	■	■	■	
IB _ X	37	■	■	■	■	■	■	■	
IOOut _ X	32	■	■	■	■	■	■	■	◤

■ Timing OK across 16 bits of result ◢ OK across low byte ◤ OK across low nibble

Figure 7. Allowable X-bus Operations

2.4 Main Memory Interface

This section discusses the interface between the CP and the memory system. As outlined earlier, a memory address is sent to the Memory Controller in c1, any data to be written is sent during c2, and returning data is available in c3. Every click is a potential memory operation: if the Emulator kept the memory 100% busy and there were no I/O, it would have available up to 2.4 megawords/s (38 Mbits/s) of bandwidth.

The memory system accepts two types of addresses: real or virtual. Real references result in a read or write to the addressed location itself. Virtual references cause the memory system to ignore the low byte of the address and then, using the remaining 16 bits, read or write the Map, located at real address 10000 hex.

For both reference types, when the mem field is set in c2 a write occurs (MDR_) and when set in c3 a read occurs (_MD). If both a read and write are specified in the same click, the original value is returned and then the location is overwritten. Furthermore, if a click specifies a MDR_ or _MD without a corresponding MAR_ then memory is not written and a potential memory Error trap does not occur.

As outlined in section x.xx, the memory system is available in a variety of sizes: real address size from 192K to 768K words and virtual address size from 4 to 16 megawords. This section assumes the maximum of both ranges: 20-bit real addresses and 24-bit virtual addresses.

2.4.1 Real Address References

When the mem bit is true in cycle 1, a real reference is caused. The microcoder specifies a real reference by using the MAR_ macro in c1. The memory address is sent to the Memory Control card on the YH and Y buses. The Y bus can be driven from either the 2901's F bus or A-bypass; hence addresses can be either pre or postmodified. The YH bus, which supplies the high-order address bits, is always driven by the RH register addressed by rB. Furthermore, YH[0-3] are ignored by the memory.

Several important things happen with a MAR_: the 2901 is divided such that the high half executes a fixed function, a special "address-overflow" branch is enabled, and an MDR_ or IBDisp in the next cycle is canceled if the branch is taken. Moreover, if a MAR_ is executed with YH[4-7] = 0 and the display controller is enabled and actually transferring bits to the monitor, then the click is suspended (See sec. 2.5.6.5).

MAR_ Effect: Split 2901

If mem=1 in c1, the 2901 is divided such that the high half executes with its aS and aF inputs equal to (0,B) and (aF *or* 3), while the low half executes the aS and aF values given by the microinstruction. This causes the high byte of the ALU output to equal the high byte of the R register addressed by rB (or its complement if aF is in [4..7]). Thus, assuming the Y bus is driven from the F bus, the 20-bit real address is rhB[4-7],,rB[0-7],,F[8-15].

This change in normal ALU function was required by the fact that the most significant memory address bits must be ready very early in the click. Only logical operations would allow the address to pass through the ALU quickly enough. The requirements are not so strict on the low order bits, so arithmetic operations are allowed on the bottom byte. This change also facilitates the combining of the virtual page number returned by a Map reference with the offset into the page contained in the low byte of an R register (see the *DMR* for examples).

An outcome of this bipartition is that a carry out from the low half does not propagate into the high half: the high byte of rB remains unchanged after a MAR_ (unless aF is in [4..7]), even if A-bypass is utilized.

The real address modes are illustrated below. In summary, if A-bypass is not used, the upper 12 bits of the memory address (the page address) come from the RH/R pair named by the rB field, while the lower 8 bits (the page displacement) are defined by the desired ALU operation. This feature can be used to combine the real-page number, as read from the Map in the previous cycle, with a displacement into the page. If A-bypass is specified, the lowest 16 address bits come from the R register addressed by rA. Hence, the 20-bit real address is rhB[4-7],,rA[0-15].

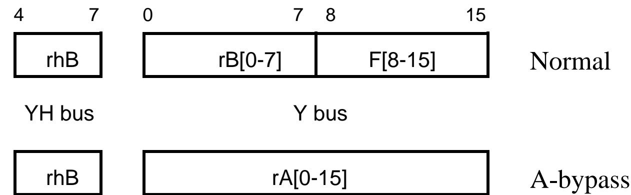


Figure 8. MAR Address Types

MAR_ Effect: pageCross Branch

The second effect of a MAR_ is that it automatically specifies a pageCross branch: 1 is *or'd* into INIA[10] if the ALU operation results in a carry out from the low half. Thus, although the carry out from the low byte does not propagate into the high byte, as discussed above, it can be detected as a transfer of control. A true pageCross branch can imply that the real address is invalid and that a remapping of the virtual address which originally generated it is necessary. Since pageCross is not *or'd* into INIA[11], other simple branches can be concurrently specified.

pageCross is defined to be (pageCarry *xor* aF[2]), where pageCarry is the carry out from the low 2901 byte. The *xor* has the effect of toggling pageCarry when doing subtraction; pageCross equals pageCarry when doing addition. The aF = (R-S) form of subtraction does not cause pageCarry to be inverted since aF[2] = 0; however, the aF = (R-S) form covers the most common subtraction requirements. See the *DMR*.

A complication of the MAR_ automatic pageCross branch is that pageCross can indeed equal 1 if the 2901 executes a logical, instead of arithmetic, function. See the *DMR*.

MAR_ Effect: Cancelation of c2 Functions

The third effect is that if pageCross = 1 during a MAR_, then a following MDR_, IBDisp, or AlwaysIBDisp in c2 is ignored. This mechanism can be used to prevent writing into the wrong page or dispatching on the next Emulator instruction when the corresponding virtual address should be remapped. This effect increases the need to avoid logic functions during a MAR_. See the *DMR*.

2.4.2 Virtual Address References

When either the fX or fY fields equal Map_ in cycle 1, a memory reference to the virtual-to-real, page-translation Map is caused. The Map is a table whose first entry is at location 10000 hex, just after the display bank. During a Map reference, the memory system uses the upper 16 bits of the virtual address (14 bits in the case of a 22-bit virtual address) to index into the table. Each entry of the table contains a 12-bit real-page number and four flags pertaining to the virtual page. Currently, a 16K table is used by the Emulator. Figure 10 illustrates the process.

The virtual address is made available to the Memory Control card on the YH and Y buses. The low byte of the Y bus is ignored and, unlike MAR_, there are no ALU side effects. Since the Y bus can be driven from either the 2901's F bus or A-bypass, addresses can be either pre or postmodified:

Figure 9. Map Address Types

For 24-bit virtual references, all of the YH bus is used. However, with early versions of the CP, which assumed a maximum 22-bit virtual address, if either YH[0] or YH[1] are 1, an Error trap resulted.

The following figure shows the format of a Map entry. See the *DMR* for a description of how the referenced, dirty, and present Map flag bits are maintained.

The mem field should not be set in c1 along with a Map_ unless MAR_'s side effects are explicitly desired. Moreover, if YH[4-7] = 0, such clicks will be suspended due to display bank contention.

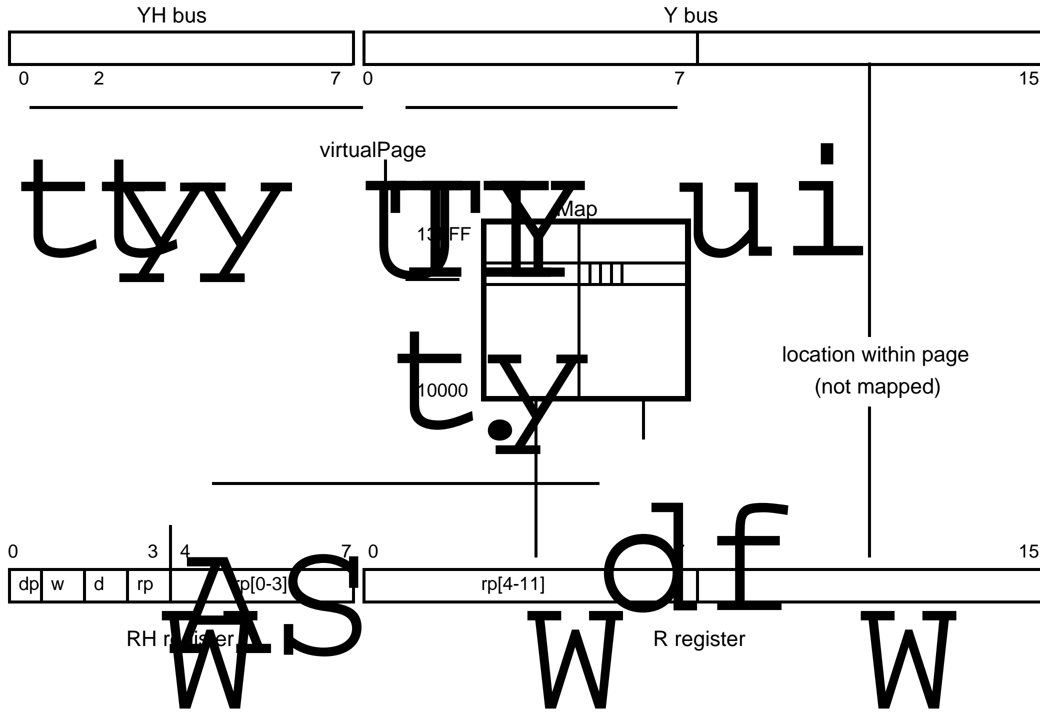
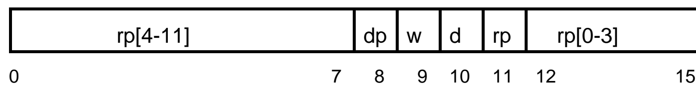


Figure 10. Virtual to Real Address Mapping



rp[0-11]	Real Page Number
dp	Dirty & Present flag
w	Write Protect flag
d	Dirty flag
rp	Referenced & Present flag

Figure 11. Map Entry Format

2.5 CP Control Architecture

This chapter discusses the algorithms used for controlling the execution of microinstructions and the interface between the IOP and the CP. Figure 12 is a block diagram of the control paths and registers.

As presented in the introduction, cycles are illimitably executed c1, c2, and c3. Every cycle, one microinstruction is decoded and executed while the next is being read from the control store (except in those clicks which have been suspended due to display bank contention). Since a device task does not execute in consecutive clicks, there is hardware to save the microprogram counter of each task while it is not running.

We first look at branching, dispatching, the Link registers, and the Error traps, as they can be described without reference to the tasking structure.

2.5.1 Conditional Branching and Dispatching

Every microinstruction can potentially branch: during each cycle, condition bits specified by the executing microinstruction are *or'd* into the next instruction's "goto"-address field (INIA) being read from control store. At the end of the cycle, this results in an address (NIA) which is used to read the next microinstruction. If the executing microinstruction does not specify a branch function, then 0 is *or'd* into INIA and, accordingly, a branch does not occur. When a microinstruction specifies a dispatch function, up-to-four bits are *or'd* into the INIA field; selecting one of up-to-sixteen target microinstructions. (The maximum of four dispatch bits was chosen in order to minimize the number which must be saved between task switches.)

Thus, all branches and dispatches take two cycles to complete: one cycle to specify the branch and one to read out the target microinstruction. The microinstruction bits required to specify a branch are fS[0-1] = DispBr and the fY field which names the branch or dispatch (Figure 13).

The notation used to specify the branching behavior is as follows: A microinstruction is located in control store at its Instruction Address, IA; the Next Instruction Address, NIA, is the control store address register; and the Intermediate Next Instruction Address, INIA, is the 12-bit "goto" address present in each microinstruction. Every cycle, the hardware *or's* the condition bits specified by fY (abbreviated DispBr) and together with a Link register specified by fX into INIA, thereby producing the NIA value used for the next cycle:

NIA[0-11] _ INIA[0-11] or DispBr[0-3] or Link[0-3].

In the case of dispatches, it is not always necessary for the microcoder to provide target instructions for each possible outcome. Any particular condition bit can be ignored by placing a 1 in its corresponding position in INIA. This method can also be used to cancel unwanted, pending branches. See the *DMR*.

Figure 13 enumerates the available branches and dispatches. Note that, in some cases, there is more than one way to branch on a particular bit and that any bit on the low half of the X bus can be branched on. The NZeroBr exists so that code can be more readily shared.

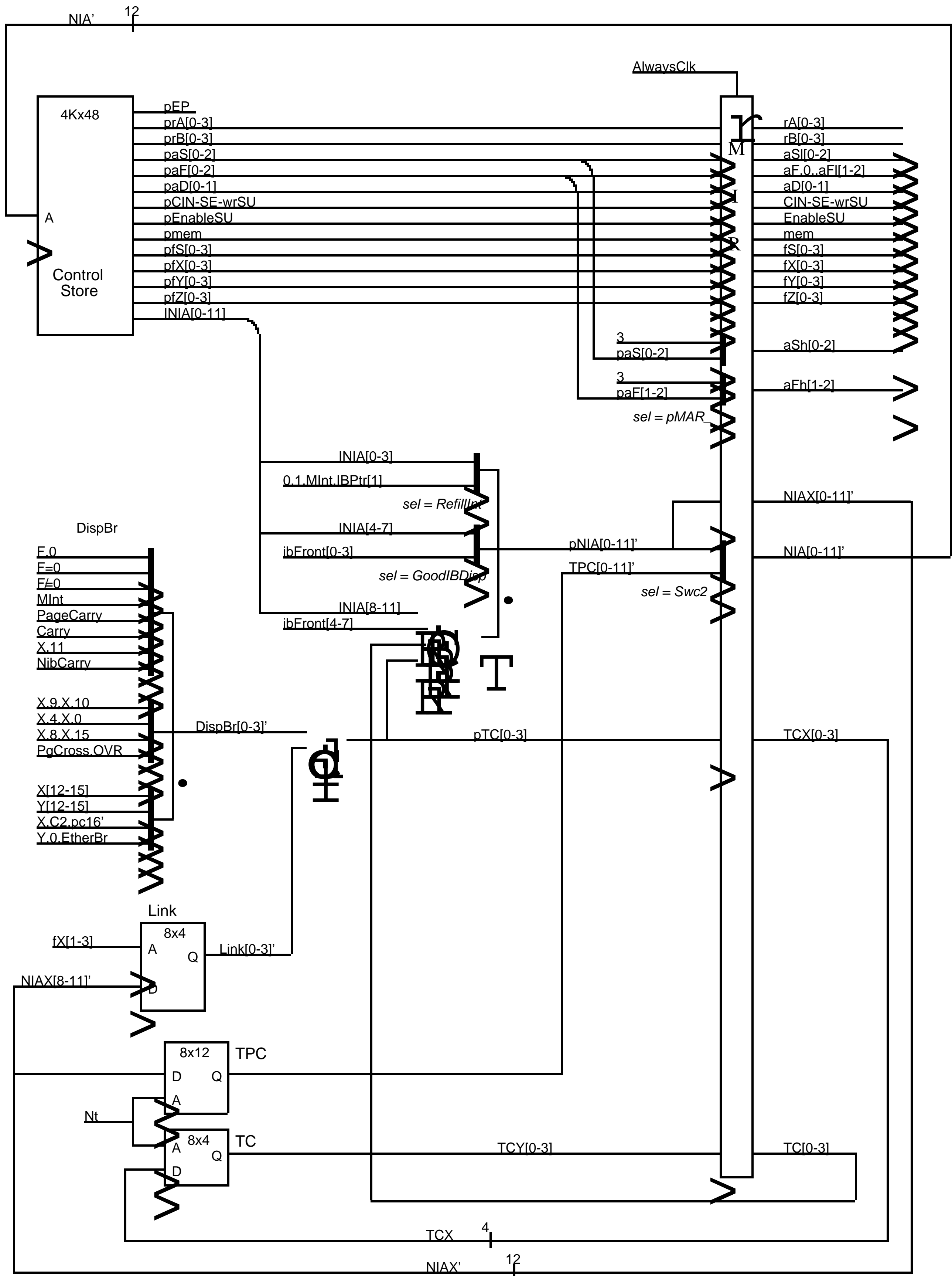


Figure 12. CP Control Paths

	source	INIA	
NegBr	F[0]	11	sign of alu result (not necessarily Y[0])
ZeroBr	F=0	11	alu output equal to zero
NZeroBr	F=0	11	alu output not equal to zero
CarryBr	Cout[0]	11	alu carry out
NibCarryBr	Cout[12]	11	alu carry out from low nibble
PgCarryBr	Cout[8]	11	alu carry out from low byte
XRefBr	X[11]	11	present & referenced Map bit
MesaIntBr	MInt	11	Emulator Interrupt (see 2.5.3)
XwdDisp	X[9],,X[10]	[10-11]	write protect & dirty Map bits
XHDisp	X[4],,X[0]	[10-11]	X (high) bus
XLDisp	X[8],,X[15]	[10-11]	X (low) bus
PgCrOvDisp	PgCross,,OVR	[10-11]	pageCross & alu overflow
XDisp	X[12-15]	[8-11]	low nibble of X bus
YDisp	Y[12-15]	[8-11]	low nibble of Y bus
XC2npcDisp	X[12-13],,c2,,~pc16	[8-11]	X bus, cycle2, inverse of pc16
YIODisp	Y[12-13],,bp[39],,bp[139]	[8-11]	I/O branches (bp=backplane pin)
IBDisp	ibFront	[4-11]	Instruction Buffer
LnDisp	Linkn	[8-11]	Link register (n=0..7)

Equivalent names: EtherDisp = YIODisp, XDirtyDisp = XLDisp.

Figure 13. Branches and Dispatches

2.5.2 Instruction Buffer Dispatch

The instruction buffer dispatch, IBDisp, is a special dispatch since more than four bits are *or'd* into INIA. Consequently, IBDisp can only occur in c1 or c2, and, by convention, it is restricted to c2. See section 2.3.5 for a discussion of the instruction buffer.

Assuming that the instruction buffer is full, IBDisp can cause a 256-way dispatch based on the value of ibFront: NIA[4-7] is set to the high nibble of ibFront and the low nibble of ibFront is *or'd* with INIA[8-11]. (Due to the *or* operation into the low nibble of INIA, simultaneous Link register dispatches are possible.[6]) INIA[0-3] is unaffected by the IBDisp (except by the four IB-Refill trap values); therefore, up-to-twelve 256-way dispatch tables can be concurrently used.

If the buffer is not full (ibPtr = full) when an IBDisp is executed, or there is a pending interrupt, then an IB-Refill trap occurs (See 2.5.5.1).

A special version of IBDisp, called AlwaysIBDisp, never IB-Refill traps: AlwaysIBDisp dispatchs on ibFront even if there is a pending interrupt (MInt = 1) or the buffer is not full. It is used in the Emulator refill and jump microcode (sec 2.6.4) to dispatch on ibFront while the buffer is still being filled. AlwaysIBDisp is encoded as fY = IBDisp and fZ= IBPtr_1.

If the microinstruction executed before an IBDisp or AlwaysIBDisp causes an IB-Empty Error trap, or it contains a MAR_ and the 2901 computation results in pageCross = 1, then the IB dispatch (or possible IB-Refill trap) does not occur and ibPtr remains unaffected. Since INIA is not modified in this case, control transfers to the first entry of the macroinstruction dispatch table. (Accordingly, Emulator opcode 0 should not be assigned to a macroinstruction.)

2.5.3 MInt Register

The 1-bit MInt register can be used to interrupt the contiguous execution of Emulator macroinstructions. When MInt is set in a antecedent cycle, IBDisp traps instead of dispatches (1.5.5.1). MInt is set with fY = MesaIntRq and cleared with fY = ClrIntErr. (ClrIntErr also resets the EKErr register.) See the *DMR* for user conventions.

2.5.4 Link Registers

The CP has eight, 4-bit Link registers which can be loaded from the low four bits of the control store address. Generally, these Link registers can be used to hold four bits of state information derived directly from the flow of control. Thus, previously determined state information can be easily recalled by dispatching on a Link register. Moreover, macroinstructions can share common code at various stages of their execution and Link registers can be used for subroutine call and return structures. See the *DMR*.

The Link register addressed by fX is written with the low nibble of NIAX (which equals NIA except during a task switch in c2. see 2.5.6.4). A Link register is written when fX is in [0..7] and NIA[7] = 0: Link[fX] _ NIAX[8-11].

A Link register is *or'd* into the low nibble of INIA when fX is in [0..7] and NIA[7] = 1, causing a potential 16-way dispatch. Since the Link register is designated by an fX function, the fY field is free to specify other condition bits which can be *or'd* into INIA[8-11].

If the preceding microinstruction does not specify a branch or dispatch condition, then the Link register is loaded with a constant. However, if the prior instruction contains a branch or dispatch, the value loaded depends on the outcome of the branch or dispatch. (The low four bits of the IB dispatch value can also be recorded in this way.) See the *DMR*.

2.5.5 Microcode Traps

There are two general classes of microcode traps: IB-Refill and Error. The former only occurs as the result of IBDisp's; hence between the execution of macroinstructions. There are four IB-Refill trap locations which are a function of ibPtr and MInt. Error traps can occur in any cycle and always trap to location 0 in c1. The Error traps have priority over the IB-Refill traps and cannot be disabled.

2.5.5.1 IB-Refill Traps

If an IBDisp is executed and ibPtr = full or MInt = 1, then the ibFront dispatch does not occur and instead an IB-Refill trap is caused. Specifically, ibPtr is unaffected, INIA[4-11] is not modified, and NIA[0-3] is set to the 4-bit quantity 0,,1,,MInt,,ibPtr[1]. The following table summarizes the interpretation of the IB-Refill trap locations. (If an IB-Refill trap occurs and MInt = 0, then ibPtr can not equal full.)

NIA[0-3]	MInt	ibPtr
4	0	empty
5	0	not empty (i.e., byte or word)
6	1	empty or full
7	1	byte or word

AlwaysIBDisp never IB-Refill traps and a MAR_ caused pageCross branch or IB-Empty Error trap cancels a potential IB-Refill trap.

2.5.5.2 Error Traps

Error traps can result when one or more predefined error conditions are detected in the CP or memory. All error traps cause the instruction at microstore location 0 to be executed in c1 by the Emulator or Kernel, depending on the error type. Error traps cannot be disabled.

The EKErr register, read onto X[8-9] with _ErrnIBnStkp, names the type of error:

EKErr	Type
0	control store parity error
1	Emulator memory error
2	stackPointer overflow or underflow
3	IB-Empty error

If, coincidentally, two or more errors occur at the same time, smaller values of EKErr are reported. The error types are also accumulated until EKErr is reset: the minimum value is reported when EKErr is read. Error traps have priority over the IB-Refill trap. See the *DMR* for example error-handling microcode.

EKErr is reset by the ClrIntErr function which, as a side effect, also resets any pending interrupts.

With early CP modules, an EKErr value of 1 can also imply that a 23- or 24-bit virtual address had been used by the Emulator. In this case, the ErrorLogging register in the Memory Controller is read to determine whether the error is actually a double-bit memory error. Since the Memory Controller can now accept 24-bit virtual addresses, this interpretation of EKErr=1 is no longer necessary (beginning with CP etch 4, Rev N).

CS Parity Error

If the parity of a microinstruction read by any task is odd, then control is transferred to location 0 at the Kernel task level. Since the Kernel is the highest priority task, no other microcode tasks can execute. The CS-parity-error signal is sampled by the IOP, which can consequently sense a failed control store chip.

If the instruction read from microstore in c1 has bad parity, then the Kernel runs at location 0 in the next c1. If the parity error occurs in c2 or c3, then there is a one click delay before the Kernel executes at location 0 in c1. This intervening click can be executed by any task.

Emulator Memory Error

If the Memory Controller indicates a double-bit memory error in c3 during an _MD executed by the Emulator, then a trap to location 0 in c1 occurs at the Emulator task level.

The hardware requires the execution of one additional Emulator click between the c3 which errored and the trap at location 0. Thus, other tasks and an additional Emulator click can intervene between the occurrence of the error and the trap code.

This trap only occurs for memory errors incurred by the Emulator task: device tasks must explicitly utilize the ErrorLogging register in the Memory Controller. Yes, the memory address is lost (as well as the syndrome if other memory reads occurred since the error).

Stack Pointer Overflow or Underflow

If a pop or push is executed with the values of the stackPointer given in the following table, then a trap to location 0 in c1 at the Emulator task level occurs (the stackP is still modified).

The hardware requires the execution of one additional Emulator click before the trap at location 0. Thus, other tasks and an Emulator click can intervene between the occurrence of the error and the trap code.

Multiple pop's and push's can be specified per microinstruction in order to ameliorate the detection of Stack overflow or underflow. For instance, fXpop (i.e., the pop in the fX field), fZpop, and push executed together leave the stackPointer unmodified, yet simulate two pop's with respect to stack underflow detection. fXpop with push checks for stack overflow while not moving the stackPointer, and, likewise, push and fZpop check for underflow. The following table enumerates the cases.

functions	stackP	Trap is	if stackP is
pop	-1	underflow	0
push	+1	overflow	15
fXpop, push	0	underflow	0
push, fZpop	0	overflow	15
fXpop, fZpop	-1	underflow	0 or 1
fXpop, fZpop, push	0	underflow	0 or 1

If the Emulator top-of-stack (TOS) element is kept in an R register and the rest of the Stack in the U registers, and it is assumed that TOS can always be stored away into the Stack, then these values imply a maximum stack size of 14 words.

IB-Empty Error

If an _ib, _ibNA, _ibLow, or _ibHigh is executed when ibPtr=empty, then an IB-Empty Error trap occurs to location 0 in c1 at the Emulator task level. If the IB-Empty Error occurs in c1, a MDR_ in the next cycle is canceled. (Furthermore, an IBDisp is ignored, but this fact is of no particular value.)

In normal operation (sec. 2.3.5) the IB is always guaranteed to have enough operand bytes (two) before a macroinstruction begins executing. However, when the macroprogram counter points to the last word of a page, the buffer is intentionally not refilled by the Emulator "refill" microcode and the IB-Empty trap can occur, indicating that control has actually proceeded across a page boundary. See the *DMR*.

If the IB-Empty error occurs in c1, then control transfers to location 0 in the next Emulator c1. However, if the error occurs in c2 or c3, the hardware requires the execution of one additional Emulator click before the trap at location 0. Consequently, other tasks and an Emulator click can intervene between the occurrence of the IB-Empty error in c2 or c3 and the trap code. In particular, if such a click executed a MDR_ with an address which was a function of an IB value read in the previous c2 or c3, then a random memory location can be written.

The IB is not read during c2 or c3 of a macroinstruction's last click. However, the microcoder must ensure that, immediately following an _ib, _ibNA, _ibLow, or _ibHigh function executed in c2 or c3, there is not a memory write with a MAR_ or Map_ address which is a function of the IB value read in c2 or c3. (This is not checked for by MASS.)

2.5.6 Task Scheduling and Switching

A task is the microcode which supports an IO device or the Emulator. A device task runs whenever the device controller in the Dandelion asserts its "wakeup" request. Since a device task can only run during its pre-allocated clicks, a controller's maximum memory latency and maximum memory bandwidth is an outcome of its preassigned location within the round.

The Emulator and Kernel tasks behave differently than device tasks. The Kernel task is a special task used for communication between the CP and IOP (see 2.5.6.6). The Emulator task has no fixed assigned slot in the round: it executes during a click which a controller has opted not to use. Since devices do not utilize all of the bandwidth implied by the round structure, there is always a minimum number of clicks available to the Emulator.

2.5.6.1 Task Allocation

The CP can control a maximum of 8 tasks. Currently, there are 6 wakeup lines (5 of them on the backplane) which can request microcode service. The eight task numbers are allocated between the devices, Emulator, and Kernel as follows:

0	Emulator
1	Display or LSEP or MagTape
2	Ethernet
3	Refresh (Auxiliary)
4	Disk (Rigid)
5	IOP
6	IOP control store read/write address
7	Kernel

The Dandelion is configured at boot time so that either the Display, or the LSEP, or the MagTape can use task number 1, but all three can not simultaneously use task 2. Normally, the Display task controls the refreshing of memory, but when the LSEP or MagTape (or other Option board controller) is active instead of the Display, then the Refresh task has this responsibility. Similarly, the Disk task cannot be simultaneously used by both the SA4000 and SA1000. Task 6 is currently not assigned to an actual device: instead it is used by the IOP as an address register when reading or writing the control store (see 2.5.6.7).

2.5.6.2 Click Allocation

There are two types of rounds: a standard 5-click round and an extended 10-click round. The standard round is used with the HSIO board (Shugart SA4002 or SA1002 disks) and the extended round with the HSIO-LD board (LDC, or LargeDiskController: Trident drives). The extended 10-click round is an "even" 5-click round followed by an "odd" 5-click round. In the even rounds, the Ethernet task has claim to click 3, and in the odd rounds the Trident disk controller does.

Click 4 is special because the Display Controller hardware guarantees that memory references to the display bank can never abort in this click. In order to refresh memory and maintain the cursor, the Display and Refresh tasks are assigned to this click. When the Display is on, the Display task will start in click 4 of the 11th round of a Display line. In contrast, the Refresh task will begin with the 1st round of a Display scan line.

The LSEP also uses click 4 since its band buffers are located in the Display Bank. Moreover, because of hardware pin limitations, the LSEP and Display wakeup requests are *or'd* together (on the HSIO board). Thus, if both the Display and LSEP are enabled, their wakeup requests will be irresolvable. (Note the single microcode function, ClrDPRq, is used to reset *both* their wakeup requests.) Also in click 4, the Display-LSEP wakeup request has priority over the Refresh request. Conversely, due to special hardware in the MagTape controller, the Refresh request has priority over the MagTape request.

The following tables show the standard and extended rounds:

Standard Round:

Click	Task
0	Ethernet
1	SAx000 Disk
2	IOP
3	Ethernet
4	Display-LSEP-MagTape OR Refresh

Extended Round:

Click	Task
0-0	Ethernet
0-1	Trident Disk
0-2	IOP
0-3	Ethernet
0-4	Display-LSEP-MagTape OR Refresh
1-0	Ethernet
1-1	Trident Disk
1-2	IOP
1-3	Trident Disk
1-4	Display-LSEP-MagTape OR Refresh

2.5.6.3 Click Bandwidth Utilization

The following table summarizes the bandwidth availble to each device and the percentage which remains for the Emulator when the controller is transferring data. (Pre- and post-data-transfer overhead, which normally utilizes 100% of device clicks, is not included.) Note that the IOP only transfers one byte per click, so its maximum available rate is actually 3.9 Mbits/s.

Device	BW allocated (Mbits/s)	BW used (Mbits/s)	% remaining for Emulator
Ethernet(w/SAx000)	15.6	10.0	36
Ethernet(w/Trident)	11.7	10.0	15
SA4000	7.8	7.14	9
SA1000	7.8	4.27	45
Trident	11.7	9.6	18
Display (microcode)	7.8	1.1	86
IOP	7.8	2.0	26
LSEP & Refresh	7.8	3.7+1.1	38
MagTape & Refresh	7.8	.6+1.1	78

Even with the Ethernet, SA1000, and IOP concurrently transferring data and the Display microcode refreshing memory, the Emulator still executes 60% of the time.

2.5.6.4 Tasking Hardware

The CP control hardware was designed to hide the details of task switching from the programmer. Since tasks are frequently resumed and suspended by controller wakeup requests, the hardware performs all the necessary start upand stop functions: every click it saves the current task's microprogram counters and pending condition bits and, when it is scheduled to run again, it restores them. Figure 14 illustrates the process, outlined below.

Every c2 the Schedule Prom in the CP, on the basis of the controller wakeups and click number, decides which task (Nt) will run in the next click. Also in c2, the Switch Prom, on the basis of Nt, the currently executing task (Ct), and Wait (x.xx), decides whether to activate the task switching logic (and, if so, sets Swc2 _ 1). A task switch has two parts dealing with the outgoing and incoming microprogram counter and conditions: (1) a restore process and (2) a save process.

(1) The Temporary Program Counter (TPC) array holds the eight 12-bit task microprogram counters. If it is cycle 2 and a task switch is occuring, the TPC, as addressed by the next task number, is the source of the control store address. The next task's first micronstruction is subsequently read in c3 and executed in the following c1. In short, NIA _ TPC[Nt] at the end of c2.

At the same time the next task's microprogram counter is being read from TPC[Nt], the saved condition bits are read out of the Temporary Conditions array, TC, and latched into the TC regsiter. During c3, TC is *or'd* with the next task's first microinstruction INIA field, which is being read from the microstore. In summary, the saved condition bits are read during c2 from TC[Nt], latched into the TC register, and in c3 *or'd* with INIA.

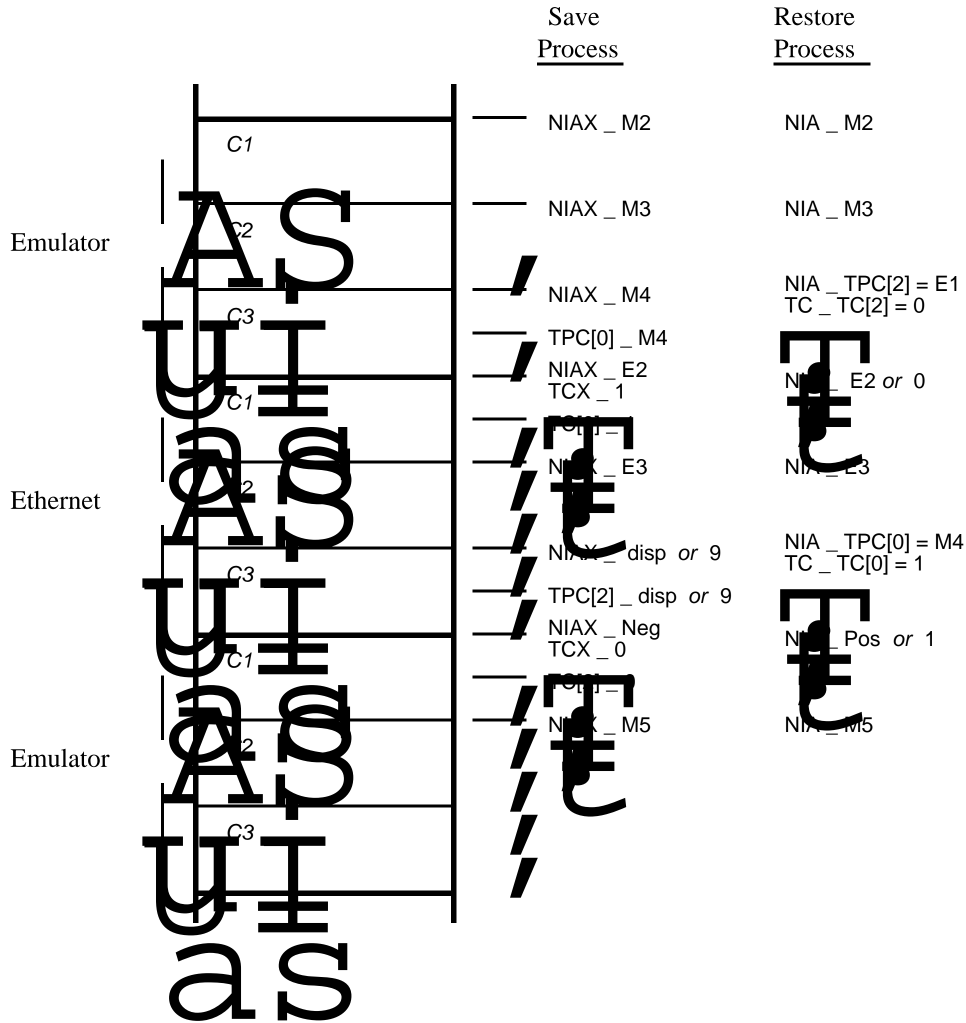
(2) The current task's Next Instruction Address (which would have been loaded into NIA if there were no task switch) is latched into the NIAX register at the end of c2 and then saved in the current task's TPC location during c3. In general, every c3, TPC[Nt] _ NIAX. (Note that in c3, Nt equals the task currently executing.)

Furthermore, the condition bits of the task currently executing (which would have been *or'd* into INIA) are latched into the TCX register at the end of c3 and then saved into the TC array in c1. In general, every c1, TC[Nt] _ TCX. (In c1, Nt actually equals the task which executed in the previous click. The condition bits are saved in c1 because there is not enough time in c3 to write them into a RAM.)

The following table summarizes when the TPC and TC are read and written and the interpretation of Nt:

cycle	operation	Nt
end of c2	NIA _ TPC[Nt]	next task
c3	TPC[Nt] _ NIAX	current task
end of c3	NIA _ INIA *or* TC	
end of c3	TCX _ DispBr *or* Link	
c1	TC[Nt] _ TCX	previous task

The TPC and TC RAMs are written every click (except suspended clicks) even if there is not a pending task switch. Consequently, if the Emulator is suspended because of Display bank interference, it's correct restart address is available in the TPC.



{Emulator microcode for above example.}

```
M1:     Noop,                 c1;
M2:     Noop,                 c2;
M3:     [] _ -1, NegBr,       c3;

M4:     BRANCH[Pos, Neg],     c1;
Pos:    GOTO[ME]              c2;
Neg:    Noop                  c2;
M5:     Noop                  c3;
```

{Ethernet microcode for above example}

```
E1:     Noop                  c1;
E2:     XBus _ 9, XDisp,      c2;
E3:     DISP4[disp],          c3;
```

Figure 14. Demonstration of Tasking Mechanism:

Where the Emulator task (0) is preempted by the Ethernet task (2) for one click.

This example demonstrates a pending branch across the task switch for the Emulator and shows when the TPC and TC arrays are written and when NIAX is not equal to NIA.

The Save Process refers to the writing of the TPC & TC arrays, while the Restore Process refers to the reading out of TPC & TC.

2.5.6.5 Display Bank Interference

If any task references the dual-ported Display bank (lowest 64K of real memory) and the Display controller is reading the bank, then the task is suspended for the duration of that click; that is, no microinstructions are executed during the suspended click. Click suspension is always in multiples of clicks and the c1-c2-c3 structure is not modified. Device tasks should not reference the Display bank (unless the Display is off).

In particular, the Emulator task is suspended until either it is scheduled for click 4 or the Display controller relinquishes the low bank. This reduces the Emulator's maximum possible bandwidth into the low bank by about half (47%) when the Display is active: from 38.9 to 18.3 Mbits/s (1.1 megaword/s).[7]

Clicks are suspended by the signal Wait which gates off all clocks which can change sensitive state information. In the schematics, such clocks are labeled WaitClock, in contrast with the normal AlwaysClock. Wait is defined

> Wait _ (MAR_ *and* YH[4-7]=0 *and* Disp-Proc'=0) *or* (IOPWait *and* c1)
> *or* (Wait *and* c2) *or* (Wait *and* c3).

When Wait is true, the following registers and RAMs are not written: R, Q, U, RH, stackP, IB[0], IB[1], ibFront, ibPtr, Link, TC, TPC, MInt, pc16', and Errors (Memory, stackPointer, CSParity, IBEmpty). By contrast, the following are unaffected by Wait: MIR, NIA, NIAX, TCX, TC, KernelReq, EKErr, and schedular task states (Nt, Ct, Pt, Swc3).

Since the Microinstruction (MIR) and Next Instruction Address registers' (NIA) clocks are unaffected during suspended cycles, the decoded signals from the MIR can change during an aborted click. However, this does not result in a random sequence of decoded microinstructions: the MIR output in c1, c2, and c3 is equal to the values it would have had if the click were not suspended. This is because the microinstruction loaded into MIR is always defined by an NIA which is unaffected by any invalid states generated during the suspended click: cycle 1's MIR output is defined by the NIA read from the TPC (in c2), cycle 2's by the value of INIA *or* TC (computed in c3), and cycle 3's by INIA *or'd* with conditions bits specified in c1 (which are not effected by WaitClock in c1). Furthermore, if the Emulator is suspended for consecutive clicks, the MIR output is the same for each click since NIA is reloaded from the TPC during suspended clicks.

2.5.6.6 Kernel Task

The Kernel task is used for supporting the debugging of the CP (e.g., breakpoints, reading/writing CP registers) and handling the CP-IOP communication while booting (e.g., memory refresh during control store read/write). When the Kernel task is enabled, it executes in all clicks, preempting all device tasks and the Emulator.

The Kernel task runs if there is a CSParityError, IOPWait is true (2.5.6.7), or the microcode function EnterKernel is executed. If EnterKernel is executed in c1, the Kernel runs in the next click. However, if executed in c2 or c3, an Emulator or device click can intervene before the Kernel runs. When the Kernel task is started, the Switch Prom does not cause a task switch; hence, a breakpoint microinstruction can specify an entry point into the Kernel.

The Kernel task request remains active until reset by the ExitKernel function. An ExitKernel is overridden by a pending IOPWait or CSParityError. When ExitKernel is executed in c1, the next click can be executed by another task (depending on which click the ExitKernel is in and the wakeup requests).

2.5.6.7 CP-IOP Interface

The IOP interfaces with the CP as both a standard I/O controller and as a boot loader/debugger. This section deals with the booting interface: the control lines used to load the control store and initialize the tasks' microprogram counters (TPCs). The following signals are used between the IOP and CP:

SwTAddr	high level causes Nt = IOPTPCHigh[0-2] and NIAX[0-4] = IOPTPCHigh[3-7] and NIAX[5-11] = IOPData bus
IOPWait	high level sets Kernel wakeup request and WaitClock is suspended
WrTPCHigh	positive edge writes IOPTPCHigh with IOPData bus
WrTPCLow	pulse causes TPC[Nt] _ NIAX
CSWE[n]'	pulse writes a control store byte with IOPData bus
ReadCSEn'	places CS byte, TPC, & TC onto IOPData bus
ReadCS[n]	selects CS, TPC, & TC bits to use with ReadCSEn'

The basic algorithm for reading or writing control store is to first write TPC[6] with the address of the location to be accessed and then read or write data bytes (addressed by CSWE[n]' or ReadCS[n]) while allowing the Kernel to Refresh memory if necessary. Although all of the tasks' TPCs can be initialized, the TC registers cannot be loaded by the IOP.

In general, when reading or writing a TPC location or CS byte, both SwTAddr and IOPWait must be high and the value of Nt (loaded into IOPTPCHigh) must be 6 or 7. When SwTAddr is true, Nt and NIAX are defined by the IOPTPCHigh register instead of their normal sources. This allows the IOP to address and supply data directly to the TPC RAM.

Setting IOPWait causes the Wait line to be high. Thus, registers clocked by WaitClock cannot be loaded with spurious data while a TPC or CS location is being written. (Moreover, the CSParityError trap cannot occur.) IOPWait also sets the Kernel wakeup request so that the Kernel task runs when IOPWait is removed.

While IOPWait=1 and Nt = 6 or 7, the Switch Prom causes a continuous task switch; that is, Swc2 is always true and NIA is set to the value of TPC[6] or TPC[7]. In this state, the Kernel microcode does not run and its TPC does not change. However, after writing one byte of control store or one TPC location, it may be necessary to refresh main memory. In this case, IOPWait and SwTaddr are reset and, since the IOPWait caused the Kernel wakeup request to be set, the Kernel begins running at the saved TPC location and executes the required number of Refresh functions or performs a function enumerated by the IOP via the normal I/O interface (e.g., _IOPIData, _IOPStatus).

The following table shows which control store bytes are read or written with ReadCSEn' and CSWE[n]'. Note that when writing the control store the inverse of the data must be supplied on IOPData.

ReadCS	CSWE[n]	IOPData[0-7]
0	a	rA, rB
1	b	aS, aF, aD
2	c	ep, Cin, EnableSU, mem, fS
3	d	fY, INIA[0-3]
4	e	fX, INIA[4-7]
5	f	fZ, INIA[8-11]
6		TC, TPC[0-3]
7		TPC[4-11]

2.6 Input/Output Interface

The CP and the high speed devices were mutually designed within one framework and are inexorably bound together: the I/O bus is the same as the CP's main data bus (the X bus), the I/O register control is directly encoded into the microinstruction format, and the devices depend on the preallocated click structure for guaranteed memory latency and bandwidth. This intimate relationship between the devices and the processor exists in order to absolutely minimize the overall system cost. By sharing the ALU among several controllers, overlapping memory accesses with ALU computation, and guaranteeing memory latency, very small IO controllers can be built. This section exists because it is possible to design different disk or display controllers on the HSIO board, new high speed controllers on the Option board, and new Memory systems.

2.6.1 CP-IO Interface

The following signals and buses are used between the CP and a typical device controller, called Dev:

X bus	16-bit data to or from memory or 2901
Y bus	16-bit data from 2901
DevReq'	task wakeup request to CP Schedule Prom
DevCtl_'	signal from CP to load controller control register from X or Y Bus
DevOData_'	signal from CP to load controller data register from X Bus
_DevStatus'	signal from CP to place controller status onto X Bus
_DevIData'	signal from CP to place controller data onto X Bus
ClrDevRq'	signal from CP to reset controller wakeup request
DevStrobe'	signal from CP for general use by controller
IODisp	CP branch on a controller state
Wait	level from CP to gate off WaitClock

Normal CP-Controller interaction (for input) goes something like: (1) A controller receives a word of data, (2) the controller activates its wakeup request, (3) the controller's task runs in its allocated click, (4) the microcode reads the data from the controller to main memory or 2901, and (5) the controller resets its wakeup request. In general, the wakeup request is either explicitly turned off by the task via ClrDevRq' or is turned off by the controller when it senses a _DevIData', DevOData_', or DevStrobe'. It is explicitly assumed that a controller only causes wakeups when data transfers are pending (or when directed by its task) in order to minimize the impact on the Emulator.

A device's wakeup request must be turned off by the end of the cycle 1 which follows the service click in order to prevent a task from accidentally running again. Since the device's wakeup request must be 2-level synchronized, this implies that the reset-wakeup function must be executed in c1 or c2 for those devices which have a two-click minimum separation.

In general, all controller control registers should be clock'd with WaitClock so that spurious device actions are prevented while writing control store. If a control signal can be used by an Emulator click which could be suspended, it should also be gate'd with WaitClock. Device tasks should not reference the Display bank unless the Display is off.

2.6.2 Controller Latencies

A controller's data buffer size depends on how often the buffer is serviced and what kind of wakeup scheme is employed. There are two basic wakeup strategies: post and prerequesting. In the former case, the wakeup request is raised after the device buffer is available to be read/written by the CP. In prerequesting, the wakeup request is raised before the device buffer is actually available. Only the SAx000 disk uses prerequesting. Where a task must process some of the data and cannot transfer a word per click, then a FIFO is usually used as a buffer (as in the Ethernet). However, when little or none of the data must be examined by the microcode, then a simple register buffer is sufficient (as in the rigid disk controllers and LSEP).

In order to avoid overruns with the postrequesting scheme, the maximum microcode service latency plus the wakeup-request synchronizer delay must be less then the data rate:

$$L_{max} + s_{max} < b/r,$$

where b is the number of bits of buffering, r is the data rate of the device (in Mbits/s), L_{max} is the maximum latency (in mseconds), and s_{max} is the synchronizer delay (equal to 2T, where T = .137 msec). If the task microcode transfers one word per click, then

$$L_{max} = 3dT + 4T \quad \text{for output, and}$$
$$L_{max} = 3dT + 3T \quad \text{for input,}$$

where d is the maximum seperation between device clicks. If the microcode does not always transfer a word per click, then L_{max} is correspondingly greater.

For prerequesting, the wakeup request cannot be made too early, thus the constraint

$$s_{min} + L_{min} - t_{handoff} > 0,$$

where $t_{handoff}$ is the time for the CP to read the buffer (equal to T) or the controller to read the buffer (about .05 msec)). If prerequesting begins p device bit times before the buffer is ready, then

$$s_{min} = 2T - p/r, \text{ and}$$
$$s_{max} = T - p/r.$$

Since L_{min} = 5T for output and 4T for input, p must satisfy the following conditions in order for prerequesting to work ($t_{handoff}$ = 0 for output):

$$[rT(3d + 6) - b] < p < 6rT \quad \text{for output, and}$$

$$[rT(3d + 5) - b] < p < 4rT \quad \text{for input.}$$

For SA4000 write or verifty operations: $4.54 < p < 5.51$!

2.6.3 IO Controller Design Rules

Since replacement or augmented controllers are being designed for the Dandelion, the following design rules should be followed in order to guarantee correct operation. Figure 15 illustrates the proper application of the CP interface signals.

(1) CP control signals such as DevReq', DevCtl_', _DevIData', ClrDevRq', and DevStrobe' originate from an SN74S138 decoder and therefore must not be used in an asynchronous way, such as the clock input of a register. These CP signals must be synchronized to the CP clock or gate'd with pAlwaysClk or pWaitClk.

(2) Controller input buffers must be either an SN74S240 or SN74S374 (or equiv) and the CP control signal which enables them onto the X bus, such as _DevIData' or _DevStatus', must be connected directly to the output enable input without being gate'd in any way.

(3) If there is more than one output register on the board, the X bus must be buffered with an SN74S241 (or equiv) before routed to the registers. The CP control signals which load the output registers, such as DevOData_' or DevCtl_', can be modified per the constraints of a clock qualifier signal (see (5)).

(4) The device wakeup request signal, DevReq', must come from an SN74S374 (or S74, or equiv) and must be synchronized by at least 2 such FF's.

(5) The clock qualifying structure shown in figure 8 must be used: the S02 is located in the position nearest backplane pins 1-10 and the "qualifier" gates are no further away then the center of the board, their preferred location. Clock qualifier terms should be valid by 94 nanoseconds after the positive (active) edge of AlwaysClk. Clock'd registers should be no more than 10" from their qualifier gate.

pWaitClk must be used for any register which, if spruiously loaded during a control store boot, can activate a device function (e.g., disk write enable). Such registers should also be reset by IOPReset' which is *or'd* with the power supply on/off reset.

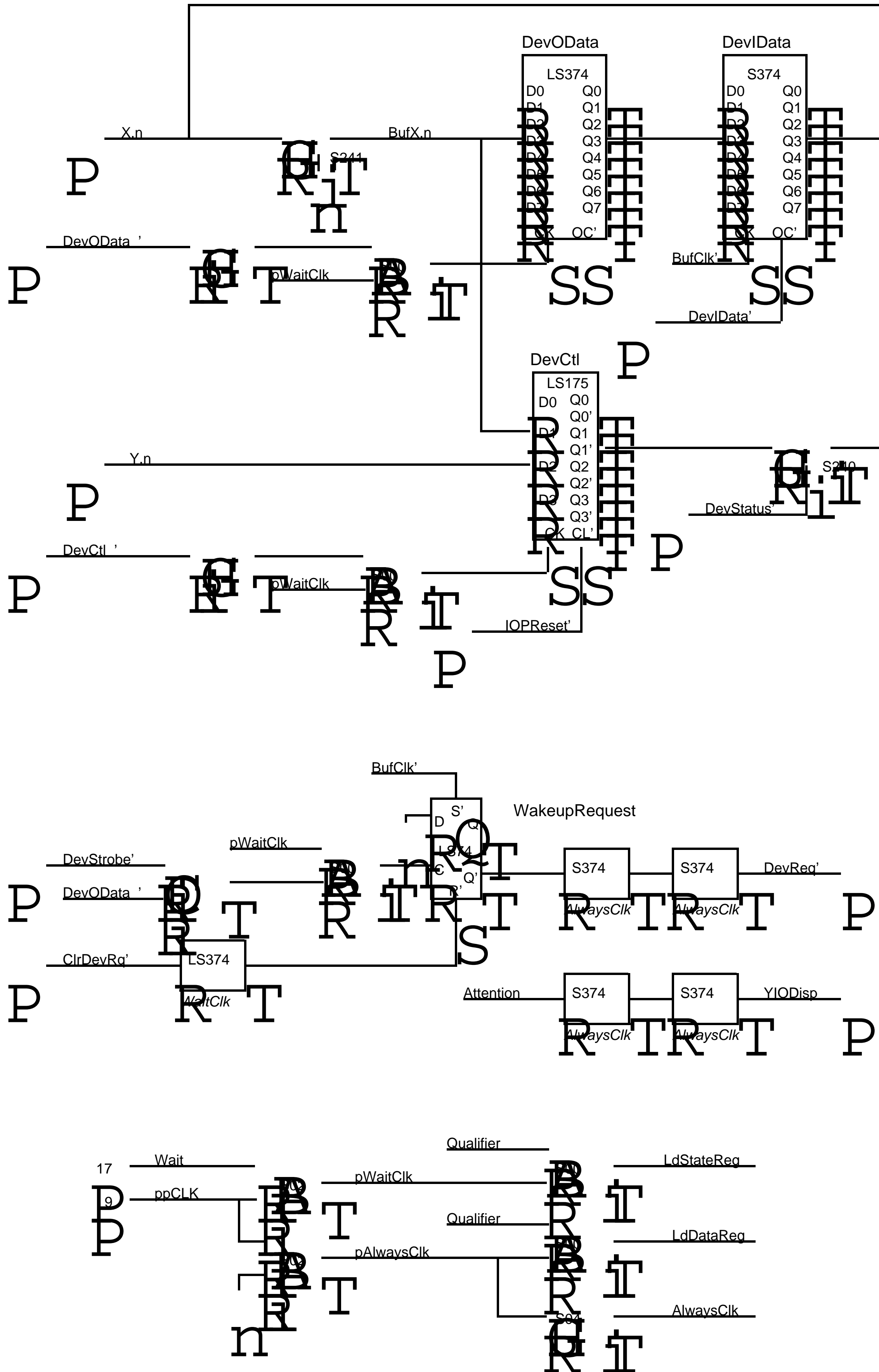


Figure 15. Controller Hardware Demonstrating I/O Rules & CP Interface

2.7 Example Microcode

Just as a melody, in order to be heard, requires both notes and intervals, the CP hardware should be viewed in light of its corresponding microcode. The following microcode examples illustrate how and in what time frame certain elementary functions are accomplished. There are seven examples, some simplified: Mesa Emulator Load Local n, Read n, Jump n, Refill, and the Ethernet, Disk, and LSEP inner loops. See the *DMR* for a description of the microcode format.

(1) The Mesa Emulator Load Local 1 (LL1) macroinstruction indexes the local frame pointer and then push's the addressed word from memory onto the Stack. It executes in one click if the indexing operation does not cross a page boundary and in three if a page cross occurs. If the Map flags must be updated (RMapFix), another two clicks are required.

```
@LL1:        MAR _ Q _ [rhL, L+1], L1_L1.PopDec, push,                            c1, opcode[1'b];
LLn:         STK _ TOS, PC _ PC+PC16, IBDisp, L2_L2.LL, BRANCH[LLa,LLb,1],        c2;
LLa:         TOS _ MD, push, fZpop, DISPNI[OpTable],                              c3;
LLb:         Rx _ UvL,                                                            c3;

LSMap:       Noop,                                                                c1;
             Q _ Q - Rx, L2Disp,                                                  c2;
             Q _ Q and 0FF, RET[LSRtn],                                           c3;

LLMap:       Map _ Q _ [rhMDS, Rx+Q],                                             c1, at[3,10,LSRtn];
             Noop,                                                                c2;
             Rx _ rhRx _ MD, XRefBr,                                              c3;

             MAR _ [rhRx, Q + 0], L0_L0.R, BRANCH[RMUD,$],                        c1;
             IBDisp, GOTO[LLa],                                                   c2;
RMUD:        CALL[RMapFix],                                                       c2;
```

(2) The Mesa Emulator Read 1 (R1) macroinstruction indexes the virtual address on the top of Stack and then push's the addressed word from memory onto the Stack. It executes in two clicks. Four are required if the page has been read the first time; that is, the Map flags must be updated.

```
@R1:         Map _ Q _ [rhMDS, TOS + 1], L1_L1.Dec, pop,                          c1, opcode[101'b];
             push, PC _ PC + PC16,                                                c2;
             Rx _ rhRx _ MD, XRefBr,                                              c3;

             MAR _ [rhRx, Q + 0], L0_L0.R, BRANCH[RMUD,$],                        c1;
             IBDisp, GOTO[LLa],                                                   c2;
```

(3) The Mesa Emulator Jump 2 (J2) macroinstruction increments the PC by 2 bytecodes and then refills the instruction buffer. It executes in two clicks. Five are required if the jump crosses a page boundary.

```
@J2:         MAR _ PC _ [rhPC, PC+1], push,                                       c1,opcode[201'b];
             STK _ TOS, L2 _ L2.Pop0IncrX, Xbus_0, XC2npcDisp, DISP2[jnPNoCross], c2;

jnPNoCross:  IB _ MD, pop, DISP4[JPtr1Pop0, 2],                                   c3, at[0,4,jnPNoCross];
jnP1Cross:   Q _ 0FF + 1, L0 _ L0.JRemap, CANCELBR[UpdatePC, 0F],                 c3, at[2,4,jnPNoCross];

JPtr1Pop0:   MAR _ [rhPC, PC + 1], IBPtr_1, push, GOTO[Jgo],                      c1, at[2,10,JPtr1Pop0];
JPtr0Pop0:   MAR _ [rhPC, PC + 1], IBPtr_0, push, GOTO[Jgo],                      c1, at[3,10,JPtr1Pop0];
Jgo:         TOS _ STK, AlwaysIBDisp, L0 _ L0.NERefill.Set, DISP2[NoRCross],      c2;
```

(4) The Mesa Emulator instruction buffer refill code executes in one click if the buffer was not empty and in two if it was. Four to six clicks are required if the refill occurs across a page boundary.

```
{Buffer Empty Refill.  Control goes from NoRCross to RefillNE since RefillE+1 does not contain an IBDisp.}
RefillE:     MAR _ [rhPC, PC], PC _ PC-1, L0 _ L0.ERefill,              c1, at[400];
             PC _ PC+1, DISP2[NoRCross],                                 c2;

{Buffer Not Empty Refill.}
OpTable:         {"Noop" location of Instruction Dispatch table}
RefillNE:    MAR _ [rhPC, PC + 1],                                       c1, at[500];
             AlwaysIBDisp, L0 _ L0.NERefill.Set, DISP2[NoRCross],        c2;

NoRCross:    IB _ MD, uPCCross _ 0, DISPNI[OpTable],                     c3, at[0,4,NoRCross];
RCross:      Q _ 0FF + 1, GOTO[UpdatePC],                                c3, at[2,4,NoRCross];
```

(5) The Ethernet input inner loop transfers one word per click until either a page boundary is crossed (ERead+2 or ERead+3), the maximum sized packet has been exceeded (EITooLong), or the controller has signaled an abnormal condition (ERead+1 or ERead+3).

```
{main input loop}
EInLoop:     MAR _ E _ [rhE, E + 1], EtherDisp, BRANCH[$,EITooLong],     c1;
             MDR _ EIData, DISP4[ERead, 0C],                             c2;

ERead:       EE _ EE - 1, ZeroBr, GOTO[EInLoop],                         c3, at[0C,10,ERead];
             E _ uESize, GOTO[EReadEnd],                                 c3, at[0D,10,ERead];
             E _ EIData, uETemp2 _ EE, GOTO[ERCross],                    c3, at[0E,10,ERead];
             E _ EIData, uETemp2 _ EE, L6_L6.ERCrossEnd, GOTO[ERCross],  c3, at[0F,10,ERead];
```

(6) The SAx000 disk write and verify inner loop transfers one word per click until the required number of words have been sent.

```
WrtVerLp:    MAR _ [RHRCnt, RCnt], RCnt _ RCnt+1,                        c1, at[0,2,FinWrtVer];
             RAdr _ RAdr-1, ZeroBr, CANCELBR[$, 2],                      c2;
             KOData _ MD, BRANCH[WrtVerLp, FinWrtVer],                   c3;
```

(7) The LSEP output inner loop outputs a band buffer entry from the display bank and then clears the entry. This continues until the required number of words have been transferred, which is detected by aligning the data on a page boundary.

```
scan:        MAR_ [displayBase1, rX+0], ClrDPRq,                         c1;
             MDR_ rY{= zero}, rX_ rX+1, PgCarryBr,                       c2;
             POData_ MD, BRANCH[scan, endLine],                          c3;
```

2.8 Footnotes

[1] All of the microcode-related specifications and rules presented in this chapter are validated by the microcode assembler and control-store-allocation program (MASS).

[2] The writeable control store is expensive: out of the 160 chips total, 70 are microstore chips.

A special version of the CP has been built which has a 16K control store partitioned into four, 4K banks. The 2-bit Bank register can be loaded from NIAX with fZ = Bank_. All non-Emulator tasks are forced to execute from bank 3. Error trap location 0 exists in each bank.

[3] Where did this (prime) number come from? All system timing is based on the Display's bit time, 19.59 nS (51.04 MHz, + .05%). There are 7 bit times in a cycle and 210 cycles (14 rounds) in one horizontal display line. More precisely, the cycle time is 137.14 + .57 nsec.

Alternatively, the cycle time (137) equals the inverse of the fine structure constant: a fundemental dimensionless constant equal to 2p times the square of the electron charge in electrostatic units, divided by the product of the speed of light and Planck's constant $(2pe^2/ch)$!

[4] This sequence has been likened to the triple time meter of a waltz!

[5] Because there are so many sources and sinks on the X bus, it has a nonnegligible capacitance: it has been measured at 337 pF!

[6] The *or*ing of a Link register with the low 4 bits of the IB byte during an IBDisp is not encouraged as this feature will not exist in a future version of the processor.

[7] The 18.3 Mbits/s into the display bank is approximated as follows: There are 70 clicks per display scan line and, of these, the Display controller uses 4*10 = 40 clicks for a normal scan line. Furthermore, the display microcode uses 2 clicks for memory refresh. During 808 of the total 897 scan lines, the display controller is actually pumping bits out to the monitor. Thus, the Display controller and microcode use about (808/897)(42/70)(38.4 Mbits/s) = 20.6 Mbits/s of the bandwidth, leaving 38.9-20.6 = 18.3 Mbits/s for the Emulator.

3.0 Memory System

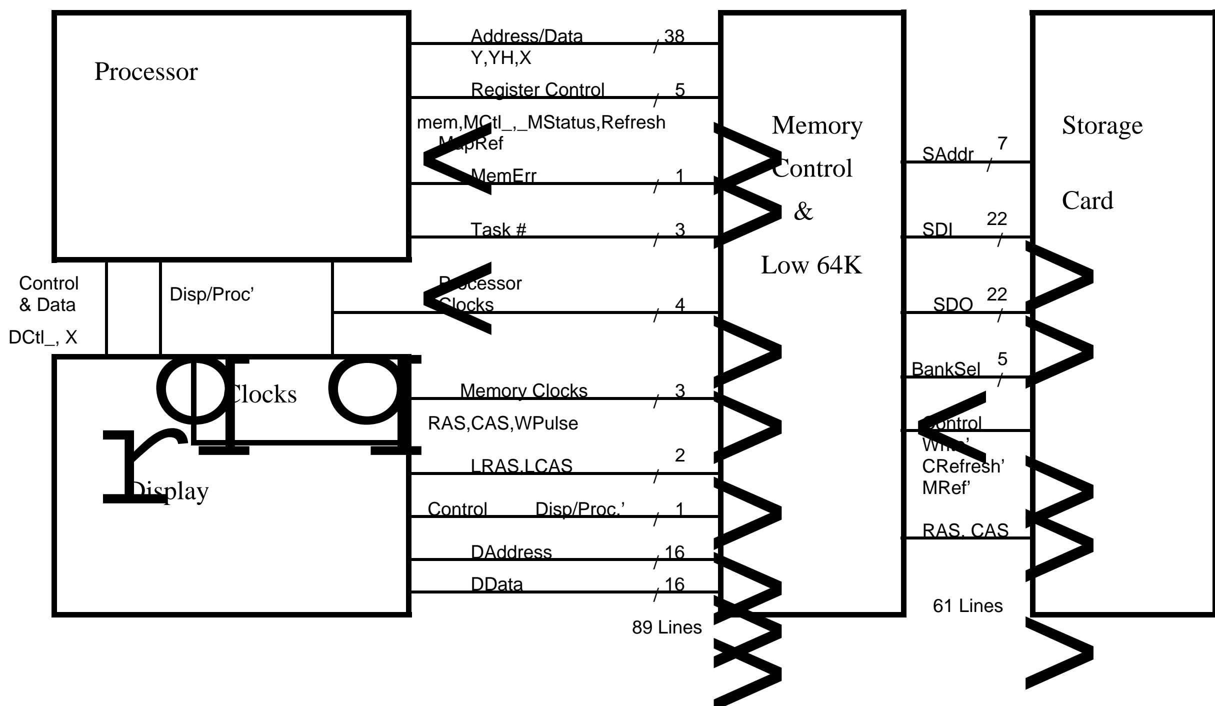
The memory system has two, 16-bit ports: one to the central processor (CP) and one to the display controller The CP shares the lowest 64K bank with the display and has exclusive use of the upper banks. Single-bit error correction and double-bit error detection is performed on all words delivered to the CP, but words used by the display are not corrected. The memory cycle time for the CP is 411 nanoseconds (nS), but for the display controller is either 293 (full) or 215 (page) nS.

The memory can be configured in at least five different sizes depending on the mix of Memory Control Cards (MCCs) and Memory Storage Cards (MSCs). The lowest 64K words (Display bank) are located on the memory control card along with the error correction and port logic. The storage card holds additional memory chips plus data and address drivers. The timing signals for the memory system are generated by display controller (sec. x.xx) and are synchronous to the processor clocks. Figure 17 is a block diagram of the memory controller.

The MCC comes in one of two sizes: 64K or 256K words. Likewise, the MSC has either 128K or 512K words (the large version is called MSC-X). With some modifications, the 256K MCC card (called MCC-X) can be used with the 128K storage card. The maximum real memory size is 1,048,576 words. The following configurations are standard:

MCC	MSC	Total size (words)
64K	none	65,536 (64K)
64K	128K	196,608 (192K)
256K	none	262,144 (256K)
256K	128K	393,216 (384K)
256K	512K	786,432 (768K)

From the micropogrammer's perspective, the CP controls all accesses to the memory: the CP's X, Y, and YH buses are used to supply addresses and transfer data. Device controllers can only use memory via their corresponding microcode tasks. (See section 2.4, "Main Memory Interface.") The Display controller is the excemption: it actually constructs its own memory timing signals (RAS and CAS) in order to acheive the maximum bandwidth possible through its port (sec. x.xx). The Display controller does not use the X and Y buses, but has its own 16-bit address and data buses. The following figure is a block diagram of the memory system:



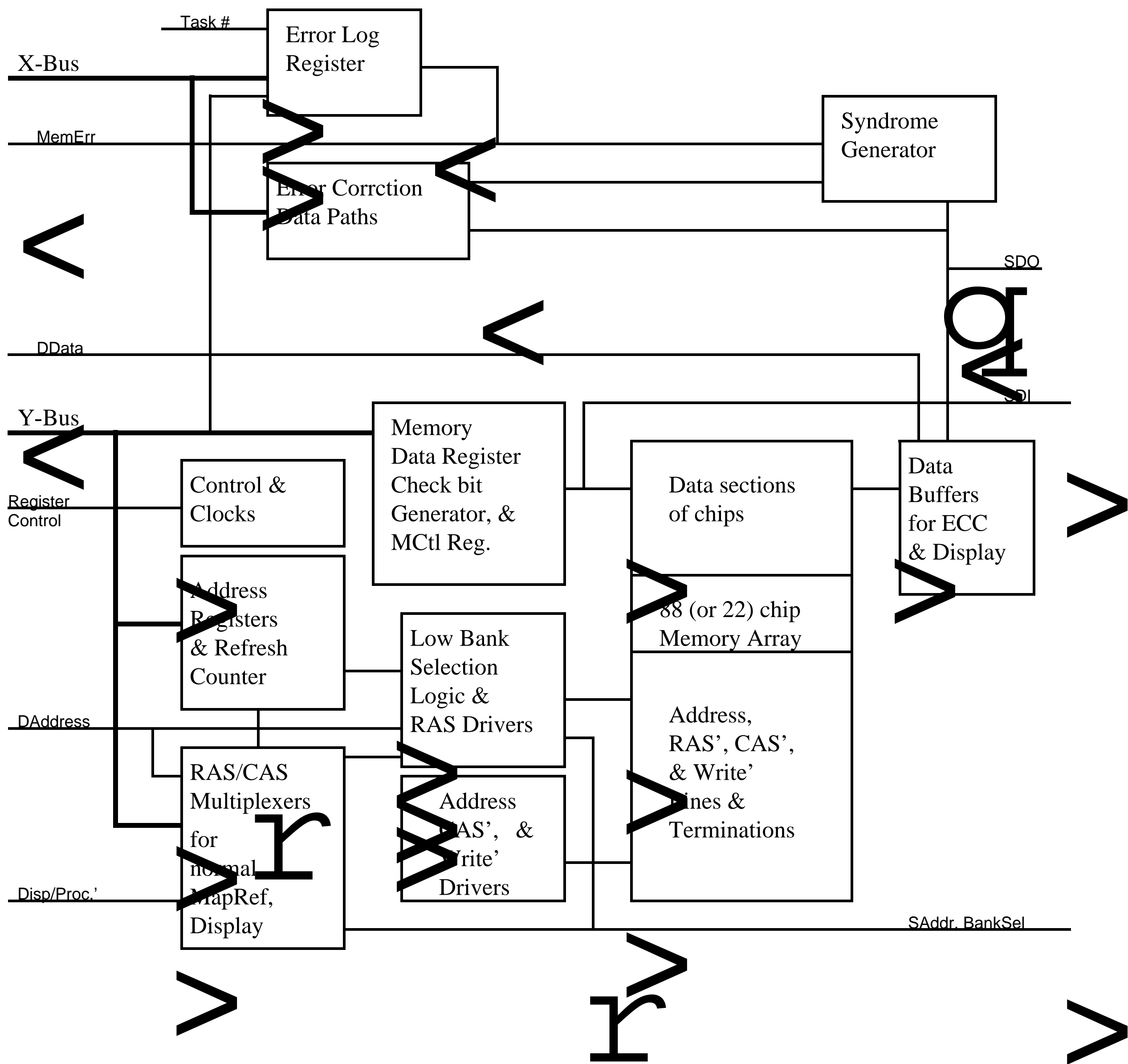


Figure 17. Memory Control and Low 64K Bank

3.1 CP Interface Summary

This section provides a summary of each of the functions of the memory system as viewed from the central processor. Figure 18 summarizes the functions. For a complete description of the microcode interface, see section 2.4.

Read

A read operation is started by placing the memory address on the Y and YH busses and asserting mem in the first cycle of a click. The data can be read back to the X bus during the third cycle by asserting mem then. All data read by the processor is error corrected unless the correction inhibit bit is set in the Memory Control (MCtl) register.

Write

The first cycle of a write operation is dedicated to sending the address to memory. It is identical to the first cycle of a read operation. The data to be stored must be delivered to the memory during the second cycle of a click, by asserting mem in the second cycle, and placing the data on the Y bus. Error correction check bits are always calculated and stored automatically by the memory system. If a write operation in the second cycle is followed by a read in the third, the data existing before the write is returned.

Map Reference

The Dandelion's virtual memory map is kept in main memory. A map-reference-type memory read is identical to a standard read, except the bits supplied by the Y and YH busses are shifted to facilitate indexing into the Map. Microcode uses this feature to provide a 22-bit virtual memory system with the MCC and a 24-bit system with the MCC-X.

The virtual memory is divided into 256 word pages. The Map_ function discards the low 8 virtual address bits (since they reference the word location on the page), moving the high 14 bits (virtual page number) to the low 14 or 16 bits used for the real map address. The location of the 16K map is fixed between locations 10000 and 13FFF (hex) in real memory.

Each 16 bit entry in the Map contains 10 to 12 bits of real page number and four flags describing the page (present, dirty, referenced, etc). To derive a real address from a virtual one, the microcoder uses the map function (Map_), checks the flags and appends the original low order 8 bits to the 10 or 12 bits fetched (sec. 1.4.2). The presence of a Map_ function in cycles 2 or 3 has no effect on the memory. mem should not be asserted, unless its side effects are desired (sec. 1.4.2).

Refresh

The memory controller contains circuitry to facilitate memory refresh. Each memory chip is organized as a 128x128 (or 256x128x2) bit matrix. When the row address is received, all bits in the specified row are read. The column address is used to select one of them. At the end of the memory cycle, all 128 bits are rewritten to perform a refresh. Hence, a row of a chip may be refreshed by reading any bit in that row. If the column address is suppressed during refresh, a substantial section of the chip remains quiescent, saving power. During each refresh cycle, the memory controller broadcasts only a 7 (or 8) bit row address and row address strobe (RAS) to every memory chip. This row address is supplied by a counter on the MCC that is incremented at the end of the cycle.

Refresh is initiated by asserting the Refresh function from the CP during cycle 1 of a click when the display is quiescent. The Refresh line is ignored during cycles 2 and 3 and whenever the display accesses memory. All memory chips require that 128 rows be refreshed at least every 2 milliseconds. A horizontal line on the display takes 28.8 microseconds, hence, the memory should

be refreshed at least 1.85 times per horizontal line. The standard display code performs two refresh cycles each line. The display microcode was chosen to do this because it can guarantee that the display hardware is inactive. Note that any displayless configuration of the Dandelion must contain some combination of hardware and microcode to perform the refresh task. The Refresh task is used in this case.

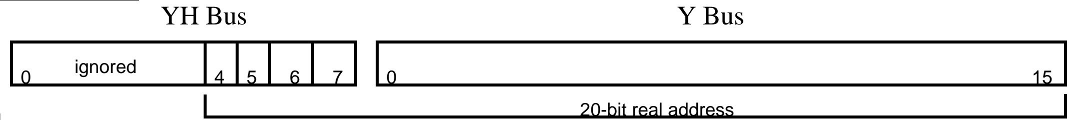
Display Lockout

The low 64K of the memory is shared between the display and the CP. The display has priority. When actually scanning a line, the display consumes clicks 0 through 3, leaving click 4 for the CP. Thus, one click out of 5 is available for use by display handling microcode and accessesby the Emulator to the low bank. As discussed in section 2.5.6.5, "Display Bank Interference," the lockout (plus refresh & display microcode functions) reduces by about half the Emulator’s maximum possible bankdwidth into the display bank: from 38.9 to 18.3 MBits/s.

Lockout occurs only if the processor and display attempt to access the low bank at the same time. Accesses to the high banks are not affected. Lockout does not occur during retrace intervals (horizontal and vertical), or during any other period of display inactivity (such as when the display is disabled). By convention, time critical hardware tasks using the first 4 clicks must never attempt access to the low (display) memory bank since a lockout could occur causing extra delay. In particular, one could not fill the bit map directly from an I/O device such as the disk or Ethernet without first disabling the display. See the display controller description for exact details of display timing.

Lockout is implemented by generation of a wait signal in the CP whenever a bank 0 (low 64K bank) access is attempted and the display is already using the low bank. The processor suspends the microcode which started in that click, and continues the normal arbitration of what runs in the next click. In this manner, lockout in one click does not hold up operation in the following click.

MAR_ Memory Address Register



MAR_ mem during c1

Action: Contents of YH[4-7],,Y[0-15] is used as memory address.

MDR_ Memory Data Register

MDR_ mem during c2

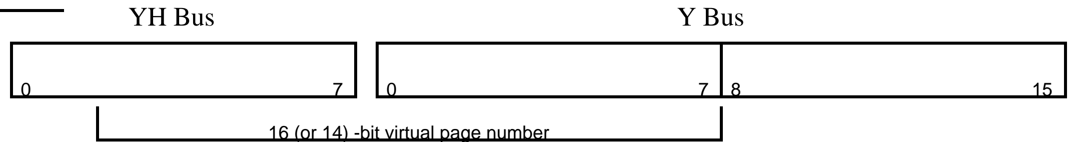
Action: Contents of Y Bus go into memory location specified by contents of MAR as loaded during first cycle of click.
No write occurs if the low 64K bank is selected and it is already being used by the display.

_ MD Memory Data

_MD mem during c3

Action: Memory data to X-Bus is single error corrected if MCtl bit 15 is set. The status of a given read operation can be found by looking in MStatus before the next memory read (_MD) is done. The occurance of both single and double errors are indicated here. This operation gives the contents of the memory cell specified during cycle 1, independent of whether a write was specified during cycle 2.

Map_ Map Reference



Map_ in c1 only

Acton: This action is the same as a MAR_ except that the physical address is derived differently.
An access is started in the 65K - 80K bank of memory. The location accessed is specified by the page number.

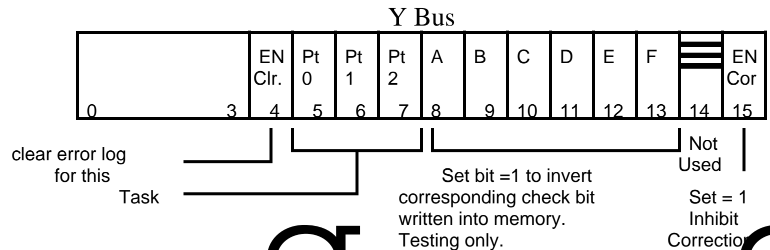
Refresh

Refresh during cycle 1 of click. Ignored in c2 and c3.

Action: A RAS only cycle is initiated in all memory chips. Row Address is supplied from an internal 7 (or 8) bit counter which is incremented once per occurance of refresh.
DO NOT USE refresh if the display is using the low bank of memory during that cycle. No refresh will occur.
This can be guaranteed if used only in click 4.

MCtl_ Memory Control Register

MCtl_ during any cycle.



Action: Normally this register is set to 0. A-F can be set to one to test syndrome bits and error indications.
Individual bits of the error log can be cleared by setting bit 4 and using Pt0-2 to specify the bit to be cleared.
Bit 15, Inhibit correction, affects only the data being read. Check bits are always generated and stored in memory during writes.

_ MStatus Memory Status

_ MStatus during any cycle.

Action:
This register is loaded every time memory data is read by the processor (_MD). High byte has status of most recent memory access. Low byte latches any occurance of double error on a per task basis. Register is 0 if no errors logged.

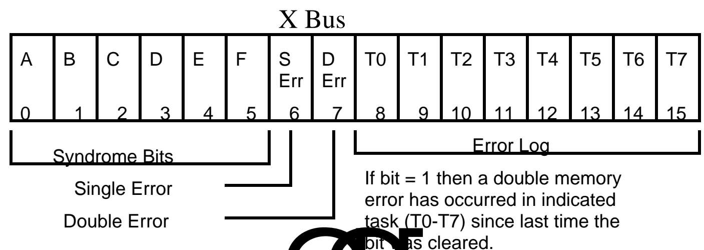


Figure 18. CP Memory Interface Summary

3.2 Error Correction

Since soft errors can occur in the memory (alpha particles from the package, etc.) error correction circuitry is included in the memory system. Six check bits added to the 16 bit word provide single error correction and double error detection (SEC-DED). No explicit indication of single errors is provided, although the status of any particular operation can be read from the Status & Errors (MStatus_) register after an operation. Error correction can be disabled, and the check bit positions in memory selectively set by writing into the MCtl register and reading the MStatus register.

A double error signal is available and also latched on a per task basis in the MStatus register. Thus, a task, upon entering a critical data transfer phase, could clear its particular bit, perform the task, and then check to see if its bit was set (double error). If an error did occur, its effect would be limited to events in that interval, over which some corrective action might be taken. If the emulator task caused the double bit error, a kernel trap is taken to location 0. See section 2.5.5.2, "Error Traps."

The following calculations yield probabilities of errors due to independent random processes in each chip. They do not include correlated events such as power line transients or static discharges which could affect all of the chips at the same time. A memory with 22 bits/word is assumed.

If the chips are assumed to (hard) fail at a rate of one per 2.5 million years (.04%/1000hr), then the mean time to a chip failure in a memory system with 12 banks (192K or 768K) is 9470 hours (13 months). By contrast, the mean time to failure with 4 banks (256K) is 28,410 hours (3 years, 3 months).

The soft error rate for the chips is assumed to be 1%/1000 hours. Following are the probabilities of 0, 1, and 2 soft errors in a 22 bit word in a 10 hour period. 10 hours was selected as the interval over which errors could accumulate, with the system being reset after 10 hours. The mean time between single errors is 38 intervals and the mean time between double errors is approximately 36,200 intervals. (It should be pointed that these probabilities are those that one would expect to measure with a program which continually scans through all memory cells looking for an error. If a program is confined to a small segment of memory, it would perceive a proportionately smaller probability of soft error.)

Prob.(1 single error in 22 bit word in 12 bank system in 10 hr. interval)	=.0263
Prob.(1 double error in 22 bit word in 12 bank system in 10 hr. interval)	$=2.76 \times 10^{-5}$

The following table shows the interpretation of the syndrome bits which can be read with the _MStatus function after a memory read. The code table shows how the syndrome bits A-F are generated. When checking, syndrome bit F is parity over the entire word.

or (A-F)	F	Meaning
0	0	no errors or >2 errors
0	1	not possible
1	0	double bit error
1	1	single bit error

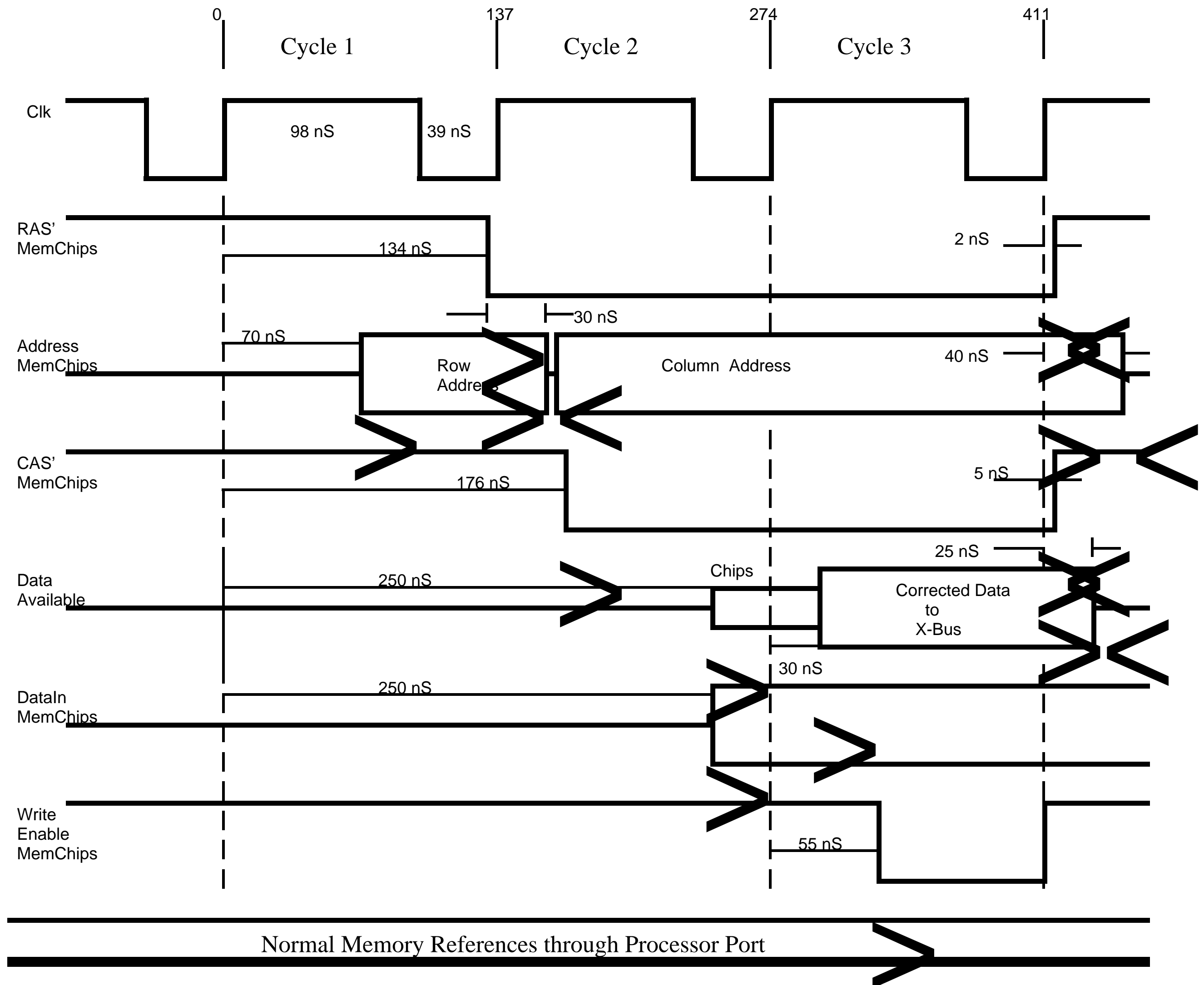
The SEC-DED code was optimized for 9-input parity chips. The following code table shows how the syndrome bits A-F are generated. Each row represents the inputs to a single parity chip. For example, syndrome bit A is the *xor* of data bits 0-3 and 10-13. Bit 0 will be inverted (corrected) during reading when A-F equals 110001 (from the column under 0). Any of the syndrome bits can be inverted when being generated by setting the corresponding bit in the MCtl register.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	a	b	c	d	e	f
A	+	+	+	+							+	+	+	+			+					
B	+				+	+		+	+	+				+		+		+				
C		+			+		+		+	+		+	+		+				+			
D			+			+	+	+		+	+		+		+					+		
E				+			+	+	+		+	+		+		+					+	
F	+	+	+	+	+	+									+	+						+

3.3 Memory Timing

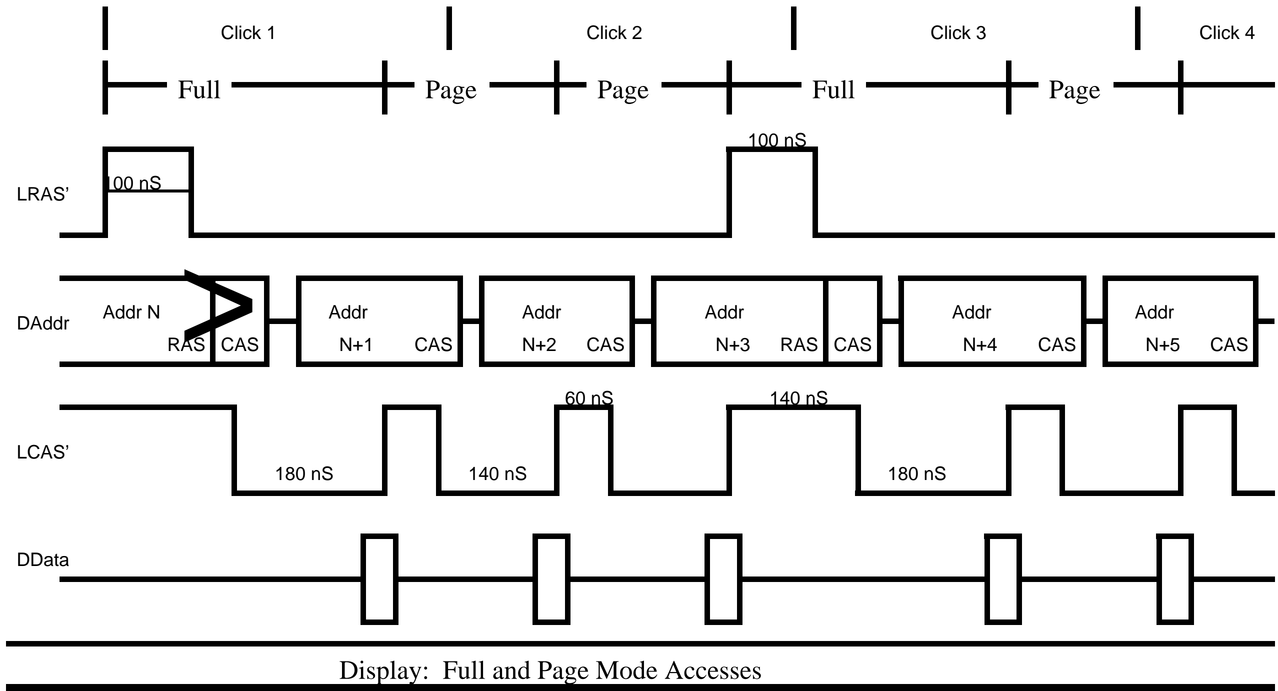
Typical processor timing is shown in figure 19 below. The memory address must be valid on the Y and YH busses early enough that the proper bank is selected and address lines valid for RAS’ (row address strobe). The column address bits are latched by the RAS’ signal. The CAS’ (column address strobe) signal occurs 42 nS after the RAS’ signal and latches the column address in the memory chips. Data becomes valid at the output of the chips at a maximum of 150 nS after RAS’ or 100 nS after CAS’, whichever is later.

When writing into memory, the data to be written must be supplied during the second cycle of a click. The data is actually written in the latter half of the third click. Notice that up until the presence of the write pulse, all signalling is identical to a read cycle. The memory chips latch and hold the old data on their outputs during a write pulse if it occurs more than 150 nS after the RAS’ signal. Thus, it is possible to write into a location and read data from it, all in the same memory cycle.



The display port supports both full and page mode accesses. The data delivered to the display port is not error corrected. The full access cycle time is 280 nS and the page mode access time is 200 nS. While the full access time is smaller than that specified in the data sheets (320 nS) for continuous operation, it is the average that is important, and the average cycle time in this case is 342 nS (6 full accesses per round, counting click 5). A page mode access occurs when the RAS' signal goes low and the CAS' signal cycles several times, strobing several different column addresses into the memory chips while retaining the same row address. (Because bit 12 is used during RAS, the maximum number of sequential page mode accesses between full accesses is 7, since bit 12 will change on every 8th access. The insertion of full accesses at the appropriate times is handled by the display controller.)

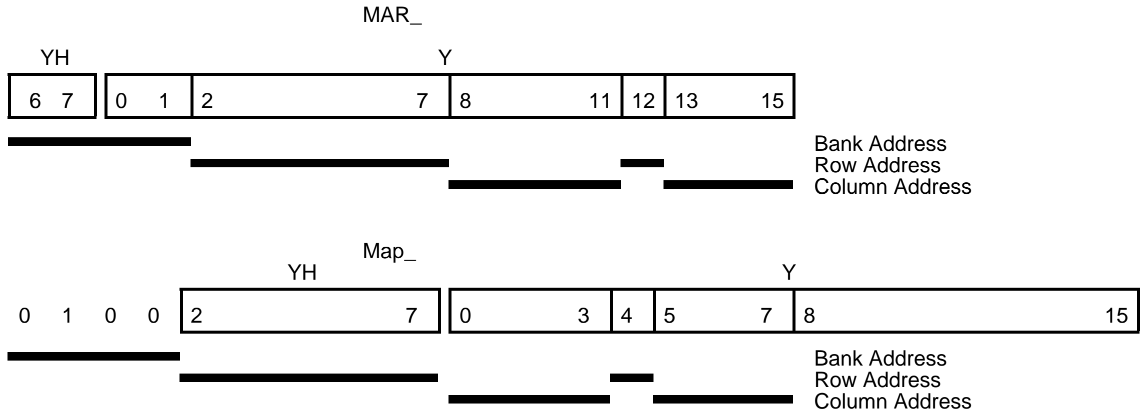
In normal operation, the display controller will seize the low bank of memory for 4 clicks of every round. It will start with a full access which is aligned on a click boundary, and then proceed with page and full accesses until the end of click 4. The other page or full accesses will not necessarily be synchronized with any click or cycle boundaries. They are packed so as to maximize the number of accesses during the 4 clicks the display has the memory.



Display: Full and Page Mode Accesses

3.4 Row and Column Addressing

In the case of 16K chips (to which the Dandelion was originally desinged), one of the seven bits for the row address must come from the low byte. The maximum settling time of the high nibble of the low byte is too long if a carry from the low nibble occurs (sec. 2.3.8). Consequently, bit 12 (instead of bit 8) of the low byte is used during RAS. Consistent juggling occurs for map references so that this is invisible to the microcoder. The following figure shows how the row and column address bits map into the Y and YH buses for 16K chips:



When 64K chips are used, the row and column bits are "correct." The following table shows how they are derived from the Y and YH bus. (If it is known that only 64K chips are present in the system, the restriction that X bus arithmetic can not occur with Map_ is no longer valid.)