Dandelion Microcode Reference

14 Feb 80
R. Garner

This document describes Rev. D of the CP.
(previous revisions: 31 Oct 79, 8 Jan 80, 22 Jan 80)

XEROX

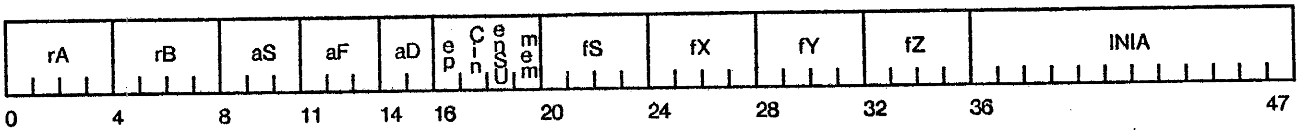
OPD SYSTEMS DEVELOPMENT
3408 Hillview Ave. / Palo Alto / California 94304

Contents

Where the files are saved:

[Iris]<Workstation>Mass>RevC>Mass.image ,.bcd -- or
[Iris]<Workstation>Mass>RevD>Mass.image ,.bcd
[Iris]<Workstation>Jarvis>Burdock.image
[Iris]<Workstation>mc>KernelDlion.fb

[Iris]<Workstation>LH>DMR.press
[Iris]<Workstation>LH>dmr1.bravo ,dmr2.bravo
[Iris]<Workstation>LH>DMRSil.dm



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
1	S − R
2	R − S
3	R or S
4	R and S
5	~R and S
6	R xor S
7	~R xor S

sh,,aD	B[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

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	MapRef
E	pop
F	push

fY	fYNorm	DispBr	IOOut
0	ExitKern	NegBr	IOPOData←
1	EnterKern	ZeroBr	IOPCtl←
2	ClrIntErr	YOddBr	KOData←
3	IBDisp	MesaIntBr	KCtl←
4	MesaIntRq	PgCarryBr	XOData←
5	stackP←	CarryBr	XCtl←
6	IB←	XRefBr	DCtlFifo←
7	cycle	NibCarryBr	DCtl←
8	Noop	XDisp	DBorder←
9		YDisp	PCtl←
A	Refresh	XC2npcDisp	MCtl←
B	push	YIODisp	
C	ClrDPRq	IODisp	
D	ClrIOPRq	XHDisp	
E	ClrXRq	XLDisp	
F	ClrKFlags	PgCrOvDisp	POData←

fZ	fZNorm	IOXIn
0	Refresh	←XIData
1	IBPtr←1	←XStatus
2	IBPtr←0	←KIData
3	Cin←pc16	←KStatus
4	MapRef	←PStatus
5	pop	←MStatus
6	push	←KTest
7	AltUaddr	
8	Noop	←IOPIData
9	Noop	←IOPStatus
A	Noop	←ErrIBStkp
B	Noop	←RH
C	LRot0	←ibNA
D	LRot12	←ib
E	LRot8	←ibLow
F	LRot4	←ibHigh

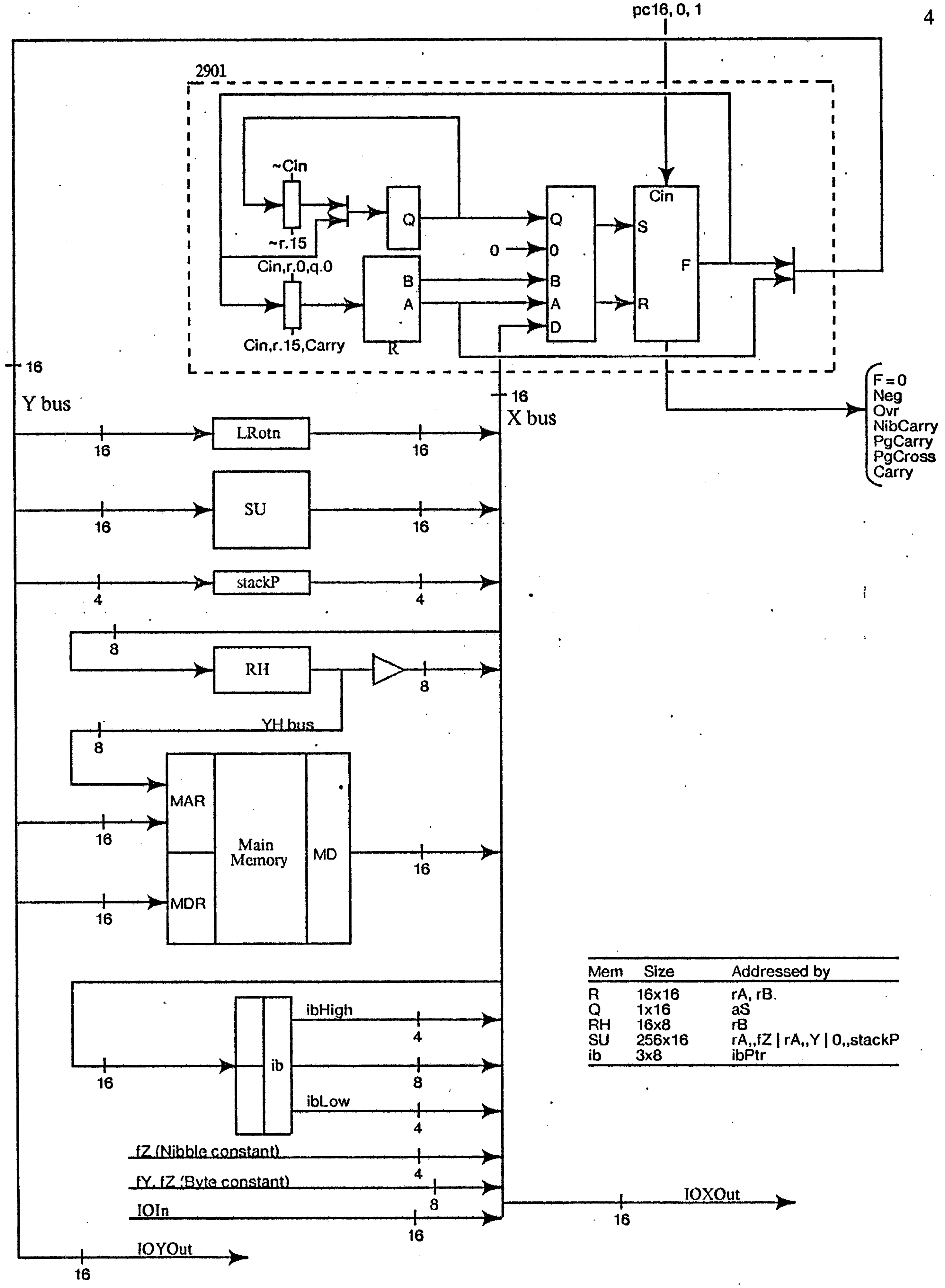
Notes

1. pCall when NIA[7]=0, pRet when NIA[7]=1.
2. When writing SU (Cin=1, enSU=1) and doing Cin←pc16, fXCin←pc16 must be used.
3. MAR← causes aS←"0,B" & aF←"RorS" in bits[0-7] of ALU; tests for PageCrossBr; cancels MDR← or IBDisp if PageCross true.
4. XBus[0-7] is zeroed for fZ=[7..F] and fS=Nibble/Byte.

* Refers to fZ and Y[12-15] of previous microinstruction.

Figure 1. Dandelion Microinstruction Format

7 Feb 80
Filed: <Workstation>LH>MicroInstFormat-D.sil
Workstation>LH>DMR.press

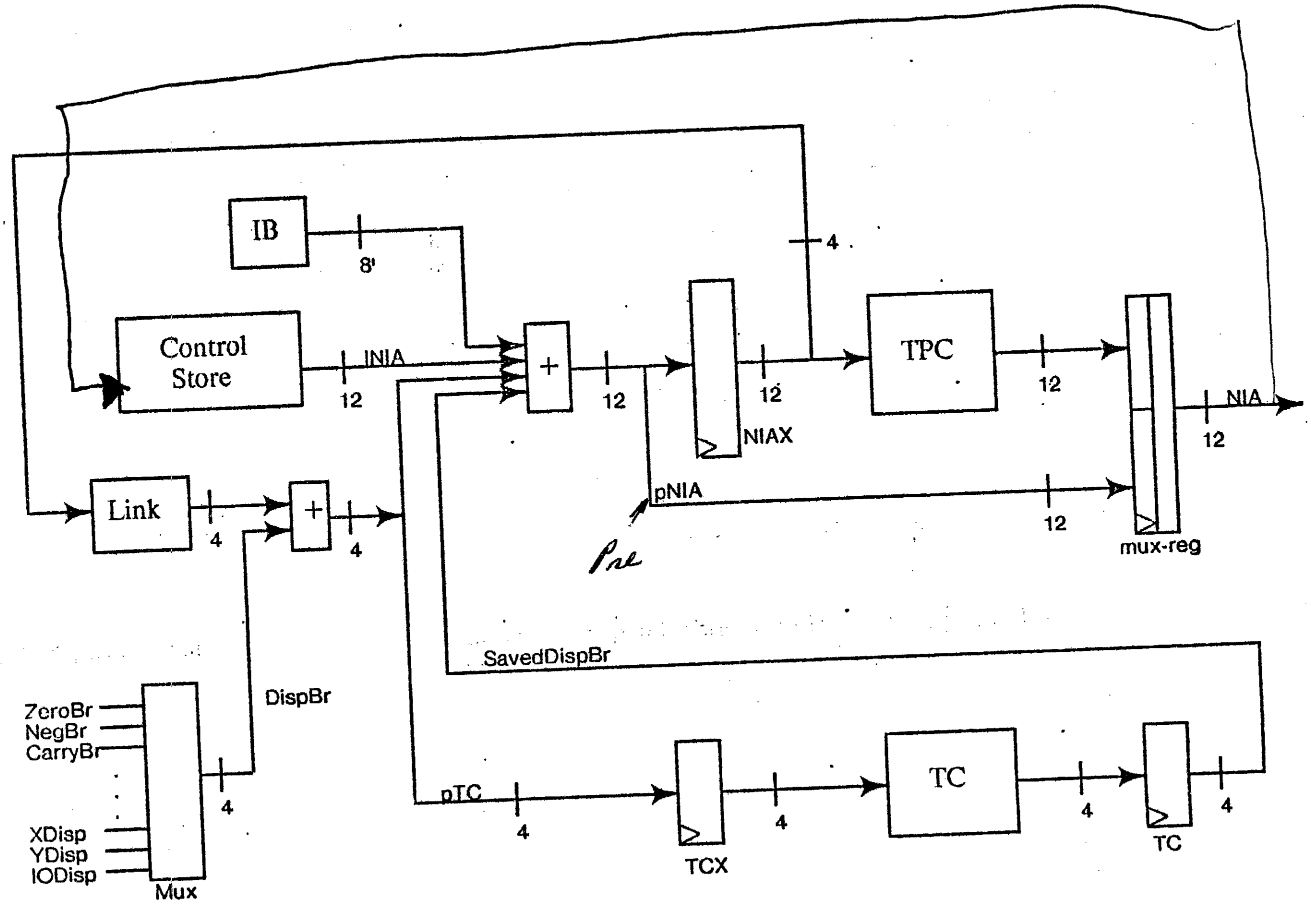


Mem	Size	Addressed by
R	16x16	rA, rB.
Q	1x16	aS
RH	16x8	rB
SU	256x16	rA,,fZ | rA,,Y | 0,,stackP
ib	3x8	ibPtr

Figure 2. Dandelion Processor Data Paths

4 Jan 80
Filed: <Workstation>LH>CPDataPaths.press
<Workstation>LH>DMR.press

NA = next immediate address of control store



Mem	Size	Address	Read	Written
Link	8x4	fX	any c	any c
CS	4Kx48	NIA	all c	all c
TPC	8x12	Nt	c2	c3
TC	8x4	Nt	c2	c1

> denoates a register which is loaded at end of each cycle

+ denotes logical OR
(IB[4-7] is ored with INIA[8-11] and IB[0-3] replaces INIA[4-7])

Figure 3. Dandelion Processor Control Paths

24 Oct 79
<Workstation>LH>CPControlPaths.sil
<Workstation>LH>DMR.press

A. Branches and Dispatches

	source	INIA dest	
NegBr	F.0	11	sign of alu result (*Not* Y.0)
ZeroBr	F=0	11	alu output equal to zero
YOddBr	Y.15	11	least significant Y bus bit
MesaIntBr	MesaInt	11	Mesa Interrupt bit
NibCarryBr	Cout.12	11	alu carry out of low Nibble
PgCarryBr	Cout.8	11	alu carry out of low Byte
CarryBr	Cout.0	11	alu carry out
XRefBr	X.11	11	present & referenced map bit
IODisp	bp.38,,bp.138,	10,,11	IO branches (bp = Backplane pin)
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	Page Cross & Overflow bits
XDisp	X[12-15]	8,,9,,10,,11	X bus
YDisp	Y[12-15]	8,,9,,10,,11	Y bus
XC2npcDisp	X[12-13],,C2,,~pc16	8,,9,,10,,11	X bus, cycle2, ~pc16
YIODisp	Y[12-13],,bp.39,,bp.139	8,,9,,10,,11	IO branches (bp = Backplane pin)
IBDisp	ib	[4-11]	Instruction Buffer
L0Disp, pRet0	Link0	8,,9,,10,,11	Link0 dispatch (NIA.7 = 1)
:	:	:	:
L7Disp, pRet7	Link7	8,,9,,10,,11	Link7 dispatch (NIA.7 = 1)

1. PageCross branch (called PgCrossBr) is defined to be "PageCarry xor aF.2." This has the effect of toggling PageCarry when doing subtraction (aF=S−R). PageCross equals PageCarry when doing addition (aF=R+S). Thus, assuming one uses positive displacements, such as R+1 or R−1, PgCrossBr will consistently indicate when a page boundary has been crossed.

The aF=R−S form of subtraction, unlike aF=S−R, does not cause PageCarry to be toggled on subtraction (since aF.2=0). However, the aF=S−R form covers most of the common subtraction cases: B−1, A−1, B−A, A−constant, and Q−constant. It does not include D−1. In "A+B", if either A or B is negative, PgCross branch will always be true. Moral: Always add or subtract positive displacements and PgCross branch will be true to you.

2. Even in the absence of ALU arithmetic, the NibCarryBr, PageCarryBr, and PgCrOvDisp branches can produce non-zero results (i.e., branch). When aF="RandS", NibCarry and PageCarry are the logical inner product of R with S. If aF=notRandS and aS="0,B", "0,A" or "0,Q", then NibCarryBr tests for the low nibble equaling zero and PgCarryBr tests for the low byte equaling zero. If aF="RorS", NibCarry is the logical inner sum of R with S. If aS="0,B", "0,A", "0,Q" or "D,0", then NibCarryBr tests for the low nibble equaling 0F. For the final coup de grace, if aF="RxorS" and aS="0,Q", "0,A", or "0,B", then NibCarryBr is true if the low nibble is 8, 0C, 0E, or 0F.

Notes:

1. Branches take two microinstructions to specify. In the first microinstruction the branch or dispatch condition (abbreviated DispBr) is declared by an fY. The second instruction should contain a "BRANCH[Label0, Label1]" phrase.

```
          Reg ← Reg xor RegA, ZeroBr,                          c1;
          BRANCH[NotZero, Zero],                               c2;

NotZero:  Noop    {here if result nonzero},                    c3;
Zero:     Noop    {here if result zero},                       c3;
```

2. The "at[x, y, Label]" macro is used to constrain the location of instructions. It tells MASS to place the instruction at a control store location which is "x MOD y" and in the same "MOD group" as the instruction labeled "Label." Thus, the above example could be rewritten as (The "at"s are NOT required.):

```
          Reg ← Reg xor RegA, ZeroBr,                          c1;
          BRANCH[NotZero, Zero],                               c2;

NotZero:  Noop,   {here if result nonzero}                     c3, at[0,2,Zero];
Zero:     Noop,   {here if result zero}                        c3, at[1,2,NotZero];
```

3. A dispatch specifies more than a single bit which is OR'd into INIA. Instead of a "BRANCH" macro, dispatches are specified by "DISP2[Label]", "DISP3[Label]" or "DISP4[Label]" (abbreviated DISPn), where n specifies the number of bits used for the dispatch. "at" clauses ARE required.

```
          [] ← MD, XLDisp,                                        c3;
          DISP2[Table],                                           c1;

Table:    Noop,  {here if MD.8,,MD.15 = 0}                        c2, at[0,4,Table];
          Noop,  {here if MD.8,,MD.15 = 1}                        c2, at[1,4,Table];
          Noop,  {here if MD.8,,MD.15 = 2}                        c2, at[2,4,Table];
          Noop,  {here if MD.8,,MD.15 = 3}                        c2, at[3,4,Table];
```

4. A two-way branch on a dispatch field is notationally accomplished by specifying a mask which has 1's in those bit positions of the dispatch which should be ignored in the branch. The mask should be the same width as the one implied by the dispatch. The mask is a third argument to the "BRANCH" macro. The only legitimate values for the third argument have exactly one zero in their binary (and a leading zero is used if needed); they are [1,2,3,5,6,7,0B,0D,0E]. "0F" is illegal since it has no zero in its binary. "at" clauses are NOT required.

```
          [] ← Reg LRot8, XDisp,                                  c1;
          BRANCH[NotSet, Set, 0B], {branch on bit 13 of X bus}    c2;

NotSet:   Noop,  {here if bit 5 of Reg = 0},                      c3, at[0B,10,Set];
Set:      Noop,  {here if bit 5 of Reg = 1},                      c3, at[0F,10,NotSet];
```

5. A dispatch on a sub-field of a dispatch is again specified with a mask which says which bits of the larger dispatch should be ignored. The mask is a second argument to the "DISPn" macro. "at" clauses are NOT required.

```
          [] ← RHReg, XDisp,                                      c3;
          DISP4[Table, 9],                                        c1;

Table:    Noop,  {here if RHReg.13,,RHReg.14 = 0},                c2, at[9,10,Table];
          Noop,  {here if RHReg.13,,RHReg.14 = 1},                c2, at[0B,10,Table];
          Noop,  {here if RHReg.13,,RHReg.14 = 2},                c2, at[0D,10,Table];
          Noop,  {here if RHReg.13,,RHReg.14 = 3},                c2, at[0F,10,Table];
```

6. The "CANCELBR" macro is used to cancel pending branch/dispatch conditions by forcing the argument address to have ones where pending bits would normally be OR'd in. CANCELBR may be necessary after a path of two instructions which specify branching or after a MAR← (see Memory section). MASS will give a warning message where it thinks there should be a CANCELBR. (It uses the principle that all DispBr's or pRet's should be followed by either a BRANCH, DISPn, RET, or CANCELBR)

```
          ZeroBr,                                                 c3;
          NegBr, BRANCH[NZ, Z],                                   c1;

NZ:       BRANCH[Pos, Neg],                                       c2;
Z:        CANCELBR[Zero],                                         c2;

Zero:     Noop,    {placed "at[1,2]" by MASS}                     c3;
```

7. Pending bits of a dispatch or "pRet" are canceled by a mask which says which bits should be ignored. The mask is the second argument to the CANCELBR macro. Thus mask=0F causes the argument address to be placed "at[0F,10]."

```
          ZeroBr,                                                 c3;
          pRet0, BRANCH[NZ, Z],                                   c1;

NZ:       RET[NZReturn],                                          c2;
Z:        CANCELBR[NotYet, 0F],                                   c2;

Zero:     Noop,    {placed "at[0F,10]" by MASS}                   c3;
```

A general rule for the branch masks described above: The "mask" always indicates bits which should be ignored.

8. The "GOTOABS" macro sends control to an absolute control store location.

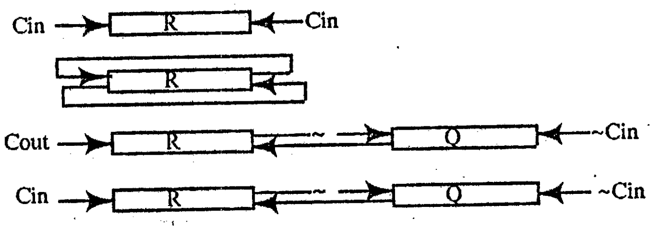
```
        GOTOABS[0],                                          c3;
```

B. Shifting

Single-bit shifts and rotates occur at the output of the ALU and the results can only go to an R register (or Q on double length shifts). Four-bit rotates occur between the Y bus (ALU F output or A bypass) and the X bus. If the result of the 4-bit rotate is destined for an R register, it must have been placed onto the Y bus via the A bypass (which implies that aD=2). Single-bit shifting uses the fX field, single-bit rotating fX or fY, and 4-bit rotates fZ.

LShift1, RShift1	Left, Right Shift R by 1 (fX = shift)
LRot1, RRot1	Left, Right Rotate R by 1 (fX = cycle or fY = cycle)
DALShift1, DARShift1	Double Arithmetic Left, Right Shift R,,Q by 1 (fX = shift)
DLShift1, DRShift1	Double Left, Right Shift R,,Q by 1 (fX = cycle or fY = cycle)

aD.1	fX
1	shift
1	cycle
0	shift
0	cycle



aD.0 = 0 implies right shift

Notes:

1. The notation "SE←0", "SE←1", or "SE←pc16" is used to specify the shift ends. "SE←" is equivalent to "Cin←".

```
Reg ← LShift1 Reg, SE←1, {puts 1 into Reg.15}          c1;
Reg ← DALShift1 Reg, SE←1, {puts 1 into Q.15}          c1;
```

2. A "DARShift1" shifts Cout into the left side of the double length R,,Q. (This is used in the multiply instruction.) The 2901 can cause carries on logical operations (believe it or not). Therefore, if you want to shift a 0 into the left side you must specify an arithmetic operation which produces no carry.

```
Reg ← DARShift1 (Reg + 0),                              c1;
```

3. The single bit shifting operations use Cin for the shift ends. Therefore, if SU is being read and shifted, or SU is being read and there is a shift operation in the ALU, the shift ends must be zero. Note that SU can not be written simultaneously with any type of shifting operation (A bypass and shift are not a legal aD combination). If shifting is combined with arithmetic, the shift ends must be 0 unless a +1 operation is desired. Note that "Reg−Reg" implies a SE of 1.

```
Reg ← RShift1 (RegA + Reg), SE←1,                       c1;
Reg ← RShift1 (Reg + 1), SE←1,                          c1;
Reg ← RShift1 (RegA − Reg − 1), SE←0,                   c1;
Reg ← Rshift1 Ureg, SE←0,                               c1;
```

4. LRotn, when used in conjunction with A bypass, allows the ALU to be used for other purposes. For instance, an R register can be rotated and placed onto the Xbus (where it can be branched on or sent to RH or IOOut) while arithmetic is performed in the ALU. Note that the R register given by rB must always be written when A bypass is used.

```
            IOOut ← RegA LRot8, Reg ← Reg + 1,                      c1;
            rhReg ← Reg LRot12, Reg ← ~Reg, XDisp,                  c1;
            STK ← RegA, rhReg ← RegA LRot0, Reg ← Reg + 1,          c1;
```

5. An arbitrary 16 bit rotate takes 3 cycles (plus 1 to specify it). This example uses 2 R registers and assumes the shift count is in RH.

```
            [] ←rhReg, XDisp,                                       c1;
            T ← LRot1 R,  DISP4[Rot],                               c2;

Rot:        GOTO[Shift0],                                           c3, at[0,10,Rot];
            R ← T, GOTO[Shift0],                                    c3, at[1,10,Rot];
            R ← LRot1 T, GOTO[Shift0],                              c3, at[2,10,Rot];
            R ← RRot1 R, GOTO[Shift4],                              c3, at[3,10,Rot];

            GOTO[Shift4],                                           c3, at[4,10,Rot];
            R ← LRot1 R, GOTO[Shift4],                              c3, at[5,10,Rot];
            R ← LRot1 T, GOTO[Shift4],                              c3, at[6,10,Rot];
            R ← RRot1 R, GOTO[Shift8],                              c3, at[7,10,Rot];

            GOTO[Shift8],                                           c3, at[8,10,Rot];
            R ← LRot1 R, GOTO[Shift8],                              c3, at[9,10,Rot];
            R ← LRot1 T, GOTO[Shift8],                              c3, at[A,10,Rot];
            R ← RRot1 R, GOTO[Shift12],                             c3, at[B,10,Rot];

            GOTO[Shift12],                                          c3, at[C,10,Rot];
            R ← LRot1 R, GOTO[Shift12],                             c3, at[D,10,Rot];
            R ← LRot1 T, GOTO[Shift12],                             c3, at[E,10,Rot];
            R ← RRot1 R, GOTO[Shift0],                              c3, at[F,10,Rot];

Shift0:     GOTO[Done],                                             c1;
Shift4:     R ← R LRot4, GOTO[Done],                                c1;
Shift8:     R ← R LRot8, GOTO[Done],                                c1;
Shift12:    R ← R LRot12, GOTO[Done],                               c1;
```

C. Link Registers & Subroutines

Link registers, besides being used for subroutines, can be used to store 4-bits of state information which can be branched on later. Constants or branch condition bits can be stored in Link registers. Later, current branch conditions can be simultaneously OR'd with the saved state bits. "pRetn" acts like a dispatch/branch (DispBr) and "pCalln" is used to load a link register.When either a "pRetn" or "pCalln" is specified, the following instruction must be constrained in some way.

Address bits of the following instruction are indicated by "IA.n", where n varies from 0 to 11 ("IA" stands for "instruction address"). "NIA" ("next instruction address") is the 12-bit quantity which addresses the control store -- while instruction "n" is executing, "n+1" is being accessed from the control store. "INIA" refers to the contents of the 12-bit microinstruction field. In the CP, "inia" is OR'd with the currently specified dispatch/branch bits to form "NIA."

1. Link registers are loaded from the low 4 bits of NIA--the control store address which is currently being used to fetch the next microinstruction. Notationally, the instruction after an "L1←" must be constrained such that its low 4 bits equal the constant to be loaded into the Link register. In addition, IA.7 of the next instruction must be 0 (MASS does this allocation). "Ln←" is equivalent to "pCalln". The "at" is NOT required.

```
Set[FlagBB, 6];

L1 ← FlagBB,  {loads a 6 into Link1}                         c1;
Noop,                                                        c2, at[6,10];
```

2. If the microinstruction before the "pCalln" specifies a branch, dispatch or "pRetn", then the specified bits will be OR'd into the value stored into the link register. The "at" is NOT required.

```
[] ← Reg xor RegA, NegBr,                                           c1;
pCall5, BRANCH[Pos, Neg],  {bit 3 of Link5 ← IF negative THEN 1 ELSE 0} c2;
Noop,                                                               c3, at[0,2];
```

3. "LnDisp" is used to dispatch on the value of link register "n" and is equivalent to "pRetn". Branch or dispatches can be simultaneously specified. The instruction after a "pRetn" must be constrained so that the "BRANCH", "DISPn" or "RET" has the desired affect. In addition, IA.7 of the next instruction must be 1 (MASS does this allocation).

The following example dispatches on "(0,,0,,Ureg.8,,Ureg.15) OR Link3" and places the result in Link2:

```
[] ← Ureg, L3Disp, XLDisp,                                   c1;
pCall2, DISP4[Table],                                        c2;
```

4. Each subroutine has an associated table of 16 possible return locations. On exit, the subroutine uses a link register (specific to the subroutine) to dispatch into the return table. Thus a subroutine usually only has 16 possible return locations (usually implying 16 possible call locations). Each location of the return table also has ia.7 set to 1 since the table is preceded by a pRetn. Similarly, each location of the call table has IA.7 set to 0 since each "CALL" is preceded by a "pCalln". Thus, the "pCalln" can not immediately precede a return point since IA.7 can not be resolved. (See example below). It is possible to have a "CALL" on a return point if RH registers are used instead of Link registers for subroutines (See C.6).

```
           pCall7,   {Link7 loaded before call point}          c2;
           Noop,                                               c3, at[0,10];
           Noop,                                               c1;
           CALL[Sub],                                          c2;
ReturnA:   Noop,   {return point 0}                            c2, at[0,10,ReturnA];

           pCall7,                                             c1;
           CALL[Sub],                                          c2, at[1,10];
ReturnB:   Noop,   {return point 1}                            c2, at[1,10,ReturnA];

           pCall7, CALL[Sub],   {only 1 call to Sub can be of this form}   c2;
ReturnC:   Noop,   {return point 2}                            c2, at[2,10,ReturnA];

{The following type of call IS NOT POSSIBLE:
           pCall7,                                             c2;
ReturnC:   CALL[Sub],   {call and return point}                c2, at[x,10,ReturnA];}

Sub:       Noop,                                               c3, at[2,10];

           pRet7,                                              c3;
           RET[ReturnA],                                       c1;

{Sub's return dispatch table is
           0: ReturnA
           1: ReturnB
           2: ReturnC}
```

5. Since condition bits can be simultaneously specified with a "pRet", there can be conditional return points. The same is true of "pCall" so conditional entry points are possible. If the least significant bit of the return address is not masked, conditional calls always imply conditioned returns (since the condition bits are saved in the Link register).

6. RH registers can also be used for subroutine calling. This format is easier to use since there are less address constraints: The call points don't need to be "at'd" and return addresses need not have IA.7=1.

```
           rhRet ← 0,                                          c2;
           Noop,                                               c3;
           Noop,                                               c1;
           CALL[Sub],                                          c2;
ReturnA:   Noop,   {return point 0}                            c2, at[0,10,ReturnA];

           rhRet ← 1,                                          c1;
           CALL[Sub],                                          c2;
ReturnB:   Noop,   {return point 1}                            c2, at[1,10,ReturnA];

           rhRet ← 2, CALL[Sub],                               c2;
ReturnC:   Noop,   {return point 2}                            c2, at[2,10,ReturnA];

Sub:       Noop,                                               c3;

           [] ← rhRet, XDisp,                                  c3;
           RET[ReturnA],                                       c1;

{Sub's return dispatch table is
           0: ReturnA
           1: ReturnB
           2: ReturnC}
```

7. By using RH registers more than 16 return points can be accomodated through multiple return-dispatch tables. Two tables imply 32 call/return locations.

```
Sub:        Noop,                                                   c3;

            [] ← rhRet, XRefBr,                                     c2;
            [] ← rhRet, XDisp, BRANCH[Table1, Table2],              c3;
Table1:     RET[ReturnA],                                           c1, at[0,2,Table2];
Table2:     RET[ReturnQ],                                           c1, at[1,2,Table1];
```

D. SU Registers

The Stack-U registers (abbreviated "SU") are addressed implicitly by the fS field. The "Cin" field determines whether SU will be read or written. The SU addressing mode and Cin field only effect the SU registers if the "EnSU" (Enable SU) field is 1.

1. When writing SU, Cin must be 1. If SU is being written via the A bypass, ALU arithmetic must assume a Cin of 1. Similarly, if SU is being placed on the X bus only, Cin must be 0.

```
                RHreg ← Ureg, Reg ← Reg + RegA,                          c3;
                Ureg ← RegA, Reg ← Reg + 1,                              c1;
```

2. If fS.2 is 0, the SU address comes from the stack pointer (stackP). The fZ field is free to be interpreted as either fZNorm or a Nibble.

```
                STK ← RegA, Reg ← RegA + 0FF + 1,                        c3;
                Reg ← STK, rhReg ← RegA LRot0, XLDisp,                   c1;
```

3. If fS.2 is 1, the SU address is rA,,fZ. Since the aS value which combines a U with an R register is "D,A" and since rA is also used to specify the high four U address bits, a given R register can only be combined (in one statement) with U registers which are in the block of 16 given by the value of the R register. If A bypass is used in a statement which uses a U register, the same restriction is true. Statements which only read or write U registers are not affected.

```
                Reg ← Ureg,                                              c3;
                Reg ← Ureg xor RegA,   {RegA = Ureg[0-3]}                c1;
                Ureg ← RegA, Reg ← MD,   {RegA = Ureg[0-3]}              c1;
```

4. If fS.2 is 1 and fZ of the previous instruction was "AltUaddr", then the SU address is rA,,Y[12-15]. Here Y[12-15] is for the previous instruction (the same one which contained the AltUaddr). This U register indexing mode can be used to efficiently load a block of 16 U registers from memory (such as from an IOCB). The individual U registers can be used later, one at a time. The following example assumes the 16 words in memory are hex aligned (rAddr is 0 mod 16).

MASS expects a register of type UY, where the 4-bit register number references the block of 16. "AltUaddr" can not occur in c3, and a UY register should not be used in c1--the addressing mode can't be used across clicks.

```
                RegDef[Ublock, UY, 0E];

Cont:           MAR ← [rhAddr, rAddr], rAddr ← rAddr + 1,                c1;
                [] ← ~0 and rAddr, AltUaddr, NibCarryBr {tests for 0 nibble},  c2;
                Ublock ← rData, rData ← MD, BRANCH[Cont, Exit],          c3;
```

5. The Alternate U addressing mode can be used with IOXIn (but not fZNorm or Nibble).

```
                [] ← <Uaddress>, AltUaddr,                               c2;
                Ublock ← rData, [] ← RH, XDisp,                          c3;
```

E. Mesa Stack

1. For the PrincOps stack, the stackP equals the number of words on the stack. Thus, the stackP=0 for an empty stack; and stackP=8 for a full stack. Also in the PrincOps stack, the stackP points one above the top of stack, thus a PrincOps Pop must decrement the stackP & return the top of stack and a PrincOps push must write first, then increment the stackP.

In the Dandelion, the top of stack is kept in TOS and TopOfStack-1 is kept on the top of the stack in the U register file (STK). The stackP always points at TOS-1 in the STK. Thus, to pop STK one moves STK[stackP] to TOS and decrements the stackP, and to push one increments the stackP and then moves TOS to STK[stackP]. In order to keep the values of stackP identical for the two Stack representations, PrincOps stack locations 1-8 should be mapped into U locations 2-9. For example, If the PrincOps stack has one entry, then TOS is full, stackP=1, and (with a stack push) TOS could be saved in STK[2]. If the stack is empty, then TOS is empty, stackP=0, and (with a stack push) TOS could be saved in STK[1].

Figure 4 shows the stack from empty to overflowing.

U[9] is necessary if one assumes one can always save TOS into STK, i.e., if the stack is full (stackP=8) and we save TOS, the place it will go is U[9]. For example, if JEQn is executed on a full stack, then U[9] is necessary (This would *not* be stack overflow). If a Mesa Push is tried on a full stack, the write into U[9] would occur before it could be stopped (This would be a stack overflow).

Since stackP=8 & Push does not define stack overflow, we define overflow to be stackP=9 and overflow. This implies that whenever a true Mesa Stack Push is desired, the stackP must be incremented twice and decremented once. The idea is that stackP can always be incremented once (to save TOS into STK) without fear of overflow,. but if we are truly putting one more word on the Stack, we must increment it once more.

Stack underflow occurs when stackP=0 and a Pop is attempted.

The maximum sized Mesa stack is 14 words (overflow at stackP=15 and Push, underflow at stackP=0 and Pop).

If a stack error occurs, one additional emulator click beyond the one which erred can execute before the emulator begins executing control store location 0 in c1.

2. To ameliorate checking for stack overflow or underflow, the pop function fields have been asymmetrically encoded. The following tables show the allocation of pops and pushs among the function fields and their effect on the stackP and the StackErrProm when multiple pops and pushs are specified in the same microinstruction.

fX	fY	fZ
push	push	push
pop		pop

functions	stackP	Check For
pop	-1	underflow
push	+1	overflow
fXpop, push	0	underflow (simulates a Pop)
push, fZpop	0	overflow (simulates a Push)
fXpop, fZpop	-1	underflow (simulates a Pop-Pop)
fXpop, fZpop, push	0	underflow (simulates a Pop-Pop)

3. In general, the previously executed Mesa instruction may complete executing without saving TOS into STK. Therefore each Mesa instruction implementation must, if neccessary, save TOS into STK (at STK[stackP+1]) before it modifies TOS. According to PrincOps, if TOS is an argument to the bytecode, TOS should be saved away (so it can be recovered by a Mesa PUSH) if either the Mesa bytecode does not change the contents of the Stack or does not change the value of the stack pointer. SLn and JEQn are two examples.

Note that as a part of normal Stack maintenance, TOS must be saved into STK if the Mesa opcode is merely pushing data onto the Stack.

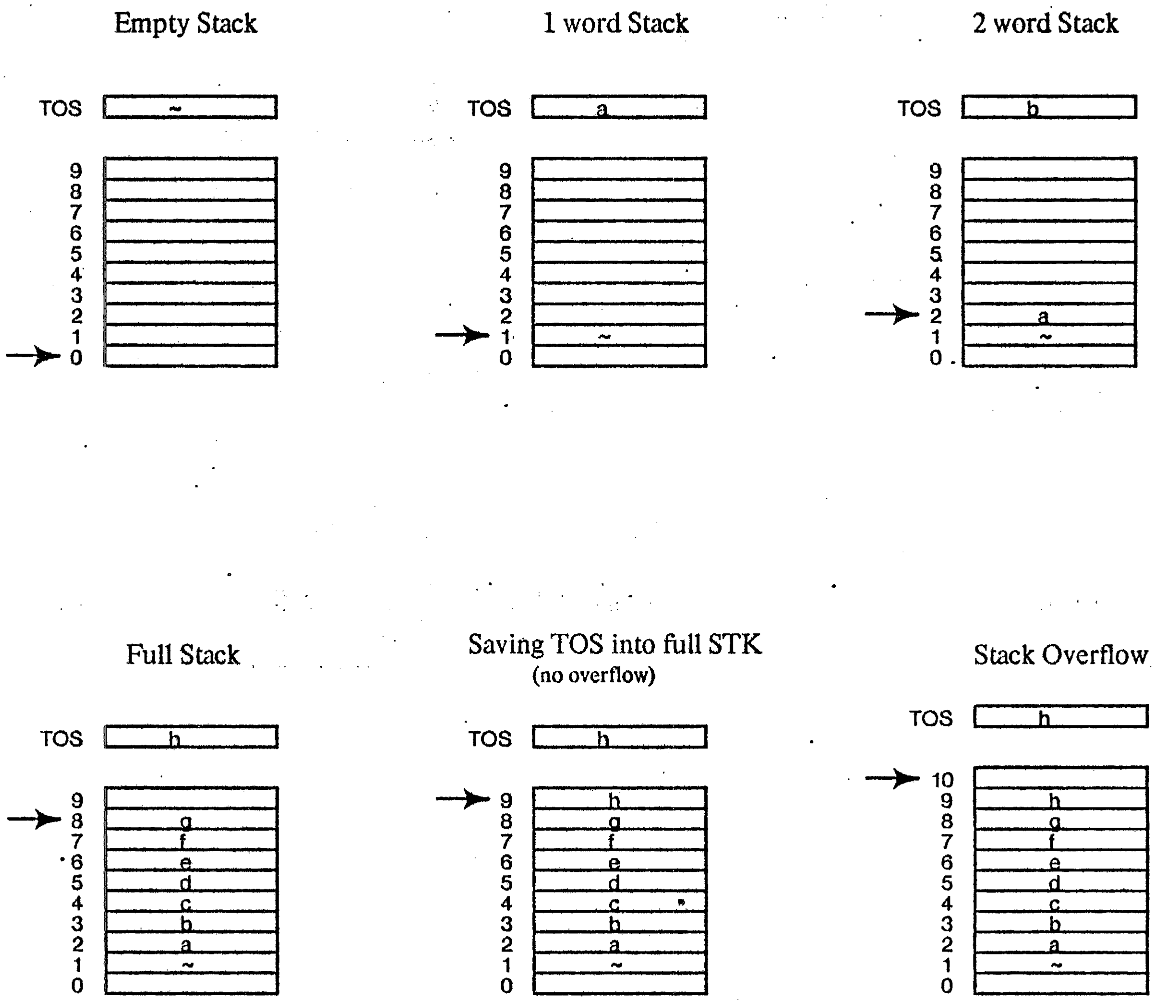


Figure 4. Mesa Stack examples

8 Jan 79
MesaStack.sil

F. Mesa Instruction Buffer

The instruction buffer holds a maximum of 3 bytes--the minimum number necessary to complete a Mesa instruction. Whenever a Mesa opcode completes and there are not 3 bytes in the buffer, a microcode trap is caused which results in refilling of the buffer. The so-called "refill" microcode executes in one click if 2 more bytes are needed and in two clicks if 4 are needed. The Refill code also dispatches on the front byte of the buffer (so if 4 bytes were fetched, 3 are retained).

The instruction buffer has four possible states as given by the 2-bit "ibPtr" register:

state	ibPtr
3 bytes (full)	0
2 bytes	1
1 byte	3
0 bytes (empty)	2

Thus the ibPtr counts 0, 1, 3, 2, 2, 2, ...

"←ib", "IBDisp", and "AlwaysIBDisp" cause the ibPtr to "increment" by 1. However, "←ibNA", "←ibHigh", and "←ibLow" do not change the ibPtr. "IB←" causes ibPtr to be set to 1 if it was 2 and otherwise sets it to 0. "IBPtr←0" sets it to 1 and "IBPtr←1" sets it to 3 (See Figure 5).

As shown below, the 3 bytes of the buffer are labeled IB[0], IB[1], and ibFront. IB[0],,IB[1] is parallel-loaded by a word from the X-bus when fY="IB←". ibFront is loaded if fY="IBDisp", "AlwaysIBDisp", "←ib" or "IB←". "IB←" only loads ibFront if the old ibPtr=2. When it is loaded, its value comes from IB[0] if the old ibPtr.1 was 0 and from IB[1] if the old ibPtr.1 was 1. When ibFront is loaded by fZ="IBPtr←n", its value comes from IB[n].

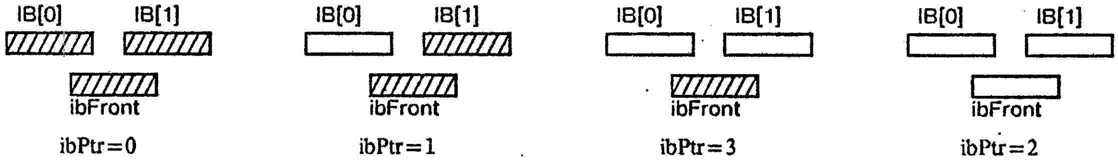


Figure 6a: Mesa Instruction Buffer States

"←ib" and "←ibNA" cause ibFront to be placed onto the X-bus, while "←ibHigh" puts the high 4 bits, and "←ibLow" the low 4 bits, of ibFront onto the X-bus.

```
MAR ← [rhL, L + ib],                                   c1;
rhT ← ibLow + 1,                                       c2;
```

"IBDisp" causes a 256-way dispatch based on the value of ibFront. It can only occur in c2 (since only 4 bits of branch/dispatch bits are saved across clicks). The high 4 bits of ibFront replace INIA[4-7], while the low 4 bits of ibFront are OR'd with INIA[8-11]. INIA[0-3] are unaffected, so there are 16 possible 256-way dispatch tables which can be used. The macro "DISPNI" is equivalent to "GOTO".

```
IBDisp,                                                c2;
DISPNI[OpTable],                                       c3;
```

If IBDisp is executed and ibPtr does not equal 0 (full), a microcode trap is caused to location 400'x for a buffer-empty refill or 500'x for a buffer-not-empty refill. If there is a pending Mesa interrupt request (MInt=1), a microcode trap is caused to location 600'x or 700'x. If either trap occurs the buffer state does not change. "←ib", "IB←", "IBPtr←0", and "IBPtr←1" do not cause an instruction buffer trap. The Error microcode trap to location 0 has priority over the buffer traps.

"AlwaysIBDisp" will never trap--it ignores a pending Mesa interrupt request and a non-full buffer. AlwaysIBDisp is encoded by fY="IBDisp" and fZ="IBPtr←1". If the microinstruction before either an IBDisp or AlwaysIBDisp is a "MAR←" and a PageCross occurs, the IBDisp or AlwaysIBDisp will be canceled and the state of the buffer will remain unchanged.

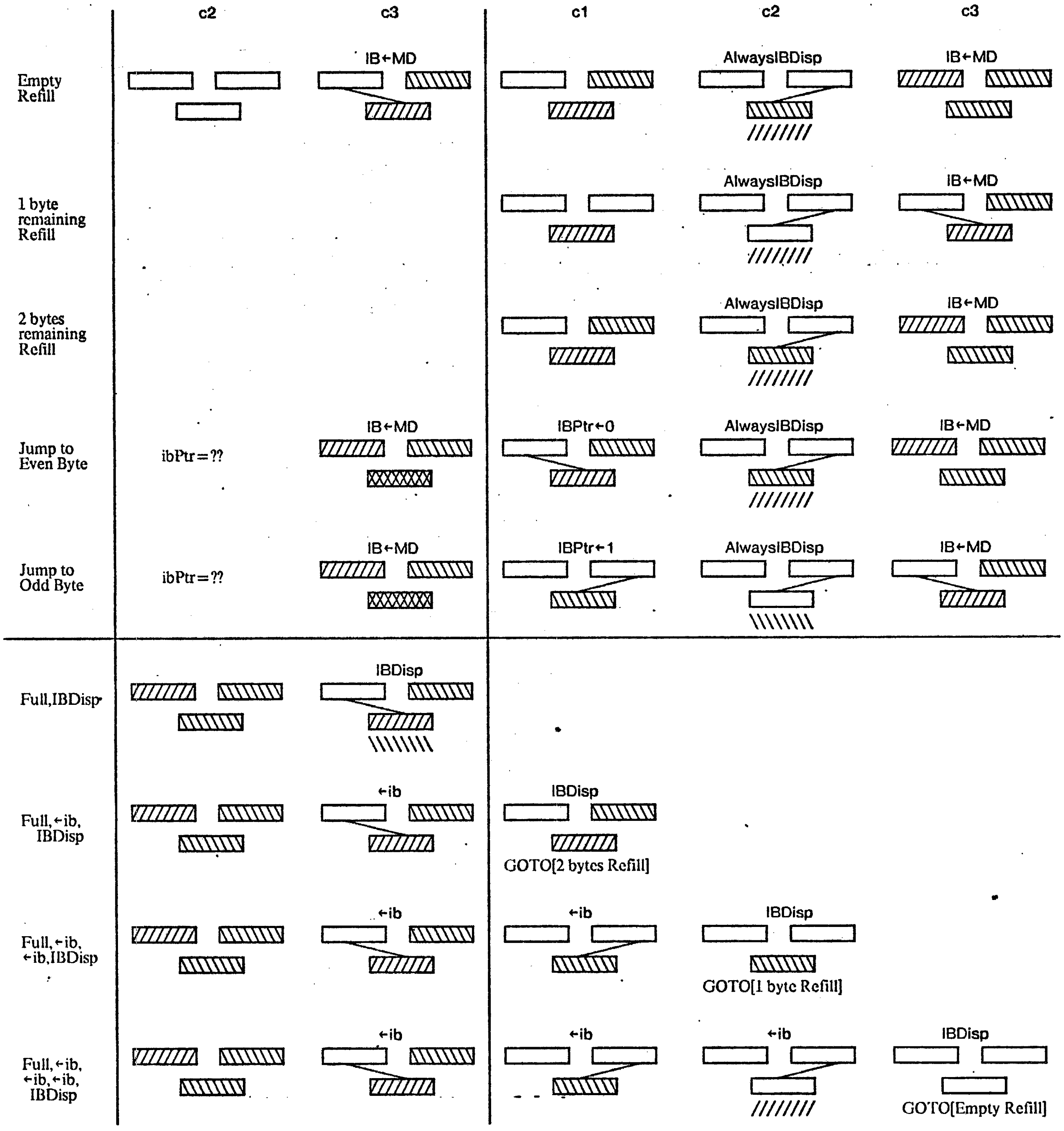


Figure 7. Mesa Instruction Buffer Sequences

G. Memory

The memory address register is 18 bits wide (expandable to 20 bits) for real addresses and 22 bits for virtual addresses (expandable to 24). Using 16K chips, the memory size is 192K, using 64K chips it is 768K.

1. "MAR←[rhReg, <arithPhrase>]" implies a real address reference. "MAR←" can only occur in c1. The first argument specifies which RH register holds the high 2 address bits. The rB field is set to the value of rhReg. Notationally, <arithPhrase> is anything that can occur on the right side of an arithmetic statement.

An RH register should not be loaded simultaneously with a MAR←.

The memory data register can only be loaded in c2. A "MDR←" has no effect unless the previous c1 contained a MAR←. MDR can be loaded by any register, including SU and IOIn.

Data is only delivered from the memory in c3. A "←MD" has no effect unless the previous c1 executed a MAR←. Besides R and RH registers, MD can be loaded into IOOut or the Instruction Buffer or be the source of a branch/dispatch.

If a memory location is both read and written in the same click, the old contents of the location is returned:

```
MAR ← [rhReg, Reg + 0],                                   c1;
MDR ← RegB,                                               c2;
RegB ← MD,                                                c3;
```

2. "MAR←" has 3 important side effects:

(a) "MAR←" forces aS="0,B" and aF="RorS" for the high half of the ALU. This causes the output of the high half of the ALU (bits 0-7) to equal the contents of the R register given by the rB field. Thus, if A bypass is not used, the upper 10 bits of the memory address (the page address) come from the RH-R pair given by the rB field, while the lower 8 bits (the displacement within a page) come from the source defined by <arithPhrase>.

```
MAR ← [rhReg, RegA + 0],                                  c1;
```

causes

```
YHbus ← rhReg,
Ybus[0-7] ← Reg[0-7],             {NOT RegA[0-7]}
Ybus[8-15] ← RegA[8-15].
```

If A bypass is used, the Y bus (and MAR) receive the complete R register given by the rA field, but the ALU still delivers rB in the high half and the high 2 bits of address still come from RH[rB]. Thus,

```
MAR ← [rhReg, RegA], Reg ← RegA + 1,   {Reg[0-7] unchanged}   c1;
```

causes

```
YHbus ← rhReg,
Ybus[0-7] ← RegA[0-7],             {NOT RegA+1}
Ybus[8-15] ← RegA[8-15] + 1.
```

A consequence of forcing aS="0,B" and aF="RorS" in the high half of the ALU is that page carries do *not* propagate into the high half.

```
            MAR ← Reg ← [rhReg, Reg+0F+1],  {Reg[0-7] unchanged}            c1;
```

(b) "MAR←" automatically specifies a PageCross branch (See Sec. A). Thus, a change in the flow of control will occur if a page boundary crossing has been indicated by the PageCross branch. This usually implies that a remapping of the real address is necessary.

The PageCross branch occurs in INIA.10 (*Not* the usual INIA.11 of 2-way branches). Thus, the "BRANCH[LabelX, LabelY, 1]" form must be used after a "MAR←".

```
            MAR ← [rhReg, Reg+0F+1],                                        c1;
            BRANCH[Continue, ReMap, 1],                                     c2;

Continue:   Reg ← MD,                                                       c3;
ReMap:      Noop,                                                           c3;
```

The implied PageCross branch can be canceled by a "CANCELBR[Label, 2]".

```
            MAR ← [rhCnt, RegA], Cnt ← Cnt + 1,                             c1;
            CANCELBR[Cont, 2],                                              c2;
Cont:       Reg ← MD xor Reg,                                               c3;
```

Since the 2901 produces carries on logical operations (believe it or not), you must explicitly say "MAR ← [rh, R+0]" (where "MAR ← [rh, R]" was desired) in order to prevent a PageCross branch. In general, if MASS doesn't see a "Reg+0", then the "MAR←" must be followed by either a BRANCH, DISPn, or a CANCELBR. If "Cin←pc16" is present with "MAR←", then any "MAR←" is an implied branch.

```
            MAR ← [rhReg, RegA+0],                                          c1;
            Noop,  {CANCELBR not required here}                             c2;
```

Since the automatic PageCross branch occurs in INIA.10, other branch conditions can be simultaneously specifed.

```
            MAR ← Reg ← [rhReg, Reg+1], ZeroBr,                             c1;
            DISP2[Table],                                                   c2;

Table:      GOTO[Cont],   {here if no PgCross, Cnt#0}                       c3, at[0,4,Table];
            GOTO[Done],   {here if no PgCross, Cnt=0}                       c3, at[1,4,Table];
            GOTO[ReMap],   {here if PgCross, Cnt#0}                         c3, at[2,4,Table];
            GOTO[Done],   {here if PgCross, Cnt=0}                          c3, at[3,4,Table];
```

(c) If a PageCross occurs in a "MAR←", then a following "MDR←" or "IBDisp" is canceled. This prevents writing into the wrong memory page (if A bypass was not used in the MAR←) or dispatching on the next Mesa instruction if a page crossing has been indicated.

```
            MAR ← Reg ← [rhReg, Reg+1],                                     c1;
            MDR ← RegA, BRANCH[Cont, ReMap, 1],  {MDR← canceled if we go to ReMap}   c2;
```

3. (a) A memory address can be incremented either before or after being sent to MAR.

```
            MAR ← Reg ← [rhReg, Reg+1],                                     c1;
            MAR ← [rhReg, Reg], Reg ← Reg+1,                                c1;
```

(b) The automatic PageCross branch can be used to indicate the end of a count sequence, where the initial Cnt equals 256-count. rhCnt is used for bank select:

```
            MAR ← [rhCnt, Reg], Cnt ← Cnt+1,                                c1;
```

H. The Map

The Map is a 16K linear table which is indexed by a 14-bit virtual page number and contains a 10-bit real page number (expandable to 12 bits) and some flags pertaining to the virtual page. The Map is located immediately above the low 64K display bank, real addresses 10000'x to 13FFF'x. Figure 6 illustrates the mapping process.

1. References to the Map are notationally indicated by "Map←". Either the fX or fZ field must be set to "MapRef". "Map←" causes NO side affects (such as those caused by "MAR←").

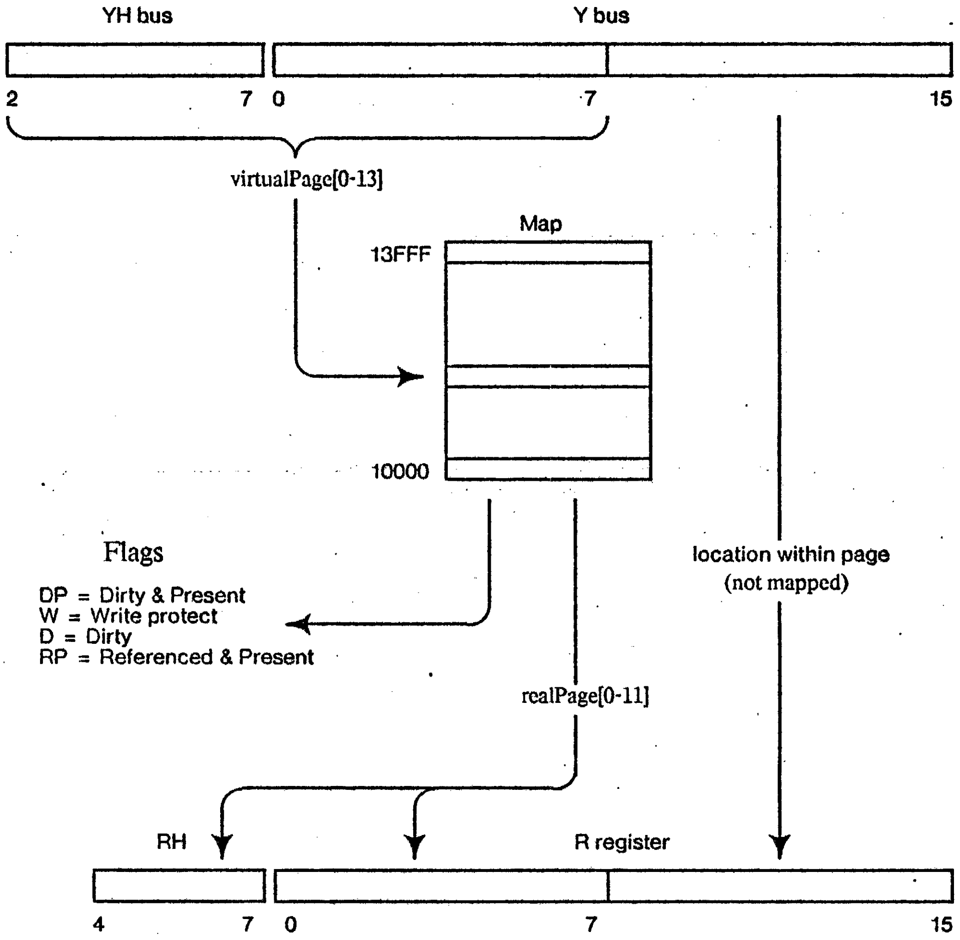
"Map←" supplies to the memory system a 22-bit virtual address in YH,,Y. An RH register, as addressed by field rB, holds the high 6 bits of the virtual address and an R register typically supplies the low 16 bits. The upper 14 bits of this address are used to index into the Map.

```
Map ← [rhReg, Reg],                                   c1;
Map ← Reg ← [rhReg, Reg+0F+1],                        c1;
Map ← [rhReg, Reg], Reg ← Reg + 1,                    c1;
```

2. If "MapRef" is specified during a "MAR←", all of MAR←'s side affects occur (see above) and the MapRef correctly causes a read from the Map. If a "MDR←" or "←MD" is executed without a preceeding "Map←", the "MDR←" or "←MD" have no effect.

3. A "Map←" will cause a microcode trap to location 0 (and set the EKErr register) in the Emulator if either bits 0 or 1 of an Emulator virtual address are non-zero. (There is no trap for IO microcode). The flow of normal Mesa byte code execution stops and a Mesa Xfer occurs.

If a virtual address error occurs, one additional emulator click beyond the one which erred can execute before the emulator begins executing location 0 in c1. Since the Mesa PC and stackP can change in this additional click, they can not be backed up to their original values. The memory address is also lost.



Map Entry Format:

realPage[4-11]	dp	w	d	rp	realPage[0-3]
0	7				15

Figure 5. Dandelion Map Reference

9 Feb 80
MapRef.sil

4. The following description is contained in [Iris]<Workstation>ExampleMap.mc.

{A map entry has the following format:

rp[4-11]	dp	w	d	rp	rp[0-3]
0	8	9	10	11	

where,

rp[0-11]	[12-15],,[0-7]	real page number. Implies max memory addr of 20 bits => 1,048,576 words
dp	8	Dirty & Present
w	9	Write Protected
d	10	Dirty
rp	11	Referenced & Present

The map flags have the following interpretation:

W	D	R = RP	DP	Present	
0	0	0	0	1	Untouched, unprotected page
0	0	1	0	1	Unwritten, read page
0	1	0	1	1	{reserved for software}
0	1	1	1	1	Written page
1	0	0	0	1	Clean, protected page
1	0	1	0	1	Protected, read page
1	1	0	0	0	Vacant page
1	1	1	0	1	{reserved for software}

PrincOps defines vacant to be W,,D,,R = 6. (Version 1.0 has 6 or 7). Note that Referenced is equivalent to Referenced and Present.

In addition to the three standard bits, W, D, and RP, the bit DP is maintained by the microcode. DP is true if the page is Dirty and Present.

These bits are utilized as follows: When the microcode is doing a MapRef with the intent of reading a word from the page it branches on the RP bit (by using "XRefBr"). If RP is 0, then either the page has not been referenced yet or it is not present; in either case more work must be done. If RP is 1, the microcode can proceed assuming no map maintenance of any kind is required.

If RP was 0, then the microcode does a 4-way dispatch on W,,D and acts according to the following table:

W	D	action
0	0	set RP and do a real memory read
0	1	set RP and do a real memory read (changes flags=2 to flags=3)
1	0	set RP and do a real memory read
1	1	Page Fault

Similarly, when the microcode is referencing the Map with the intent of writing a word into the page it branches on the DP bit (by using "XDirtyDisp" = "XLDisp". Note that the branch occurs in INIA.10). If DP is 0, then either the page has not been written yet, or it is write protected, or it is not present; in any case more work must be done. If DP is 1, the microcode can proceed assuming no map maintenance of any kind is required.

If DP was 0, then the microcode does a 4-way dispatch on W,,D and acts according to the following table:

W	D	action
0	0	set D and DP and do a real memory write
0	1	{Control shouldn't reach here}
1	0	Write Protect Fault
1	1	Page Fault

IO microcode can quickly check whether the page is present and the flags are corrrect by using XDirtyDisp or XRefBr.

"MAR←"'s feature of forcing "0 or B" in the high half of the ALU is used to combine the real page number from the Map with the displacement into the page given by the low byte of the virtual address.

Mesa Emulator Note: For PageFaults and WriteProtectFaults, the Mesa PC and the stackP must be restored to what they were at the begining of the Mesa instruction. The STK (and TOS), along with the rest of the Mesa machine state, must be restored if it changed.

Example 1 assumes we have been given a virtual address in rhV,,V and we want to map it into a real address and do a read. Example 2 is a possible implementation of the W1 opcode: a virtual write within an MDS. Example 2 uses a subroutine to update the map entry.
}

```
{Example 1:
Memory Read given virtual address in rhV,V.  R holds real address and data from memory.}

Start:        Map ← [rhV, V],                                          c1; {rA, fY, fX or fZ unused}
              Noop,                                                    c2;
              R ← MD, rhR ← MD, XRefBr,                                c3; {rA, fZ unused}

{The MAR← causes the ALU to output R[0-7],,V[8-15] onto the Y bus, where R[0-7] holds the low byte of the
real page number and V[8-15] holds the location in the page from the original virtual address. The high 4 real
page bits are put onto the YH bus from rhR.}

RedoR:        MAR ← [rhR, V], BRANCH[MapUpDate, MapOK],                c1; {rA, fX, fY, fZ unused}
MapOK:        Noop,                                                    c2;
              R ← MD, GOTO[ProcessData],                               c3; {rA, fX, fY, fZ unused}

{Either this is the first time we have referenced the page, or it is mapped out}
MapUpDate:    Noop,                                                    c2;
              [] ← LRot12 R, XDisp,                                    c3;

              Map ← [rhV, V], DISP4[FixRFlags, 8],                     c1;

FixRFlags:    MDR ← R or 10, GOTO[ReRead],  {Set RP}                   c2, at[8,10,FixRFlags];
              MDR ← R or 10, GOTO[ReRead],  {Set RP}                   c2, at[0A,10,FixRFlags];
              MDR ← R or 10, GOTO[ReRead],  {Set RP}                   c2, at[0C,10,FixRFlags];
              GOTO[RestoreStateAndPageFault],                          c2, at[0E,10,FixRFlags];

{Set a pending branch so the BRANCH at RedoRead goes to MapOK}
ReRead:       [] ← 0, ZeroBr, GOTO[RedoR],                             c3;

{Example 2: Mesa Opcode W1
Memory Write in MDS. Q holds virtual address. Subroutine to update map: Link0 holds the return address and
Link1 encodes the necessary state fixup.}

W1:           Map ← Q ← [rhMDS, TOS + 1], L1 ← n,                      c1;
              T ← STK, pop, pCall0,                                    c2, at[n,10];
              R ← MD, rhR ← MD, XDirtyDisp,                            c3, at[0,10];

RedoW:        MAR ← [rhR, Q + 0], {CALL}BRANCH[WMapUD, WMapOK, 1],     c1, at[0, RedoW];
WMapOK:       MDR ← TOS, TOS ← T,                                      c2;
              PC ← PC + 0, Cin←pc16, DISPNI[@Noop],                    c3;


{Subroutine to do map updates for writes.}

WMapUD:       Noop,                                                    c2;
              [] ← LRot12 R, XDisp,                                    c3;

              Map ← [rhMDS, Q], L1Disp, DISP4[FixWFlags, 1],           c1;

FixWFlags:    MDR ← R or 0A0, pRet0, CANCELBR[ReWrite, 0F],            c2, at[0,10,x];
              MDR ← R or 0A0, pRet0, CANCELBR[ReWrite, 0F],            c2, at[2,10,x];
              T ← sWriteProtect, DISP4[WTrap],                         c2, at[4,10,x];
              T ← sPageFault, DISP4[WTrap],.                           c2, at[6,10,x];

ReWrite:      [] ← 2, XDisp, RET[RedoW],                               c3;

WTrap:        push, GOTO[PageFault],                                   c3, at[n,10,WTrap];
```

I. Bus Sinks

Destination	Source Bus	
MAR←	YH,,Y	Memory Address Register
MDR←	Y	Memory Data Register
STK←	Y	Stack (SU reg addr = 0,,stackP) (enSU = 1, Cin = 1, fS.2 = 0)
stackP←	Y	Stack Pointer
U←	Y	U register (SU reg addr = fA,,fZ) (enSU = 1, Cin = 1, fS.2 = 1)
IB←	X	Mesa Instruction Buffer
RH←	X	RH registers
KOData←	X	Rigid Disk Output Data reg
XOData←	X	Xerox Wire Output Data reg
POData←	X	Printer Output Data reg
IOPOData←	X	IOP Output Data reg
DCtlFifo←	Y	Display Output Data Control Fifo
DBorder←	Y	Display Output Data border register
KCtl←	X	Rigid Disk Control reg
XCtl←	X	Xerox Wire Control reg
MCtl←	Y	Memory Control reg
IOPCtl←	X	IOP Control reg
DCtl←	X	Display Control reg
PCtl←	X	Printer Control reg

J. Bus Sources

Note that Xbus bits [0-7] are set to zero when Nibble, Byte, or IOXIn values 8 through 0F are specified.

Source	Destination Bus	
← Y LRot0	X	Left Rotate Y by 0
← Y LRot4	X	Left Rotate Y by 4
← Y LRot8	X	Left Rotate Y by 8
← Y LRot12	X	Left Rotate Y by 12
fZ	X	0-0F 4-bit constants
fY,,fZ	X	0-0FF 8-bit constants
←MD	X	Memory Data
←STK	X	Stack (SU regs addr = 0,,stackP) (implies enSU = 1, Cin = 0, fS.2 = 0)
←U	X	U register (SU regs addr = fA,,fZ) (implies enSU = 1, Cin = 0, fS.2 = 1)
←RH	X	RH registers
←ib	X	Instruction Buffer
←ibNA	X	Instruction Buffer (doesn't advance IBPtr)
←ibLow	X	ib[4-7]
←ibHigh	X	ib[0-3]
←KIData	X	Rigid Disk Input Data
←XIData	X	Xerox Wire Input Data
←IOPIData	X	IOP Input Data
←ErrIBStkP	X	EmuErrAB,,IBPtr,,stackP
←KStatus	X	Rigid Disk Status
←XStatus	X	Xerox Wire Status
←PStatus	X	Printer Status
←MStatus	X	Memory Status (Syndrome, SE, DE, ErrLog)
←IOPStatus	X	IOP Status reg

K. Miscellaneous Functions

Cin←pc16	Carry in to ALU or into Shift Ends is pc16
SE←pc16	Carry in to ALU or into Shift Ends is pc16
Cin ← 0	Carry in to ALU or into Shift Ends is 0
Cin ← 1	Carry in to ALU or into Shift Ends is 1
SE ← 0	Carry in to ALU or into Shift Ends is 0
SE ← 1	Carry in to ALU or into Shift Ends is 1
IBDisp	Causes a microcode trap to 600x or 700x if MesaInt pending, trap to 400x or 500x if buffer not full.
MesaIntRq	Sets Mesa Interrupt Request flag
IBPtr←0	ibFront ← IB[0]
IBPtr←1	ibFront ← IB[1] (With IBDisp, changes to AlwaysIBDisp)
AlwaysIBDisp	Encoded by fY = IBDisp & fZ = IBPtr←1 (prevents IBDisp from trapping)
ClrIntErr	Clear error bits & interrupt req bit
EnterKern	Breakpoint - go to Kernel task
ExitKern	Leave Kernel task and commence regular task scheduling
Refresh	Memory refresh(only in cycle1. Ignored in c2 or c3)
push	Increment stackP by 1 (trap to loc. 0 if overflow)
pop	Decrement stackP by 1 (trap to loc. 0 if underflow)
ClrDPRq	Reset Display and Printer Requests
ClrKFlags	Reset Rigid Disk Flags
ClrIOPRq	Reset IOP Request
ClrXRq	Reset Xerox Wire Request
Noop	assure noop F fields

L. Microcode Conventions:

The file [Iris]<Workstation>mc>form.dm embodies the following suggested conventions.

1. All microcode should be in either font Helvetica8 or Helvetica 10. (The [Hardcopy] section of user.cm should be updated to cause Empress to use this font instead of Gacha8. Returns should be used for spacing since Empress doesn't know about paragraph looks and will otherwise squish the paragraphs together. Empress will also change the tabs, but the source is still readable.)

2. The tab stops for Helvetica8 microcode are 140 pt., 160 pt., and 420 pt. The stop at 160 keeps lines with long labels from jumping over to the comment field when Bravo is not in hardcopy mode. (160-140 is less than the length of most statements.) In form.mc, the tab stops for comments are 140, 160, 180, 200, and 420 pts.)

3. The top of the microcode file should have at least the following information;

```
{File name: <Name>.mc
 Description: <what file contains>,
 Author: <being>,
 Created: <date>,
 Last Edited: <date & time>}
```

4. The general format of a line is:

```
<Label:>        <Arrow clause>, <functions>, <DispBr>, <Call/Ret>, <GOTOfield>,          <cycle>, <at>;
```

There should always be one space after a comma. Those statements which are executed together within the same click should be preceded and followed by a blank line. The <cycle> and <at> clauses should be located after the 420 pt. tab. It also helps to locate the last comma at the 420 stop. For example:

```
                TOS ← TOS + 1,                                                c3;

Shift:          [] ← ~17 and TOS                                              ,c1;
                PC ← PC+0, Cin←pc16, pCall1                                   ,c2;
                TT ← STK {TT ← v}                                             ,c3;

                TOS ← T LRot1, DISP4[MaskTbl],                                c1;
```

5. Comments can be imbedded within a line.

6. MASS generates the required "at" phrases after BRANCH's, but not for dispatch tables. For readability, it helps if "at" clauses are explicitly placed to identify their partners when more than one line away. For example, the "at" clause of "MulLoop" points to the instruction "MLDEnd", which then points back to MulLoop."

```
MulLoop:        [] ← Q, YDisp,                                                c1, at[0,2,MLDEnd];
                TT ← TT - 1, ZeroBr, BRANCH[MPlier0, Mplier1, YOdd],          c2;

MPlier0:        T ← T, DRShift1, BRANCH[MulLoop, MLDEnd],                     c3, at[];
MPlier1:        T ← T + TOS, DRShift1, BRANCH[MulLoop, MLDEnd],               c3, at[];

MLDEnd:         STK ← T {long.high/rem}, pop {point at long.low/quot},        c1, at[1,2,MulLoop];
```

7. If names consist of multiple parts, the first letter of each sub-name should be capitalized. All names should not be all capitalized. For example: MulLoop, JumpSign, MapOK. "i" should be used instead of the capital form "I", which is indistinguishable from "l" (small L) and similar to "1" (one). Try to avoid a solitary "O" in a name since it looks like a "0" (zero).

8. All instances of user names and reserved words must follow the same capitalization, else MASS will not recognize them. (Searching problems in Bravo are reduced.)

9. U register names should start with the letter "U" or "u", and RH register names with "RH" or "rh". For example: Uinterrupt, uTemp, RHrK, rhRtemp.

10. Arithmetic clauses (i.e., arrow clauses) should have a space before and after the ←. The — character can be used for the minus sign instead of -, which is microscopic in size (in all fonts). In Bravo type-in mode, this minus sign is entered by typing: n Ctl-S ESC Look g (see p.57 Bravo manual). It looks like a ▮ in bravo. For example:

```
T ← ~17 xor T, stackP ← 17,                              c2;
TT ← TOS — T,                                            c3;
```

11. The allocation and definition of registers used in the complete microcode system are given in [Iris]<Workstation>Dandelion.df.

Dandelion.df should *ONLY* be accessed via the Librarian Access Tool. Thus, "Mesa Access checkOutReason/r Dandelion/o" is used to check Dandelion.df out and move it to your disk, and "Mesa Access Dandelion/i" is used to return it to Iris.

The following files are needed on your disk: RunMesa.run, Mesa.image (or BasicMesa.image), and [Igor]<AlphaTools>Access.bcd.

Your user.cm should include the following entry:

```
[Librarian]
Server: Marion
NamePrefix: WorkStation
NameSuffix: df
```

[Iris]<Tools>Documentation>Access.press is a one page summary of the Access tool.

M. MASS

1. Overview

There are two types of source files. There are macro-and-defs files and there are microcode source files. The macro-and-defs files have a ".df" extension and microcode files have a ".mc" extension. ".df" files do not contain microinstructions and are global to an assembly. ".mc" files may have macros and defs. Intermediate files are produced for each micro-code source file to allow subsequent partial re-assemblies: ".ml", ".si", and ".eb" files are produced from every ".mc" file.

The ".fb" is the Burdock loadable file of allocated microinstructions. The ".ft" is a human readable form of the ".fb" file. Burdock also reads the MASS produced ".st" symbol table file.

While MASS is running, the cursor is reversed once per statement processed. During the allocation phase, a square cursor is displayed with the number of enclosed dots increasing to six. If errors are found in the file, MASS pauses with the cursor showing "hit me". As soon as any keyboard character is struck, MASS returns to the Alto executive (which may continue on to Bravo if the S macro was previously invoked).

The ".er" status file contains MASS version information and the errors encountered. Along with the complete statement which is in error, the phrase which caused the error is written into the status file. MASS actually displays an error message for each possible (and unsuccessful) way to encode the statement.

2. File Summary

The following files are used (u) or generated (g) by MASS in pass 1 or 2.

.mc	u	MicroCode source
.df	u	macro and Defs source File
.eb	ug	Early Binary (output of Pass 1, one per ".mc" file)
.ml	ug	Label constraint records & Reserves (output of Pass 1, one per ".mc" file)
.si	ug	Symbol Intermediate (output of Pass 1, one per ".mc" file)
.fb	ug	Final Binary (output of Pass 2)
.ft	g	Final binary Text (output of Pass 2)
.st	g	Symbol Table (output of Pass 2)
.er	g	status-error (output of Pass 2)

3. Command Line Switches

noswitch	use name for ".mc" input file and ".eb", ".ml", & ".si" output files.
/d	designates a ".df" file
/o	use name for ".er", ".fb", ".ft", and ".st" output files
/2	pass 2 only for this file (it has been previously assembled)
/x	satisfy imports from this file & exclude its locations from allocation

4. Sample Assembly Descriptions

```
MASS regs/d  macros/d  acode  aout/o
        Pass1:
                reads: regs.df, macros.df, acode.mc
                writes: acode.eb, acode.ml, acode.si
        Pass2:
                reads acode.ml, acode.si
                writes aout.st
                allocates acode
                reads acode.eb
                form final binary of acode
                writes aout.fb, aout.ft, aout.er

MASS config1a/d  source1/2  source2  source3/x  sourceout/o
        Pass1:
                reads: config1a.df, source2.mc
                writes: source2.ml, source2.eb, source2.si
        Pass2:
                reads source1.ml, source1.si
                        source2.ml, source2.si
                        source3.fb
                writes sourceout.st
                allocates source1, source2
                reads source1.eb, source2.eb
                form final binary of source1, source2
                writes sourceout.fb, sourceout.ft, sourceout.er
```

5. MASS Format descriptions:

Comments: All text between squiggly bracket pairs "{" and "}" is comment. The brackets nest, so they must be properly paired. (This is so sections of code which already contain comments can be commented out)

Labels & numbers: The case of letters in names and macros is important and must be consistent. Hexadecimal numbers are assumed, although a decimal number is specified by a trailing 'd and an octal number by a trailing 'b. Hex numbers which start with A-F must be prefixed by a zero: 0F = 0F'x = 15'd = 17'b.

Source Line Format: A line of source which defines a single microinstruction is a list of clauses which are terminated by semicolons. Clauses are separated from one another by commas. A clause is either a function field name, a macro invocation, or an arrow (arithmetic) clause. A name followed by a colon is the label of the microinstruction. Microinstructions can have multiple labels. Spaces are ignored (and this does not mean they are significant in variable names). Parentheses are used in arithmetic clauses (where they may improve readability but are ignored by MASS) and in macro definitions. Brackets are used to denote the argument list to macros (and used in "MAR←[...]", "Map←[...]", and "[]←").

Cycle Numbers: Each source line must have a cycle number macro: c1, c2, c3, or c*. c* inhibits wrong cycle warning from MASS. This is primarily used by the Kernel, but can be used in loops which are not a multiple of three micro-instructions and short subroutines which don't contain cycle constrained operations. Such loops must always exit on the same cycle number and should not contain "MAR←", "MDR←", or "←MD" since these depend on executing in cycle 1, 2, or 3, respectively. Burdock can not breakpoint instructions with a c*.

Register definition: All registers must be defined before their names are used. The macro "RegDef" is used to associate a name with a type (R, RH, U or UY) and a register number.

```
RegDef[Reg, R, 4];
RegDef[Ureg, U, 47];
RegDef[rhReg, RH, 4];
RegDef[Ublock, UY, 4];
```

Arrow formats: Arrow statements, like all clauses, are isolated by commas. They may have one or two destinations, as in

```
B ← B or A,                                              c1;
B ← SU ← B or A,                                         c1;
```

The right side of an arrow clause (the source) consists of a single entry, a double entry with an operator, or a triple entry with two (identical) operators. It is also possible for an entry to have a qualifying unary operator ("~" or "−"). For example:

```
B ← ~A,                                                  c1;
B ← A + 1,                                               c1;
B ← A + B + 1,                                           c1;
B ← ~A and B,                                            c1;
```

Constants can be 4 or 8 bits long. Note that a special form must be used for −1:

```
Reg ← 0FE,                                               c1;
Reg ← ~0F xor Reg,                                       c1;
Reg ← −5,                                                c1;
Reg ← ~Reg xor Reg, {used instead of Reg ← −1, or Reg ← ~0},   c1
```

MASS expects the single-bit-shift phrases to precede any arithmetic clauses and 4-bit rotate phrases (LRotn) to follow the quantity to be rotated. Parentheses are ignored. See sections A.1 and A.2.

```
Reg ← LShift1 Reg,                                       c1;
Reg ← RShift1 (Reg and 0F),                              c2;
Reg ← LRot1 (Reg + 1),                                   c3;

Reg ← Reg LRot4,                                         c1;
Reg ← (Reg LRot8) or Reg,                                c2;
rhReg ← (RegA + Reg) LRot12,                             c3;
```

Default Labels: A "$" can be used in BRANCH and CANCELBR macros to refer to the location of the next statement.

```
ZeroBr, BRANCH[MapUpDate, $],                            c1;
CANCELBR[$],                                             c2;
```

External variables: The macro "IMPORT" indicates which labels of the ".mc" file have been defined elsewhere. The "EXPORT" macro specifies labels which other modules will import. When MASS is assembling a group of files together, called an "assembly unit", (a combination of ".mc" and/or ".ml"/".si" files) it is illegal to have the same label defined twice or imported and defined. Labels defined in IMPORT macros will be assigned values from the EXPORT of a previous assembly unit by using the /x switch on its name.

Macro Definition: A macro is defined by supplying a name for the macro and a text string which will replace the name. Macros return nothing or a single number or variable (never a pair of arguments, for instance). Arguments (up to 9) may be supplied in the macro call and are referred to in the expansion by "#n" for n=0 to 9. The "#0" is replaced by the number of arguments in the macro call, and #1-9 will be replaced with the appropriate argument (or null if there is none). Parentheses must be used to enclose arguments containing commas. For example:

```
MacroDef[NewMacro, (OldMacro1[#1], OldMacro2[#2, #3])];
```

Macro Invocation: The invocation of a macro is caused by supplying the macro and optionally up to nine arguments. Arguments are separated by commas, and if an argument contains a comma, it must be enclosed within parentheses. Macro calls can not appear on the left side of arrow clauses, and only unargumented macros (such as macros defined by Set) can appear on the right side.

If an ".mc" file contains macro definitions, they remain valid for subsequent ".mc" files which are part of the same assembly unit.

Reserved Names: None of the function field or builtin macro names should be used as user names. For example, RH, ib, Q, XStatus, c1, Refresh, xor, and pop are all reserved. Null and Apass are also reserved names. "←" should not be present in user names.

6. Bravo S macro:

The S macro facilitates ping ponging between Bravo and MASS. It assumes "standard" file naming conventions. In particular, corresponding to a "Name.mc" microcode file, there is a "Name.cm" command file, and errors are placed into a "Name.er" file. A typical "Name.cm" file is:

```
MASS Dandelion/d Name/o Name
```

The macro is invoked by "BRAVO/S Name" This causes the ".mc" and ".er" files to be fetched into bravo. When leaving bravo (and the top file window contains Name.mc), it is invoked by typing "QS<return>", which causes deletion of the old ".er" file, execution of the ".cm" command file, and re-entry to bravo via the S entry macro. [Iris]<Workstation>MASS>user.cmslice contains the bravo macros.

7. Conditional Code Generation

The "IfEqual","IfGreater", "IfAndZero" and "SkipTo" macros can be used so that MASS will not assemble sections of code. The values of the variables used by the macros can be set on the command line by phrases of the form "[varName,value]" (no spaces). MASS treats the bracketed pair as an argument to the "Set" macro.

8. Builtin Macros:

Add[arg1, ..., arg9]	-- adds up to 9 arguments
Mul[arg1, ..., arg9]	-- multiplies up to 9 arguments
And[arg1, ..., arg9]	-- ands up to 9 arguments
Or[arg1, ..., arg9]	-- ors up to 9 arguments
Xor[arg1, ..., arg9]	-- xors up to 9 arguments
LShift[arg1, arg2]	-- left shift arg1 by arg2
RShift[arg1, arg2]	-- right shift arg1 by arg2
Sub[arg1, arg2]	-- arg1 — arg2

GOTO[label]	
CALL[label]	
GOTOABS[value]	
BRANCH[label0, label1, mask]	-- mask optional
CANCELBR[label, mask]	-- mask optional
DISP2[label, mask]	-- 2-bit dispatch, mask optional
DISP3[label, mask]	-- 3-bit dispatch, mask optional
DISP4[label, mask]	-- 4-bit dispatch, mask optional
DISPNI[label]	-- 8-bit IBDisp dispatch.
RET[label]	

IfEqual[x, y, equalVal, unequalVal]	-- IF x=y THEN equalVal ELSE unequalVal
IfGreater[x, y, equalVal, unequalVal]	-- IF x>y THEN equalVal ELSE unequalVal
IfAndZero[x, y, equalVal, unequalVal]	-- IF x&y=0 THEN equalVal ELSE unequalVal
SkipTo[label]	-- MASS skips code until label! (! must be appended to label)
EXPORT[label1, ..., label9]	-- Declares up to 9 values exportable
IMPORT[label1, ..., label9]	-- Declares up to 9 values importable
at[offset, modulo, label]	-- modulo defaults to 4096, label to current label
Reserve[LowAddr, HighAddr]	-- micro-insts can't be allocated within this range.
Set[varName, value]	-- sets variable named in arg1 to value of arg2; "-" not allowed.
MacroDef[macroName, expansion]	
RegDef[regName, regType, regAddress]	-- regType = {U, R, RH, UY}
PrintVar[varName]	-- print variable in ".er" file
Print[Message]	-- puts "Message" in ".er" file (only letters, numbers, and ".")
SetTask[task]	-- in effect untill the next SetTask. task in [0..7]
StartAddress[label]	-- Burdock initilizes task's TPC to this value.

N. Burdock-Kernel

1. Overview

This section deals with 4 of Burdock's numerous windows: the CP Panel, State Analyzer, Files, and an Empty window. The Empty window can be used to load source files such as ".mc" or ".ft" files. The Files window accepts the file names for the ".fb", ".st", and ".cpr" files. The Analyzer window reads and formats the output of the Tektronics 7904 (DF1 formatter) logic analyzer. The CP Panel reads and writes CP registers and executes CP related commands.

2. CP Panel - Commands

The CP Panel can execute one of the following commands: Boot, Load, Start, Stop, Continue, Break, Unbreak, or LoadReal.

The "Boot" command boots the IOP, if necessary, and then loads the CP kernel (KernelDlion.fb). All the TPC's are initialized to "0FDF", which currently is an microinstruction which loops on itself and resets the display controller. Note that while the kernel is executing at the kernel task level, no other tasks in the Dandelion can run.

"Load" loads the ".fb" file into the control store and reads the ".st" symbol table file into Burdock. The file name come from the Files window. If the original source files contained "SetTask" and "StartAddress" macros, the specified TPC's are initialized.

"Start" loads the emulator (task 0) TPC with the value of the given label and then causes the CP to stop executing the kernel. Normal task scheduling begins, and if multiple tasks are enabled, it is not known which will begin executing first (since the kernel will stop executing on an arbitrary click within a round).

Since the CP-task-specific register which holds condition bits between clicks (the TC register) can not be written, the microinstruction which will be started must contain a "CANCELBR[Label, 0F]", i.e., the address of the second instruction executed must be "at[0F,10]".

"Stop" causes the kernel task to run, thereby blocking all other tasks from running. The tasks which were running are interrupted on a click boundary, so the TPC's and TC's for the tasks remain valid. It is not possible to determine which click boundary the stop occured on.

"Continue" exits the kernel given the current values of the TPC's. Like Start, it is not known which task will begin executing first.

"Break" sets a breakpoint at the address given by the label. Burdock saves the breakpointed microinstruction in an internal table and replaces it with the appropriate instruction which will cause entry into the kernel. If the control store is examined from Burdock, the breakpoints will be visible and not the original instructions. "UnBreak" restores the breakpointed location with its original contents.

There are 4 types of breakpoints: cycle1, cycle2, shortCycle3, and cycle3. Their breakpoint identification numbers are [10..1F], [20..2F], [1..7], and 0, respectively. Thus, there are 16 cycle1, 16 cycle2, and 8 cycle3 breakpoints available. (Burdock can be expanded to accomodate 256 cycle1 and 256 cycle2 breakpoints). The cycle1 and cycle2 breakpoints correctly save all possible pending dispatch/branch bits (i.e., they correctly breakpoint an instruction containing a DISP4[] or RET[]). However, there is only one c3 breakpoint (id_num 0) which has this property. The other seven c3 breakpoints (called shortCycle3 breakpoints) only correctly handle instructions containg at most 2-way BRANCH's, i.e., it may not be possible to correctly resume from a shortCycle3 breakpoint set on a statement which specifies a DISP4, DISP3, DISP2, or RET. (This restriction is necessary in order to save memory data if the click causes a memory read. The kernel could be changed in order to provide more versatile cycle3 breakpoints at the cost of losing memory data.)

Clicks which contain memory reads ("←MD") *can* be correctly restarted after a breakpoint, but cycle2 of clicks which contain "MDR←" can *not* be (since the memory address is lost). Clicks containing memory reads are restarted by loading main memory location 0 (0FFDD instead?) with the data which was arriving from memory during the breakpointed click and then restarting the memory to read from mem[0] when resuming from the breakpoint.

If a cycle1 breakpoint is executed, the kernel will always run in the following task. However, if a cycle2 or cycle3 breakpoint is executed, the breakpointed task must run for at least one more click in order for the breakpoint to take effect. This is never a problem with the emulator (since it will eventually run), but could be a problem with an IO task if the task's request were disabled during the breakpointed click, thereby preventing it from executing again. Note that fo: cycle2 and cycle3 breakpoints it is not known which other tasks may have run between the breakpointed click and the kernel entry.

If "Start" is used instead of "Continue" after a breakpoint, the *second* instruction executed must be "at[0F,10]." This requirement is not necessary if the TC register of the task does not have any non-zero bits where the low 4 bits of the address of the *second* instruction has zero bits. For example, if the TC register is 0, then any microinstruction can be Started. If the TC register is 1, then only microinstructions which are followed by an instruction at an odd address can be started. (The TC register for a task can be written by executing a single microinstruction at the task level which sets the bits accordingly.)

3. CP Panel - Registers

After the CP has been booted, its registers can be read and written via absolute addresses. After a program has been loaded, Burdock can read and write registers given their symbolic names. Burdock ignores capitalization in all names. The control store, TPC, and TC registers can be read or written before the CP is booted.

The format for reading and writing a register with absolute addresses is ".name address" or "address .name". An address without a ".name" is assumed to be a real memory address. Burdock recognizes the following register names:

.name	address range	
.q		
.r	[0..0F]	
.rh	[0..0F]	
.u	[0..0FF]	
.tc	[0..7]	read only
.tpc	[0..7]	
.link	[0..7]	
.ioxin	[0..0F]	address is value of fZ--read only
.ioout	[0..0F]	address is value of fY--write only
.cr0	[0..0FFF]	control store real--first word
.cr1	[0..0FFF]	control store real--second word
.cr2	[0..0FFF]	control store real--third word
.mr	[0..3FFFF]	memory real address
.mv	[0..3FFFFF]	memory virtual address
.map	[0..3FFF]	index into Map
.ib		doesn't effect ibPtr
.ibPtr		can only be set to 1 or 3
.stackP		
.pc16		
.MInt		Mesa Interrupt
.EKErr		error register--read only

Currently, if the display task (1) was executing at the time of a breakpoint, a 2 (or 0) should be written into ".ioout 7" in order to disable the display controller, thereby preventing it from reading the low bank so that the low bank can be read/written correctly by the kernel. (This is also necessary for proper resumption after a cycle2 or cycle3 breakpoint of a click which reads memory.) This problem will be fixed.

4. State Analyzer Window

For CP debugging, the low 12 bits of the state analyzer are connected to the control store address lines (NIA[0..11]'). The upper 4 bits are available for other inputs (such as cycle number, task number, etc). The Analyzer Tool can format, filter, and search the addresses for matches. The "mask" is and'd with all displayed addresses and the "xor" mask is used to invert appropriate bits. The default for the xor mask is 0FFF since the control store address lines are inverted.

The addresses can be displayed in one of 4 modes: NIA (symbolic labels), hex, octal, or binary. Since NIA' is displayed, the occurrence of a label corresponds to the cycle when the instruction is being *read* from the control store--the indicated instruction is executed in the following cycle. Thus, if c1 is being displayed in the top bits, it actually corresponds to microinstructions labeled c2 in the source file.

5. Real Memory Loader

The LoadReal command loads real memory with the ".cpr" file given by the Files window. Such a file contains one or more blocks in the following format:

<count>,<addrHigh>,<addrLow>,<data1>,....<dataCount>.

A program exists (stored on [Iris]<DDavies>MakeBinFile.bcd) which translates text files into this format.

6. Command. files

<Will be documented later>

7. Microcode Error Traps

If a microcode trap occurs to location 0, the kernel will treat it as a breakpoint (id=0FF'x). "Reserve[0, 0]" must be present in your program unless you are going to handle the error. The .EKErr register identifies the microcode trap:

0	Control Store Parity Error
1	Emulator Double-bit Memory Error
2	Stack Overflow or Underflow
3	Emulator Virtual Address Out of Range (>22 bits)

O. Timing Constraints - Allowable Bus Operations:

The following two figures should answer the question: "Can my microinstruction accomplish its task in time (i.e. before the end of the cycle)?" The first table deals with the X-bus, and the following with the Y-bus.

Note that ALL operations which do *not* output onto the X or Y bus (i.e. internal 2901 register to register operations) complete on time (see schematics for timing information). (Note that +1 or -1 do *not* imply use of the X-bus.)

If your microinstruction contains an X-bus operand, then Figure 7 will tell you whether that microinstruction will complete in time. At the intersection of the appropriate "X Source" column and the "X Operation" row, if the number is not in bold type (i.e. any cycle times less than 140 nS), then the microinstruction is OK from a timing viewpoint.

MASS checks for the timing violations given in the following tables. The macro "SuppressTimingWarning" can be used to prevent an error message where the timing is OK for a partial result (for example, addition in the low 4 bits).

The ALU performs arithmetic at 3 different speeds depending on which bits of the result you're looking at. Bits[0-7] are the slowest (they depend on a carry from the lookahead unit; Bits[8-11] are next (they depend on a ripple carry from the low nibble); and Bits[12-15] are fastest (Cin arrivies *very* early relative to Xbus sources). The low nibble always has the timing of a corresponding ALU logic operation. For example "Reg ← Ureg or Reg" has the same timing as "Reg ← Ureg + Reg" for the low 4 bits *only*.

```
Rreg ← Ureg - 1, YDisp, {only low 4 bits OK}                c2;
Rreg ← RHreg + 1, YDisp, {only low 4 bits OK}               c2;
```

			X Source										
		D etup	SU	MD	* RH	* Nibble	* Byte	* IB	* ErrIBStkP	IOIn	A LRotn	(A .or. B) LRotn	(A+B) LRotn
	X ←		75	97	64	50	56	59	59	63	91	102	
	max (59, X←)		75	97	64	59	59	59	59	63	91	102	131 127 111
X Operation	B←X .or. A	40	115	137	104	99	99	99	99	103	131	—	—
	B←X .or. A, ZeroBr	58	133	*155*	122	117	117	117	117	121	*149*	—	—
	B←X .or. A, NegBr	58	133	*155*	122	117	117	117	117	121	*149*	—	—
	[]←X .or.A, YDisp	68	*143*	*165*	132	127	127	127	127	131	*159*	—	—
	B←X .or. A, LShift1	50	125	*147*	114	109	109	109	109	113	—	—	—
	B←X .or. A, LRot1	59	134	*156*	123	118	118	118	118	122	—	—	—
	MAR←X .or. A	78	*153*	—	*142*	137	137	137	137	*141*	—	—	—
	MDR←X .or. A	45	120	—	109	104	104	104	104	108	—	—	—
	SU←X .or. A	87	—	*184*	*151*	*146*	*146*	*146*	*146*	*150*	—	—	—
	IOYOut←X .or. A	64	139	*161*	128	123	123	123	123	127	—	—	—
	B←X + A	74 65 40	*149* *140* 115	*171* *162* 137	133 129 104	133 124 99	133 124 99	133 124 99	133 124 99	137 128 103	*165* *156* 131	—	—
	B←X + A, ZeroBr	95	*170*	*192*	*154*	*154*	*154*	*154*	*154*	*158*	*186*	—	—
	B←X + A, NegBr	87	*162*	*184*	*151*	*146*	*146*	*146*	*146*	*150*	*178*	—	—
	B←X + A, OvBr	90	*165*	*187*	*154*	*149*	*149*	*149*	*149*	*153*	*181*	—	—
	B←X + A, NibCarry	58	133	*155*	122	117	117	117	117	121	*149*	—	—
	B←X + A,PgCarryBr	65	*140*	*162*	129	124	124	124	124	128	—	—	—
	B←X + A,PgCrossBr	77	*152*	*174*	*141*	136	136	136	136	*140*	*168*	—	—
	B←X + A, CarryBr	80	*149*	*171*	*145*	139	139	139	139	*144*	*147*	—	—
	B←X + A, YDisp	68	*143*	*165*	132	127	127	127	127	131	*171*	—	—
	B←X + A, RShift1	89 80 50	*164* *155* 125	*186* *177* *147*	*148* 144 114	*148* 139 109	*148* 139 109	*148* 139 109	*148* 139 109	*152* *143* 113	—	—	—
	B←X + A, RRot1	99 90 60	*174* *165* 135	*196* *187* *157*	*158* *154* 124	*158* *149* 119	*158* *149* 119	*158* *149* 119	*158* *149* 119	*162* *153* 123	—	—	—
	MAR←X + A	– 78 78	*153*	—	*142*	137	137	137	137	*141*	—	—	—
	MDR←X + A	77 70 45	*152* *144* 120	—	136 134 109	136 129 104	136 129 104	136 129 104	136 129 104	135 133 108	—	—	—
	SU←X + A	119 112 87	—	*216* *209* *184*	*178* *176* *151*	*178* *171* *146*	*178* *171* *146*	*178* *171* *146*	*178* *171* *146*	*182* *174* *150*	—	—	—
	IOYOut←X + A LS374	96 89 64	*171* *164* 139	*193* *186* *161*	*155* *153* 128	*155* *148* 123	*155* *148* 123	*155* *148* 123	*155* *148* 123	*159* *152* 127	—	—	—
	IOYOut←X + A S374	80 73 48	*155* *148* 123	*177* *170* *145*	139 137 112	139 132 107	139 132 107	139 132 107	139 132 107	*143* 136 111	—	—	—
	[] ← X, XDisp	32	107	129	96	91	91	91	91	94	123	134	*163* *159* *143*
	RH ← X	36	111	133	100	95	95	95	95	99	127	138	*167* *163* *147*
	IB ← X	37	112	134	101	96	96	96	96	100	128	139	*168* *164* *148*
	IOXOut←X (LS374)	22	97	117	84	79	79	79	79	83	111	122	*151* *147* 131

* Timing for bits[0-7] of these sources is that of Nibble. stackP← has timing of the slow IOYOut.
The 3 numbers for arithmetic operations correspond to bits[0-7], bits[8-11], & bits[12 15], respectively.

Figure 7. Allowable X-bus Operations

9 Feb 80
Filed on <Workstation>LII>AllowableXbus.sil
<Workstation>LII>DMR.press

			Y Source		
		setup	A .or. B	A (bypass)	A + B
	Y ←		80	69	109 105 83
Y Operation	MAR ← *	36 11 36	116 91 116	105 80 105	114 116 119
	MDR ←	3	83	72	112 108 86
	SU ←	45	125	114	*154* *150* 128
	stackP ←	6	86	75	115 112 89
	[]← , YDisp	32	112	101	121
	Uaddr[4-7] ←	15	95	84	124 120 98
	IOYOut ← (S374)	6	86	75	115 112 89

* Bits[0-7] have timing of Y ← (B .or. 0), except in the A bypass case.

The 3 numbers for arithmetic operations correspond to bits[0-7], bits[8-11], & bits[12-15], respectively.

Figure 8. Allowable Y-bus Operations

9 Feb 80
Filed on <Workstation>LH>AllowableYbus.sil
<Workstation>LH>DMR.press

Appendix: Antithetical List of MicroInstructions

This appendix contains a list of some example microinstructions in addition to examples of illegal ones. Those microcode statements which can be written but don't work as intended have three possible reasons for their shortcomings:

1. Timing error,
2. Syntax error, or
3. Characteristic of a processor data path.

```
SU :: = STK | U register;
A :: = the R register addressed by rA;
B :: = the R register addressed by rB;
R :: = an R register addressed by either rA or rB;
Rot1 :: = LRot1 | RRot1;
Shift1 :: = LShift1 | RShift1;
LRotn :: = LRot0 | LRot4 | LRot8 | LRot12;
constant :: = Nibble | Byte;
ArithBr :: = NegBr | ZeroBr | OvBr | CarryBr | PgCarryBr;
LogicBr :: = NegBr | ZeroBr;
XDispBr :: = XpcDisp | XhDisp | XwdDisp | XlDisp | XDisp;
° :: = alu logic operation (R or S, R and S, ~R and S, R xor S, ~R xor S);
± :: = alu arithmetic operation (R + S, S - R, R - S);
⊕ :: = alu arithmetic or logic operation;
```

General:

Possible:

```
B ← R ° R,
B ← R ± R,
Q ← R ⊕ R,
YBus ← A, B ← R ⊕ R,
```

Not Possible:

```
B ← Q ← R ⊕ R,                  (2)
YBus ← A, [] ← R ⊕ R,           (2)
YBus ← A, Q ← R ⊕ R,            (2)
```

SU registers:

Possible:

```
B ← SU ° A,
B ← SU ° Q,
SU ← R ° R,
B ← SU ← R ° R,
SU ← A, B ← R ° R,
SU ← A, B ← R + R + 1,
SU ← A, B ← R - R,


SU ← A, B ← R ± R, Cin←pc17,
```

Not Possible:

```
B ← SU ± A,                     (1)

SU ← R ± R,                     (1)


SU ← A, B ← R + R, Cin←0,       (2)
SU ← A, B ← R - R - 1,          (2)
SU ← A, B ← SU,                 (2) & (3)
SU ← SU ⊕ A,                    (2) & (3)
```

Constants:

Possible:

```
B ← −constant,
B ← A ⊕ constant,
B ← 0,
{B ← 100,}          B ← 0FF + 1
{B ← 0FF00,}        B ← ~0FF
{B ← 0FFFF,}        B ← ~B xor B
{B ← 7FFF,}         B ← RShift1 (~B xor B)
{B ← 1FF,}          B ← LShift1 0FF, SE←1


SU ← 0
{SU ← 0FFFF,}       SU ← ~A xor A
```

Not Possible:

```
STK ← constant,              (1)
U ← constant,                (2)
B ← SU ⊕ constant,           (2) & (3)
```

RH registers:

Possible:

```
B ← RH[B] ∘ A,
B ← RH[B] ± A,
RH[B] ← SU,
RH[B] ← SU, B ← SU ∘ A,
RH[B] ← SU, B ← R ⊕ R,
RH[B] ← constant,
RH[B] ← constant, B ← R ⊕ R,
RH[B] ← ib,
RH[B] ← ib, B ← R ⊕ R,
STK ← A, B ← RH[B] ⊕ A,
```

Not Possible:

```
A ← RH[B],
RH[B] ← RH[B]                (2)


U ← A, B ← RH[B] ⊕ A,        (2)
```

Memory:

Possible:

```
R ← MD,
R ← R ∘ MD,
Q ← R ∘ MD,
RH[B] ← MD,
RH[B] ← MD, B ← R ⊕ R,
RH[B] ← MD, B ← MD, SU ← A,
MAR ← [rh[B], A], B ← B ⊕ R,
MAR ← [rh[B], A], B ← IOIn,
MAR ← [rh[B], A], B ← RH[B],
MAR ← [rh[B], A], B ← SU,
MAR ← [rh[B], A], B ← constant,
MAR ← [rh[B], B ⊕ R],
MAR ← B ← [rh[B], B ⊕ R],
Map ← Q ← [rh[B], R ⊕ R],
MAR ← [rh[B], constant],
MDR ← R ⊕ R,
MDR ← A, B ← R ⊕ R,
MDR ← SU ∘ A,
MDR ← RH[B],
```

Not Possible:

```
R ← R ± MD,                  (1)
R ← MD Shift,                (1)
SU ← MD,                     (1)


MAR ← [rh[B], SU], (1)


MDR ← SU ⊕ A, {OK in bits[12-15]}   (1)
```

IOIn/IOOut:

Possible:	*Not Possible:*	
B ← IOIn ⊕ A,		
RH[B] ← IOIn,		
MDR ← IOIn,		
RH[B] ← IOIn, B ← IOIn ⊕ A,		
	SU ← (IOIn ⊕ A) LRotn,	(1)
IOOut ← (R ∘ R) LRotn,	IOOut ← (R ± R) LRotn,	(1)
IOOut ← Q LRotn,		
IOOut ← MD,		
IOOut ← IOIn,		
IOOut ← RH[B],		
IOOut ← SU,		
IOOut ← A LRotn,		
IOOut ← ib,		

stackP:

Possible:	*Not Possible:*	
stackP ← R ⊕ R,		
stackP ← stackP ⊕ A,	stackP ← stackP + constant,	(2) & (3)
stackP ← Nibble,	stackP ← MD,	(1)
stackP ← IOIn,		
stackP ← RH[B],		
RH[B] ← stackP,		

LRotn:

Possible:	*Not Possible:*	
B ← A LRotn, {A bypass}	B ← (R ⊕ R) LRotn,	(2) & (3)
[] ← (A ∘ B) LRotn,	Q ← A LRotn,	(2) & (3)
[] ← Q LRotn,	B ← Q LRotn,	(2) & (3)
B ← A ∘ (A LRotn),	B ← B ⊕ (A LRotn),	(2)
	SU ← (R ⊕ R) LRotn,	(2) & (3)
	SU ← A LRotn,	(2) & (3)
	MDR ← A LRotn,	(2) & (3)
	B ← SU LRotn,	(2) & (3)
	B ← (A LRotn) Rot1,	(2) & (3)
	B ← (R ⊕ R LRotn) Shift1,	(2) & (3)
	RH[B] ← A LRotn,	
RH[B] ← A LRotn, B ← R ⊕ R,		
RH[B] ← (R ∘ R) LRotn,	RH[B] ← (R ± R) LRotn, {OK in [12-15]}	(1)
	RH[B] ← (RH[B] ⊕ A) LRotn,	(2) & (3)
STK ← A, RH[B] ← A LRotn, B ← R + R + 1,		
STK ← A, RH[B] ← (R ∘ R) LRotn,	U ← A, RH[B] ← (R ∘ R) LRotn,	(2)

Single Bit Shifting:

Possible:	*Not Possible:*	
B ← (R ° R) Shift1,	Q ← (R ° R) Shift1,	(2)
B ← (R ° R) Rot1,	Q ← (R ° R) Rot1,	(2)
	B ← (R ± R) Shift1,	(1)
	B ← (R ± R) Rot1,	(1)
	YBus ← A, B ← R Shift1,	(2)
	[] ← R Shift1,	(2)
B ← (R + R) Shift1, SE←0,	B ← (R + R + 1) Shift1, SE←0	(2)
B ← (R - R - 1) Shift1, SE←0,	B ← (R - R) Shift1, SE←0	(2)
B ← (SU ° A) Shift1, SE←0,	B ← (SU ° A) Shift1, SE←1,	(2)
	SU ← B Shift1,	(2)
	SU ← A, B ← R Shift,	(2)
B ← (constant ± A) Shift1, {only [8-15]}	B ← (constant ± A) Rot1 {OK in [12-15]}	
B ← R ± R, DRShift1,		
B ← R ± R, DLShift1,		

Branching:

Possible:	*Not Possible:*	
B ← R ⊕ R, Branch,		
YBus ← A, B ← R ⊕ R, Branch,		
B ← Xbus ° A, LogicBr,	B ← MD ° A, LogicBr,	(1)
B ← Xbus ± A, CarryBr, {except for SU, MD}	B ← Xbus ± A, ZeroBr,	(1)
B ← Xbus ± A, PgCarryBr, {except for SU, MD}	B ← Xbus ± A, NegBr,	(1)
B ← R ± 1, ArithBr,		
B ← R ⊕ R, YDisp,		
B ← RH[B] ° A, YDisp,	B ← SU ° A, YDisp,	(1)
	B ← MD ° A, YDisp,	(1)
YBus ← A, B ← R ⊕ R, YDisp,		
[] ← SU, XDispBr,		
[] ← IOIn, XDispBr,		
[] ← constant, XDispBr,		
[] ← ib, XDispBr,		
[] ← RH[B], XDispBr,		
[] ← stackP, XDispBr,		
[] ← MD, XDispBr,		
B ← A LRotn, XDispBr,		
B ← A ° (A LRotn), XDispBr,		
[] ← (R ° R) LRotn, XDispBr,	[] ← (R ± R) LRotn, XDispBr, {bits[12 = -15 OK}	(1)