MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo 528

July 21, 1980

CADR

by
Thomas F. Knight, Jr.
David Moon
Jack Holloway
and Guy L. Steele, Jr.

竹岡尚三 訳
(日本語版
2010/MAR/20 ドラフト 1
2010/MAY/20 日本語訳 1 版)

要約

CADR マシン、CONS マシンの更新版、は、汎用で、Lisp マシン・システムの基礎である 32bit のマイクロプログラム可能なプロセッサであり、Laboratory (※脚注:AI ラボ) で開発中の、Lisp の高性能で、経済的な実現である新しい計算機システムである。この文書は、CADR プロセッサと、関連あるハードウェアと、低レベルのソフトウェアについて記述する。

このレポートは MIT の AI ラボで行われた研究を記述している。ラボの AI 研究のサポートは、"Office of Naval Research contract N00014-75-C-00643" の DoD (Department of Defense) の ARPA (Advanced Research Projects Agency) の一部として提供されているものである。.

目次

Overview 概要

CADR は、複雑な命令コード、特にスタックとポインタに関する操作、のエミュレーションを容易にするように設計された汎用プロセッサである。
それは、Lisp マシンプロジェクトの中央プロセッサであり、Lisp マシン・コンパイラによって生成される、ビット効率の良い 16bit 命令コードを解釈(インタープリット)する(用語「Lisp マシン」と「CADR マシン」は時々、混同される。この文書では、「CARD マシン」は特定のマイクロ・プロセッサの設計であり、よって Lisp マシンは、Lisp マシン命令コードを解釈するマイクロコードを CADR に加えたものである。)

CADR のデータパスは 32bit 幅である。48bit 幅の各マイクロコード命令は、様々な内部スクラッチ・レジスタからの、2 つの 32bit データ・ソースを指定する。
2 つのデータ操作命令は、デスティネーション(あて先)・アドレスも指定する。内部スクラッチパッドは、1K のポインタでアドレス可能な RAM を含み、それは、キャッシュと同様の作法で、エミュレートされたスタックのトップを格納することを、想定している。Lisp マシンでは、メインメモリの参照の多くのパーセンテージが、スタックへの参照であろうから、この部分は、機械を加速する。

CADR マシンは、伝統的なプロセッサと非常によく似た動きをする 14bit のマイクロ・プログラムカウンタを持ち、16k の書換え可能なマイクロプログラム・メモリが使える。また、32 個のマイクロコード・サブルーチン・スタックを含んでいる。

メモリは、24bit 仮想アドレスを 22bit の物理アドレスにマップする、2 階層の仮想ページング・システムを通しアクセスされる。

4 クラスのマイクロ・命令がある。それぞれ 2 ソース(A と M)を指定し; ALU とバイト操作はデスティネーション(A か、M に加えて機能的デスティネーション)も指定する。
A バスは 1024 ワードのスクラッチパッド・メモリからのデータを供給し、M バスは 32 ワードの M スクラッチパッド・メモリ(A スクラッチパッドの最初の 32 個の場所のコピー)か、または、他の様々な内部レジスタからのデータを供給する。
4 つのクラスの命令は:

ALU
デスティネーションは、2 つのソースのブーリアンか算術演算の結果を受け取る。

BYTE
デスティネーションは、バイトの抽出、バイト生みつけ、1 つのソースから他へのフィールドの選択的置き換え、を受け取る。バイトは 0 でない幅でしか操作できない。

JUMP
M バスでアクセス可能などれかのビットの値による条件、か、ALU や割込みペンディングやページ・フォールトのような他の内部の様々な条件により、制御を移す。

DISPATCH
M バスから抽出された 7bit までのバイトによって選択されたディスパッチ・メモリ、それからのワードによって決定される場所へ制御を移す。

ロードとそれの使用がマイクロプロセッサに特殊な動作を起動するいくつかのソースとデスティネーションが存在する。これらには、メモリ・アドレスとメモリ・データ・レジスタを含み、それらの使用はメイン・メモリ・サイクルを初

期化する。

ALU 操作のいくつかは条件付きであり、Q レジスタの下位ビットと A ソースの符号に依存している。これらの操作は乗算と除算のステップで使用される。

この機械が Lisp マシン命令コードのインタープリットするのを適切にしている主な特徴は、動的に書換え可能なマイクロコード、柔軟なディスパッチとサブルーチン呼び出し、素晴らしいバイト操作能力、内部スタック記憶、である。
CADR の設計は Lisp マシン設計の要求に強く影響されたので、極端に特殊用途になるような特徴は意識的に避けるようにした。

この機械のゴールは、Lisp マシンの特別な命令コードのインタープリットにおいて良くなる事柄を見出すことであるし、また、他のものもほとんど同じぐらい良くインタープリットするぐらい十分に汎用であるというものである。特に、Lisp マシン設計のクリティカルでない部分 (Lisp マシン命令形式のような) は、布線"wired in"された; このように、Lisp マシン設計へのどんな変更も、CADR によって容易に受け入れられた。
しかしながら、いくつかの素晴らしいハック ("efficiency hacks") がハードウェアの中にある、他のマイクロコードでは役に立たないような、Lisp マシン・マイクロコードのいくつかの共通操作を高速化するような設計。この文書の後の節でそれは述べる。

Notational Conventions 記述慣習

ビット位置、フィールド幅、メモリ・サイズなどを記述するすべての数は十進である。八進数は、フィールドの「値」を記述することのみに(そして排他的に)使用される。ワード中のビットは、一貫して右から左に番号付けされている、LSB(最下位ビット)は ビット<0> である。フィールドは最上位と最下位ビットで記述される(例 "ビット <22-10>")。

不正("illegal")と書かれる特別なフィールドの値は、いつであっても、その値を指定することは機械の運用を台無しにするであろうことで、意味が無い。単に値はある機能を起こすことを示し、それが即座に便利であると思われないものでも、機械の内部での働きが、他の理由によって、いくつかのセレクタをその値へ決めているだろう。そして、通常、まったく便利ではないが、ユーザはこの値を選ぶことができる。イリーガル値の記述は、内側の動作を推測しようとする誰かの役に立つだろう。

非使用("unused")と書かれたフィールド値は設計が拡張可能なように予約されており、プログラムによって使用されてはならない。非使用("unused")と書かれたビット・フィールドは、プログラム中では、0 でなければならない、将来の互換性のために。

レジスタと命令の参照での"micro"という術語の使用は冗長になるから、この文書のこの部分では、それの使用を省略する。議論されているすべての命令はマイクロ命令である。

以下のビットは、各命令で同じに扱われる。それらは、各命令記述ごとに繰り返さない。

IR<48> = 奇数パリティ・ビット

IR<47> = Unused 不使用

IR<46> = 統計(統計カウンタの記述を参照) これは、マイクロコードの指定した領域を何回実行したかをカウントすることや、マイクロコード・ブレークポイントを実現するためや、ある回数"time"で機械を止めるのに、使用する。

IR<45> = ILONG (1 で遅いクロック)

IR<44-43> = オペコード(Opcode) (0 ALU, 1 JUMP, 2 DISPATCH, 3 BYTE)

IR<42> = POPJ 遷移。マイクロ・サブルーチンからのリターンを起こす。場合によっては、加えて一つの命令を実行後に。

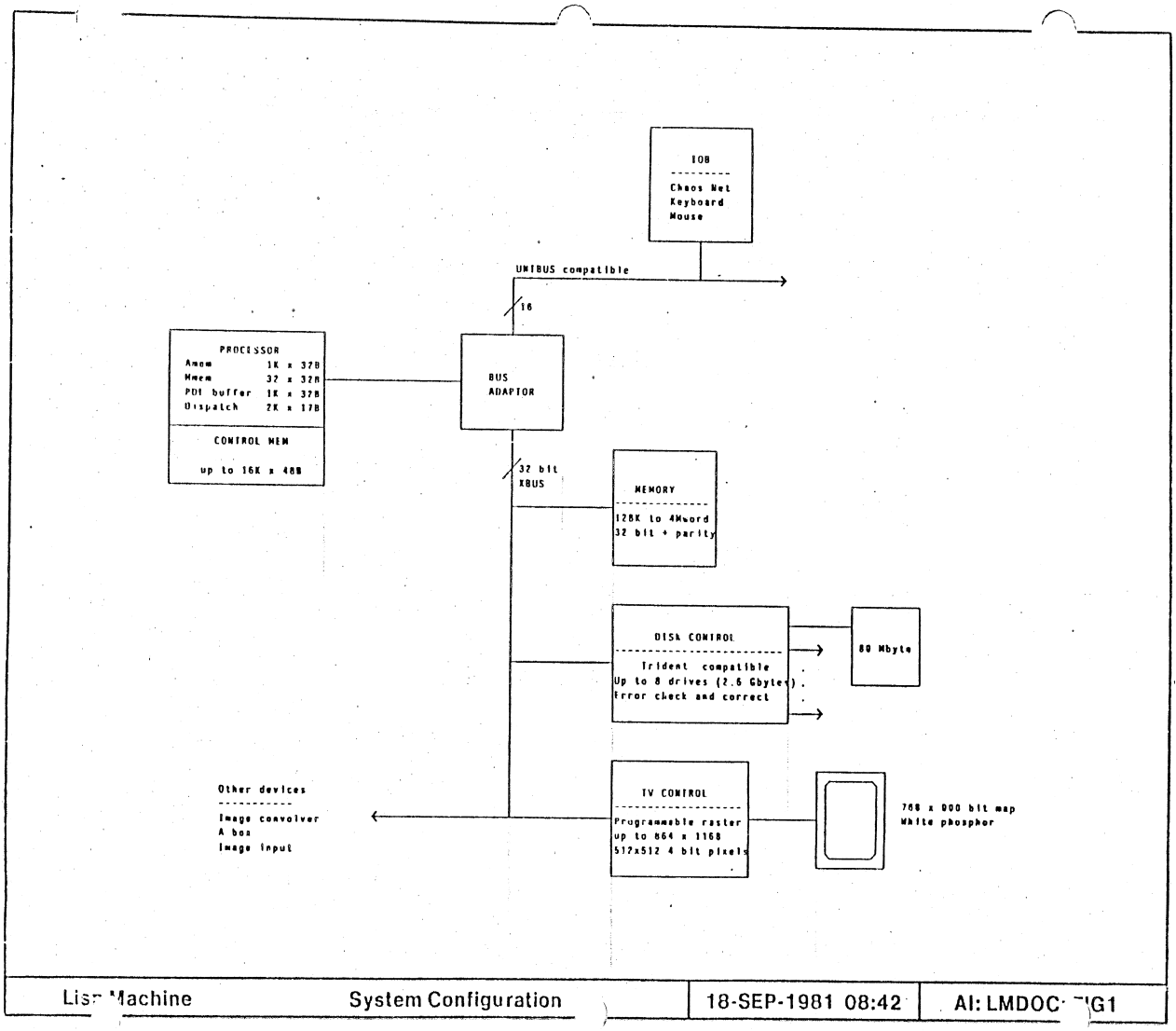
IR<11-10> = その他の機能

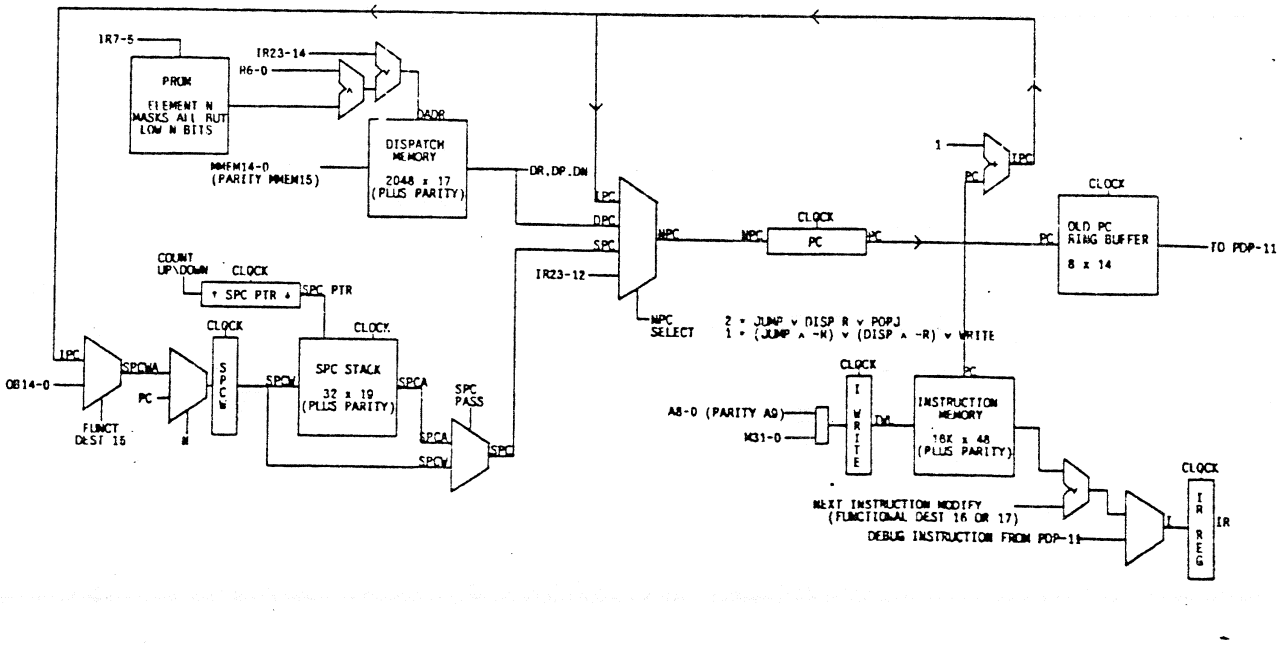
0 通常

1 不使用

2 ディスパッチ・メモリへの書き込み、もし opcode が DISPATCH であるなら。

3 ロケーション・カウンタ(LC)による M-ROTATE フィールドの変更を許可。命令ストリーム(instruction-stream)・ハードウェアの説明を参照。

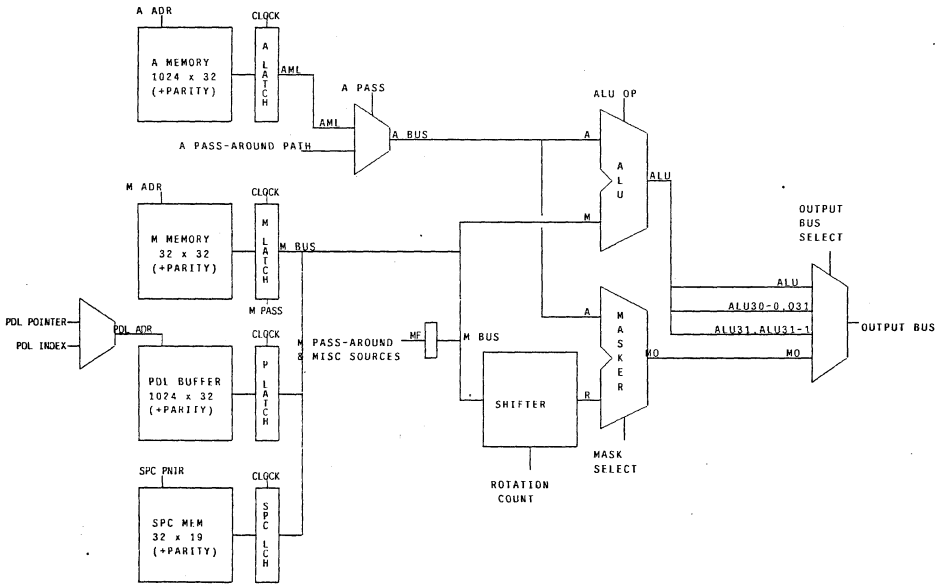

IOB
Chaos Net
Keyboard
Mouse
UNIBUS compatible
16
PROCESSOR
Amem 1K x 32B
Mmem 32 x 32B
PDI buffer 1K x 32B
Dispatch 2K x 17B
CONTROL MEM
up to 16K x 48B
BUS ADAPTOR
32 bit XBUS
MEMORY
128K to 4Mword
32 bit + parity
DISK CONTROL
Trident compatible
Up to 8 drives (2.6 Gbyte)
Error check and correct
80 Mbyte
Other devices
Image convolver
A bus
Image input
TV CONTROL
Programmable raster
up to 864 x 1168
512x512 4 bit pixels
768 x 900 bit map
White phosphor
Lisp Machine
System Configuration
18-SEP-1981 08:42
AI: LMDOC;FIG1



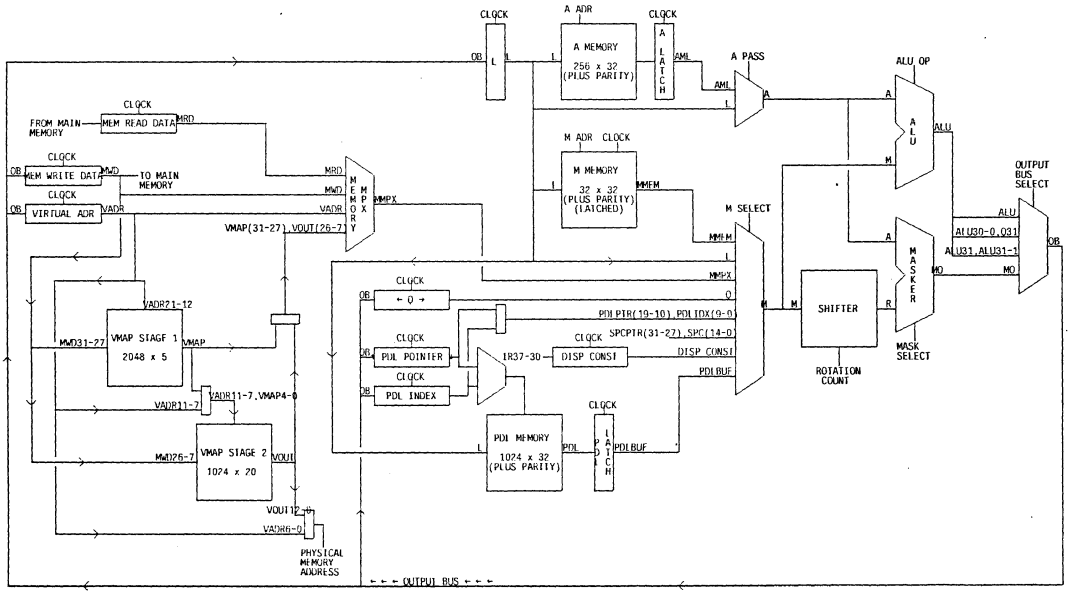
CADR Main Control Paths

Data Paths データパス

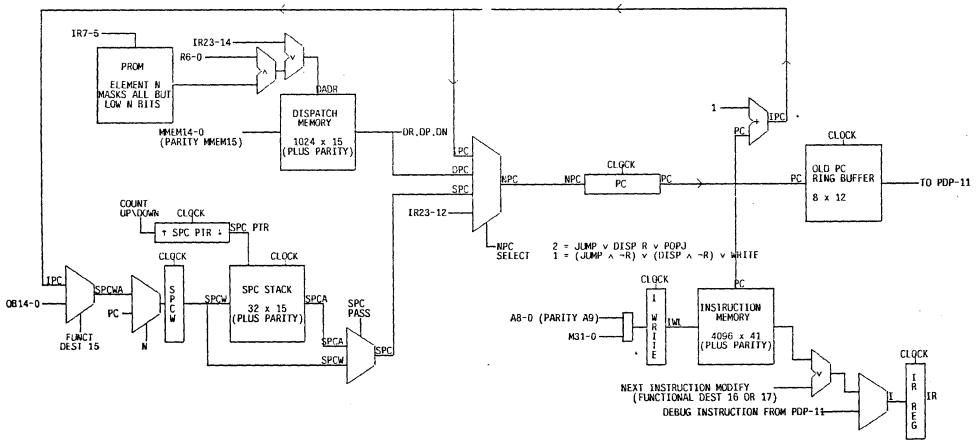
マシンのデータパスは、ALU とバイト抽出機 (byte extractor) へデータを供給する 2 つのソースバス A と M と、ALU (オプションとして左か右のシフト結果) からか、バイト抽出機からの出力を選択した、一つの出力バス OB、そしてその出力データは様々なデスティネーションに経路付けできる、から成っている。ここでは最初に、すべての命令で同様に指定されるソースバス； そして、結果がどこに格納されるかの制御を指定するデスティネーション；最後に、ALU とバイト抽出機を制御する 2 つの命令の仕様を説明する。



CONS	A AND M BUSSES	17-OCT-1979 08:21	AI:LMDOC; CHODAM



CONS	MAIN DATA PATHS	17-OCT-1979 08:20	AI:LMDOC; CHOD1



CONS	MAIN CONTROL PATHS	17-OCT-1979 08:21	AI:LMDOC; CHODI

Sources ソース

全命令でソースは同じ方法で指定される。マシンの中には A バスと M バス、2 つのソース・バスがある。A バスは 32bit で 1024 ワードのスクラッチパッド・メモリからのみドライブされる。M バスは、32bit で 32 ワードの M スクラッチパッドと他の様々なソース、メインメモリ・データと制御レジスタ， PC スタック（トラップ後のプロセッサ・ステータス（状態）の復帰のための）， 内部スタック・バッファとそのポインタ・レジスタ， マイクロコード・ロケーション・カウンタ， Q レジスタからドライブされる。A と M スクラッチパッドのためのアドレスは命令から直接取られる。M ソースのためのデータのもう一つのソースは、M ソース・フィールド中の付加（アディショナル・）ビットで指定される。

```
IR<41-32> = A ソース・アドレス
IR<31-26> = M ソース・アドレス
     If IR<31> = 0,
         IR<30-26> = M スクラッチパッド・アドレス
     If IR<31> = 1,
         IR<30-26> = M "機能(functional)"ソース
               0 ディスパッチ定数（下を参照）
               1 SPC ポインタ<28-24>， SPC データ<18-0>
               2 PDL ポインタ <9-0>
               3 PDL インデックス <9-0>
               5 PDL バッファ（インデックスでアドレスされる）
               6 OPC レジスタ（下を参照） <13-0>
               7 Q レジスタ
              10 VMA レジスタ（メモリ・アドレス）
              11 MAP[MD]
              12 MD レジスタ（メモリ・データ）
              13 LC （ロケーション・カウンタ）
              14 SPC ポインタとデータ， pop
              24 PDL バッファ,ポインタから指される， pop
              25 PDL バッファ,ポインタから指される
```

上のリストに載っていない機能ソースは使用してはならないし、副作用があるだろう。ソース 15,16,17 は将来の拡張のために予約済み。ソース 4 は PDL インデックスで指される PDL バッファ、PDL ポインタはデクリメントされる、おそらく役に立たない操作である。

プログラミング・ヒント： ひとつの A メモリ・ワードとひとつの M メモリ・ワードを予約し、そこに定数 0 を入れて、各ソース・バスに 0 を供給できるようにすることは、しばしば便利である。また、全部が 1 になった M メモリ・ワードを一つ持つことも便利だ。これらは、バイト抽出、マスク、ビット・セット、ビット・クリアの操作で特に役立つ。実際のところ、CONSLP アセンブラは、A メモリ番地 2 と M メモリ番地 2 が 0 のソースであると想定している。UCONS マイクロコードは、番地 3 にオール 1 を格納している。

M スクラッチパッドは通常、A スクラッチパッドの最初の 32 個の番地の複製コピーを保持している。この効果は、あたかも、最初の 32 番地がデュアル・ポート化（※訳注：同時に 2 つ読み出しできる）されている一つのスクラッチパッド・メモリがあるのと同様である。これは、ALU とシフタの両側からこれらのロケーションをアクセス可能にする

ことで、プログラミングをより手軽にしている。

Destinations デスティネーション

BYTE と ALU 命令中の 12bit のデスティネーション・フィールドは、命令の結果がどこに行くかを指定する。最上位ビットによる 2 つの形式のうちの 1 つに、それはある。もし最上位ビットが 1 なら、下位 10bit は A メモリ番地のアドレスで、残りのビットは不使用。もし最上位ビットが 0 なら、下位 10bit は 5bit の機能デスティネーション・フィールドと、5bit の M スクラッチパッド・アドレスに分割され、これらのフィールドで指定される 2 つの場所は"両方とも"書き込まれる。デスティネーション・フィールド中の「次が最上位ビット」(next-to-highest bit)は、不使用。

```
IR<25-14> = デスティネーション
    If IR<25> = 1,
        IR<23-14> = A スクラッチパッド・アドレス
    If IR<25> = 0,
        IR<23-19> = 機能デスティネーション・ライト・アドレス
            0  None
            1  LC (ロケーション・カウンタ)
            2  割込み制御 <29-26>
               Bit 26 = シーケンス-ブレーク要求
               Bit 27 = 割込み-イネーブル
               Bit 28 = バス-リセット
               Bit 29 = LC バイト-モード
           10  PDL (ポインタによりアドレスされる)
           11  PDL (ポインタによりアドレスされる),push
           12  PDL (Index によりアドレスされる)
           13  PDL インデックス
           14  PDL ポインタ
           15  SPC データ,push
           16  次命令修飾
               ("OA register"), bits <25-0>
           17  次命令修飾
               ("OA register"), bits <47-26>
           20  VMA レジスタ (メモリ・アドレス)
           21  VMA レジスタ, メイン・メモリ・リード開始
           22  VMA レジスタ, メイン・メモリ・ライト開始
           23  VMA レジスタ, ライト・マップ
               マップは MD からアドレスされ、VMA から書かれる。
               VMA<26>=1 なら VMA<31-27>から level 1 map を書く
               VMA<25>=1 なら VMA<23-0> から level 2 map を書く
           30  MD レジスタ(メモリ・データ)
           31  MD レジスタ, メイン・メモリ・リード開始
           32  MD レジスタ, メイン・メモリ・ライト開始
           33  MD レジスタ, 23 と同様に map を書く
        IR<18-14> = M スクラッチパッド書き込みアドレス
```

リストに載っていない機能デスティネーションは、おかしな結果を出すだろう。
デスティネーション 3-7 は拡張のために予約済み。

注意：もし M メモリへの書き込むならば、マシンは、対応する A メモリ・アドレスへも書き込むだろう。したがって、あなたは、A メモリの 0-37 番地へ絶対に書き込んではいけない； この方式は、A メモリの最初の 40(八進)個の場所は M メモリに"マップされて"いるから。

機能デスティネーションのより立ち入った完全な詳細は後の節に記述されている。Q レジスタは、機能デスティネーションを使用するのではなく、ALU 命令の Q 制御フィールドを使用してロードされる。加えて、それは、出力バスではなく、ALU からの出力からロードされる。これが意味することは、左と右シフト操作は Q へロードされるデータには効果が無いということである。

プログラミング・ヒント： もし機能デスティネーションが指定されたら、M スクラッチパッドの場所も必ず指定しなければならない。"ゴミ garbage"のために、M スクラッチパッドに一つの場所を予約しておくと便利である；機能デスティネーションに書き込む必要があり、しかし M スクラッチパッドのどの場所にも書くべきでないときに、この場所を指定できる。CONSLP アセンブラはデフォールト値で M ライト・アドレスを 0 にするから、0 をゴミの場所にするのが、ベストである。A スクラッチパッドの番地 0 は同様にゴミ番地として予約され、書き込まれるだろう。

The ALU Instruction ALU 命令

ALU 操作は機械の中の算術計算のほとんどを実行する。32bit 数値の 2 つのソースと、ALU によって実行される操作を指定する。操作は、2 変数の 16 ブーリアン関数のどれでもが可能であり、2 の補数の加減算、左シフトと、いくつかのあまり役に立たない操作。ALU へのキャリーは、強制的に 0 か 1 を指定できる。ALU の出力は、オプションで 1 シフトでき、出力バスを通して、指定したデスティネーションに書き込める。加えて、ALU 命令は、Q レジスタに対する 4 つの操作のうちの一つを指定できる。それは、何もしない、左シフト、右シフト、ALU 出力からのロードである。
ALU 操作フィールド中の付加ビット(additional bit)は、条件操作を示すようにデコードされる;これは"乗算ステップ(multiply step)"と"除算ステップ(divide step)"操作を指定する方法である。(乗算と除算は別な節で非常に詳しく解説される)

IR<44-43> = 0 (ALU オペコード)
IR<41-32> = A ソース
IR<31-26> = M ソース
IR<25-14> = デスティネーション
IR<13-12> = 出力バス制御
0 バイト抽出機出力(不正)
1 ALU 出力
2 ALU 出力右 1 シフト,正しい符号シフト・イン付き,オーバフローは気にしない。
3 ALU 出力左 1 シフト,Q<31>が右からシフト・イン。
IR<9> = 不使用
IR<8-4> = ALU 操作
If IR<8> = 0,
IR<7-3> = ALU オペコード(表を参照)
If IR<8> = 1,
IR<7-3> = 条件 ALU オペコード
0 乗算ステップ
1 除算ステップ
5 剰余補正
11 除算ステップ初期化
IR<2> = ALU の下位からキャリーを入れる
IR<1-0> = Q 制御
0 何もしない
1 Q 左シフト,ALU 出力の符号(ALU<31>)の”反転”をシフトインする。
2 Q 右シフト,ALU 出力の下位ビット(ALU<0>)をシフトインする。
3 ALU 出力から Q をロード

ALU 操作コード(74181 仕様の表 1 から)。全部の算術演算は 2 の補数である。注意として、論理操作が、Lisp BOOLE 関数によって使われるのと同じオペコードになるように、ビットは並べられている。
角括弧([])の中の名前は、操作の CONSLP ニーモニックである。

	Boolean (IR<7>=1)		Arithmetic (IR<7>=0)			
IR<6-3>			Carry in = 0		Carry in = 1	
0	ZEROS	[SETZ]	-1		0	
1	M&A	[AND]	(M&A)-1		M&A	
2	M&~A	[ANDCA]	(M&~A)-1		(M&~A)	
3	M	[SETM]	M-1		M	
4	~M&A	[ANDCM]	M|~A		(M|~A)+1	
5	A	[SETA]	(M|~A)+(M&A)		(M|~A)+(M&A)+1	
6	M*A	[XOR]	M-A-1	[M-A-1]	M-A	[SUB]
7	M|A	[IOR]	(M|~A)+M		(M|~A)+M+1	
10	~A&~M	[ANDCB]	M|A		(M|A)+1	
11	M=A	[EQV]	M+A	[ADD]	M+A+1	[M+A+1]
12	~A	[SETCA]	(M|A)+(M&~A)		(M|A)+(M&~A)+1	
13	M|~A	[ORCA]	(M|A)+M		(M|A)+M+1	
14	~M	[SETCM]	M		M+1	[M+1]
15	~M|A	[ORCM]	M+(M&A)		M+(M&A)+1	
16	~M|~A	[ORCB]	M+(M|~A)		M+(M|~A)+1	
17	ONES	[SETO]	M+M	[M+M]	M+M+1	[M+M+1]

The BYTE Instruction　BYTE 命令

BYTE 命令は ALU 命令と同様に、2 つのソースと一つのデスティネーションを指定する。ただし、操作は、M ソースからのバイトフィールドを、A ソースからのワードの同じ長さのフィールドへの選択的挿入のひとつを実行する。M ソースのローテートは、0 か、または ROTATE フィールドの内容と同じであるかを、SR ビットによって指定される。置換されるビットを選ぶために使用されるマスクのローテートは、0 か、または ROTATE フィールドの内容と同じであるかを、MR ビットによって指定される。置換に使用されるマスク・フィールドの長さは LENGTH MINUS 1 フィールド中で指定される。
SR と MR ビットの 4 つの状態は次の操作になる:

MR=0 SR=0 利用可能ではない（他のモードのサブセットである）

MR=0 SR=1 LOAD BYTE
PDP-10 LDB 命令（マスクされていないビットは、A ソースから来ることを期待する）。M ソースからの任意の位置のバイトが、出力では右詰めされている。

MR=1 SR=0 SELECTIVE DEPOSIT
M ソースからのマスクされたフィールドは、A ソースからのワード中の同じ長さと位置のバイトを置き換えるために使用される。

MR=1 SR=1 DEPOSIT BYTE
PDP-10 DPB 命令。M ソースからの右詰されたバイトは、A ソースからのワードの任意の位置のバイトの置き換えに使用される。

BYTE 命令は、出力バス・セレクト・コードに 0（バイト抽出機出力）を強制的に指定することで、自動的にバイト抽出機の出力を利用可能にする。

IR<44-43> = 3 （バイト操作）
IR<41-32> = A ソース
IR<31-26> = M ソース
IR<25-14> = デスティネーション
IR<13>　= MR = マスク・ローテート（上を参照）
IR<12>　= SR = ソース・ローテート（上を参照）
IR<9-5>　= length minus 1 ,バイトの長さ-1 （0 は長さ 1 のバイトを意味する,など）
IR<4-0>　= マスク や/かつ M ソースの（左への）ローテート・カウント

バイト操作は、M ソースを、0（もし SR=0 なら）か、ローテーション・カウント（もし SR=1 なら）によってローテートし、R と呼ばれる結果を生成する。それは、加えて MR ビット、ローテーション・カウント、length minus 1 フィールドを使用して、セレクタ・マスク（下の説明を参照）を生成する。このマスクは A ソースと R をマージするのに使用され、もしマスクが 0 なら A からの、もしマスクが 1 なら R からのビット、という選択をビットごとに行う。そして、この結果は指定されたデスティネーション（複数の場合アリ）に書き込まれる。

マスク・メモリの出力：

右マスク・メモリは 0(MR=0) かローテーション・カウント(MR=1)によってインデックスされる。

左マスク・メモリは、

((右マスク・メモリへのインデックス) + (length_minus_1 フィールド))mod 32

によってインデックスされる。

八進 インデックス	左マスク・メモリ内容	右マスク・メモリ内容
0	00000000000000000000000000000001	11111111111111111111111111111111
1	00000000000000000000000000000011	11111111111111111111111111111110
2	00000000000000000000000000000111	11111111111111111111111111111100
3	00000000000000000000000000001111	11111111111111111111111111111000
4	00000000000000000000000000011111	11111111111111111111111111110000
5	00000000000000000000000000111111	11111111111111111111111111100000
6	00000000000000000000000001111111	11111111111111111111111111000000
7	00000000000000000000000011111111	11111111111111111111111110000000
10	00000000000000000000000111111111	11111111111111111111111100000000
11	00000000000000000000001111111111	11111111111111111111111000000000
12	00000000000000000000011111111111	11111111111111111111110000000000
13	00000000000000000000111111111111	11111111111111111111100000000000
14	00000000000000000001111111111111	11111111111111111111000000000000
15	00000000000000000011111111111111	11111111111111111110000000000000
16	00000000000000000111111111111111	11111111111111111100000000000000
17	00000000000000001111111111111111	11111111111111111000000000000000
20	00000000000000011111111111111111	11111111111111110000000000000000
21	00000000000000111111111111111111	11111111111111100000000000000000
22	00000000000001111111111111111111	11111111111111000000000000000000
23	00000000000011111111111111111111	11111111111110000000000000000000
24	00000000000111111111111111111111	11111111111100000000000000000000
25	00000000001111111111111111111111	11111111111000000000000000000000
26	00000000011111111111111111111111	11111111110000000000000000000000
27	00000000111111111111111111111111	11111111100000000000000000000000
30	00000001111111111111111111111111	11111111000000000000000000000000
31	00000011111111111111111111111111	11111110000000000000000000000000
32	00000111111111111111111111111111	11111100000000000000000000000000
33	00001111111111111111111111111111	11111000000000000000000000000000
34	00011111111111111111111111111111	11110000000000000000000000000000
35	00111111111111111111111111111111	11100000000000000000000000000000
36	01111111111111111111111111111111	11000000000000000000000000000000
37	11111111111111111111111111111111	10000000000000000000000000000000

2 つのマスクが選択された後、最終のマスクを得るために、それらは一緒に AND される。バイトを定義する連続した 1 のフィールドを除いて、このマスクはオール 0 である。

例えば、もしMR=1, ローテート・カウント=5, legth minus 1 = 7であるとき、右マスク・インデックスは5で、左マスク・インデックスは14(八進)である。
ファイナル・マスク中のこれの結果は次のとおり:

```
        右マスク    5          11111111111111111111111111100000
        左マスク   14          00000000000000000001111111111111
 それらのAND                   --------------------------------
        ファイナル・マスク     00000000000000000001111111100000
```

バイトは8ビット幅であり、右から5の位置である。

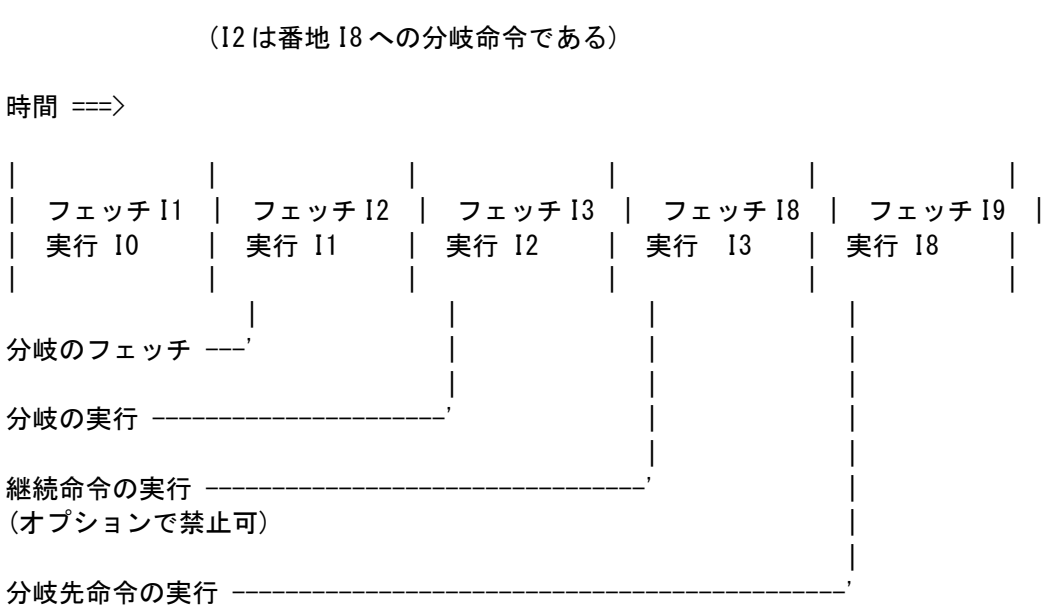
プログラミング・ヒント:
もしバイトが"大き過ぎ"(つまり、その位置とサイズの指定がワードの左側の端を越えてはみ出ることが起き)たら、マスカ(masker)はバイトを左端で切り詰めない。代わりに、0のマスクを生成する、すべてについてバイトを選択しない; よって、バイト操作の出力はAソースと同じである。この理由は、左マスク・メモリへのインデックスの計算でオーバーフローを起こしたら、ファイナル・マスクは0である、ということである。例えば、もしMR=1, ローテーション・カウント=20(八進), lentgh minus 1 = 27(八進)であれば、右マスク・インデックスは20で左マスク・インデックスは477 mod 32である。
これのファイナル・マスク中の結果は次:

```
        右マスク 20          11111111111111110000000000000000
        左マスク   7         00000000000000000000000011111111
AND them together            --------------------------------
        ファイナル・マスク    00000000000000000000000000000000
```

Control 制御

プロセッサの制御部 DISPATCH 命令の間使用される、14bit プログラム・カウンタ(PC)、32 番地の PC スタック(SPC)とスタック・ポインタ(SPCPTR)、2K ディスパッチ・メモリ、から成っている。いくつかのマイクロプロセッサとは違い、多くの伝統的な機械のように、操作の通常モードは PC のインクリメントにより次に続く命令を実行する。

プロセッサは一つの命令ルックアヘッド(先読み)を使用する。つまり、次の命令の読み出しは現在の命令の実行とオーバーラップされる。これは、もし、分岐が成功したとしても、分岐命令の後に、プロセッサは、引き続く命令を普通に実行することを暗示している。これらの命令にはこの実行を(N ビットを使い)禁止するための機能がある。しかし、それに使われるべきであったサイクルは浪費されるだろう。



2 つの型の命令が、機械中の制御のフローに影響を与える。条件 JMUP は、命令そのものの中で新 PC と移行型(transfer type)を指定し、DISPATCH 命令は新 PC と移行型を 2K ディスパッチ・メモリから見つける。新 PC が PC レジスタにロードされるか、3bit の移行型で指定される操作が実行されるか、のどちらかのケースがある。これらの操作は:

N bit
もし ON なら、次の命令の実行を禁止する,すなわちそれは移行命令のアドレスより一多いアドレスの命令のことである。(もし制御の移行がすでに進行中であれば、この命令は一大きいアドレスの命令を本当には必要としない。(※訳注:すでに後続の命令もフェッチされているから)) その命令によって実行されるべきであったサイクルは、浪費される。

P と R ビットは次のようにデコードされる:

P=0 R=0 BRANCH(分岐)　　通常プログラム移行。

P=1 R=0 CALL(コール)　　正しい戻りアドレスを SPC スタックにセーブし、新 PC アドレスにジャンプ。

P=0 R=1 RETURN(リターン)　　新 PC を無視； 代わりに SPC スタックから PC をポップする。

P=1 R=1 FALL THROUGH or I-MEM WRITE(そのまま抜けるか I-MEM 書き込み)
DISPATCH 命令では、ディスパッチしない。
JUMP 命令では；ジャンプをせずに、命令メモリに書き込む。

BRANCH(分岐)移行型は通常のプログラム移行、戻りアドレスをセーブしない。

CALL 移行型は、適切な戻りアドレスを SPC スタックにプッシュする。スタックは 32 番地の長さがある。オーバーフローを避けるのはプログラマの責任である。戻りアドレスは、PC+2 であるか、または、もし N ビットが ON だと PC+1 である。実際のところは、もし N ビットが ON なら NOP にされた命令のアドレスがセーブされるが、もし制御の移行がすでに進行中であるとそれは PC+1 と同じではないだろう。もし N ビットが ON でなければ、命令のアドレス+1 がセーブされる。ディスパッチの場合、もし N ビットが ON でなく、かつ命令のビット 25 が ON であれば、PC、ディスパッチ命令そのもののアドレス、をセーブする； これはディスパッチが、リターン時に再度実行されることを可能にする。(実際には、パイプライン動作しているので、上の段落で言う PC は本当の PC を意味していない。)

RETURN 移行型は、戻り PC を SPC スタックからポップし、命令中で指定されている PC やディスパッチ・テーブルを無視する。

FALL THROUGH(そのまま抜ける)移行型はディスパッチのためであり、ディスパッチ・テーブル中のエントリのいくつかが、すべての後にディスパッチを起さないことを指定できるようにする。引き続く命令は実行され(禁止されていない限り)、その後も何かに引き続く(続きの最初のものが分岐か、後続の禁止でなければ)。

I-MEM WRITE(I-MEM 書き込み)移行型はマイクロプログラム命令メモリへの命令書き込みの機構であり、後の節で説明する。(ディスパッチ・メモリは、命令メモリと違い、P と R ビットをセットすることでは書き込めない(すべての後、ディスパッチ命令中のこれらのビットはディスパッチメモリからやってくる!)； 代わりに Miscellaneous Function(その他機能)フィールドが使用される)

ALU と BYTE 命令を含む、各命令の中の付加ビットは POPJ ビットと呼ばれ、どんな命令の実行とも一緒に RETURN 移行型の同時の実行を指定できるようにしている。つまり、移行ではないなんでもに加えて、この命令があたかも N ビットを ON せずに RETURN 移行型ジャンプを実行したのと、同様のことを行う。他の移行型と同時にこのビットを使用した衝突を避けるのは、プログラマの責任である

POPJ ビットは、JUMP 命令の中では、RETURN 移行型と連結した中でしか使用できない。これは、2 つのどちらかのケース、引き続く命令の実行が条件付き以外か(※訳メモ：条件付き以外か、条件付きか、後で再チェックすべし)、条件 JUMP 命令と N ビットによって制御されているか、のどちらで RETURN 操作を起こす。POPJ ビットは、DISPATCH 命令中で使用されている時には、JUMP と CALL 移行型によって、特別にくつがえされる。例外的ケースで他のコードへジャンプすることを除いては、普通に RETURN を行い、リターンしたくない他のディスパッチ命令と同じディスパッチ・テーブルを使用することを可能にする。POPJ ビットは、ディスパッチか命令メモリへの書き込みと連結された中での使用と、SPC のポップとプッシュ機能ソースとデスティネーションと一緒の使用は、してはならない。これらのケースでは、マシンはなにか意味のあることをしようと悩んだりしない。(※訳注：意味の無いことが起こる)

The DISPATCH Instruction ディスパッチ命令

ディスパッチ命令はMバス上で利用可能な どのソースの選択もできる[データパス節のMバス・ソースの説明を参照], そして、選択されたワードからの7bitまでのどんなサブ・フィールドでもディスパッチができる。選ばれたサブ・フィールドは、11bitアドレスを作るために、命令の"ディスパッチ・アドレス(dispatch address)"フィールドと一緒にORされる。このアドレスは、ディスパッチ・メモリ中の14bitのPCと3bitの移行型を見つけるのに使用される。SPC-pointer-and-data-popソースは、ディスパッチ命令とつながれた中では、意味のある操作をしないだろう。

IR<44-43> = 2 (DISPATCH操作)
IR<41-32> = ディスパッチ定数 (またD-MEMの書き込みの時は、Aソースも)
IR<31-26> = Mソース
IR<25> = もしNビットがセットされていたら、次の命令ではなく、この命令のアドレスが戻りアドレスとして、CALL移行型によってSPCにプッシュされる。
IR<24> = 命令ストリーム(instruction-stream)・ハードウェアを許可する(後述)
IR<23> = 不使用
IR<22-12> = ディスパッチ・メモリのアドレス
IR<9-8> = オフ・ザ・マップ・ディスパッチの制御, 下を参照
IR<7-5> = Mソースからディスパッチに載せるバイトの長さ(-1ではない!)
IR<4-0> = Mソースのローテート・カウント(左へ)

ディスパッチ操作は、指定されたMソース・ワードを取り、それをローテート・カウントで指定された分だけ左へローテートする。下位Kビットを除いたすべては、マスクされる。ここでKは長さ(length)フィールドの内容である。その結果はディスパッチ・アドレスとORされ、そして、これは2K ディスパッチ・メモリをアドレスするのに使用される。ディスパッチ・メモリは新PC,R,P,Nビットを供給する。

もし、IRのビット8と9が0でなければ、ディスパッチ・アドレスの最下位ビットは、ローテータとマスカではなく、仮想メモリ・マップから来る。この場合、マップへのアドレス入力はMDから来る。これは、ガーベジ・コレクタの慣習を尊重し、メインメモリからフェッチされたばかりのポインタの正当性をテストするのに一番便利である。IR<8>は第2レベルのマップのビット14を選択し、IR<9>はビット15を選択する。両方のビットを選択は、それらを一緒にORする。

各ディスパッチ命令で、ディスパッチ定数フィールドはDISPATCH CONSTANT(ディスパッチ定数)レジスタにロードされる。このレジスタはMソースとしてアクセスできる。ディスパッチ定数フィールドはディスパッチ操作と一緒だとどんなものであろうと何もしない; 何をしている時であろうと、完全な乱数レジスタをロードするためのデバイスとして本当に手軽である。(この機能を使うことは、あとの節で議論されている。)

その他機能2は、命令の通常動作を禁止し、代わりに、ディスパッチ・メモリに、Aソースによって指定されたAメモリ・スクラッチパッド番地の下位の内容をロードする。注意として、Aソース・アドレスは、ディスパッチ定数フィールドと同じである。ディスパッチ定数はどうやってもロードされるが、これは無視することができる。パリティ・ビット(ビット17)もロードされ、メモリに正しい(奇数)パリティをロードすることは、プログラマの責任である。ディスパッチ・メモリの通常のアドレシングは有効であり、よって、長さ(length)フィールドが0を持つようにするのが賢明である、そうすると、命令中のディスパッチ・アドレスによって、変更されるべきディスパッチ・メモリ番地が一意に指定される。

The Jump Instruction ジャンプ命令

ジャンプ命令は、M ソースのどのビットでも条件の元にでき、また、ALU 出力を含む、様々なプロセッサ内部の条件を元にした条件分岐ができる。(DISPATCH も一つの M ソース・ビットをテストするのに使えるので、JUMP の使用はディスパッチ・メモリを節約する。) JUMP 命令は、トリックによって、命令メモリへの書き込みでも使用される。

```
IR<44-43> = 1 (ジャンプ操作)
IR<41-32> = A ソース
IR<31-26> = M ソース
IR<25-12> = 新 PC
IR<9>     = R bit (1 であると、新 PC を SPC スタックからポップして取り出す)
IR<8>     = P bit (1 であると、戻り PC を SPC スタックへプッシュする)
IR<7>     = N bit (1 であると、もしジャンプが成功したら次の命令を禁止)
IR<6>     = もし 1 なら、ジャンプ条件を反転する
IR<5>     = もし 0 なら、M ソースの 1 ビットをテスト;もし 1 なら内部条件をテスト
IR<4-0>   = IR<5>=0 なら M ソースのためのローテーション・カウント
            IR<5>=1 なら、条件番号:
                        0  シフタ出力の下位ビット(不正)
                        1  M ソース < A ソース
                        2  M ソース <= A ソース
                        3  M ソース = A ソース
                        4  ページ・フォールト
                        5  ページ・フォールトか割込みペンディング
                        6  ページ・フォールトか割込みペンディングかシーケンス・ブレーク・フラグ
                        7  常に真
```

ページ・フォールト、割込み、シーケンス・ブレークは後の節に記述。

ジャンプの条件は次のように決定される。もし IR<5>=0 ならば、ローテーション・カウントにより M ソースは左にローテートされる; そして、結果の下位ビットがテストされる。つまり、符号ビットをテストするには、ローテーション・カウントには 1 が使用されるべきである。もし下位ビットが 1 ならば、ジャンプ条件は真 (true) である。もし IR<5>=1 ならば、指定された内部条件がテストされる。どちらの場合も、IR<6>=1 ならば、ジャンプ条件は反転して見られる。これによって、特に、M と A ソースの間の 6 つの算術的関係のすべてをテストできるようにしている。

反転後も含んだ最終ジャンプ条件が、もし真なら、新 PC フィールドと R,P,N ビットが新しい PC の内容を決定するために使用される。条件が真でなければ、次の命令の実行を続ける、ただし POPJ ビットの効果を除いて。(※訳注:原文の modulo は、Hacker 用語で、except for の意味)

R と P ビットの両方がセットされていたら (WRITE)、A と M ソースが (条件により!) 命令メモリに書かれる。ビット<47-32>は、A ソースのビット<15-0>から取られ; ビット<31-0>は M ソースのビット<31-0>から取られる。注意として、これは、"次命令修正 next instruction modify"機能デスティネーション (16,17) で使用されるビットのアラインメント (整列) とは異なる。セットされる命令の WRITE の番地が奇数である理由は、それが動作する方法だからである。(※訳メモ 奇数番地の意味を再確認 The reason for the odd location of

WRITE in the instruction set is due to the way it which it operates.) それは、CALL 移行型でも同じことを起こし、結果として、古い PC に 1 か 2 を加えたものを SPC スタックにセーブしながら、修正されようとするアドレスが PC レジスタにロードされる。そして、命令メモリが普通にその場所から実行されるための命令をフェッチしようとするとき、書き込みパルスが生成され、A と M ソースからセーブされたデータを命令メモリへと書き込む。一方、マシンは RETURN 移行命令をシミュレートし、SPC スタックをポップし PC を復旧し、命令実行を残りの位置から進める。注意として、この余分の命令は、SPC スタック上の 1 ワードを使用し、余分の 1 サイクルが必要である。強くお薦めするのは、JUMP 命令中の N ビットを ON にするべきである。なぜなら、書き込みに引き続く命令の実行の間、プロセッサは RETURN 移行型を無条件に実行するからである。けれども、もし、これが、これに続く命令が指定する他の事柄と衝突しないのなら、続く命令が実行されてしまうだろう。手当てが必要である。

Program Modification プログラム変更

プログラム命令の可変フィールド(variabilizing fields)では斬新なテクニックが使用されている。出力バスの"機能デスティネーション"の 2 つは、(概念的には)レジスタ(時には、ひとまとめに OA レジスタとして参照される)であり、それらの内容は次に実行される命令を OR して得ている。シフタ/マスカと組み合わせた能力はどんな連続したビットの集合でも任意のフィールドに入れる， この機能は、例えば、可変なローテーション・カウントとプログラムがレジスタのアドレス決定をするのに使用する能力を提供する； 例えば、それは A スクラッチパッド・メモリをインデックスするのに使用することができる。

機能デスティネーション 16(OA-REG-LOW)、書き込みの時、効果(有効?)的にビット<25-0>を次の命令のビット<25-0>に OR する；機能デスティネーション 17(OA-REG-HIGH)は効果(有効?)的にビット<21-0>を次の命令のビット<47-26>に OR する。ビット <26>と<25>の間の位置は、命令の全クラスにおける自然な分割線である。注意として、ある特定の命令の半分だけが、変更可能である。なぜなら、両方の機能デスティネーションを同時に書き込むことは不可能であるから。

この機能が使用される時、命令メモリからのフェッチされるワードのパリティ・チェックは禁止される。なぜなら、パリティがチェックされる前に、OA"レジスタ register"がメモリの出力への OR されるからである。

この機能は特に、書き込もうとする命令メモリの番地かディスパッチメモリのアドレスを供給するためや、A と M メモリ上の可変なアドレスを指定するためや、可変長や可変な位置でのバイトの操作のために便利である。これらの例は、後の節に詳しい。

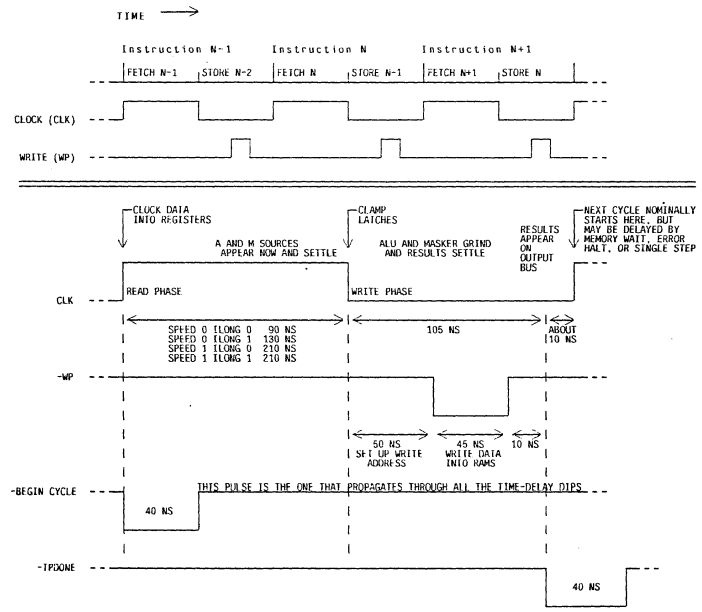
Clocks クロック

CADR プロセッサはただ一つのクロック信号を使用している。このクロックは、出力データを指定されたレジスタにロードし、新 PC と命令もロードする。クロックと同期せずに起こるイベントは、A,M,PDL スクラッチパッドの制御信号と SPC スタックだけである。これらのデバイスのために、2 ステージ・サイクルが実行される。最初のフェーズの間、各デバイスのソース・アドレスがアドレス入力にゲートされている。出力データが安定した後、これらのデバイスの出力がラッチされる。そして、アドレスを、「前の」命令から書き込み番地として指定されたものに変更する。アドレスが安定した後、書き込みパルスがスクラッチパッド・メモリのために生成され、書き込みが実行される。Pass-around path(パス回りパス)が A,M メモリのために提供されている(プログラマには見えない)。それは、前のサイクルで書き込まれたが、しかし、実際にはまだスクラッチパッドに書き込まれていない番地への読み出し参照に注目し、正す。PDL メモリのためには Pass-around path(パス回りパス)のようなものが無いので、どの PDL メモリへの書き込みサイクルでも、M スクラッチパッドにも書き込まなければならない。そうすれば、次の命令は M スクラッチパッド番地を参照することができる。そこで M pass-around path(M パス回りパス)が使用される。SPC スタックは、RETURN 移行型によって使われるときは、一つの pass-around path(パス回りパス)を持つが、M ソースとして使用される時は pass-around path(パス回りパス)を持たない。RETURN pass-around path はサブルーチンがただの 2 命令だけの長さであることを可能にする。失われている pass-around path を提供するには余分のハードウェアが要る。そして、実際のマイクロプログラムの検査で、それが非常にまれにしか使用されないことが判るだろう。

クロック・サイクルは可変長である。サイクルの前半分の期間(読み出しフェーズ"read phase")は、命令の ILONG ビット(IR <45>)によってと、診断インターフェースからの 2 つの"speed"ビットによって制御される。後半の期間(書き込みフェーズ"write phase")は通常、固定である。クロックは、プロセッサ・クロックと、バス・インターフェース、メモリ、外部デバイスのためのクロックの両方に配られる。

クロックは、いくつかの理由から、どちらかのフェーズの最後で停止することができる。通常、クロックは読み出しフェーズの最後でストップし、"ウェイト wait"として参照される。これはクロックをインアクティブなハイ(high)状態にしておく、そしてメモリのラッチを開いたままにする。クロックは、次の、診断インターフェースによるマシンの停止の指令、診断インターフェースによるシングル・ステップの指令でそれを完了した、パリティ・エラーのようなエラー、統計カウンタのオーバフロー、メモリ・ウェイト状態、によってマシンをウェイトすることができる。もし、前のサイクルが進行中の間にメイン・メモリ・サイクルが初期化されるか、あるいは、もし、バス・コントローラが読み出しサイクルの実行に必要なバス・アクセスを許可する前に、プログラムがメイン・メモリ読み出しの結果を要求する時、のどちらかで、この後半の状況(※訳注:後半のフェーズでウェイトする状況)が発生する。クロック・ウェイトの間、プロセッサ・クロックは止まるが、システムの残り(バス・インターフェースと XBUS デバイス) へのクロックは走り続け、それらの動作はできる。プロセッサがウェイトを終了する時、プロセッサ・クロックは外部クロックと同期してスタートアップする

クロックは書き込みフェーズの最後でも止めることができ、"ハング hang"として参照される。これは、メモリ読み出しの間でだけ使用される。もしプロセッサが、まだ完了していない進行中の読み出しの結果を要求するとき、データがメモリからやってきて、そして、データがデータパスを通って流れ、出力バスに出現するのに十分な時間が経つまで、ハングする。これは、データのパリティがチェックされるのに十分な時間でもある。
ハングの場合、両方のクロックが停止し、それはどんな余分の遅延も無く、同期が再開することを可能にしている。このようにして、プロセッサの速度はメモリの速度に正確に合致するように、調整される。



All times shown are nominal and subject to tweaking.
Diagram is not precisely to scale.

Accessing memory メモリのアクセス

メイン・メモリへのアクセスは、いくつかの機能ソースとデスティネーションの使用を通して成し遂げられる。これらは 3 つの機能を実行する； 1 番， VMA（仮想メモリ・アドレス virtual memory address）と MD（メモリ・データ）2 つのレジスタにアクセスできる。2 番，メモリ操作を初期化が可能。3 番， メモリ操作の完了までウェイトできる。実際には、この機構はメイン・メモリをアクセスするためだけではない； Xbus や Unibus 上のデバイス、それにはメモリだけでなく周辺機器も含まれる、をアクセスするのに使用される。だが、簡単のために"メモリ"という言葉を使用する。

8 つの機能デスティネーションが、メモリ・システムに結び付けられている。それらの 4 つが、VMA へのデータをロードし、他の 4 つが MD へデータをロードする。各グループは 4 つ、一つは副作用なし、一つは読み出しサイクルをスタート、一つは書き込みサイクルをスタート、一つは仮想アドレス・マップへの書き込み、から成っている。

メモリ読み出し操作中、メモリからのデータは、それが使用可能になったら MD レジスタに置かれ、それから、プログラムによって（機能ソースを使用して）ピックアップ可能である。メモリ書き込み操作では、プログラムは書き込まれるべきデータを MD レジスタに（機能デスティネーションを使用して） 置く、そしてそれはメモリに渡される。

VMA レジスタは参照される番地の仮想アドレスを持つ。これは、24bit 長；レジスタの上位 8bit は存在するが、ハードウェアにより無視される。VMA は"仮想"アドレスを保持する； メモリに送られる前に、それは、"map"を通して渡される。"map"は 22bit の物理アドレスを生成し、要求された読み出しと書き込みの操作が許されるかを制御し、そして、ソフトウェア（マイクロコード）が自分自身の目的のために使用することができる 8bit を記憶する。

メモリ・サイクルの開始時を除いて、マップされるべきアドレスは、VMA レジスタではなく、MD レジスタのビット<23-0>から来る。これは、読み出しや書き込みしようしているポインタが指している、メモリのどの"空間 space"であるかの、チェックのためのマップの使用を単純にするためである。マップのチェックは、Lisp マシンのガーベジ・コレクション（ごみ集め） アルゴリズムで頻繁に必要とされる操作である。

マップは 2 つのスクラッチパッド・メモリから成っている。第一レベルマップ（First Level Map）は 5bit で 2048 個の番地を持ち、それは VMA か MD のビット<23-13>によりアドレスされる。第二レベルマップ（Second Level Map）は 24bit で 1024 個の番地を持ち、それは、第一レベルマップの出力と VMA か MD のビット<12-8>をつないだものによりアドレスされる。仮想アドレス空間は、各々 32 ページを含む、2048 ブロックを持つ。各ページは 256 ワード（もちろん、32bit の）を持つ。仮想アドレス空間の各ブロックは第一レベルマップの中に対応する場所を持つ。第二レベルマップ中の位置は、恒久的に特定のアドレスが割り当てられているのではない；その代わり、現在、記述（※訳注：アドレス変換やページの属性を記述している）されているそれらのアドレスが第二レベルマップ中のどこになるかを、仮想アドレスのブロックのための第一レベルマップの場所が示している。
第二レベルマップは 32 個のブロックを記述するために十分な空きを持っており、よってどんなときも、ほとんどのブロックの記述は"情報無し no information available"にされているであろう。これは、第二レベルマップの最後の 32 個の場所をこの用途のために、"no information available"ページ記述子でそれらを埋めて、予約することによって、行われる。ほとんどの第一レベルマップはここを指すことであろう。

第二レベルマップの出力は次の構成：

MAP<23>　　= アクセス許可（アクセス・パーミッション）
MAP<22>　　= 書き込み許可
MAP<21-14> = ソフトウェアで利用可能。特にビット 15,14 は DISPATCH 命令によってテストすることができる。
MAP<13-0>　= 物理ページ番号

メモリに送られる物理アドレスは、物理ページ番号と VMA のビット 7-0 をつないだものである。

2 つのマップは MD に適切なアドレスを置くことで読み出すことができ、
そして、読み出しの機能ソース MEMORY-MAP-DATA (11) は、次の通り：

MAP<31> = 1 であれば、一番最近の書き込もうとしたメモリ・サイクルが、
書き込み許可 (write permission) が無かったため、実行されなかった。
すなわち第二レベルマップのビット 22 が 1 であった。
MAP<30> = 1 であれば、一番最近のメモリ・サイクルが、アクセス許可が無かったため、実行されなかった。すなわち第二レベルマップのビット 23 が 1 であった。MAP<30>が 0 ならアクセス・フォールトは無いが、書き込みのフォールトはあるかも知れない。注意として、ビット<31-30> は最後に行おうとしたメモリ・サイクルに適用し、MD の内容によってアドレスされたマップの場所には何もしない。
MAP<29> = 常に 0。
MAP<28-24> = 第一レベルマップ
MAP<23-0> = 第二レベルマップ

マップは機能デスティネーション VMA-WRITE-MAP (23),MEMORY-DATA-WRITE-MAP (33) の一つを使用して書き込める。MD は書き込むべきマップの番地のアドレスを供給し、VMA は書き込むデータを供給し、どのレベルのマップに書き込むかを言う。直前の命令中で 1 つのレジスタが準備されなければならない、他方は機能デスティネーション経由で書き込まれ、マップへの実際の書き込みは引き続くサイクルで起こる。パス回りパス (pass-around path) やマップのためのラッチはない。よって、引き続く命令はそれを使用してはならない。

第一レベルマップは、VMA<26> が 1 の時に、VMA のビット <31-27>から書き込まれる。(これは MEMORY-MAP-DATA 機能ソースを使用した時に読み込むものとは違うビットである。)
第二レベルマップは、VMA<25> が 1 の時に、VMA のビット <23-0>から書き込まれる。
注意として、第二レベルマップを書き込むとき、第一レベルマップはアドレスの一部を供給し、そしてそれは先んじて書かれていなければならない。そして、それは両方を同時に書くときに使い物にならない、けれども、両方のビットを 1 にすることは可能である。

メイン・メモリ操作は、機能デスティネーション VMA-START-READ (21), VMA-START-WRITE (22), MEMORY-DATA-START-WRITE (32) のひとつを使用して開始する。MEMORY-DATA-START-READ (31) というものもあるが、たぶん役に立たない。
書き込みの場合、VMA がアドレスを供給し、MD がデータを供給する。よって、一つのレジスタが先立ってセットアップされなければならず、他方は操作をスタートするための機能デスティネーションによってセットアップされる。
メイン・メモリ読み出しは、後述するマクロ instruction-stream ハードウェアによってもスタート可能である。

VMA か MD と名づけられたレジスタは、命令が実行された期間の最後のサイクルで、(出力バスからの) その命令の結果と一緒にロードされる。続くサイクルの間、マップは読み出される。このサイクルの期間、実行される命

令は必ず JUMP 命令であり、それはページ・フォールト条件をチェックする。このサイクルの終わりに、ページ・フォールトの発生がなければ、メモリ操作が開始する。プロセッサはメモリ操作が起きている間も実行を続ける。しかし、もし、ビジーであるメモリと衝突する操作が行われようとしたら、マシンは、メモリ操作が完了するまで、ウェイトかハングする。こういう参照には、機能ソース MEMORY-DATA (12)を使用することによって読み出しサイクルの結果を尋ねること、VMA,MD,MAP を参照するようなすべての機能デスティネーションの使用、instruction stream ハードウェアを通して読み出しサイクルをスタートしようとすること、が含まれている。

ページ・フォールトの有無は、次のメモリ・サイクルが開始される時まで、記憶される。よって、サイクルの開始直後に、即座にページ・フォールトのチェックをすることは、厳格には必要ではない、が、それはいい慣習である。

VMA 中にずっと正しい値があるので、MEMORY-DATA-START-WRITE デスティネーションは read-followed-by-write(ライトが引き続くリード)操作の後半を行うのに便利である。
注意として、書き込み開始後の書き込みフォールトのチェックが続けて必要である、なぜなら、読み出しパーミッションがあったとしても、書き込みパーミッションがないかも知れないから。

メイン・メモリのパリティ・エラーをマイクロコードにトラップできる機能が存在している。診断インタフェース中のビットは、これを許可するか否かを制御する。MEMORY-DATA 機能ソースが使われる時、MD にロードされるべき最後のものがメモリからのデータで、それが偶数パリティを持っていたら、メイン・メモリ・パリティ・エラーが発生した。もしトラップが許可されていたら、現在の命令は NOP にされ、0 番地への CALL 移行が強行される。引き続く命令も NOP にされる。トラップ・ルーチンは、もし戻ることを計画しているならどこへ戻るかを決定するのに、OPC レジスタを使わなければならない。なぜなら、もし移行操作が進行中だった場合、トラップによって、SPC スタックへプッシュされたアドレスは、トラップを起こした命令のアドレスについて何もすることがないだろうから。マイクロコードが検出したプログラミングのエラーのためのエラー・ハンドラでも、これは真である。もし、メイン・メモリ・パリティ・エラーが発生し、かつ、トラップが禁止であって、もし「エラー停止 (error-halting)」が許可されていれば、内部メモリ中のパリティ・エラーに対するのとまったく同様にマシンは停止する。

半導体メイン・メモリを使用する時、それは 1 ビットのエラー訂正を持っている。パリティ・エラー・トラップは訂正不能な複数ビットのエラーが発生したことを示す。1 ビットエラーはハードウェアによって自動的に訂正され、そして割り込みを発生する。プロセッサは、余裕のある時、エラーを記録し、悪い番地の内容を再書き込みしようとするだろう。

The Instruction-Stream Feature 命令ストリーム機構

CADR プロセッサは、CADR のワード・サイズより小さな単位でやってくる命令ストリームの解釈を助けるハードウェアの小さな集合を持っている。例えば、Lisp マシンのコンパイルされたマクロ命令セットは 16bit が単位である。このハードウェアは、いくつかの予定管理をするルーチンのマイクロコードを取り除くことによって、フェッチと命令のデコードを高速化する。

8bit (バイト) と 16bit (ハーフ・ワード) 命令がサポートされていて、モード・ビット ("割込み制御 Interrupt Control"レジスタのビット 29, 機能デスティネーション 2) に依存している。ハードウェアは、新しいメイン・メモリのワードをフェッチするときに、命令ストリームの次が、2 か 4 単位を持つかを判断し、マイクロプログラム制御のフローを変更する。ハードウェアはローテータ制御の機能を提供しており、それによって、命令ストリームの現在の単位を選ぶことができる;これは、命令を解釈しディスパッチする時と、BYTE マイクロ命令を経由して命令のフィールドを抽出する時に使用される。

ロケーション・カウンタ (Location Counter (LC)) と呼ばれる 26bit レジスタのレジスタがあり、機能ソース 13 から読み出せ、機能デスティネーション 1 から書き込める。それは常に命令ストリームの次の単位のアドレスを持つ。アドレスは 8bit の区切りである。ハーフワード・モードでは、LC<0>は強制的に 0 になる。LC は、1 か 2 (バイトかハーフワード・モードによって) をカウントすることができ、そして、VMA へ特別な接続を持つ; 命令フェッチが起きたとき、VMA には LC を 4 で割ったものがロードされる、。

機能ソース 13 の上位 6bit は、本質的に LC の部分ではない、しかし、関係のある様々なステータスを持っている:

31 フェッチが必要。もし、次回に命令ストリームを進めるならば、これが 1。そして、新しいワードがメイン・メモリからフェッチされるだろう。これが、LC 下位ビットの、バイト・モードの、そして、命令ワードが最後にメイン・メモリからフェッチされてから、LC が書き込まれたかどうかの、機能である、

30 不使用。0。

29 LC バイト・モード。もし、命令ストリームが 8bit 単位なら 1、16bit 単位なら 0。これは、割り込み制御レジスタのビット 29 に反映される。

28 バス・リセット。これは、割り込み制御レジスタのビット 28 に反映される。これが 1 にセットされたならバス・インタフェース, Unibus と Xbus をリセットする。

27 割り込みイネーブル。1 なら外部割込み要求が JUMP 条件に寄与できる。これは、割り込み制御レジスタのビット 27 に反映される。

26 シーケンス・ブレーク。もし 1 なら、シーケンス・ブレーク (マクロコード割り込み信号) がペンディングである。このフラグは JUMP 条件に寄与する以外、何もしない。これは、割り込み制御レジスタのビット 26 に反映される。

SPC スタックのビット 14 は、メインの命令解釈ループのアドレスとして戻りアドレスを持つことを示すフラグとして使用される。ハードウェアは SPC<14>=1 のある RETURN 移行を一つの命令の解釈を完了と認識し、次の (命令) 解釈を初期化する。RETURN 移行に続くサイクル中で、命令ストリームは、その次の単位 (バイトかハーフワード) に進められるだろう。(色々なタイミングの理由から、それは 1 サイクル遅延される。) このサイクルは、、LC, VMA, MD や関係のあるハードウェアを使わない、役に立つマイクロ命令を実行してもよい。

命令ストリームを進めるのは、LC を 1 か 2、インクリメントする。もし、新ワードがメイン・メモリからのフェッチを必要としたら、LC はインクリメントされず、4 で割られ、VMA に送られ、読み出しサイクルが開始される。LC がワードの最初の単位を指しているか、(分岐が発生し) 最後の命令ストリームを進めることによって LC が変更されたか、のどちらかの理由によって、フェッチは要求される。LC を変更する RETURN 移行を実行する命令は正当であり、そし

て、フェッチは常に要求される。もしフェッチが要求されないなら、RETURN 移行は強制的に SPC<1>を 1 にすることによって切り替えられ、2 つのマイクロ命令を飛び越し、そしてフェッチの場合、ページフォールト（か割り込みかシーケンス・ブレーク） をチェックし、そして新しい命令ストリーム・ワードを MD からスクラッチパッド番地に転送する。

命令ストリームはビット 24 がセットされた DISPATCH 命令によっても進められることができる。この場合、SPC 戻り番地の置き換えは発生しない。ディスパッチは、まさに新しいワードがフェッチされるかどうかを決定するために LC 機能ソースのビット 31 の NEEDFETCH 信号をチェックしなければならない。フェッチが発生したら、DISPATCH は、ページ・フォールトをチェックするためにサブルーチンを呼ばなければならず、新しい命令ストリーム・ワードを MD からスクラッチパッド番地に転送する。フェッチが発生しなければ、DISPATCH はそのまま下へ抜ける。DISPATCH の後の命令は命令ストリームの次の単位を操作するだろう。この機構はマルチ単位（複数単位）命令の使用を容易にするために提供されている。

命令ストリーム機構の残りのハードウェアは、その他機能 3 を実現しており、命令ストリームの現在の単位を現在のワードから選択するために M ローテート・フィールドを切り替え、M ソースから供給されなければならない。ローテータを使用するどの操作：BYTE 命令、DISPATCH 命令、ビットをテストする JUMP 命令、にもこれは適用される。命令は単位（バイトかハーフワード）ごとに、ワードの右側の終わりの位置にコード化されなければならない。ハーフワード・モードでは、IR<4>は LC<1>と XOR されローテート・カウントの上位ビットになる。バイト・モードでは、IR<4>が（LC<1> XOR LC<0>）と XOR され、そして IR<3>は LC<0>と XOR される。LC は常に次の命令のアドレスを持ち、ビットは右から左に番号が付いているので、その効果は、望んだとおりである。ハーフワード・モードでは、LC<1>=1 の時、M ソースの下半分が偶数命令としてアクセスされ、LC<1>=0 の時、上半分が奇数命令としてアクセスされる。

Multiplication, Division, and the Q register 乗算、除算、Qレジスタ

Qレジスタは CADR 中の乗算と除算のための主たるものとして提供されている。それは、時々、他の事柄でも便利なことがあり、ALU 命令の結果を置く余分の場所にしたり、ALU 命令中で OUTPUT-SELECTOR-RIGHTSHIFT-1 操作が使われるときにシフト・アウトするビットを溜めるのに使えたり。

Qレジスタは、ALU 命令中の 2 つのビット(IR<1-0>)で制御される。その操作は、何もしない、左シフト、右シフト、ALU の出力からロード、である。(電子的な理由から、出力バスではなく、ALU からロードする。) Qレジスタは左シフトする時、Q<0>は -ALU<31>を受け取る、ALU 出力の符号の反転である。Qレジスタは右シフトする時、Q<31>は ALU<0>を受け取る、ALU 出力の下位ビットである。Qレジスタは出力バス・シフタにも接続されている; 出力バスが左シフトさせられる時、OB<0>は Q<31>を受け取り、それは Q の符号である。これらの内部接続は乗算と除算の必要性によって、作成されている。

CADR の乗算は単純である、一度に 1 ビット、シフトと加算を行う。
ハードウェアは条件付き ALU 操作である、MULTIPLY-STEP、もし Q<0>=1 なら ADD、そうでなければ SETM を行う、を提供している。これは、SHIFT-Q-RIGHT と OUTPUT-SELECTOR-RIGHTSHIFT-1 と組み合わせて使用される。まず最初に、A スクラッチパッド番地に被乗数を置き、乗数を Q に置く。MULTIPLY-STEP 操作を 32 回実行する; Q は乗数のビット毎に右へシフトするごとに、Q<0>に現れる。もしビットが 1 なら、被乗数は加算される。各操作の結果は、M スクラッチパッド番地に行く、それは次のステップにフィードバックされる。各結果の下位ビットはシフトされ Q に入っていく。そして、32 ステップが完了したら、Q は積の下位 32bit を持ち、M スクラッチパッドは上位 32bit を持つ。

このアルゴリズムは、2 の補数を扱うのに、わずかな変更が必要である。2 の補数の符号ビットが負の重みを持っている、よって、最終ステップがもし Q<0>=1 なら、乗数は負で、加算の代わりに減算を行わねばならない。ハードウェアはこれを提供しない、よって、代わりに我々は最終ステップの後に減算を行う、最終ステップは加算であり、そして加算して多すぎる分の減算を 2 回、それで減算の効果がある。注意として、この修正は積の上位 32bit だけに効果があり、もし単精度結果(※訳注:下位 32bit の結果)だけを求めているなら、修正は省略できる。
次のコードについて考察する。(CONSLP アセンブラ・フォーマットはこの文書の後方で述べられている)

```
; 乗算サブルーチン。 A-MPYR 掛ける Q-R, 積の下位が Q-R, 上位が M-AC.

MPY       ((M-AC) MULTIPLY-STEP M-ZERO A-MPYR))      ;部分解 = 0 最初のステップ
(REPEAT 30. ((M-AC) MULTIPLY-STEP M-AC A-MPYR))      ;Do 30 ステップ
     (POPJ-IF-BIT-CLEAR-XCT-NEXT                     ;もし A-MPYR が正なら、次の後にリターン
          (BYTE-FIELD 1 0) Q-R)
     ((M-AC) MULTIPLY-STEP M-AC A-MPYR)              ;最終ステップ
      (POPJ-AFTER-NEXT
       (M-AC) SUB M-AC A-MPYR)                       ;負の乗数の補正
     (NO-OP)                                         ;ジャンプのディレイ
```

32bit より小さい数の乗算も可能である。同じ初期化条件で、n ステップ後に、Q の上位 n ビットは積の下位 n ビットを持ち、積の残りのビットは M スクラッチパッドの下位ビット中にある。2 つの BYTE 命令が、これらのビッ

トを抽出してまとめて、右詰の積を生成するために使用できるだろう、もし数値が無符号 (unsigned) であれば。

除算は乗算よりもう少し複雑である。それは一時にもう少しやり過ぎる、引きはなし法だと、各ステージで加算か減算を行う。基本アイデアは、筆算による長い除算と同様で、被除数から除数を減ずることを続ける、差の残りが超えるように除数をシフトしつつ。もし減算結果が正であれば、"そのまま続ける goes in"で、商のビットが 1 に生成される。もし、減算結果が負であれば、"失敗 fails to go in"であり、商のビットに 0 が生成される。バックアップ (※訳注：引きすぎたら、元の値に戻す) と減算をしない代わりに、「減算で引きすぎましたフラグ too much has been subtracted」をセットし、次回、代わりに加算を行う。これは、次のステップの除数の重みが半分であるから、B - (A / 2) = B - A + (A / 2) として働く。この"フラグ"は単純に、商のビットとして生成されたものを反転である、ただし、最初のステップだけは、例外として、フラグは強制的に OFF にされなければならない。

除算は乗算ほど簡単には 2 の補数を扱えない。アルゴリズムは基本的にすべて正の数を要求する、
しかし、もし除数が負なら、ハードウェアは自動的に加算と減算を交換して、除数の絶対値を取る。
前もって被除数を正にし、終わった後に、商と剰余の符号を補正するか決定するために、マイクロコードに上げる。
商の符号は被除数と除数の符号の XOR である。剰余の符号は、被除数の符号と同じである。

最初に Q レジスタに正の被除数を入れ、符号付の除数を A スクラッチパッド番地に入れる。適切な条件付き ALU 操作が SHIFT-Q-LEFT と OUTPUT-SELECTOR-LEFTSHIFT-1 機能の連結したものとして使用される。M スクラッチパッド番地は各ステップの結果を受け取り、次のステップにフィードバックされる。この番地は、初期化時に倍長の被除数の上位 32bit を持ち、被除数が単精度では 0 とする。各ステップでは、OUTPUT-SELECTOR-LEFTSHIFT-1 操作は Q の上位ビットを M スクラッチパッドの下位ビットに入れる、被除数の他のビットは (シフトして) 持ち上げられる。各ステップでは、ALU 出力の符号の反転は商のビットを表し、Q の最下位にシフトインされる。33 ステップの後、Q は正の商を持つ (Q が Q-for-quotient レジスタと呼ばれる由縁である)。なぜ 32 ステップでなく、33 ステップが必要かということは、記述するには少し困難がある。最初のステップで作られる商のビットは、もし 1 なら"除算オーバフロー divide overflow"を示し、それは商の本当の部分ではない。単精度の被除数を使用している時、"除算オーバフロー divide overflow"は除数が 0 の時にしか発生しない。なぜなら、最初の操作は 0 から除数の絶対値を減じる、それは除数が 0 以外では負になるから。

すべての減算の後の、被除数の残りが正の剰余である。最終ステップは OUTPUT-SELECTOR-LEFTSHIFT-1 を使わない、従って M スクラッチパッドは剰余の 2 倍ではなく、剰余を受け取る。
もし"引きすぎ too much has been subtracted"フラグがセットされていれば、剰余を補正するために最後に一回加算が必要である。この加算は単純に前の減算を打ち消す、新しい減算を行うのではない、よって左シフトは省略する。

除算のための ALU 操作は：

DIVIDE-STEP
: 上に述べた条件付き加算か減算、SHIFT-Q-LEFT と OUTPUT-SELECTOR-LEFTSHIFT-1。
Q<0>=0 は"引きすぎ too much has been subtracted"フラグをサービスする。

DIVIDE-FIRST-STEP
: "too much has been subtracted"を強制的に off にすることを除いて、
DIVIDE-STEP と同じ。

DIVIDE-LAST-STEP
: OUTPUT-SELECTOR-LEFTSHIFT-1 を省略することを除いて
DIVIDE-STEP と同じ。

DIVIDE-REMAINDER-CORRECTION-STEP
: 条件付き加算か減算論理が使用される、乗算論理の一部の呼び出しによって

減算が SETM へ振り向けされる以外は。
除数が負であれば、加算が減算に交換が適用され、正しく物事が行われる。
シフトは発生せず、Q は変化しない

32bit より小さな数値の除算は、十分に注意した入力と出力のシフト操作によって、33 より少ないステップで遂行できる。

すべてが一緒にどのように適合しているかを解説するために、また、符号の補正がどうやって行われているかを示すために、ここに倍精度の被除数を持つ 32bit 除算のコードを掲げる。後述の CONSLP 形式で記述されている:

```
;  除算サブルーチン
;  M-AC と M-1 は各々、被除数の上位と下位ワードである。
;  M-2 は除数。商は M-AC, 剰余は M-1 に。

DIV       (JUMP-GREATER-OR-EQUAL M-AC A-ZERO DIV1)          ;負の被除数をチェック
          (JUMP-NOT-EQUAL-XCT-NEXT M-1 A-ZERO DIV0)         ;そうなら、符号を変更
         ((M-1 Q-R) SUB M-ZERO A-1)
          ((M-AC) SUB M-AC (A-CONSTANT 1))                  ;もし下位が 0 なら、上位からボロー
DIV0      ((M-AC) SETCM M-AC)                               ;1 の補数 上位被除数
          (CALL DIV2)                                       ;ここで、正の除数の場合をコールする
          (POPJ-AFTER-NEXT (M-1) SUB M-ZERO A-1)            ;剰余のネガティブを取る
         ((M-AC) SUB M-ZERO A-AC)                           ;商の符号を変更

;  正の被除数のための除算ルーチン
DIV1       ((Q-R) M-1)                                      ;被除数の下位を Q-R へ
DIV2       ((M-1) DIVIDE-FIRST-STEP M-AC A-2)               ;最初の除算ステップ
           (JUMP-IF-BIT-SET (BYTE-FIELD 1 0) Q-R DIVIDE-OVERFLOW) ;エラーチェック
(REPEAT 31. ((M-1) DIVIDE-STEP M-1 A-2)                     ;中間除算ステップ
           ((M-1) DIVIDE-LAST-STEP M-1 A-2)                 ;最終ステップ、商は Q-R に
           ((M-1) DIVIDE-REMAINDER-CORRECTION-STEP M-1 A-2) ;M-1 には剰余
           ((M-AC) Q-R)                                     ;Q-R から商を抽出
           (POPJ-AFTER-NEXT                                 ;次が終わったらリターン, ただし、もし
            POPJ-GREATER-OR-EQUAL M-2 A-ZERO)               ;除数が負なら、
           ((M-AC) SUB M-ZERO A-AC)                         ;商の符号を変更
```

The Bus Interface バス・インターフェース

バス・インターフェースは CADR マシンに 2 つのバス、Unibus と Xbus を接続する。Unibus は通常の PDP11 のバスであり、周辺機器、特に PDP11 ラインのために設計された商用デバイス、を接続するのに使用される。Xbus はメモリとディスクのような高性能周辺デバイスを接続するために使用される 32bit バスである。バス・インターフェースは、マシンのオペレーションを制御するために、PDP10、PDP11、他の Lisp マシンのような Unibus オペレータを許可する診断インターフェース、Unibus と Xbus からプロセッサへ割り込みを渡すハードウェア、Xbus を調停するロジック、そして Unibus 上に PDP11 が存在しないので Unibus の調停を行うロジック、を含む。

バス・インターフェースは、CADR マシンが Xbus 上のメモリと Unibus 上のデバイスをアクセスできるようにし、独立した Xbus 上のデバイスが Xbus (だけ) にアクセスできるようにし、Unibus デバイスが Xbus メモリに (Unibus アドレス空間は十分に大きくないので、map を通して) アクセスできるようにする。Unibus が Xbus にアクセスする時、32bit ワードを 16bit ワードのペアに変換するためのバッファが提供される。

CADR マシンは 32bit ワードの 22bit 物理アドレス空間を見る。これの先頭の 128K、番地 17400000-17777777, は Unibus を参照する。各 32bit ワードは 16bit の Unibus ワードをビット 0-15 に持ち、ビット 16-31 は 0 である。Unibus 上のバイト・アドレシングの使用も、リード-ポーズ-ライト (読み出し、待ち、書き込み) サイクルも、提供されていない。Unibus のすぐ下の 128K、番地 17000000-17377777, は、Xbus I/O デバイスのために予約されている。番地 0-16777777 は Xbus メモリである。

バス・インターフェースは Unibus の様々な機能を制御するいくつかの Unibus レジスタを含む:

スパイ機能 (Spy Feature)

Unibus 番地 766000-766036 は、Spy 機能のために使用される。詳細はどこかよそに書かれている。これらの番地は、CADR マシン中の様々な内部信号を読み出し、書き込み、そして、マイクロコードのロードと診断のために必要なフックを提供する。

2 マシン・間に合わせ (Two-Machine Lashup)

メンテナンス目的で、2 つのバス・インターフェースは、一つの 50 芯フラット・ケーブルでお互いにケーブル接続されるだろう。一つのマシンは、デバッガであり、他のマシン、すなわちデバッグ対象、での Unibus での読み出しと書き込みを実行できる、Unibus 上のレジスタを通して (スパイ機能のような)、デバッグ対象に診断と検査がなされる。デバッグ対象の Unibus マップ (下に記述) を使用することにより、デバッグ対象の Xbus を検査することもできる。デバッガの Unibus 上の次の番地はこの機能を制御する:

766100

後述のレジスタによってアドレスされるデバッグ対象の Unibus の番地を読み書きする。

766114

(書き込みのみ) デバッグ対象の、アクセスされるべき、Unibus アドレスの 1-16 ビットを持つ。アドレスのビット 0 は常に 0。

766110

(書き込みのみ) 付加モディファイア・ビットをもち、次のごとく。これらのビットはデバッグ対象の Unibus がリセットされた時、0 にリセットされる。

1 デバッグ対象 Unibus アドレスのビット 17

2 デバッグ対象の Unibus とバス・インターフェースをリセットする。ここに 1 を書き、その後 0 を書く。

4 タイムアウト禁止。これはデバッグ対象バス・インターフェースによって行われる (デバッガによって指令されるものだけではなく) 全 Xbus と unibus サイクルの NXM タイムアウトをオフにする

766104

（読み出しのみ）これらはデバッグ対象のバスで実行されるバス・サイクルのステータスを持つ。これらのビットはデバッグ対象の Unibus での 766044（エラー・ステータス）への書き込みでクリアされる。これらは電源投入ではクリアされない。これらのビットは下の"エラー・ステータス"に記述されている。

エラー・ステータス（Error Status）

766044

この番地の読み出しは、前のバス・サイクルから累積したエラーの状態ビットを返す。この番地への書き込みは、書き込みデータを無視し、ステータス・ビットをクリアする。注意、電源投入ではこれらのビットはクリアされない。

1 Xbus NXM エラー。応答が無く Xbus サイクルがタイムアウトした時、セットされる。

2 Xbus パリティ・エラー。Xbus 読み出しがバッド・パリティとともにワードを受け取り、Xbus パリティ無視－線をアサートしていなかった時、セットされる。パリティ・エラーは Xbus NXM エラーによってもセットされる。

4 CADR アドレス・パリティ・エラー。プロセッサから受け取ったアドレスがバッド・パリティを持っていた時、セットされる。プロセッサとバス・インターフェースの間の通信のトラブルを示す。

10 Unibus NXM エラー。応答が無く Unibus サイクルがタイムアウトした時、セットされる。

20 CADR パリティ・エラー。プロセッサから受け取ったデータがバッド・パリティを持っていた時、セットされる。プロセッサとバス・インターフェースの間の通信のトラブルを示す。

40 Unibus マップ・エラー。Unibus マップを通して Xbus サイクルを実行しようとし、マップが不正か書き込み禁止（write-protected）に指定されていて、それが拒絶された時に、セットされる。

残りのビットはランダム（0 である必要がない）である。

割り込み（Interrupts）

バス・インターフェースは CADR マシンに Unibus 上の割り込みをさばかせるようにする、PDP11 が無い時。PDP11 が存在するとき、それのプログラムが割り込みを透過な方法で CADR マシンにフォワードする。Xbus は CADR マシンに割り込みを掛けられる。次の Unibus 番地が、割り込みと Unibus 調停機を制御する：

766040 この番地の読み出しは割り込みステータスを返す，次のような：

1 割り込み受付を禁止。もしこれがセットされていたら、Unibus 調停機は BR4， BR5， BR6， BR7 要求を受け付けない。NPR 要求の受付は続ける。電源投入で 0 に。

2 ローカル許可（読み出しのみ）。1 はバス・インターフェースが Unibus を調停することを意味し、0 は、バス上に PDP11 が存在し、それが調停をすることを意味する。

1774 ビット 9-2 が、バス・インターフェースによって受付けられたか、または PDP11 プログラムによってシミュレートされた、最後の Unibus 割り込みのベクトル・アドレスを持つ。

2000 Unibus 割り込みをイネーブル。ここを 1 にすると、バス・インターフェースが Unibus 割り込みを受け付けた時、ビット 15（Unibus 割り込み）がセットされるようになる。このビットは電源投入ではリセットされない。

4000 割り込み受付を停止(Interrupt Stops Grant)。ここを1にすると、バス・インターフェースがUnibus割り込みを受け付けた時、ビット0(割り込み受付を禁止 Disable Interrupt Grant)がセットされるようになる。つまり、CADRマシンが最初の割り込みを処理し終わるまで、より一層の割り込みを防止する。このビットは電源投入ではリセットされない。

30000 ビット13-12はUnibusの受付の用途のための"割り込みレベル interrupt level"である。通常のPDP11のレベルに対応している:0->0, 1->4, 2->5,3->6。レベル7をシミュレートするために、「割り込み受付を禁止 Disable Interrupt Grant」をオンにする。これらのビットは電源投入ではリセットされない。

40000 Xbus割り込み(読み出しのみ)。このビットはXbusの割り込み要求ラインである。

100000 Unibus割り込み。1はUnibus割り込みが、バス・インターフェースによって受け付けられたか、PDP11プログラムによってシミュレートされて、CADRプログラムによる処理待ちであることを示す。このビットは電源投入でクリアされる。注意、CADRマシンへの割り込み要求信号はビット14と15のORである。

766040

この番地への書き込みは、上のレジスタのビット0と10-13(マスクは36001)に書き込む。これは、"割り込みレベル interrupt level"を変更するためと、割り込み処理後にUnibus割り込みの受付を再度許可するために使用される。

766041

この番地への書き込みは、上のレジスタのビット2-9と15(マスクは101774)に書き込む。これは、Unibus割り込みのシミュレートと割り込み処理後にビット15(Unibus割り込み)をクリアするのに使用される。

上で述べていない番地766040と766136の間には、他の番地の複製があり、それらは使用してはいけない。

Unibusマップ(Unibus Map)

Unibus番地の140000-177777は16ページに分割され、それはXbus物理アドレス空間のどこにでもマップできる。各ページは16bitワードの512個か、32bitワードの256個であり、CADR仮想メモリのページ・サイズと同じである。最初の8ページはPDP11によってアドレス可能であり、次の8ページはPDP11のI/O空間の下に隠れている。Unibusマップは、Xbusへの診断パスと、メモリアクセスをするUnibus周辺機器の操作のための、両方に使われることを意図している。

各Xbus番地は4 Unibusバイト・アドレスを占める。1つの32bit Xbus番地への読み書きに、2つの16bit Unibusサイクルがかかる。2つのUnibusサイクルの間データを保持するために(各ページごとに1つの)16個のバッファが提供されている。各ページが、一つのバス・マスタにより使用される限り、物事は正しく起こるだろう。

付加機能として、Unibusマップを通したXbusアドレス17400000以上への書き込みは、CADRのMDレジスタへの書き込である。これは、診断目的の、プロセッサへの32bitパラレル・データ・パスを提供する。これらのXbusアドレスは違うことには使用できない。なぜなら、それらはプロセッサがUnibusをアドレスするのに使用されているから。

37777 Bits 13-0 contain the Xbus page number. These bits are concatenated with bits 9-2 of the Unibus address to produce the mapped Xbus address.

Unibus の 766140-766176 番地は 16 個のマッピング・レジスタを持つ。注意、これらは電源投入では内容はランダムであり、初期化ルーチンでクリアされるべきである。
ビット配置は：

100000 ビット 15 はマップ-バリッド（正当）ビット。もしこれが 0 なら、このマッピング・レジスタはセットアップされていなく、Unibus へ応答しない；NXM タイムアウトが発生し、エラーステータス・ビットがセットされるだろう。

40000 ビット 14 は書き込み-許可ビット。もしこれが 0 なら、このマッピング・レジスタは Unibus の書き込みに応答しない；NXM タイムアウトが発生し、エラーステータス・ビットがセットされるだろう。

37777 ビット 13-0 は Xbus ページ番号を持つ。これらのビットは、Unibus アドレスのビット 9-2 とつながれて、マップされる Xbus アドレスを生成する。

The Xbus

Xbus は CADR プロセッサで標準の 32bit 幅のデータ・バスである。メイン・メモリと高速周辺機器、ディスク・コントロールや TV ディスプレイのようなものは Xbus にインターフェースされている。Xbus の制御は Unibus と同様で、転送は明確にタイミング合わせをされ（デバイスが関係する限り）非同期である。バスは両端で、390 Ω の抵抗でグランドに 180Ω で+5v にプルアップされ、123Ω で+3.42V にターミネートされている効果がある。グランドでは、各ターミネーションは 28mA （ミリ・アンペア）が流れ、トータルで 56mA の負荷である。バスはオープン・コレクタで、どのデバイスも 56mA の付加をハンドリングする能力を持ち、バスを駆動するだろう。お勧めのドライバは AMD 26S10 で、これはバス・レシーバでもある。

典型的な読み出しサイクルは、転送のためのアドレスを-XADDR ライン上に置き、アドレスのパリティを -XBUS.ADDRPAR 線上に置いて、始まる。それから -XBUS.RQ 線が low にされ、要求が初期化される。
応答するデバイスは、要求されたデータを 32 本の -XBUS 線とそのデータのパリティを-XBUS.PAR 線上に置く。パリティを生成するのが簡便でないデバイスでは、（I/O レジスタの場合のような時）、バス・マスタに転送がパリティ補正をチェックしないように知らせるために、デバイスは、 -XBUS.IGNPAR をアサートする。（※訳注：ここでの「アサート」とは、ハードウェア信号をアクティブにすること。） そして、応答するデバイスは -XBUS.ACK をアサートし、-XBUS.RQ 信号がマスタによって消されるまで、それをアサートし続ける。

マスタが、-XBUS.WR を、そして-XBUS 線に書かれるべきデータをアドレス線とともにアサートすることを除いて、書き込み要求も同様に進む。すべてのバス・マスタは、書き込みでは良いパリティ・データの生成を要求される。

遅延の補正（デ・スキュー）はバス・マスタの責任である。特に、-XBUS.RQ をアサートするに先立って、バス・マスタには、正しくアドレス線、書き込み線、データ線を 80ns アサートする責任がある。これらの線は、マスタへの -XBUS.RQ が落ちたことに対する応答として、-XBUS.ACK 信号が落ちるまで、保持されなければならない。
応答するデバイスは、-XBUS 線上で読み出しデータをドライブするのと同時に、-XBUS.ACK をアサートすることができる。そして、マスタは、-XBUS.ACK を受け取った後 50ns の遅延を必ず置き、-XBUS.RQ を落とし、データをストローブ（※訳注：データ線上にデータを出力すること）する。応答するデバイスは、 -XBUS.RQ がアサートされなくなった後、即座に-XBUS.ACK を落とす必要がある。

CADR プロセッサと Unibus 要求の間の通常のバス・マスタ調停は、バス・インターフェースによって取り扱われる。バス・マスタになるべき Xbus 上のデバイス、ディスク・コントローラのような、は、-XBUS.EXTRQ 信号をアサートすることによってそれを行う。バスがフリーになる時、バス・インターフェースは -XBUS.EXTGRANT をアサートして応答する。この信号は、Xbus 上のバス・マスタ・デバイスの間を、-XBUS.EXTGRANT.IN ピンから入力され、-XBUS.EXTGRANT.OUT ピンから出力されるデイジーチェーン（※訳注：数珠つなぎ）で結んでいる。各デバイスでは、次のデバイスに受付（グラント）を渡すか否かを決定が行われる。
Unibus の構造とは違い、グラントを渡すか、そしてバス・マスタになる動作をするかの決定は、-XBUS.SYNC 線で分配されているマスタ・クロック信号に同期して起こる。

デバイスが要求を初期化する時、即座に -XBUS.EXTRQ をアサートする。-XBUS.SYNC の立下りエッジが、クロックとなり、要求信号を REQ.SYNC と呼ばれる D-フリップフロップに入れる。-XBUS.EXTGRANT.IN がロー（low）になるとき、デバイスは、-XBUS.EXTGRANT.OUT をアサートする、が、ただし、REQ.SYNC フリップフロップをセットされたか、またはすでにバス・マスタになっているか、ではない場合に。次の-XBUS.SYNC の立下りエッジで、 -XBUS.EXTGRANT.IN と REQ.SYNC の両方をセットするデバイスはバス・マスタになる。
デバイスは即座に、-XBUS.BUSY をアサートしなければならず、転送のためのアドレス線を即座にアサートし始めるだろう。

スレーブ・(下位の)デバイスが、マスタの要求への応答で -XBUS.ACK を落とした後に、-XBUS.BUSY は非同期的に落とされるだろう。

-XBUS.EXTGRANT.IN 信号は、各デバイスで、単純に -XBUS.EXTGRANT.OUT.に接続されていなければ、180 オームの抵抗で+5v にプルアップでターミネートされなければならない。

XBUS 信号概要:

Data Lines (データ線):
  -XBUS<31:0> 32 本のデータ線、データが 1 の時ロー(low)。
  -XBUS.PAR 32 本のデータ線のパリティ、書き込みで必要。
  -XBUS.IGNPAR パリティ無視信号、読み出しのどんなデバイスでもアサートできる。

Address Lines(アドレス線):
  -XADDR<21:0> 22 本のアドレス線、アドレス・ビットが 1 の時ロー(low)。
  -XADDR.PAR アドレスの奇パリティ。

Cycle control lines(サイクル制御線):
  -XBUS.RQ 読み出しか書き込み要求で、マスタによってアサートされる。-XADDR, -XBUS.WRITE, -XBUS data の安定のために引き続く最小 80ns が必要である。
  -XBUS.ACK スレーブによって、-XBUS.RQ の応答でアサートされる。正しい読み出し時の引き続くアサートには、なんの遅延も必要ない。
  -XBUS.WR 書き込みサイクルの間、マスターによってアサートされる。

Mastership control lines(マスタ権制御):

  -XBUS.BUSY バス・インターフェース以外のデバイスがバス・マスタの時アサートされる。バス・インターフェースだけがこの線を検査する。-XBUS.SYNC クロックのエッジで、アサートされ、-XBUS.ACK が落ちた後、非同期的に落とされる。
  -XBUS.EXTRQ バス・インターフェース以外のデバイスがバス・マスタになりたい時、アサートされる。非同期的にアサートされ、デバイスがマスタになった後、非同期的に無くなる。ただし、-XBUS.BUSY.が落ちる前を除いて。
  -XBUS.EXTGRANT.IN デイジーチェーンのマスタ権が受け付けられた信号。+5V に 180Ω の抵抗でプルアップされなければならない。
  -XBUS.EXTGRANT.OUT バス・インターフェースによって最初に、-XBUS.SYNC のエッジと同期してアサートされる。この信号は、-XBUS.SYNC に同期していない-XBUS.EXTRQ のクロック化版であるから、同期機構(シンクロナイザ)の不調(lossage)は信号の問題であろう。

Miscellaneous(その他):

  -XBUS.INIT ローの時、すべてのデバイスをリセットする。これは電源 ON 時と OFF 時の間、またマシンがリセットの時、ローである。
  -XBUS.SYNC マスタ権引渡しと他の必要な用途のための同期用クロックである。バス・マスタになるデバイスは、この信号のエッジで同期する。要求は通常、エッジに 80ns 続く。
  -XBUS.INTR これをローにドライブすると、割り込み要求である。非-割り込み許可状態に初期化することを全デバイスが要求され、そして、そのデバイスからの割り込みを選択的に許可できる割り込み許可ビットと禁止ビットを持つことを要求される。"割り込み要求中 requesting interrupt"状態はデバイス制御レジスタのビットの一つで読み出し可能でなければならない。
  -XBUS.POWER.OK 電源が安定しているとき、この線がハイ(HIGH)。電源が立ち上がった後 3 秒間ローになり、また、電源がオフになる前の 3 秒間ローになる。

Error Checking エラーチェック

CADR マシンの中の全部の内部メモリはパリティ・チェックを持っている。もし、悪いパリティが検出され、パリティ・チェックがイネーブルならマシンは停止する。停止する前に、プロセッサは常に現在の命令を完了する、そして次の命令を IR にラッチ(clock)させる。これは、タイミングを簡単にさせ、スクラッチパッド・メモリのラッチを空け(※訳注:つつぬけにし)ながら停止させる。これは、一度マシンが停止したら、悪いパリティのデータは、もはやバス上にはないことを、意味する。加えて、一つの誤った命令が実行されているだろう。何が起こったかを推測するのに、OPC レジスタが助けになるだろう。

最初の電源投入時、エラー停止は、禁止になっている。が、しかし、ブートストラップ・プログラムが全部の内部メモリを初期化したら、即座にエラー停止を許可することが期待されている。

メイン・メモリのパリティはチェックされ、マシンを停止したり、マイクロコードのトラップを起こしたり、無視したり、することができるが、それは診断インターフェース内のモード・フラグによって制御される。

データパスはなんの冗長チェックも持っていない。マシンがブートストラップされる時、それは、メモリとデータパス中の固定的な障害を検出するように設計された単純な診断を走行させる。

Self Bootstrapping 自己ブートストラップ

マシンが電源投入された時、それ自身とUnibusをリセットするが、自動的にはスタートアップしない。ブートストラップ・シーケンスはいくつかの方法のうちの一つで初期化される。診断インターフェースは、一つの指令ができる。診断ディスプレイ・パネルは、1本の線をグランドに落とすことによって、一つをスタートできる。これはプッシュ・ボタンに接続するものである。バス・インターフェースは、1本の線をグランドに落とすことによって、ブートストラップをスタートできる。
Chaosネットワーク・インターフェースは、もし、それがネットワークからあるメッセージの列を受け取ったら"push the boot button."する。
I/Oボードはキーボード・コマンドの特別な組み合わせ(左と右のControl-Meta)を、ブート信号と認識する。左-右Control-Metaと供にタイプされる文字は、ソフトウェア・オプションの選択のためのブートストラップで利用できる。

ブートストラップ・シーケンスはマシンをリセットすることによってスタートする、もしそれが走行していたら、それを止めるだろう。それがRUNになったら、クロックを止める以外何もできない。それは、マシンをもっとも遅いスピードに設定し、パリティ・トラップとエラー停止と統計カウンタを禁止し、そして、PROM(読み出し専用)制御メモリをイネーブルする。ブート信号の立ち上がりエッジでスタートのクロックが許可され、メモリ・パリティ・トラップと同じように、マイクロコード番地0へのトラップが発生する。PROMの番地0が制御を受けとる。それは、内部メモリの全部のクリア(正しいパリティで埋める)を行わねばならない、(最初にそれを使う前に)Unibusのリセットを行い、エラー停止を許可し、マシンのスピードを通常値に設定し、マシンがある領域で確かに動作しているかをチェックするいくつかの診断チェックを走らせ、ディスクからマイクロコードをロードし、そして、
Unibusを超え、診断インターフェースを操って、そのスタート・アドレスから通常のマイクロコードへ制御を移す。

もし自己テスト診断が失敗したら、マイクロコードはループに入り、問題が何であるか判断できるように、PCの値は診断ディスプレイから読み取ることができる。

Interrupts and Sequence Breaks 割込みとシーケンス・ブレーク

割込みは、マイクロコードへのハードウェアの信号である–通常、マイクロコードはデータをメイン・メモリ中のバッファへデータを出し入れする転送を行う。信号が全部が Lisp でのコードの注意を引きたい時、シーケンス・ブレークが引き金になる。これは、A メモリ中のシーケンス・ブレーク・ペンディング・フラグ (sequence-break pending flag)をセットする、そして、もし、(やはりA メモリ中の) シーケンス・ブレーク延期フラグ (defer-sequence-break flag)がセットされていなければ、ハードウェアのシーケンス・ブレーク・フラグをセットする、ということで構成されている。このフラグはマイクロ命令フェッチのような、あちこちのやり易いポイントでテストされる。そして、マイクロコードがフラグをオフにして、シーケンス・ブレーク・ルーチンに入るということを、引き起こす。シーケンス・ブレーク・フラグは、ページ・フォールトと割込みをテストするのと同じジャンプ命令によってテストされる。

割込みは、Xbus と Unibus の両方で生成されることができる。バス・インターフェースの節で詳しいプロトコルが記述されている。

シーケンス・ブレークは、(Lisp プログラムである)スケジューラを走らせる必要があることを、示すソフトウェア信号である。シーケンス・ブレークは、条件が真 (true)になるのを待っているプロセスに条件変化を示唆する。スケジューラは全プロセスの走行可能性をチェックし、また、周期的に行動を実行する時刻が来たかをチェックするが、それは全プロセスについてではない。Lisp プログラムは、最低レベルの実時間応答性を保護するために、割込みを許している間も、きわどい領域を守るために、シーケンス・ブレークを遅延させることができる。

Lisp マシン・ソフトウェア環境では仮想記憶へのアクセスはプリミティブ操作に見える。メモリ・データの実際の番地に無頓着に、それらへのフェッチは継続する。この見方は、システムのコーディングをかなり単純にしている、しかし、シーケンス・ブレークへの応答に、高い潜在的レイテンシを穏やかに押し付けられている。
割込みはマイクロコード・レベルで完全に取り扱われ、これらのための応答時間は極めて短い。

割込み制御レジスタ、機能デスティネーション 2 で書き込み可能で LC (機能ソース 13) の上位ビットで読み出せる、は割込みに関する 3 つのビットを持っている。Bit<27>、割込み許可 (INTERRUPT ENABLE)、は JUMP 命令によって見られる、バス・インターフェースからの外部割込み信号を許可する。
Bit<26>, シーケンス・ブレーク (SEQUENCE-BREAK)、は JUMP 命令によってテスト可能なシーケンス・ブレーク・フラグ (sequence-break flag)である、

Bit <28>, バス・リセット (BUS-RESET)、は 1 を書きその後 0 を書かれた時に、
Unibus 上 (BUS INITL) と Xbus 上 (XBUS.INIT L) のリセット信号を生成し、バス・インターフェースをリセットする。電源投入時も、マシンはバスをリセットする。

Bit <29>は命令ストリーム機能のために使用される。

The Statistics Counter 統計カウンタ

統計カウンタは 32bit のカウンタであり、命令のビット 46=1 のものが実行されたら、いつでもインクリメントされる。カウンタが-1 から 0 へオーバフローした時、オーバフローを起こした命令の実行が完了した後、マシンは停止する。(停止は診断インターフェース中の許可ビットの制御下にある。)
PROM からの命令のビット 46 は常に 0 である。

統計カウンタは診断インターフェースを使用して読み書きすることができる。それは、いくつかの機能を提供している。

それはいくつの命令が実行されたかの測定のための計量に使用でき、またマイクロプログラムのいくつかのサブセットに限定することができる。マイクロコード・デバッガとコンソール・プログラムは、制御メモリ領域内の統計ビットのセットとクリアのコマンドを持つ。

カウンタを-1 にセットし、ブレークポイントをセットされるべき命令の統計ビット(statistics bit)をオンにすることで、ブレークポイントに使用することができる。

制御メモリの全番地の統計ビットをセットし、エラーが発生する丁度直前でマシン停止を起こす適切な数値を統計カウンタにセットすることによって、疑わしいマイクロコードをシングル・ステップで通すことができる。
それによって不明確なバグを見つけることに使用できる。

統計カウンタは、通常の診断バスではなく、命令書き込みレジスタ(Instruction Write Register)からロードされる、それは 32bit 幅であるから。1 クロック・サイクルの遅れと共に、M バスからそれが実効的にロードする。たぶん、マシンはそれ自身の統計カウンタを使用するのは不可能だろう、しかし賢い方法も見つかるかも知れないが。

The Diagnostic Interface 診断インターフェース

診断インターフェースは、16 個の Unibus アドレスを占める。それは、マシンの様々な部分を読み書きするのに使用することができる、16bit の診断バスを含んでいる。
16 個の読み出し番地、8 個の書き込み番地がある。同じアドレスの読み出し番地と書き込み番地は、お互いに無関係である。診断バスは、デバッギングとメンテナンスのプログラム、"コンソール(console)"プログラムを含む、とマシン自身がブートストラップ中にマシンによっていくつかのケースの中で、使用される。

まず、読み出し番地を記述する。これらは時々"スパイ機能(spy feature)"と呼ばれる。自然なこととして、マシンが走行中に読んでみても、これらの多くは意味がない。

766000 IR<15-0>. 現在実行中の命令の下位 16bit。
766002 IR<31-16>. 現在実行中の命令の中位の 16bit。
766004 IR<47-32>. 現在実行中の命令の上位の 16bit。
766006 不使用
766010 OPC. OPC は下に記述
766012 PC. 現在のプログラム・カウンタ、次に実行されるべき命令のアドレス。
766014 OB<15-0>. 出力バスの下位半分。
766016 OB<31-16>. 出力バスの上位半分。

766020 フラグ・レジスタ 1。これはマシンの開始と停止に関係ある様々な信号を提供する。ハードウェア・エラーのよりマシンが停止する時、このレジスタが何を起こったのかを示す。ビットは:

<15> = -WAIT. 1 ならマシンは走行中か、走行可能、0 ならメモリの待ち。WAIT の正確な意味については、クロックの議論を参照。
<14> = -V1PE. 通常 1、0 なら最後のクロックで、レベル 2 マップにパリティ・エラーが出た。
<13> = -V0PE. 通常 1、0 なら最後のクロックで、レベル 1 マップにパリティ・エラーが出た。
<12> = HIGHOK. 1 ならマシンのハイ・ラン(※訳注:ハイ電圧であるべきライン)はすべて正常、0 は、いくつかがそうではない。これは、基本的な電源チェックと、配線の断線チェックである。
<11> = -STATHALT. 通常 1,0 なら統計カウンタによりマシンが停止されている。
<10> = ERR. 1 ならエラー状態がある。もしモード・レジスタ中の ERRSTOP が ON なら、マシンは停止させられる。
<9> = SSDONE. 1 なら、シングル・ステップ動作が完了。
<8> = SRUN. 1 なら、マシンが走行しようとしている(しかし、パリティ・エラーか、ウェイト条件か、統計カウンタによって停止させられているだろう)
<7> = -HIGHERR. 1 なら、最後のクロックで HIGHOK があった。
<6> = -MEMPE. 通常 1, 0 ならメイン・メモリ・パリティ・エラーがあり、最後のクロックのトラップでは捕らえられていない。
<5> = -IPE. 通常 1, 0 なら最後のクロックで、制御メモリ・パリティ・エラーがある。
<4> = -DPE. 通常 1, 0 なら最後のクロックで、ディスパッチ・メモリ・パリティ・エラーがある。
<3> = -SPE. 通常 1, 0 なら最後のクロックで、SPC スタック・パリティ・エラーがある。
<2> = -PDLPE. 通常 1, 0 なら最後のクロックで、PDL-バッファ・パリティ・エラーがある。
<1> = -MPE. 通常 1, 0 なら最後のクロックで、M-スクラッチパッド・パリティ・エラーがある。
<0> = -APE. 通常 1, 0 なら最後のクロックで、A-スクラッチパッド・パリティ・エラーがある。

766022 フラグ・レジスタ 2。このレジスタはパイプラインと、デバッグ・プログラムが見たがる、いいろいろな制御信号に、関係したフラグを持っている。
ビットは：

<15> = 不使用
<14> = 不使用
<13> = WMAPD. 前のサイクルがマップへの書き込みをいい、このサイクルでそれを行うだろう。
<12> = DESTSPCD. 前のサイクルが、機能デスティネーションを使用した SPC スタックへの書き込みをした(CALL 移行とは対照的に)。
<11> = IWRITED. 前のサイクルが I-MEM WRITE 型の JUMP 命令を行い、そして、このサイクルが制御メモリを書き込み、RETURN 移行を行い、それに続くサイクルは NOT であろう。
<10> = IMODD. 前のサイクルがこのサイクルの命令を変更するのに"OA register"を使用したか、このサイクルの命令は DEBUG-IR(下を見よ)から来た。このフラグは IR のパリティ・チェックを禁止する。
<9> = PDLWRITED. 前のサイクルが PDL-buffer への書き込みを起こし、
このサイクルもそうである。
<8> = SPUSHD. 前のサイクルが SPC スタックへの書き込みを起こし、このサイクルで行うであろう。
<7> = 不使用。
<6> = 不使用。
<5> = IR<48>. IR のパリティ・ビットである。
<4> = NOP. 現在 IR にある命令は実際には実行されようとしていない；このサイクルは NOP サイクルである。
<3> = -VMAOK. 最後のメイン・メモリ・サイクルの開始は、マップがページ・フォールトを示していたため、成功しなかった。
<2> = JCOND. 1 ならジャンプ条件が満たされた。IR 中の命令が JUMP 命令でなければ、無意味。
<1-0> = PCS1-0. これらの 2 ビットは、次の PC(next PC(次の後の命令のアドレス))を選択する。
エンコード値は：
0 = SPC<13-0> SPC スタック。
1 = IR<25-12> JUMP 命令で指定されたアドレス。
2 = DPC<13-0> ディスパッチ・メモリ。
3 = IPC<13-0> PC+1。

766024 M<15-0>. 現在 IR 中にある命令によって選択された M ソースの下位半分。
766026 M<31-16>. M ソースの上位半分。
766030 A<15-0>. 現在 IR 中にある命令によって選択された A ソースの下位半分。
766032 A<31-16>. A ソースの上位半分。
766034 ST<15-0>. 統計カウンタの下位半分。
766036 ST<31-16>. 統計カウンタの上位半分。

診断インターフェースの書き込みレジスタの説明。

766000 DEBUG-IR<15-0>. 診断インターフェースによって与えられる命令の下位 16bit。

766002 DEBUG-IR<31-16>. 中位 16bit。

766004 DEBUG-IR<47-32>. 上位 16bit。

766006 クロック制御レジスタ。マシンのリセットが、これを 0 にセットする。
次のビットがある:

<4> = LDSTAT. 1 にすることで、マシンをクロックさせ(動かし)、前のクロックで M バスからロード済みの IWR<31-0>から統計カウンタをロードさせる。(※訳注: クロックする、というのは、クロックを与え、マシンを動作させることである)

<3> = IDEBUG. 1 にセットすることで、マシンがクロックされたときに、PROM や制御メモリの代わりに、DEBUG-IR から IR をロードする。診断インターフェースを通してマシンを操る第一番の方法が、この機構を使って命令を実行させることである。

<2> = NOP11. これを 1 にすることで強制的に NOP。これは、マシンをクロックさせることができるようにする。具体的には、IR の現在の内容が命令を実行して望まない副作用を起こすようなこと無しに、DEBUG-IR を IR へ転送するなど。NOP11 は PC が変更を受けるを妨げない(実際、それはインクリメントされるだろう)、また、発生した(happening)ときにスケジュールされパイプラインに入った書き込みを妨げない。

<1> = STEP. 1 にすることで、SSDONE が 0 の時、プロセッサは走行のために 1 サイクルだけクロックされ、そして SSDONE がセットされる。STEP を 0 にセットすると SSDONE はクリアされる。(これらの両方の操作は、実際には完了するのにいくつかのクロックのサイクルを要する。) STEP は診断インターフェースがマシンを"クロック"するための方法である。注意として、マシンが停止している時であろうと、メイン・クロックは常に走行している。STEP は、メイン・クロックに同期した、単一のプロセッサのクロックを生成する。

<0> = RUN. 1 にすることで、マシンの走行を開始させる。まず STEP を使用して、全レジスタとメモリ、PC、IR の状態を準備し、そして RUN に切り替える。最初に実行される命令は、IR に置いたものである。

766010 OPC 制御レジスタ。マシンのリセットはこれを 0 にする。このレジスタは、セーブされた状態からマシンの状態を完全に復帰させるためにコンソール・プログラムによって使用される必要のある、いくつかのビットを含んでいる。そのビットは:

<2> = OPCINH. 1 にすることで、プロセッサ・クロックによって、OPC がクロックされることを禁止する。このビットはクロックがハイの時(すなわち、マシンがストップしている)以外、変化させてはならない。OPC を復帰するプロセスは、OPCINH の設定、JUMP 命令を実行することによって PC へ入る 8 つの値を得て、そして OPCCLK ビットを経由して OPC へそれらの値を転送する、ということから成っている。OPC が一度復帰されても、残りのマシン状態が復帰されている間、それらが邪魔されないように OPCINH はセットされたままである。マシンをスタートする直前に、OPCINH を 0 にする。

<1> = OPCCLK. これを 1 にした後 0 にセットすると OPC だけへ 1 クロックを生成する。これは、残りのマシン状態を乱さずに、8 つの OPC レジスタを読み出すのに使用される。

<0> = LPC.HOLD. これを 1 にセットすると、マシンがクロックされる時に、PC レジスタから LPC レジスタにロードされることを防ぐ。これは LPC を復帰するのに使用される。LPC は第 1 OPC レジスタのコピーであり、DISPATCH 命令の IR<25>機能が使用する。

766012 モード・レジスタ。マシンをリセットするとこれは 0 にセットされる。このレジスタは様々な機能を許可し、クロックの速度を制御する。
ビットは：

<7> = PROG.BOOT． これを 1 にセットすると、ブートストラップ・シーケンスが開始する。
<6> = PROG.RESET． これを 1 にセットすると、マシンをリセットする。リセットは、RUN をクリアすることでマシンをストップし、RESET 操作が終わるまでクロックを強制的にストップし、次の命令中で起こるべきことを引き起こすパイプライン・フラグをクリアし、そして、クロック(Clock)、モード(Mode)、診断インターフェースの OPC レジスタをクリアする。
<5> = PROMDISABLE． ここに 1 で PROM を禁止する。ここが 0 で、制御メモリの最初の 1K 番地を PROM と置き換える。
<4> = TRAPENB． ここが 1 でメイン・メモリ・パリティ・エラーが 0 番地へのマイクロコード・トラップを起こすことを許可する。ここが 0 でメイン・メモリ・パリティ・エラーは他のパリティ・エラーと同様に扱われる。
<3> = STATHENB． ここが 1 で、統計カウンタのオーバーフローでマシンが停止することを許可する。
<2> = ERRSTOP． ここが 1 でハードウェア・エラー(HIGHERR と様々なパリティ・エラー)がマシンを停止することを許可する。0 で、のん気に継続する。
<1-0> = SPEED<1-0>． これらのビットはクロックの速度を制御する。マイクロ命令中の ILONG ビットも速度に影響を与え、それを 40 ナノ秒に落とすことでゆっくりにする。

スピード・コードは：
0 = とても遅い
1 = 遅い
2 = 通常
3 = 速い

766014 不使用。

766016 不使用。

OPC は PC の最後の 8 つの値を記憶する 8 個のレジスタの集合である。これは、デバッグのための役に立つ履歴を提供する。いくつかのトラップ・ハンドリング・ルーチン中では、マイクロコードそれ自身によっても使用される。8 つの OPC の最後は読み出し専用であり、それは 8 クロック前の PC である。
OPC のクロッキングに特別な制御が提供されているので、それらは di 無しで読み出しができ、よって、マイクロコード・デバッガによってセーブとリストアされることができる。これは、先述の 766010 の項に記述してある。

最大の柔軟性のために、OPC は、診断インターフェースによってと、機能ソースとして読み出すことができる。

バス・インターフェースは、特別なパスを提供し、それによって、MD レジスタがロードされるであろう。
これは診断入力データの並列ソースを提供する。MD をロードした後、データを所望のデスティネーションへ送るために、DEBUG-IR を通して命令は実行させられることができる。

機械の周りに散らされたいくつかのメンテナンス・インジケータ(発光ダイオード)がある。フロント・ドアの内側、左手下方の角あたりに、5 個のオクテット(日の字型, 7 セグメント)ディスプレイがある。これらは現在の PC の値を示している。これらのディスプレイの小数点は様々な興味深い状態を表示する。左から右へ:

1 - PROMENABLE. 現在の命令が WCM(writable control memory 書き込み可能制御メモリ)ではなく、PROM から来ていることを示す。

2 - IPE. 最後のクロックで制御メモリにパリティ・エラーがあったことを示す。

3 - DPE. 最後のクロックでディスパッチ・メモリにパリティ・エラーがあったことを示す。

4 - TILT0. 最後のクロックでマップかメイン・メモリにパリティ・エラーがあったことを示す。

5 - TILT1. 最後のクロックで、A-スクラッチパッド, M-スクラッチパッド、PDL バッファ、SPC スタックのどれかが、パリティ・エラーがあったことを示す。

また、様々なエラー状態、"機械が本当に走行しているか"、ディスク・インターフェースの状態のためのインジケータも提供されている。このインジケータ・パネルの位置と、すべてのマシンがそれを持っているかは、まだ決定されていない。

The Disk Controller ディスク・コントローラ

Lisp マシン・ディスク・コントローラは、「Traident」ファミリのディスク・ユニットの 1 個から 8 個を、CADR マシンの XBUS に取り付ける。1 ユニット版は 1 つのボードからなり、1 つより多くのディスク・ユニットが使われる時には、2 番目のボードが追加される。2 つの版はおおよそプログラム互換性がある。

Interface Registers インターフェース・レジスタ

ディスク・コントローラは XBUS 上の 4 つの 32bit レジスタを読み書きして操作される。それらは通常、物理アドレスの 17377774-17377777、Unibus のすぐ下にある。ジャンパを変更することで、アドレスを変えることができる。これらのレジスタの中の多くのビットが「選択されたユニット (selected unit)」として参照される。
ディスク・アドレス (disk-address) レジスタの<30:28>ビットに現在ある数字のディスク・ユニットが「選択されたユニット」である。

読み出しの場合、レジスタは：

0 STATUS (ステータス)

<24:31> The block-counter of the selected unit. 選択されたユニットのブロック・カウンタ。現在の回転位置を示す。このレジスタの読み出しはそのインクリメントと同期していない、よって、必ず、2 回読み出して、その 2 つが同じ値であるかをチェックしなければならない。

<23> Internal Parity Error. 内部パリティ・エラー。これは、ディスク上にあるビットのパリティとメモリ上にあるビットのパリティが一致しない事を示す； コントローラ内のどこかで何かが失われているに違いない。Read All と Write All コマンドは、
寄生 (まがいものの) 内部パリティ・エラーを起こす。もし、Read Compare コマンドが Read Compare Difference (bit22) をセットしディスク・データとメモリ・データのパリティが違っていたら、Read Compare コマンドは寄生 (まがいものの) 内部パリティ・エラーを起こす。このエラーは転送を停止しない。

<22> Read Compare Difference. Read Compare 差分。これは、メモリからのデータとディスクからのデータの不一致を示す。このビットは、コマンドが read-compare でなければ未定義である。このエラーは転送を停止しない。

<21> CCW Cycle. CCW サイクル。このビットが ON になる時、メモリ・パリティ・エラーかメモリ不在エラーとの組み合わせで、データの読み書きではなく、CCW をフェッチする間に発生したエラーを示す。

<20> Nonexistent Memory Error. メモリ不在エラー。メモリ (または他の XBUS デバイス) が 15 マイクロ秒以内の応答に失敗したことを示す。このエラーは転送を停止する。

<19> Memory Parity Error. メモリ・パリティ・エラー。メモリ (または他の XBUS デバイス) から偶数パリティが読み出されたことを示す。このエラーは転送を停止する。

<18> Header Compare Error. ヘッダ比較エラー。ディスクから読まれたブロック-ヘッダが、期待される値を持っていなかったことを示す。ディスク・ヘッドが適切な位置に位置決めされなかったか、ディスクが正しくフォーマットされなかったか、ヘッダが正しく読めなかったかが理由である。このエラーは転送を停止する。

<17> Header ECC Error. ヘッダ ECC エラー。ブロック・ヘッダのエラー訂正コードのチェックに失敗したことを示す。残念なことに、ほとんどのヘッダ ECC エラーは代わりに、ヘッダ比較エラーを示す。これは修正可能か? このエラーは転送を停止する。ヘッダ ECC エラーは、ディスクの最後を超えた読み出しや書き込みを続けるように指定したときにも、発生する。

<16> ECC Hard. ECC ハード。エラー訂正コードがエラーを発見し、それが訂正不能であることを示す。ディスクから読んだデータが悪い、読み出しに再トライすべし。このエラーは転送を停止する。

<15> ECC Soft. ECC ソフト。エラー訂正コードがエラーを発見し、どのデータ・ビットがエラーか決定できたことを示す。プログラムは訂正できる、ECC レジスタを見ればどうやるかがわかる。
エラー訂正コードは、11 までの誤りビットのひとつの塊を訂正できる。
このエラーは転送を停止する。

<14> Read Overrun. リード・オーバラン。ディスクからやってくるデータが、それをメモリに格納するよりも速すぎることを示す。このエラーは転送を停止する。

<13> Write Overrun. ライト・オーバラン。
メモリがディスクに十分に早くデータを供給できないことを示す。
このエラーは転送を停止する。

<12> Start Block Error. スタート・ブロック・エラー。ブロックの開始(start-of-block)(セクタ・パルス)が有ってはならない時に出現したことを示す。ディスクが正しくないフォーマットをされたか、寄生セクタ・パルスが生成されたかである。このエラーは転送を停止する。

<11> Timeout Error. タイムアウト・エラー。ディスク操作が 2.5 秒より長く掛かったことを示す。このエラーは転送を停止する。

<10> Selected Unit Seek Error. 選択されたユニットのシーク・エラー。選択されたユニットがシーク操作で失敗したことを報告した。このエラーは転送を停止する。リキャリブレート(Recalibrate)コマンドを使用することによりエラーをリセットする。

<9> Selected Unit not On-line. 選択されたユニットはオンラインではない。ヘッドはロードされていないか、ディスクの電源が入っていないか、指定されたユニット番号のディスクが無いかである。このエラーは転送を停止する。

<8> Selected Unit not On-Cylinder. 選択されたユニットはオン・シリンダではない。一般的に選択されたユニットでシークが進行中であることを示す。エラーではない。もし、書き込み操作中にオフ・シリンダ(off-cylinder)に行けば、失敗が発生する。もし、読み出し中にオフ・シリンダになったら、たぶん、ヘッダ比較エラー(header-compare error)か ECC エラーが発生するだろう。

<7> Selected Unit Read-Only. 選択されたユニットは読み出し専用。ディスク上のスイッチの状態。注意、read-only 状態がスイッチの変更を反映できるのは、ドライブが選択されていない時だけである。ディスク・アドレス・レジスタ(Disk Address register)へのストアは、瞬間的にカレント・ユニットをデセレクト(非選択状態)にするので、スイッチからの read-only 状態の更新ができるだろう。read-only のディスクへの書き込みは

フォールトを起こす。

<6> Selected Unit Fault. 選択されたユニットはフォールト。ディスクの障害か、プログラムのエラーを示す。Trident マニュアルを参照のこと。このエラーは転送を停止する。Fault Clear コマンドか Recalibrate コマンド(またはその両方)を使用して、リセットする。このエラーはドライブ上のデバイス・チェック・ライト(Device Check light)を点灯させる。

<5> No Unit Selected. どのユニットも選択されていない。このエラーは転送を停止する。選択されたユニット番号にディスクが接続されていないか、ディスク・ユニットが電源オフか"degated (ゲートされていない)"時に発生する。

<4> Multiple Units Selected. 複数のユニットが選択されている。このエラーは転送を停止する。一つより多くのディスク・ドライブが選択されているか、誤ったディスクが選択されたことを示す。

<3> Interrupt Request. 割り込み要求。1 であると、ディスク・コントローラが-XBUS.INTR をアサートしていることを示す。

<2> Selected Unit Attention. 選択されたユニットからのアテンション。At Ease コマンドを使用してリセットする。アテンション(注意)は、シークの完了、リキャブレートの完了、ヘッドのイニシャル・ロード、シーク不完全エラー、緊急ヘッド・リトラクト(emergency head retract)を示す。"暗黙の"シークは、アテンションを発生しない。

<1> Any Attention. なにかのアテンション。どれかのユニットがアテンションを持っている。どのユニットかを見つけるのに、順番にそれらをセレクトしていかなければならない。

<0> Not Active. アクティブではない。0 ならばコントローラがビジーである、1 はコマンドを受け付ける準備ができている。

1 MEMORY ADDRESS メモリ・アドレス

<31:24> 不使用

<23:22> Disk type. ディスク・タイプ。00:Trident, 01:Marksman, 10:不使用, 11:Trident(古い制御)

<21:0> the address of the last memory reference made by the disk control. ディスク・コントローラによって最後になされたメモリ参照のアドレス。これは、もしステータス・レジスタの CCW サイクルが ON であれば CCW アドレスであり、そうでなければデータ・ワードのアドレスである。

2 DISK ADDRES ディスク・アドレス

<31> 不使用

<30:28> Unit number. ユニット番号。1 ユニット版では、常に 0。

<27:16> Cylinder number. シリンダ番号。T-80 は 815 シリンダを持つ。

<15:8> Head number. ヘッド番号。A-80 は 5 ヘッドを持つ。それを超えたとき、ヘッド番号の下 6bit だけが働く(これは、Trident のフィチャーである)。

<7:0> Block number. ブロック番号。T-80 は通常、トラックあたり 17 ブロックでフォーマットされる。"ブロック(Block)"は"セクタ(sector)とほぼ同義である。

転送がエラーで打ち切られた時、ディスク・アドレス・レジスタは、エラーが発生した時に転送しようとしていたブロックのアドレスを持っている。転送が普通に終了したとき、ディスク・アドレス・レジスタは最後に転送されたブロックのアドレスを持っている。

3 ERROR CORRECTION REGISTER エラー訂正レジスタ

<31:16>　Error pattern bits. エラー・パータン・ビット

<15:0> Error bit position+1.　エラー・ビット位置+1

ソフト ECC エラーが発生した時、このレジスタは最後のブロック転送のどこでエラーがあったかを示している。ディスク・アドレス・レジスタはエラーのあるブロックのディスク・アドレスを持ち、コマンド・リスト・ポインタは CCW を指し、その CCW はエラーのあるメモリ・ページを指す。エラー・パターンは、指定されたビット・アドレスにあるメモリの内容と XOR されなければならない;ビット・アドレスはワード境界を超えてオーバラップすることがあるだろう。注意として、ビット・ポジションには 1 だけ下駄をはかせる。ブロックの最初のビットが bit1 である。

転送がアクティブな間は、どのレジスタにも書き込んではいけない。ただし、転送を途中で停止するために、Reset コマンドを使用するときだけは構わない、が、あなたは負けを覚悟すべきである。

書き込みの時、レジスタは:

0 COMMAND コマンド

多くのディスク・コントローラとは違って、コマンド・レジスタへの書き込みは転送を初期化しない。他のレジスタを設定して準備した後、レジスタ 3(START)を転送の初期化に使用する。しかし、コマンド・レジスタへの書き込みは、様々なエラーフラグをリセットする。注意として、コマンド・レジスタは読み出しができない。

<31:12> 不使用

<11> Done Interrupt Enable. Done 割り込み許可。not-active(状態レジスタの bit0)が割り込みを発生することを許可する。割り込みはあなたがこのビットをクリアするまで発生を続ける。(これは、実際のところ、Done (終了)割り込みというより、idle(暇)割り込みである)

<10> ttention Interrupt Enable. Attention 割り込み許可。any-attention(状態レジスタの bit1)が割り込みを発生することを許可する。(割り込みはコントローラがアクティブでないときだけ、発生する。コントローラがアクティブな間はそれについては、どういう方法でも何もできない)ドライブを選択し at-ease コマンドを与えるか、このビットをクリアするまで、割り込みは発生を続ける。

<9> Recalibrate. リキャリブレイト。コマンド 5 との組み合わせで、ディスクのヘッドをシリンダ 0 へ戻す。

<8> Fault Clear. フォールト・クリア。コマンド 5 との組み合わせで、ディスクのほとんどのフォールト状態をリセットする。

<7> Data Strobe Late. データ・ストローブを遅く。端っこのデータの復調のため。

<6> Data Strobe Early. データ・ストローブを早く。端っこのデータの復調のため。

<5> Servo Offset. オフセットをサーボする。端っこのデータを復調するために、ヘッドを少しオフセット(ずらす)しなければならない。bit4 がどの方向へかを制御する。注意、これはやや怪しい手当(kludgey)であり、もしヘッドをオフセットさせている間にシークをさせようとしたら、フォールトになる(まず、オフセットをクリアするためにコマンド 6 を使用すべし。) サーボ・オフセット・モードでの、一度の 1 ブロックより多くの転送、または、オフセット・クリアを最初に行っていない転送のリトライは、たぶん、フォールトを引き起こすだろう。疑う価値はあるものの、とにかく。ヘッドをオフセットさせたままの書き込みはフォールを引き起こす。

<4> Offset forward. オフセット前向き。1 はオフセットを前向きに、0 でオフセットを後ろ向きにかける。

<3> I/O Direction. I/O 方向。1 でメモリから、0 でメモリへ。以下の、「正しい組み合わせ(valid combinations)」を参照。

<2:0> Command code. コマンド・コード。以下のビットの組み合わせが正しいコマンドである(ここは八進で記述されている)。注意として、ビット 10 と 11 は常に ON にされているだろう、そして bit4 から 7 はすべての読み出しコマンドで ON にされているだろう。

0000 Read. リード。

0010 Read-compare. リード-コンペア。ディスクとメモリの両方から読み出し、そしてもし、両者が一致しないと、状態レジスタの bit22 をセットする。

0011 Write. ライト。

0002 Read All. リード・オール。指定されたローテーション位置から始まるディスクのすべてのビットを読み出す。注意として、このコマンドの間、内部パリティ・エラーが寄生的に発生するだろう。そして、それはヘッドとシリンダを自動的には進めないだろう。下のディスク・フォーマッティングを参照のこと。

0013 Write All. ライト・オール。指定されたローテーション位置から始まるディスクのすべてのビットを書き込む。これはディスクのフォーマッティングを意図している。以下を参照のこと。この READ-ALL での注意は、WRITE-ALL にも適用する。加えて言えば、最終ページの完全なすべてを、それは本当には書かない; 0 から 17 ワードの間のどれかが失われるだろう。

0004 Seek. シーク。ディスク・アドレス・レジスタ中で指定されたシリンダへのシークを初期化する。シークが完了した時、アテンションが発生する。注意として、このコマンドは論理的には必要無い; コントローラはデータ転送コマンドの開始時に、もし必要ならば常にシークを初期化する。リード、リード・コンペアとライト・コマンドは転送の最中にも、必要ならシークを行う。シーク・コマンドは複数のユニットのシークをオーバーラップできるように提供されている。

0005 At ease. 選択されたユニットのアテンションをリセットする。

1005 Recalibrate.リキャリブレイト。シリンダ 0 へシークする、ヘッドの現在位置が正しいかどうか、気にせずに。これはシーク・エラーを正すために使用される。リキャリブレイトはドライブのいくつかのエラー状態をリセットし、それが完了したらアテンションが発生する。

0405 Fault clear. フォールト・クリア。ドライブのほとんどのエラー状態をリセットする。

1405 これは、たぶん、リキャリブレイトとフォールト・クリアを実行する。

0006 Offset clear. オフセット・クリア。ヘッドのオフセット状態を解除。これは、完了を待たない、次のコマンドが待つだろう。

xxx7 これは、予約されたコマンドであり、現在は、コントローラをハングさせるだろう、タイムアウト・エラーを発生させる(ステータス・レジスタbit11)。

0016 Reset. リセット。これは現在の転送を停止し、コントローラをリセットする。このコマンドはコマンド・レジスタにストアされた途端に効果を持つ; START 中は、ストアしてはいけない。リセット・コマンドをストアした後、リセット状態をオフにするために、必ず、コマンド・レジスタに 0 をストアしなければならない。転送が進行中のリセットの使用はおかしなことを引き起こさないとは保証されていない。

xxx5 グループと Reset を除く全コマンドは、開始する前に選択されたユニットの、直前のシーク操作の完了を待つ。なお、シーク(Seek)とオフセット・クリア(Offset Clear)コマンドはコントローラが次のコマンドのために待機(ready)になる前に一定の時間がかかる。

1 COMMAND LIST POINTER コマンド・リスト・ポインタ

これは、ディスクから、またはディスクへ、転送するメモリのページと、その量を指定するチャンネル・コマンド・ワード(CCWs)のアドレスのベクトルである。CLP のビット<15:0>だけがカウントでき、よってもしこの境界を超えようとしたら、コマンド・リストはラップアラウンドするだろう(回って先頭へ戻る)。

CCW の形式は:

<31:24> 不使用

<23:8> ページのメイン・メモリ・アドレス

<7:1> 不使用

<0> 続くフラグ(More flag)。もしこのビットが 0 なら、これがリストで最終の CCW である。もし、このビットが 1 なら、別な CCW が引き続く番地にある。

2 DISK ADDRESS ディスク・アドレス

ディスク・アドレス・レジスタの読み出し中の説明を参照。注意として、1 ユニット版では、ユニット番号のビット<30:28>は無視され 0 とみなされる。

3 START スタート

このアドレスへのいかなる書き込みも、コマンド、ディスク・アドレス、コマンド・リスト・ポインタの各レジスタで指定された操作の初期化を行う。

Disk Structure ディスク構造

各ディスク・ブロックは 1 つの Lisp マシン・ページのデータの内容、すなわち 256.ワードか 1024.バイトを、持つ。65536.までの連続したディスク・ブロックを不連続なメモリ番地に、単一の操作で転送できる、が、あなたは、もっと多くのメイン・メモリをサポートした機械を用意出来るかな。(※訳注:当時 64M バイトものメモリを持つ機械は現実的ではなかった。つまり、一回の転送指令で、全メモリの転送が可能であったということである)T-80 は、815.シリンダで、各シリンダには、5 ヘッド(トラック)があり、あなたのフォーマッティングによるが、16.か 17.ブロックがある。T-300 は 19.ヘッドであることを除き同じ仕様である。

Formatting フォーマッティング

フォーマットは、ディスクをフォーマットするための Write All 操作を使用するプログラムと、ハードウェアによって決定される制約とによって決定される。トラックは(およそ)20160.バイトを持つ(T-80 か T-800 で)。ディスク中のジャンパは、トラックあたり 17.セクタのパルスを与えるか、1164 バイト毎に 1 つのパルスを与え、それはトラックの最後を少し残す、ようにセットされている。

すべては、下位ビットから先に、そして下位バイトから先に行われる。注意として、ディスク・コントローラ中のビットはドライブにはビットのコンプリメント(論理反転)が見えている。よって、トライデントのマニュアル中のすべてのビットは、反転すべし。

ブロックのフォーマットは:

(sector pulse here)
PREAMBLE - 53.バイトの 1。
VFO LOCK - 8.バイトの 1。
SYNC - 八進の 177 を持ったバイト。
HEADER - 32 ビットのワード、以下のような:
<31:30> 次のブロック・アドレス・コード(next block address code):
0 同じトラック上に続くブロック
1 次のトラックにブロック 0(next head)
2 次のシリンダのヘッド 0 にブロック 0
3 ディスクの終わり
<29:28> 不使用、0 でなければならない。
<27:16> シリンダ番号、正しいシリンダにディスクが位置したかを検証するのに使用される。

<15:8> ヘッド番号、ヘッドの選択を検証するのに使用。
<7:0> ブロック番号、ローテーション位置を検証するのに使用。

HEADER ECC - 32bit のチェック・ワード。
VFO RELOCK - 20.バイトの 1。
SYNC - 八進の 177 を持つ 1 バイト。
PAD - 八進 377 を持つ 1 バイト、これは read-compare 用の論理のバグを直すためにここにある。(うげっ)
DATA - 1024.バイトのあなたが欲しいもの。
DATA ECC - 32bit のチェック・ワード。
POSTAMBLE - 44.バイトの 1。

ディスクをフォーマットするには、1 トラックを一度に行わねばならない。トラックに書かれるビットをメモリ中に配置すること。長さはページの倍数に切り上げる、ただし、最後の 17.ワードは気にしない。(一般的に、19.ページか 19456.バイトを書くだろう、トラックの最後の約 771.バイトの後、書き込みの終了をしないだろう。操作を打ち切る時点は、あなたがどれだけ fifo を満たしたかに依存する。ブロック長の選択によっては、最終ブロックを完全に書くチャンスを無くすだろう。しかし、データ領域に関しては、大丈夫であろう。このデータの WRITE ALL(ライト・オール)コマンドは、ブロック番号フィールド(ビット<7:0>)が 0 のディスク・アドレスで行うべし。内部パリティ・エラー(ステータス・レジスタのビット 23)は無視すべし。Read All(リード・オール)コマンドを使用することでそれを検証できる(しかし、内部パリティとリード・コンペア(read-compare)機能は働かないだろう) 、または、通常の write

と read コマンドを用いることもできる。ECC チェック・ワードは手動で計算しなければならない。その多項式は、x^31+x^29+x^20+x^10+x^8+1 である。[筆者の論理の理解が正しければ]

(※訳注：「一般的に…」の開きカッコがいつまでも閉じていないが、原文がそうなっているのである)

注意、リード・オール(Read All)を使用する時、厳密にいうと、データ読み出しがスタートする位置は、曖昧さがある。バイト境界上に、バイトが整列するというのは、現実的ではない。データの最初の数マイクロ秒ぶんは失われるか壊れるだろう。

Debugging デバッギング

コネクタ J11 は、そこに以下の役立つ信号を出す LED ディスプレイへのフラット・ケーブルを提供する。それらはインアクティブの時グランドであり、アクティブの時は+3V(ボルト)で 15mA(ミリ・アンペア)である。

1 リード・アクティブ(Read Active)。コントローラがアクティブである。コマンド・レジスタの bit0 が 0。

2 ライト・アクティブ(Write Active)。コントローラがアクティブである。コマンド・レジスタの bit0 が 1。

3 シーク。選択されたユニットがオン・シリンダ(on-cylinder)ではない。

4 転送ロセージ(障害)(Transfer Lossage) タイムアウト、リード・オーバラン、ライト・オーバラン、メモリ・パリティ・エラー、存在しないメモリ・エラーの OR である。(※訳注：IOR は、普通の論理和である)

5 フォーマット・ロセージ(障害)。スタート・ブロック・エラー、ヘッダ・コンペア・エラー、ヘッダ ECC エラー、リセットの OR である。

6 ECC ロセージ(障害)。ハード ECC エラー、ソフト ECC エラーの OR。

7 ディスク・ロセージ(障害)。これは、複数ユニットの選択、ユニットが未選択、選択されたユニットがフォールト、選択されたユニットがオンラインでない、選択されたユニットがシーク・エラーの OR である。

8 スペア。これは、たぶん、点灯することはないだろう。

<<ここに 1 ページの命令フォーマットを差し込む>>
(※訳注： そんな表は見たことない)

The CONSLP Assembler　CONSLP アセンブラ

CONSLP は Maclisp で書かれたシンボリック・アセンブラであり、CADR マシンのためのソースコードを読み込み、CC デバッガによってロードされるファイルを生成する。ソースコードは、LISP S 式の形で書かれる； シンボルは、LISP のアトミック・シンボルで、命令やデータのアイテムはリストとして書かれる。コメントは Maclisp のセミコロン慣習を使用して書くことができる。入力の数の基数は 8(八進)、後ろに小数点'.'が付いたものは強制的に 10(十進)となる。

Localities(記憶場所指定)

プログラムは、命令、ディスパッチ、A と M メモリ中にロードされるデータを指定することができる。どのメモリのためにデータをアセンブルするかを指定するには、LOCALITY 疑似命令(pseudo-op)を使用する:

```
(LOCALITY I-MEM)  ;引き続くデータは命令メモリへ行く
(LOCALITY D-MEM)  ;同様に、ディスパッチ・メモリへ
(LOCALITY A-MEM)  ;同様に、A メモリへ
(LOCALITY M-MEM)  ;同様に、M メモリへ
```

Location Tags and Symbols 番地タグと記号

命令ストリームのアセンブル中にアトミック・シンボルに出会ったら、それは番地タグ(location tag)(ラベル, label)として扱われる。そのタグは通常、アセンブルされていく記憶場所中の現在のその次の番地の値として定義される。しかし、タグの値をその"通常の(normal)"位置に置くために「シフト」される。A メモリのタグの、通常位置は命令の A ソース・フィールドである； M メモリのタグも同様である。I メモリのタグの、通常位置は JUMP 命令の New PC フィールドである； D メモリのタグは、DISPATCH 命令の Dispatch Offset フィールドである。よって、もし FOO がディスパッチ・メモリの 7 番地のためのタグであれば、FOO の実効値は 70000(八進)である。

慣習によって、A メモリ中のタグは、"A-"という文字列から始まり、M メモリ中のものは"M-"である。しかし、これは CONSLP によって強制されるものではない。

記号は、ASSIGN 疑似命令の使用によって定義できる。

```
(ASSIGN <symbol> <value>)
```

例えば:

```
(ASSIGN CDR-IS-NORMAL 0)
(ASSIGN CDR-IS-ILLEGAL 1)
(ASSIGN CDR-IS-NIL 2)
(ASSIGN CDR-IS-NEXT 3)
```

一般に <value> は「式プログラム(expression program)」であろう。記号が参照された時、「式プログラム」は記号の値を生成するために評価される(たぶん、それが出現した文脈の条件で)。「式プログラム」は後の節で議論される。

Instructions 命令

一般的に、CONSLP は、リストのすべての要素を評価し、データ・アイテムにして、それらを足し合わせることによって、リストをアセンブルする。複雑な「式プログラム」を指定するためかつ、それらに記号名をアサインするためのかなり豪華な言語がある； しかし、ここでは、我々は単に CONSLP によってすでに定義された記号を使用するだろう。CONSLP は命令のフィールドを、ほぼ順不同に書くことを許す。が、しかし、我々がそれらを書くとき、慣習的な順序でのみ書く。

I-MEM 命令の一般的な形式は：

```
(<popj>  (<destinations>)  <operation>  <condition>
  <M-source>  <byte-descriptor>  <A-source>  <target-tag>  <other fields>)
```

<popj>フィールドは、POPJ ビットをセットすることを指定するために、POPJ-AFTER-NEXT とする。
<destinations> フィールドは、A か M メモリ・タグであるか、機能デスティネーション名か、M メモリ・タグと機能デスティネーションの両方か、である。
<operation> は命令タイプと、他の可能なフィールド（ジャンプ条件のような）を同様に指定する。
<condition>も分離されたフィールドである。通常は命令の一部としてエンコードされるのであるが。
<byte-descriptor>は BYTE か DISPATCH 命令で使用されるバイトを記述する。
<M-source>と <A-source>はソースを指定する； これらは、使用してるメモリ中のタグか、<M-source>のためか、M マルチプレクサ・ソースの名前であろう。
<target-tag>は JUMP 命令のための I-MEM タグか、DISPATCH 命令のための D-MEM タグである。
<other fields>は Q コントロールのような事柄や、「その他の機能（Miscellaneous Functions）」であろう。

これらの多くのフィールドは省略でき、そして、CONSLP は、適切にそれらをデフォールト（省略時の暗黙値）にするだろう。もし<operation>が省略されたら、ALU であると考えられるが、ただし、<byte descriptor>が明示的か暗黙にか存在したら BYTE と考えられる。もし、ALU 命令中にただ一つのソースがあったら、SETA のオペコードは A ソースを供給され、SETM は M ソースを、そしてデータの単純な転送が起こる。もし、BYTE 命令で A ソースが省略されたら、A メモリ中の 2 番地（それは 0 を持っていると仮定されている）と想定される。

コメント付きの、命令の簡単な例をここに示す。上で述べた A と M メモリ・タグの慣習を想定している。

```
((A-FOO) M-BAR)    ;M-MEM 中の BAR から、A-MEM 中の FOO へ転送

(CALL ZAP)  ;命令 ZAP への CALL 遷移を行う（N ビットはセットされる）

((A-FOO) SUB M-BAR A-BAZ)
            ;M-BAR から A-BAZ を減算し、結果を A-FOO へ

(JUMP-EQUAL-XCT-NEXT M-BAR A-FOO LOSE)
            ;もし M-BAR が A-FOO とイコールなら LOSE へジャンプ ;N ビットはクリア、
            ;よって、JUMP 命令の後の
            ;  JUMP が成功してもそうでなくとも実行される
```

```
(POPJ-AFTER-NEXT (M-FOO) MEMORY-DATA)
            ; メモリからのデータを、M-FOO に置く、
            ;  そして、次の命令の後で、POPJ する


((M-SAVE MEMORY-DATA-START-WRITE)
     ADD MEMORY-DATA A-ZERO ALU-CARRY-IN-ONE)
            ;読み出したメモリ・データに 1 を加え、
            ; メモリ・データ書き込みと M-SAVE に送り、
            ; そして、すでに VMA 中にあるアドレスに
            ; そのデータのメイン・メモリへの書き込みを始める
```

Literals 定数(リテラル)

CONSLP は、A と M メモリ中の定数を指定するための機能を提供している。
構築子

(A-CONSTANT <expression>)　と (M-CONSTANT <expression>)

は、A ソースか M ソース指定として現れ、CONSLP に適切なメモリ中にワードを割り当てるようにさせ、定数式をそこにアセンブルさせ、ソース・ロケーションとしてその番地のアドレスを使用させる。もし同じメモリ中の同じ定数が複数回参照されたら、CONSLP は、そのただ一つのコピーだけをアセンブルするだろう。もし 2 つの定数の最終バイナリ値が同じであれば、それらを同じだとみなす、それらの値に簡約化されるソースの式には無頓着で。定数 0 は特別に扱われ、適切なメモリの 2 番地への参照とされる(したがって、ユーザは定数ソース 0 のための番地を予約する必要はない)。同様に、定数-1 は、適切なメモリの 3 番地への参照とされる。

Byte Specifications バイト指定

BYTE と DISPATCH 命令のためのローテション・カウントと LENGTH (minus 1) フィールドの計算をユーザに要求せず、CONSLP は、ワード中のサイズと位置の項 (term) でバイト指定をする、単一形式の方法を提供する; CONSLP は適切にフィールドを計算する。

BYTE-FIELD を使用してバイトを記述する最も簡単な方法は:

```
(BYTE-FIELD <size in bits> <position from right>)
```

例えば、(BYTE-FIELD 5 0) はワードの下位 5bit であり、そして (BYTE-FIELD 7 5) はその上の 7bit である。BYTE-FIELD の 2 つの引数は整数定数でなければならない。

バイトの別な記述は:

```
(LISP-BYTE <ppss>)
```

ここで、<ppss>の八進数の下位の 2 桁はサイズであり、次の 2 つは位置である。引数<ppss>は LISP フォームとして評価される (下の、"式プログラム (Expression Programs)"を参照)。

バイト指定子が命令中に出現したら、op-code は BYTE に既定され、そして、バイト命令の型は"load byte"に既定される。もし命令外のどこかで指定されたら、op-code は代って DISPATCH とされ、ディスパッチは指定されたバイトに基づく。バイトが 1bit 幅であるときだけ、op-code は JUMP であろう; これはジャンプが M ソースの指定されたビットをテストすることを意味する。

CONSLP が最終命令へアセンブルする時、それは、バイト指定子の基本であるローテション・カウントと length minus 1 フィールドと、実行されるべき操作から、構成されている。JUMP、DISPATCH、"load byte" 型の BYTE 命令では、正しいローテーション・カウントを得るために 32 からバイト位置を減じることをする。(CADR がワードを左にローテートすることを思い出そう) その他機能 3 (LOW PC BIT がハーフ・ワードを指定する) がイネーブルの時、代わりに、位置 (それは 16 より小さくなければならない) は 16 から減じられたものになる。
"deposit byte"と"selective deposit"型の BYTE 命令では、バイト位置そのものがローテーション・カウントとして使われる。BYTE と JUMP の length minus 1 フィールドは、バイト長が 0 の時は 0 が使用され、それ以外の時は、バイト長から 1 減じることによって計算される。
(注意: CADR は、実際のところ、長さ 0 のバイトを扱うことができない、
しかし、CONSLP はそれを"next instruction modify"機能が使用中であるという仮説のもとに、それらを定義することを許している。この機能を使用するプログラムは、アセンブラがまとめるハッカー式やり方を承知しているに違いない、そして走行時 (ランタイム) のこのフィールドの実際の値として許す。)
DISPATCH 命令は length minus 1 フィールドの代わりに長さフィールド (length field) を持つ、そのために 1 の減算は行われない。

いつかのバイト指定子の使用例を示す:

```
((M-X) (BYTE-FIELD 7 4) M-Y)
            ; M-Y の右から 4bit の、7bit バイトを展開、
            ; このバイトを右詰めして M-X に。
            ; A ソースは 1 に既定され、それは定数 0 である。
            ; よって M-X の他のビットは 0 になる。
```

```
(JUMP-IF-BIT-SET (BYTE-FIELD 1 3) M-ZAP QUUX)
            ;もしM-ZAP中の"10"ビットがセットされていれば、QUUXへジャンプ

(DISPATCH (BYTE-FIELD 3 0) M-ZAP DTABLE)
            ;M-ZAPの下位3bitを、ディスパッチ・テーブルDTABLEを
            ; 指す(インデックスする)のに使用する
```

ASSIGN疑似命令を使用して、バイト・フィールドのために記号名を作ることができる:

```
(ASSIGN LOW-HEX-DIGIT (BYTE-FIELD 4 0))
```

以下は共通の操作であるから、この用途のための他の疑似命令も存在する:

```
(DEF-DATA-FIELD <symbol> <byte size> <byte position>)
```

例えば:

```
(DEF-DATA-FIELD LOW-HEX-DIGIT 4 0)
```

特定のレジスタ中のバイト・フィールドに名前を付けることもできる。バイト指定子とレジスタ名をまとめたものにこれを行うのは片方向である:

```
(ASSIGN CONDITION-CODES (PLUS (BYTE-FIELD 4 0) PDP-11-PS))
(ASSIGN TRACE-TRAP-BIT (PLUS (BYTE-FIELD 1 4) PDP-11-PS))
(ASSIGN PRIORITY (PLUS (BYTE-FIELD 3 5) PDP-11-PS))
```

以下のケースも十分に共通であり 、この用途のための特別な疑似命令のための確証となる:

```
(DEF-BIT-FIELD-IN-REG <symbol> <byte size> <byte position> <register>)
```

例:

```
(DEF-BIT-FIELD-IN-REG CONDITION-CODES 4 0 PDP-11-PS)
(DEF-BIT-FIELD-IN-REG TRACE-TRAP-BIT 1 4 PDP-11-PS)
(DEF-BIT-FIELD-IN-REG PRIORITY 3 5 PDP-11-PS)
```

注意: <register>はM-スクラッチパッド中にあった方がいい。この定義を使うと、PRIORITYとだけ言う必要がある、適切なバイト参照を行う命令中では:

```
((A-PRIORITY) PRIORITY) ;PDP-11-PSからPRIORITYバイトを抽出し
                        ; A-PRIORITYに右詰めで置く
```

特殊な調合により、デスティネーション・フィールド中でのこういう記号の使用も動作する。正確なDPBがアセンブルされる。

2 つより多くの擬似命令が、レジスタ中の、多くの一連のビットやフィールドの名前定義を簡単にしている。

```
(DEF-NEXT-FIELD <symbol> <byte size> <register>)
```

これは、<symbol>を、指定されたサイズのバイトとして定義する、DEF-NEXT-FIELD によってすでに定義されたフィールドの左の位置に。もしこれが指定されたレジスタの最初の DEF-NEXT-FIELD であれば、フィールドの位置は 0 (ワードの下の端) である。例：

```
(DEF-NEXT-FIELD REL-OFFSET 8 IBM-1130-INSTRUCTION)
(DEF-NEXT-FIELD TAG-FIELD 2 IBM-1130-INSTRUCTION)
(DEF-NEXT-FIELD FORMAT-BIT 1 IBM-1130-INSTRUCTION)
(DEF-NEXT-FIELD OP-CODE 5 IBM-1130-INSTRUCTION)
```

は、次とまったく同様に：

```
(DEF-BIT-FIELD-IN-REG REL-OFFSET 8 0 IBM-1130-INSTRUCTION)
(DEF-BIT-FIELD-IN-REG TAG-FIELD 2 8 IBM-1130-INSTRUCTION)
(DEF-BIT-FIELD-IN-REG FORMAT-BIT 1 10. IBM-1130-INSTRUCTION)
(DEF-BIT-FIELD-IN-REG OP-CODE 5 11. IBM-1130-INSTRUCTION)
```

擬似操作：

```
(DEF-NEXT-BIT <symbol> <register>)
```

は、次と完全に同じである：

```
(DEF-NEXT-FIELD <symbol> 1 <register>)
```

そして、ただ 1 ビットを割り当てる。それは、DEF-NEXT-FIELD と自由に混ぜることができる。
例：

```
(DEF-NEXT-FIELD CONDITION-CODES 4 PDP-11-PS)
(DEF-NEXT-BIT TRACE-TRAP-BIT PDP-11-PS)
(DEF-NEXT-FIELD PRIORITY 3 PDP-11-PS)
```

構成子：

```
(RESET-BIT-POINTER <register>)
```

は、DEF-NEXT-FIELD と DEF-NEXT-BIT によって使用された<register>の中の、(ビット) ポインタをリセットするために使用されるだろう。これは、<register>中のデータが、いくつかの異なる形式を持てる場合に便利である。
例：

```
(DEF-NEXT-BIT C PDP-11-PS)
(DEF-NEXT-BIT V PDP-11-PS)
```

```
(DEF-NEXT-BIT Z PDP-11-PS)
(DEF-NEXT-BIT N PDP-11-PS)
(RESET-BIT-POINTER PDP-11-PS)
(DEF-NEXT-FIELD CONDITION-CODES 4 PDP-11-PS)
(DEF-NEXT-BIT TRACE-TRAP-BIT PDP-11-PS)
(DEF-NEXT-FIELD PRIORITY 3 PDP-11-PS)


(DEF-NEXT-FIELD DST-REG 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD DST-MODE 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD SRC-REG 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD SRC-REG 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD OP-CODE 4 PDP-11-INSTRUCTION)
(RESET-BIT-POINTER PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD BRANCH-OFFSET 8 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD BRANCH-CONDITION 3 PDP-11-INSTRUCTION)
(RESET-BIT-POINTER PDP-11-INSTRUCTION)
```

Dispatch Tables ディスパッチ・テーブル

ディスパッチ・メモリへのアセンブル時 (すなわち (LOCALITY D-MEM)) 、ブロックをディスパッチ・メモリへ割り当てるために 、2 つの特殊な擬似操作、START-DISPATCH と END-DISPATCH を使う必要がある。これらの擬似操作は要求されるブロックの長さを指定する。そして、CONSLP は、様々な奇数サイズのブロックを適切な方法でディスパッチ・メモリへパックすることを保証する。

ディスパッチ・ブロックの典型的な形式は：

```
(START-DISPATCH <log2 of size> <constant data>)
<dispatch table tag>
      <first word of table>
      ...
      <last word of table>
(END-DISPATCH)
```

<log2 of size>は、ディスパッチされるであろうもののビット数で、ディスパッチ・ブロックのサイズの 2 を底とした log (ログリズム) を取ったものである。<constant data>は、ディスパッチ・テーブルの各ワードに加えられる；これは P,R,N ビットで便利である。(CONSLP では P-BIT, R-BIT, INHIBIT-XCT-NEXT-BIT と呼ばれている) 。END-DISPATCH は論理的には必要無いが、エラーチェックのために、使われている。
正確に正しいワード数が START-DISPATCH と END-DISPATCH 間でアセンブルされなければならない、
そうでなければ、CONSLP はエラーメッセージを与える。

ディスパッチ・テーブルの例、このコードを考える：

```
(LOCALITY M-MEM)
PDP-11-INSTRUCTION      (0)    ;シミュレートされた PDP-11 命令を持つ
(DEF-NEXT-FIELD DST-REG 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD DST-MODE 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD SRC-REG 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD SRC-REG 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD OP-CODE 4 PDP-11-INSTRUCTION)
      ...

(LOCALITY I-MEM)
      (DISPATCH-CALL-XCT-NEXT DST-MODE D-DST-MODE)
      ...

(LOCALITY D-MEM)
(START-DISPATCH 3 P-BIT)
D-DST-MODE
      (DST-REGISTER)                ;R0
      (DST-REG-INDIRECT)            ;@R0
      (DST-AUTO-INCREMENT)          ;(R0)+
      (DST-AUTO-INC-INDIRECT)       ;@(R0)+
      (DST-AUTO-DECREMENT)          ;-(R0)
      (DST-AUTO-DEC-INDIRECT)       ;@-(R0)
      (DST-INDEXED)                 ;N(R0)
      (DST-INDEXED-INDIRECT)        ;@N(R0)
(END-DISPATCH)
```

注意、オペコード DISPATCH-CALL-XCT-NEXT の I-MEM の使用は、純粋に整形の目的のためであり， P ビット、(N ビットでなく) が、全てのディスパッチ・テーブル・エントリ中で定数であることを示すためである； そうでなければ、それは DISPATCH オペコードと同一である。

Standard Operation Codes 標準オペレーション・コード

CONSLP は様々なオペレーション、特に様々な条件ジャンプに、多くの初期記号を供給している。それを異なるものとして定義することができるので、これらの標準のものの使用が自然に奨励される。(これらの記号はファイル LISPM;CONSYM >中で定義されている。)

ALU Operations ALU オペレーション

CONSLP によって供給されている標準 ALU オペレーション:

Boolean ブーリアン

SETCM	M の補数にセットする
ANDCB	M と A の両方の補数を AND する
ANDCM	M の補数と A の AND
SETZ	0 にセット
ORCB	M と A の両方の補数を OR する
SETCA	A の補数にセット
XOR	M と A の XOR(排他的論理和)
ANDCA	M と A の補数の AND
ORCM	M の補数と A の OR
EQV	M と A の EQV(一致、排他的論理和の補数)
SETA	A にセット
AND	M と A の AND
SETO	1 にセット
ORCA	M と A の補数の OR
IOR	M と A の OR(インクルーシブ OR)
SETM	M にセット

Arithmetic 算術演算

ADD	M+A 2 の補数の加算
SUB	M-A 2 の補数の減算
M+M	M+M 2 の補数の加算
M+M+1	M 足す M 足す 1
M+A+1	M 足す A 足す 1
M-A-1	M 引く A 引く 1
M+1	M 足す 1

Conditional Arithmetic 条件算術演算
    MULTIPLY-STEP
    DIVIDE-FIRST-STEP
    DIVIDE-STEP
    DIVIDE-LAST-STEP
    DIVIDE-REMAINDER-CORRECTION-STEP

乗算と減算のための条件 ALU オペレーションは、後の節で詳しく述べられる。出力バス・セレクタの既定は 1 (出力バスは ALU 出力となる) 。他の 2 つの選択は明示的に指定されなければならない。

    OUTPUT-SELECTOR-RIGHTSHIFT-1
    OUTPUT-SELECTOR-LEFTSHIFT-1

ALU 命令の Q 制御フィールドは、以下の記号の一つの使用によって、指定される:

SHIFT-Q-LEFT	Q を左にシフト (ALU<31>のインバースが Q<0>に入る)
SHIFT-Q-RIGHT	Q を右にシフト (ALU<0>が Q<31>に入る)
LOAD-Q	出力バスから Q へロード

これらが出現しなければ、既定では Q へは何もしない。(LOAD-Q の書き込みの代わりに、命令のデスティネーション部分に Q-R を書くだろう。これは Q が機能デスティネーションであることを意味しない; それは操作を単に強制的に ALU にし、Q 制御フィールドを強制的に LOAD-Q にする。)

キャリー・フィールドは、ALU-CARRY-IN-ZERO か ALU-CARRY-IN-ONE によって指定される。注意、SUB, M+M+1,M+A+1, M+1 操作は、それらの定義の一部として ALU-CARRY-IN-ONE を持つ、よって、それを明示的に指定する必要はない。

BYTE operations バイト操作

バイト指定子が命令中にあり、かつ、そのオペコードが明示的に JUMP か DISPATCH にする強制的な指定でなければ、その時は、オペコードは既定では BYTE であり、"load byte"型操作を実行する。

"deposit byte"型操作にするには、記号 BPD が使用される;同様に、"selective deposit"にするには、SELECTIVE-DEPOSIT が使われる。例:

```
((A-FOO) DPB M-BAR (BYTE-FIELD 3 6) A-FOO)
            ;真に PDP-10 スタイルの DPB;
            ; M-BAR の八進数の下位桁が
            ;A-FOO の下から 3 番目の八進桁を置き換える


((A-ZAP) DPB M-BAR (BYTE-FIELD 3 6) A-FOO)
            ;同様だが、結果が A-ZAP 中に置かれる。
            ; A-FOO は変えられない。


((A-ZAP) SELECTIVE-DEPOSIT M-FOO (BYTE-FIELD 16. 8) (A-CONSTANT -1))
            ;A-ZAP は、上位 8bit と下位 8bit を全部 1 に置き換えた M-FOO のコピーを得る
            ;(または、-1 の真ん中の 16bit を
            ;M-FOO からの対応するビットで置き換えたコピーを得る)
```

DISPATCH Operations ディスパッチ操作

CONSLP では、4 つのオペコードがディスパッチのために、定義されている：

```
DISPATCH
DISPATCH-CALL
DISPATCH-XCT-NEXT
DISPATCH-CALL-XCT-NEXT
```

これらは純粋に形式合わせの用途のために提供されている、本当のディパッチ行為はディスパッチ・テーブルで制御されているから。CONSLP は、与えられたディスパッチ・テーブルで"正しい"オペコードが使用されているかチェックしようとしない。習慣として、もしディスパッチ命令に続く命令が実行される(N bit がセットされていない)なら、XCT-NEXT 版が使われ、そして、もし P bit がセットされていれば CALL 版が使われる。

DISPATCH CONSTANT レジスタにロードされる 10bit の"immediate argument(即値)"の値を指定するには、ディスパッチ命令中で、次が使われるだろう

```
(I-ARG <expression>)          ;即値
```

小さな定数をアーギュメントとしてサブルーチンに渡すために、特別な擬似オペレーション DISPATCH CONSTANT を使用すると簡単である。次の式

```
((ARG-CALL FOO) (I-ARG BAR))
```

は、CALL 型の FOO への遷移を含む 1 ワードのテーブルへ、DISPATCH 命令を生成する。
そしてディスパッチ命令中のディスパッチ定数フィールドへ BAR を置く。FOO は、それから、アーギュメントを取り出して働くために、READ-I-ARG 機能ソースを使用するだろう。

Miscellaneous Function 2(その他機能 2, ディスパッチ・メモリへの書き込み)は記号 WRITE-DISPATCH-RAM によって指定される。

JUMP Operations JUMP 操作

CONSLP は様々な JUMP 操作のための名前を沢山定義している。これらは、部分の論理的組み合わせのすべてから作り出された：

```
<type>  <condition>  <xct next>
```

<type>は JUMP, CALL, POPJ のどれかであり、P bit か R bit がセットされるか，どのビットもセットされないか、だろう。<condition> は次の一つであろう：

```
IF-BIT-SET
IF-BIT-CLEAR
EQUAL
NOT-EQUAL
LESS-THAN
GREATER-THAN
GREATER-OR-EQUAL
LESS-OR-EQUAL
IF-PAGE-FAULT
IF-NO-PAGE-FAULT
IF-PAGE-FAULT-OR-INTERRUPT
IF-NO-PAGE-FAULT-OR-INTERRUPT
IF-PAGE-FAULT-OR-INTERRUPT-OR-SEQUENCE-BREAK
IF-NO-PAGE-FAULT-OR-INTERRUPT-OR-SEQUENCE-BREAK
```

もし省略されたら、<condition> は"always(常時)"と解釈される。<xct_next> がもしあれば、XCT-NEXT であり； それが無ければ、Nbit の存在を意味する。 Nbit は、もしジャンプが成功したら、ジャンプ後の命令を禁止する。3 つの部分 は"-"でつながれる。これらの操作の例：

```
CALL-LESS-THAN
JUMP-LESS-THAN-XCT-NEXT
CALL
POPJ-IF-BIT-SET
CALL-IF-PAGE-FAULT-OR-INTERRUPT
CALL-IF-BIT-CLEAR-XCT-NEXT
JUMP-XCT-NEXT
POPJ-XCT-NEXT
```

POPJ-XCT-NEXT 操作は POPJ-AFTER-NEXT と一緒でも混乱させられない、POPJ-AFTER-NEXT は POPJ ビットをセットするどの命令でも使用できる。

算術比較を実行するジャンプ命令は A と M ソースの両方を持たなければならない；そのソースが比較される。1 ビットをテストするジャンプ命令は、1-bit バイトをテストするための、一つの M ソースと、一つのバイト指定子を、持たなければならない。

Functional Sources 機能ソース

The following names are supplied by CONSLP for the various functional sources:

CONSLP によって、次の名前が機能ソースのために供給されている：

0	READ-I-ARG	ディスパッチ定数
1	MICRO-STACK-PNTR-AND-DATA	SPCPTR と SPC 定数
	MICRO-STACK-POINTER	ビット<28-24>のためのバイト指定子
	MICRO-STACK-DATA	ビット<18-0>のためのバイト指定子
14	MICRO-STACK-PNTR-AND-DATA-POP	1 と同様，ただし SPC スタックを pop する
	MICRO-STACK-POINTER-POP	1 と同様，ただし SPC スタックを pop する
	MICRO-STACK-DATA-POP	1 と同様，ただし SPC スタックを pop する
2	PDL-BUFFER-POINTER	PDL-pointer レジスタ
3	PDL-BUFFER-INDEX	PDL-index レジスタ
5	C-PDL-BUFFER-INDEX	インデックスによってアドレス指定された PDL-buffer
25	C-PDL-BUFFER-POINTER	ポインタによってアドレス指定された PDL-buffer
24	C-PDL-BUFFER-POINTER-POP	ポインタによってアドレス指定された PDL-buffer,pop
6	OPC-REGISTER	OPC
7	Q-R	Q レジスタ
10	VMA	VMA レジスタ
11	MEMORY-MAP-DATA	MAP[MD]
12	MEMORY-DATA	MD
13	LOCATION-COUNTER	LC

Functional Destinations 機能デスティネーション

CONSLP は以下の名前を機能デスティネーションのために提供している。注意、いくつかはソースと同じ名前を使用している； CONSLP は文脈によって使用法を区別している。

1	LOCATION-COUNTER	LC
2	INTERRUPT-CONTROL	割り込み制御レジスタ
10	C-PDL-BUFFER-POINTER	PDL ポインタによってアドレス指定された pdl 番地
11	C-PDL-BUFFER-POINTER-PUSH	pdl にデータをプッシュし、PDL ポインタをインクリメントする
12	C-PDL-BUFFER-INDEX	PDL INDEX によってアドレス指定された pdl 番地
13	PDL-BUFFER-INDEX	PDL INDEX レジスタ
14	PDL-BUFFER-POINTER	PDL POINTER レジスタ
15	MICRO-STACK-DATA-PUSH	データを SPC スタックにプッシュ
16	OA-REG-LOW	次の命令を変更、ビット <25-0>
17	OA-REG-HI	次の命令を変更、ビット <47-26>
20	VMA	VMA レジスタ
21	VMA-START-READ	VMA,リード・サイクル開始
22	VMA-START-WRITE	VMA, ライト・サイクル開始
23	VMA-WRITE-MAP	VMA, MAP[MD] <- VMA
30	MEMORY-DATA	MD レジスタ
31	MEMORY-DATA-START-READ	MD, リード・サイクル開始
32	MEMORY-DATA-START-WRITE	MD, ライト・サイクル開始
33	MEMORY-DATA-WRITE-MAP	MD, MAP[MD] <- VMA

記号 Q-R もデスティネーションとして使用される； "load Q from ALU output" にセットされるべき Q 制御フィールドを持つ ALU 命令を発生する； これは、命令中で LOAD-Q を指定するのと同等である。Q-R をデスティネーションとする接続に、出力バス・シフタを使用してはいけない。

Operations Common to All Instructions 全命令に共通の操作

POPJビットのための記号は、POPJ-AFTER-NEXT である。

Miscellaneous Function 3(その他機能 3)は LOW-PC-BIT-SELECTS-HALF-WD によって記述される。(前後の節で、非常に詳しくこの機能について述べられている)

Expression Programs in CONSLP　CONSLP の式プログラム

Wherever an expression may be used in CONSLP, the following arcane forms may be used. In particular, the value of a symbol is normally an expression instead of a simple number. Whenever an expression (or a symbol with an expression as its definition) is encountered, it is evaluated according to the following rules:

CONSLP 中で使用される式はなんであれ、次の神秘的な式が使われている。特に、記号の値は、通常、単純な数の代わりに式となっている。いつでも、式(またはその定義としての式を伴なう記号)に出会ったら、それは次の規則にしたがって評価される:

<number>　それ自身に、評価される。

(PLUS <exp1> <exp2>)　2 つの式を加算し、それらのプロパティにまとめる(バイト指定子的なものに)。

(DESTINATION-P <exp>)　条件: もしデスティネーションをアセンブル中に出会ったら、<exp>の値を返す、他の時は NIL。

(SOURCE-P <exp>)　条件: もしソース(M か A)をアセンブル中に出会ったら、<exp>の値を返す、他の時は NIL。

(DISPATCH-INSTRUCTION-P <exp>)
A conditional: if encountered while assembling a DISPATCH instruction, returns the value of <exp>, and otherwise NIL.

条件: DISPATCH 命令をアセンブル中に出会ったら、<exp>の値を返す、他の時は NIL。

(JUMP-INSTRUCTION-P <exp>)　条件: JUMP 命令をアセンブル中に出会ったら、<exp>の値を返す、他の時は NIL。

(ALU-INSTRUCTION-P <exp>)　条件: ALU 命令をアセンブル中に出会ったら、<exp>の値を返す、他の時は NIL。

(BYTE-INSTRUCTION-P <exp>)　条件: BYTE 命令をアセンブル中に出会ったら、<exp>の値を返す、他

の時は NIL。

(NOT <conditional>)　　否定。<conditional> は上の「条件」式の一つでなければならない。

(OR <cond1> ... <condn>)　　LISP の OR の様なもの、最初の非 NIL の条件を返す。

(BYTE-FIELD <size> <pos>)　　先に述べられた通り、サイズと右からの位置を与えられ、バイトを定義する。

(LISP-BYTE <ppss>)　　先に述べられた通り、もし ppss が八進数で書かれていたら、これは、(BYTE-FIELD ss pp)と同様であり、もし ppss が数でなければ、それは LISP 式(CONSLP 式でなく!)であり、LISP で評価される。

(BYTE-MASK <byte specifier>)　値は、指定されたバイト中が全部 1 であり、それを除いたすべてが 0 なワードである。これは条件の一種で、もしバイト指定子が、実際にはバイトを指定していないとき、NIL を返す。

(BYTE-VALUE <byte specifier> <value>)　　値は、指定されたバイトが<value>を持ち、それを除いたすべてが 0 なワードである。これは条件の一種で、もしバイト指定子が、実際にはバイトを指定していないとき、NIL を返す。

(OA-HIGH-CONTEXT <word>)　　<word>を命令としてアセンブルし、上位半分(ビット <47-26>)を返す、OA レジスタ機能(次の命令を変更する(next instruction modify)、機能デスティネーション 17) によって使用されるために。

(OA-LOW-CONTEXT <word>) <word>を命令としてアセンブルし、下位半分(ビット <25-0>)を返す、OA レジスタ機能(次の命令を変更する(next instruction modify)、機能デスティネーション 16) によって使用されるために。

(FORCE-DISPATCH <exp>)　<exp>の値を返し、命令を強制的に DISPATCH 命令にする。競合はエラーになる。

(FORCE-JUMP <exp>)　　<exp>の値を返し、命令を強制的に JUMP 命令にする。

(FORCE-ALU <exp>) <exp>の値を返し、命令を強制的に ALU 命令にする。

(FORCE-BYTE <exp>)　　<exp>の値を返し、命令を強制的に BYTE 命令にする。

(FORCE-DISPATCH-OR-BYTE <exp>)　　<exp>の値を返し、命令を強制的に DISPATCH か BYTE 命令にする。

(FORCE-ALU-OR-BYTE <exp>)　　<exp>の値を返し、命令を強制的に ALU か BYTE 命令にする。

(I-MEM-LOC <tag>) 記憶場所を I-MEM とする <tag>で表現されたアドレスを、右詰めの値として、返す。

(D-MEM-LOC <tag>) 記憶場所を D-MEM とする <tag>で表現されたアドレスを、右詰めの値として、返す。

(A-MEM-LOC <tag>) 記憶場所を A-MEM とする <tag>で表現されたアドレスを、右詰めの値として、返す。

(M-MEM-LOC <tag>) 記憶場所を M-MEM とする <tag>で表現されたアドレスを、右詰めの値として、返す。

(EVAL <lisp exp>) <exp>を LISP の S 式として評価した結果を返す。

(FIELD <name> <value>)　フィールド<name>が指定されたということは、注意すべきことであり、<name>の値と<value>を乗算する； もし<name>が、LISP の CONS-LAP-ADDITIVE-CONSTANT プロパティを持っていれば、それが足し合わされる。(この不明瞭さは、全フィールドの指定がなされるプリミティブであるからである)

(ERROR)　　もし、これがアセンブルされたら、エラー。条件の中で役立つ。

どのように条件が式の中で使用されるかの例として、次の定義を考える(それらは、CONSLP で実際に使用されるものと同様である(しかしまったく同一ではない))：

```
(ASSIGN Q-R (OR (SOURCE-P (FIELD M-SOURCE 7))
                (FORCE-ALU 3)))


(ASSIGN MEMORY-DATA
    (OR (SOURCE-P (FIELD M-SOURCE 12))
        (FIELD FUNCTIONAL-DESTINATION 30)))


(ASSIGN MEMORY-DATA-START-WRITE
    (OR (SOURCE-P (ERROR))
        (FIELD FUNCTIONAL-DESTINATION 32)))
```

Miscellaneous Pseudo-Operations　その他の擬似操作

こう書くことで、いくつかの同一の語が、連続的にアセンブルされるだろう：

```
(REPEAT <count> <word>)
```

現在の記憶場所中に、ロケーション・カウンタをセットするには、次のように

```
(LOC <value>)          ;<value>にセットする。
(MODULO <n>)           ;次の<n>の倍数まで進める
```

MODULO 操作は、A メモリ中で使用される、飛ばされた番地に定数を詰めることで浪費は起きない。

CADR Features and Programming Examples CADR の特徴と、プログラム例

この節では、CADR マシンの様々な特徴を確認し詳細を議論する。各機能がどれだけ機械の全構造の中にフィットしているか、特徴の目的がどう意図されたか、を、感じてもらえるように、試みている。各特徴を使用した短いプログラム例が各特徴を見せている。

Timing - The N Bit and the POPJ Bit タイミング - N ビットと POPJ ビット

CADR は、現在の命令を実行するのと同時に、次の命令をフェッチするので、その時間による JUMP か DISPATCH の効果として、JUMP か DISPATCH に続く命令がすでにフェッチされていることが知られている。N ビットが省略されていなければ、命令が分岐する前にこの命令は実行される。
プログラミング上では、この効果は、「分岐は、一つ前の命令のところに書け(コーディングしろ)("code the branch one instruction sooner")」となる。CONSLP ニーモニックは、通常、N ビットがセットされたさまざまな分岐操作のためのものを提供している、したがって、サイクルの浪費のコストはそのまま素直に増加する; N ビットをクリアするには、"-XCT-NEXT"をニーモニックに付けなければならない、そしてコードはマニアックになる。

例として、以下の 2 つのコード断片を考える:

```
((A-FOO) XOR M-BAR A-FOO)            ;A-FOO と M-BAR の XOR を A-FOO に
(JUMP-IF-BIT-SET MUMBLE MUMBLIFY)    ;MUMBLE bit が ON で分岐


(JUMP-IF-BIT-SET-XCT-NEXT MUMBLE MUMBLIFY)      ;MUMBLE bit が ON で分岐
((A-FOO) XOR M-BAR A FOO)            ;A-FOO と M-BAR の XOR を A-FOO に
```

これらはどちらも、XOR して、MUMBLIFY へ条件ジャンプする、しかし、一つ目は、JUMP が成功するとき、サイクルを 1 つ浪費する。XCT-NEXT の影響下で命令を"かくちょー (exdenting)"する習慣は、それをよりはっきりさせることに注目すべし。

もし CALL 遷移型が実行されたら、N ビットに依存して、戻り番地が SPC スタックにセーブされる:

```
(CALL THE-SUBROUTINE)          ;call, N bit がセットされている
((A-FOO) XOR M-BAR A-FOO)      ;call 後、ここに戻る

(CALL-XCT-NEXT THE-SUBROUTINE)       ;call, N bit がクリアされている
((A-ARGUMENT) ADD M-BAZ A-FOO)       ;サブルーチンに入る前にこれが行われる
((A-FOO) XOR M-BAR A-FOO)            ;call 後、ここに戻る
```

もし N ビットがセットされていたら、PC+1 が SPC スタックにプッシュされ、そうでなければ、PC+2 がプッシュされる。

POPJビットは、どの命令でもセットできる。それは、RETURN遷移を起こすが、次の命令が実行された後だけである：

```
ADD-THREE-WORDS          ; A-1, A-2, A-3を加えるサブルーチン
       ((M-RESULT) A-1)
       (POPJ-AFTER-NEXT (M-RESULT) ADD M-RESULT A-2)
       ((M-RESULT) ADD M-RESULT A-3)
```

もう一度。アイデアは、"命令一つ早めに"必要とされる制御を指定することである。
次のプログラムを考える：

```
START (JUMP-XCT-NEXT FOO)
     (JUMP-XCT-NEXT BAR)
       ...
FOO    (JUMP-XCT-NEXT FOO)
       ...
BAR    (JUMP-XCT-NEXT BAR)
```

STRATの位置でスタートしたとき、FOOとBARを交互に実行する無限ループに入る。実効的には、2つの「ジャンプ場所」の中で、同時にループする。

Byte Manipulation バイト操作

Mの2番地(慣習により、0のソース)をBYTE命令と一緒に使うことにより、Aメモリのどの番地の、どのビットか、どのビット・フィールドでもクリアできる：

```
       ((A-FOO) DPB M-ZERO A-FOO (BYTE-FIELD 1 31.))   ;符号ビットのクリア
```

-1(全部1)を持った別なMメモリの番地を予約するのは、たびたび、とても便利だ、ビットのセットを簡単にする：

```
       ((A-FOO) DPB M-ONES A-FOO (BYTE-FIELD 1 31.))   ;符号ビットのセット
```

同様なやり方で、符号付き24bit数を32bitに拡張するルーチンを書くことができる。

```
SIGN-EXTEND                    ;24-bit数をM-NUMに拡張
       (POPJ-AFTER-NEXT POPJ-IF-BIT-CLEAR M-NUM (BYTE-FIELD 1 23.))
    ((M-NUM) SELECTIVE-DEPOSIT M-NUM (BYTE-FIELD 24. 0) (A-CONSTANT -1))
```

```
Another way to do this, which doesn't require the use of POPJ, is to
use OA modification to select whether the M source is M-ZERO or
M-ONES:
```

POPJを必要としない、別な方法でこれを行うと、MソースにM-ZEROかM-ONEを選ぶためにOA変更を使用する：

```
       ((OA-REG-HI) (BYTE-FIELD 1 23.) M-NUM) ;下位Mソース・ビットは符号を得る
```

```
    ((M-NUM) SELECTIVE-DEPOSIT M-ZERO (BYTE-FIELD 8 24.) A-NUM)
```

これは、M-ZERO と M-ONES が 偶数/奇数の組であることを要求する。

通常、バイトは M ソースからだけロードできる。しかし、A メモリから、バイトをロードすることもできる、ワードの片方の端で可能である、次のトリックによって：

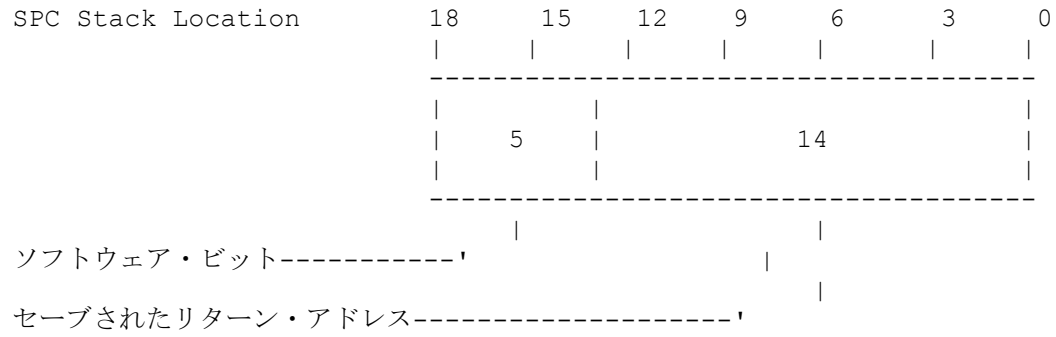
```
(DEF-DATA-FIELD X-FIELD 6 0)
(DEF-DATA-FIELD ALL-BUT-X-FIELD 32 6)

      ((DEST) SELECTIVE-DEPOSIT M-ZERO ALL-BUT-X-FIELD A-FOO)
```

The Instruction Stream 命令ストリーム

<<何か新しいことが書かれるべき>>

The SPC Stack SPC スタック

SPC スタックは 32 番地の長さ、各番地は、19bit を持つ(それに加えてパリティ)。それは、SPCPTR、5bit のアップ/ダウン・カウンタ、で指される。それは、主に、マイクロコードのサブルーチンのリターン・スタックとして使われる。しかし、マイクロコードの PC をセーブするには 14bit が必要とされ、その横に、ソフトウェア使用のための 5bit があり、上で述べたマイクロ命令のペアをフェッチする機能のために、ビットが使用される。



SPC スタック・メモリへ書き込むには、2 つの方法がある；どちらも SPCPTR をインクリメントする、つまり push 操作を起こす。CALL 遷移型 (P ビットがセットされ，R ビットがクリア) を実行する JUMP か DISPATCH が、先に述べたように、リターン・アドレスをスタックにプッシュする。5 つのソフトウェア・ビットは 0 にセットされる。機能デスティネーション 15 (MICRO-STACK-DATA-PUSH) への書き込みは、出力バスのデータの下位 19bit を SPC スタックにプッシュする。

SPC スタックは RETURN 遷移型 (R ビットがセットされ，P ビットがクリア) を実行する JUMP か DISPATCH により読まれる； 下位 14bit がスタックからポップされ、PC に置かれ、そして、NEXT-INSTR を起こすビット 14 を除

いて、ソフトウェア・ビットは無視される。それは、M 機能ソース 1 と 14 としても読むことができる。最初の (MICRO-STACK-PNTR-AND-DATA) は、単にスタック・トップ上のデータ (と SPCPTR) を読み、2 番目の (MICRO-STACK-PNTR-AND-DATA-POP) はデータを読んだ後、スタックをポップする。SPCPTR の内容を明示的にセットする方法は無い。しかし、次のループは使えるいいトリックである：

```
FOO    ((M-TEMP) MICRO-STACK-POINTER-POP)  ;SPCPTR だけを得る
       (JUMP-IF-EQUAL M-TEMP A-ZERO FOO)
```

より良いトリックは次のループを使うことだ、短いだけではなく、繰り返しの
代わりに再帰を使う、そして重要な優位性は、より分かりにくいということだ：

```
FOO    (CALL-NOT-EQUAL MICRO-STACK-PNTR-AND-DATA
             (A-CONSTANT (PLUS 1 (I-MEM-LOC FOO))) FOO)
```

これは、初期化で行うと良い、スタックが分かった場所から始まることで、診断インターフェースを通したデバッグの助けになる。

SPC スタックのオーバフローやアンダーフローを検出するための機能は存在しない。サブルーチンのネスティングが深さ 32 を超えないようにするには、プログラマの責任である。

The PDL BUFFER Memory PDL バッファ(PDL BUFFER)・メモリ

PDL バッファは、LISP マシン中で、Lisp プッシュダウン・スタックのトップ部分を保持するための、特別用途のキャッシュとして使用することを意図したものである。それは 32bit で 1024 番地を持ち、PDL POINTER か PDL INDEX で指される。PDL POINTER は 10bit のアップ/ダウン・カウンタであり、PDL INDEX は単純な 10bit のレジスタである。

PDL バッファは様々な機能ソースと機能デスティネーションを通して操作される。PDL POINTER と PDL INDEX レジスタは読み書きできる。(CONS では、これらは一緒に読み出すことしかできなかった。が、CADR では、バイト取り出しをまずやる必要無しに、それらの算術演算を行うことが容易になるように、それらは別々に読める。) PDL INDEX の内容でアドレスされる PDL バッファの番地の内容が読み書きされる。PDL POINTER の内容でアドレスされる PDL バッファの番地の内容が読み書きされ、この PUSH と POP 操作の時は、オプションで、PDL POINTER をインクリメントかデクリメントできる。ポインタのデクリメントは読み出しの後で、インクリメントは書き込みの前である、つまり、いつも最トップの有効な番地を指している。

C-PDL-BUFFER-POINTER-PUSH と C-PDL-BUFFER-POINTER-POP の両方を同じ命令中で指定するのは、動作しない。ひるがえって、C-PDL-BUFFER-POINTER をソースとデスティネーションの両方に使用することによって、単純に同じ効果を、常に達成できる。

PDL バッファのプッシュとポップのオーバフローやアンダーフローの自動的な検出は提供されていない。Lisp マシンでは、PDL POINTER は、各機能 (各関数) のエントリ部分と、数カ所の必要な場所で、チェックされる。もし、PDL バッファに最大サイズのフレームのための十分な空きが無ければ、空きを作るために、PDL バッファのいくつかは、メイン・メモリにストアされる。また、もし、PDL を割り当てるための仮想メモリに十分な空きが残っていなければ、PDL-OVERFLOW エラーがシグナルされる。同様に、関数の出口コードは、スタックをメイン・メモリから引き戻すかどうかを決定する。

--EOF