試作LISPマシンの高速化について
The high-speed LISP Machine

金田 悠紀夫* 小林 康博* 前川 禎夫* 瀧 和男**
Yukio KANEDA Yasuhiro KOBAYASHI Sadao MAEKAWA Kazuo TAKI
* 神戸大学システム工学科
* Depatment of Systems Engineering, Kobe University
** 日立製作所大みか工場
** Omika Works, Hitachi, Ltd.

1. まえがき

LISP言語の普及に伴い、LISP言語で記述されたプログラムの高速処理を目的とした専用コンピュータの研究が進められるようになった。[9]－[11] われわれはLISP言語を高速に処理できるハードウェア、ファームウェア構造をもつ計算機システムの開発を目的として、LISP処理のための機能を極力ハードウェア化し、市販のLSIを利用し、インタプリタの全てをマイクロプログラム化した、LISPマシンの研究と試作を行ってきた。[1]－[4] 本論文は、試作システムの概要を述べるとともに、高速化に役だつハードウェア機能を示し、インタプリタ、及びコンパイラの制御構造について述べ、その高速性を明らかにする。

2. LISPシステムの概要

2.1 LISPマシンシステムの構成

LISPマシンシステムのハードウェア構成を図1に示す。LISPプログラム処理の中心となるプロセッサモジュールとメモリモジュールを市販のLSI－ミニコン（DEC社LSI－11）のバスに接続した構成である。LSI－11からはメモリモジュールへページモードでアクセスでき、またプロセッサモジュールのCMR（コマンドレジスタ）やWCS（writable control storage）に対しても同様にアクセス可能である。システム構成は、LSI－11がマスターCPUであり、独自の主記憶を持ち、LISPプロセッサと並行して動作できる。LSI－11の仕事は、始動時や保守時にLISPプロセッサのプログラムロードやメモリ、レジスタの初期化、デバッグの援助を行ない、実行時には、LISPプロセッサからの割込みをうけ、入出力プログラムの一部と入出力機器の制御を行ない、時間計測も行なう。

メモリモジュールは32ビット×64K語から構成され、プロセッサモジュールは処理幅が16ビット、アドレス空間も16ビット、メモリとのデータの受け渡しは32ビット幅で行なわれる。またcar部、cdr部を独立に書き込むことも可能である。

2.2 LISPプロセッサモジュールの構成

プロセッサモジュールの構成を図2に示す。ビットスライスALUはAdvanced Micro Device 社の4ビットスライス Am2903を4個使用している。また制御部を構成するCCUはAm2910マイクロプログラムシーケンサ、WCS、PL（パイプラインレジスタ）、フラグレジスタ等から成り立っている。マイクロ命令サイクルは300nsecを基本とし、命令コードは1語56ビットでALUまわりとCCUの制御とを並列的に行なえる。[1] [4]

2.3 高速化のためのハードウェア

LISPプログラム処理を高速に行なうため次のようなハードウェア機能を組み込んだ。また、その有効性も確かめられている。[3]

ハードウェアスタック： 70nsecの高速メ

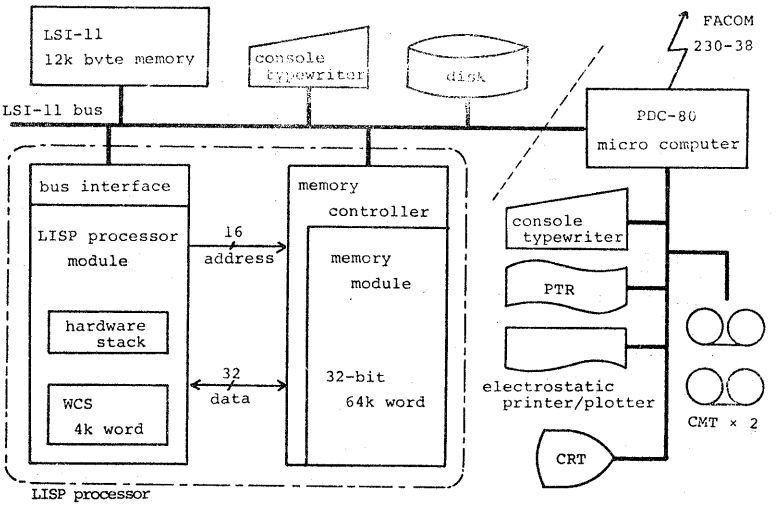

図1．LISPマシンシステムのハードウェア構成

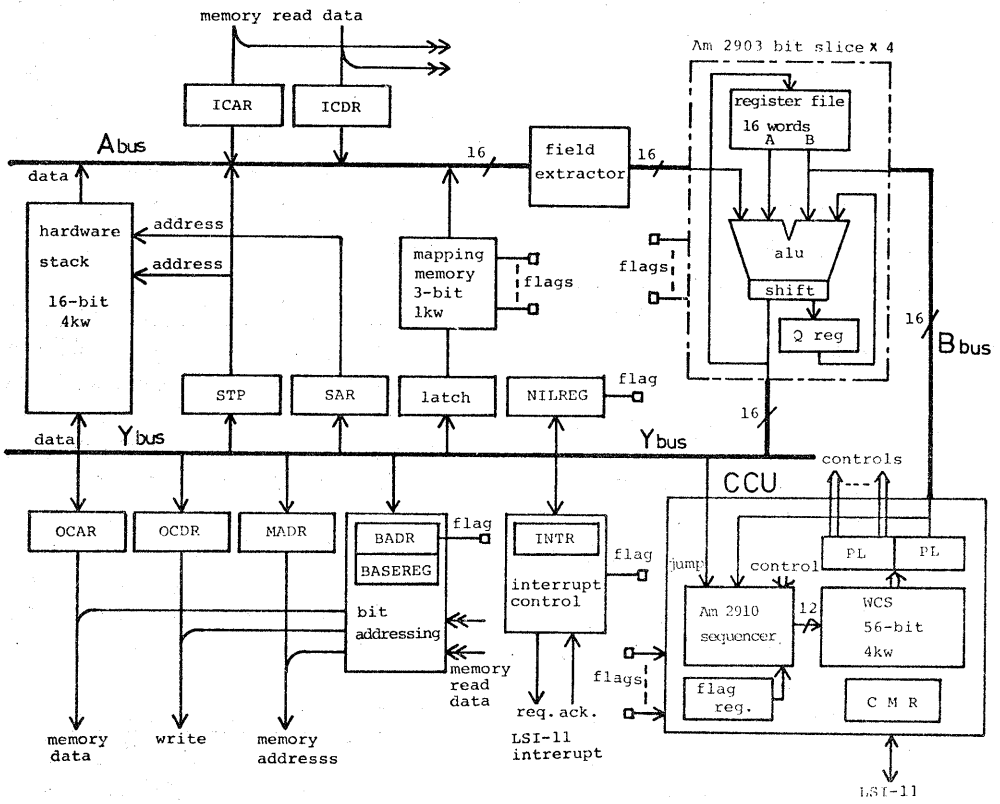

図2．LISPプロセッサモジュールの構成

モリによる4K語の固定長スタックである。スタックへのアクセスは2つのポインタ、STP, SARを用いて行なわれる。

バス構成：Aバス、Yバス、Bバスから構成されており、Yバスはマイクロプログラムシーケンサとも接続されており、演算結果の番地へのジャンプが可能である。

フィールド抽出回路：データのシフトとマスキングを同時に行なう。

マッピングメモリ回路とフラグ：マッピングメモリは3ビット×1K語の構成で、主記憶アドレスを入力として3ビットのコード（データ型に対する属性コード）を出力する。このコードやフィールド抽出回路の出力を定数（WCSアドレス）と加算することにより、マルチウェイジャンプが実現できる。またマッピングメモリの出力はデコーダを通してフラグレジスタにも接続されており、リスト、文字アトム、数値、アトムの判定が簡単に行える。

その他：WCSは56ビット×4K語から構成され、アクセスタイムは150 m sec である。NILレジスタはYバス上のデータとNILとの一致検出を行ない、フラグとして出力する。ビットアドレッシング回路は主記憶上のビットテーブルに対し、ビットテストとビットセットを行うもので、ガーベージコレクションにおけるマーキング操作に用いる。

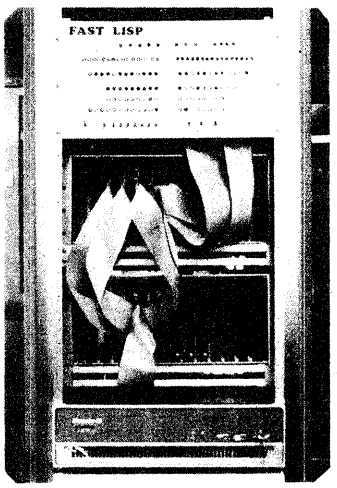

図3．LISPマシン全景

2.3 ハードウェアの製作について

LISPプロセッサ部分の構成は、17cm×22 cm の万能基板（100ピンコネクタ）15枚を用いた構成（図3）で、そのうち分けは、プロセッサモジュールが7枚、メモリモジュールが8枚で、はんだ付けとワイヤラッピングにより配線を行なっている。総IC個数は約600個で、電源容量は5V－30A，12V－4Aである。また基板コネクタは100ピンでは不足であり、基板端にフラットケーブルコネクタを増設して多用している。図4はCPU基板である。ハードウェア実装上の問題点は、第一に、ショットキーTTL・ICの高速性に起因するもので、このICはスイッチング時のスパイク雑音が標準TTLの3倍も大きいため、電源のバイパスコンデンサの強化と太く短い接地ラインが必要となった。第二には、基板分割に関する問題であり、システムが大きくなると使用IC個数が増し、高速動作を要求すると短い配線が必要となる。そのためIC実装個数を増さねばならないが、それには限界がある。許された実装方法のもとでの高密度実装、短い配線、基板間の配線本数の最小化を実現することが難しくなる。本システムでは、WCS、ALUまわりでほぼ限界の実装状態であり、処理幅を変更しようとすると実装方法自体を変更しなければならなくなると考えられる。第三は発熱の問題で、高密度実装のため、自然空冷で間に合わなくなっており、12個の小型ファンにより強制空冷を行っている。以上のことから、本システムよりも1クラス上のシステムを製作しようとすると

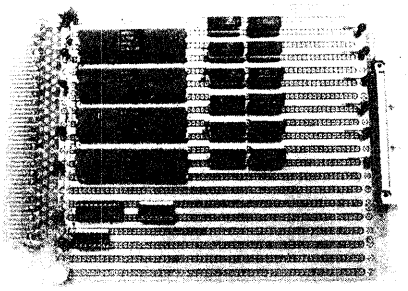

図4．ALU基板

実装設計が格段に難しくなると思われる。

本システムの稼動率は1日約5時間程度で、稼動してから約1年を経過しているが、その間の故障は、コンデンサの絶縁不良1ヶ所、ICの接触不良2ヶ所の故障があっただけである。

3. LISPマシンのインタプリタ

3.1 インタプリタの構造

本システムのインタプリタはすべてマイクロプログラム化されている。そのため、マイクロプログラムレベルでの再帰呼び出し、及びハードウェアスタックを有効に利用できるよう、LISP言語処理の制御構造を次のように構成した。関数の実行に対応してスタック上にframeが作られる。(図5) frame headには関数本体、リターンアドレスを示すMPCレジスタ、リストをたどるPCレジスタ、1つ前のframeを示すFPが含まれる。関数による評価がおわるとスタックトップの値を現FP値に置き、現フレームを消滅させ、戻り番地へ間接ジャンプしてゆく。この制御構造のもとでインタプリタを構成したが、処理速度向上のために注意をはらった点を列挙すると、

(1) EVALルーチンの改良

関数が文字アトムの時、引数評価にEVLISを用いず、スタックを用いて引数の受け渡しを行なう。またframeの作成は最少限におさえており、引数評価はARGEVALルーチンで専用に行われる。

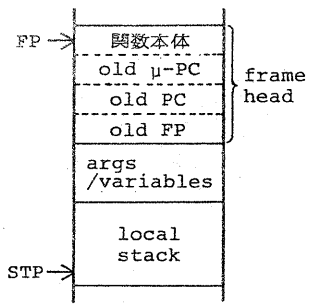

図5. frameの構造

(2) APPLY ルーチンの改良

関数がEXPR、LAMBDAに相当する場合を中心に設計しており、EVALとAPPLYとの呼び出しはループですましている。

(3) ハードウェアの活用

if-then-else型の条件ジャンプをやめ、データ型によるコードを用いてマルチウェイジャンプを実現した。またCAR、CDR等のプリミティブな関数は直接他の関数からマイクロサブルーチンとして呼び出せるよう設計した。無条件ジャンプの際にはALU関係の処理を並行して行い、ステップ数を短縮している。

3.2 インタプリタによる実行時間

第2回LISP性能コンテスト[4]に出されたプログラムを用いて、本システムにおける実行時間を測定した。その結果を表1に示す。表記方法と他の処理系の実行時間は文献[5]に従っている。比較の対象として選んだ他の処理系の特徴を簡単にまとめると、

(i) HLISP : HITAC-8800上に構成されたシステムであり、CPUサイクルタイムは50 n sec、メモリアクセスタイムは100 n secの大型計算機システムである。

(ii) OLISP : ACOS-800上に構成されたシステム

(iii) TOSBAC-5600 LISP1.9 : TSS向きのLISPシステムで、CPUサイクルタイムは900 n secである。

(iv) LIPQ : ミニコンピュータPDP 11/55上に構成されたシステムで、CPUサイクルタイムが300 n sec、メモリサイクルタイムが同じく300 n secで本システムと同等である。

表1をみると、まず本LISPマシン(表中FAST-LISP)における実行時間が他のどの大型計算機よりも短いことがわかる。本システムのサイクルタイムが300 n secであるから、HLISPと単純に比べると6倍遅いことになるが、これはハードウェア、ソフトウェア両面での改良の効果が十分に現われているためと考えることができる。[3]

3.3 Shallow-binding と Deep-binding

変数値のbind手法において従来より、

	-----INTERPRETER-----						-----COMPILER-----				
	FAST-LISP KOBE UNIV.	HLISP	OLISP	TOSBAC-5600 LISP1.9	LIPQ		FAST-LISP KOBE UNIV.	HLISP	OLISP	TOSBAC-5600 LISP1.9	LIPQ
BITA-4	6	6	14	24	90	*	2	2	4	7	18
BITA-5	17	22	46	76	300	*	9	9	15	22	63
BITA-6	55	71	155	249	1,010	*	27	30++	50	71	219
BITA-7	188	249	531	936	3,460	*	92	108+	176	248	775
BITA-8	652	869	1,849	2,925	---	*	320	377++	622	499	---
BITA-9	STACK OVER	3,111+	6,498++	15,236++	---	*	1,128	1,359++	2,229	2,705	---
BITB-4	3	6	5	11	45	*	3	1	1	4	9
BITB-5	6	14	13	23	104	*	6	3	3	6	25+
BITB-6	13	40	34	59	282	*	13	10++	9	14	75+
BITB-7	37	118+	104	172	870	*	28	33	28	39	258+
BITB-8	120	394+	338	549	3,070+	*	88	105++	91	126	1,080++
BITB-9	397	1,318+	1,130	1,832	---	*	292	356++	307	712	---
BITB-10	1,355(+)	4,542++	---	---	---	*	1,000	1,230++	---	---	---
SORT-20	73	106	287	505	1,400	*	26	12	18	31	23++
SORT-40	261	374	1,109	1,415	4,900	*	94	43	69	108	85++
SORT-60	415	---	2,459	2,061	7,800	*	150	---	153	176	140++
SORT-80	626	568	4,337	3,121	11,900	*	225	65	271	487+	210+
SORT-100	804	1,134	6,753	6,154	15,000	*	290	135	423	605+	270+
TARAI-3	55	78	197	227	900	*	26	17	36	87	13
TARAI-4	1,013	1,443	3,727	5,124	16,000	*	473	330	670	1,064	255
TARAI-5	27,538	39,068	100,734	138,819	437,000	*	12,849	8,905	18,934	43,609	6,900
TARAI-6	1,011,732	---	---	---	---	*	472,063	322,347	---	1,603,910	255,000
TPU-1	904	4,790	2,262	4,403+	12,000+	*	601	1,029	658	1,591	2,200+
TPU-2	3,426	13,411	10,262	21,148++	55,200+	*	1,957	2,871+	2,064	6,764+	6,100+
TPU-3	1,365	5,684	3,932	7,447+	21,200++	*	805	1,227++	850	2,798+	2,600+
TPU-4	1,815	8,024	5,042	10,958+	27,400++	*	1,096	1,707	1,161	3,250+	3,700+
TPU-5	238	866	759	1,461+	4,000++	*	125	181	132	555	400
TPU-6	6,728(+)	22,849	20,241	66,502++	118,000++	*	3,590(+)	4,893+	3,617	27,049++	10,500+
TPU-7	1,311	5,078	4,179	8,350+	21,800++	*	710	1,053	776	3,189+	2,300+
TPU-8	1,187	3,459	3,987	5,459+	21,000++	*	582	724+	596	1,163+	1,600+
TPU-9	819	2,663	2,716	4,133+	14,200++	*	411	545	424	964+	1,100+
PLISP-3	7,717	3,096	46,061	49,125	---	*	2,934	303	1,876	2,933	2,240
PLISP-3	163	132	575	871	2,150	*	163	132	575	871	2,150
RATIO	47.3	23.5	80.1	56.4	---	*	18.0	2.3	3.26	3.4	1.04
PLISP-4	170,471(+)	56,877	1,096,786	1,118,014	---	*	65,356(+)	4,998	44,910	66,722	48,500
PLISP-4	3,059(+)	2,020	11,384	18,119	44,000	*	3,059(+)	2,020	11,384	18,119	44,000
RATIO	55.7	28.1	96.3	61.7	---	*	21.4	2.5	3.95	3.7	1.1
PLISP-5	STACK OVER	---	---	---	---	*	2,107,056(+)	125,290	---	2,015,495	---
PLISP-5	83,321	46,431	---	503,999	1,249,000	*	83,321	46,431	---	503,999	1,249,000
RATIO	---	---	---	---	---	*	25.3	2.7	---	4.0	---

表1. bench mark program による実行時間 (m sec)

shallow-bindingとdeep-bindingのどちらが有効であるか議論されている。表1のインタプリタによる実行時間はshallow-bindingを用いた場合の結果であるが、こころみにdeep-bindingによるインタプリタの実行時間を測定してみた。その結果を表2に示しておく。本システムにおいて、deep-bindingは主記憶上にLIFOスタックを構成すること、shallow-bindingはbind時にold-valueをハードウェアスタックにつみ、rebindを行うframeを作ることで各bind手法を実現した。表2より、本システムではshallow-bindingの方が高速に処理でき、TARAI、SORTでは自由変数がないため10%程度のオーバヘッドとなっているが、TPUでは3つの自由変数があるためオーバヘッドが約40%に増大していることがわかる。deep-bindingがshallow-bindingよりも速くなるためにはshallow-bindingにおけるrebind処理に要する時間よりも、変数値を得る時間の方が速ければよいのだが、本システムではshallow-bindingを行う際、ハードウェアスタック上にアトムとそのold-valueを対にして積むためrebind処理を、比較的高速に処理できる。3変数の場合を例にとると、shallow-bindingではrebindに15ステップ、変数のアクセスに9ステップ、計24ステップですむが、deep-bindingの場合だと変数のアクセスに36ステップを要する。これはdeep-bindingではAPVALチェック、アトムの比較等がどうしても必要となるため、多くのステップ数が必要となる。この差が、表2の10%前後のオーバヘッドとなっている。以上のようにshallow-bindingを用いると高速処理は可能であるが、FUNARG機能を利用する際不都合な事が起きる。これは関数引数に自由変数が含まれ、かつ複雑個の入式中で入変数を使用した場合、不合理な結果を得てしまうのであるが、shallow-bindingで正常な結果を得るために、その「環境」を保存する機能を実現することは困難である。そこで環境を意識して書いたプログラムの処理にはdeep-binding、高速処理を望むプログラムの処理にはshallow-bindingを用いるのがよいと思われる。

表2. shallow-bindingとdeep-bindingによる実行時間 (m sec)

	SHALLOW BINDING	DEEP BINDING	RATIO (%)
TARAI-3	55	62	113
TARAI-4	1,013	1,145	113
TARAI-5	27,538	31,124	113
TARAI-6	1,011,732	1,143,501	113
SORT-20	73	78	107
SORT-40	260	277	107
SORT-60	416	440	106
SORT-80	626	663	106
SORT-100	803	850	106
TPU-1	904	1,119	124
TPU-2	3,426	4,794	140
TPU-3	1,365	1,850	135
TPU-4	1,815	2,395	132
TPU-5	238	341	143
TPU-6	6,728	9,221	137
TPU-7	1,311	1,882	144
TPU-8	1,187	1,768	149
TPU-9	819	1,205	147

4. LISPマシンのコンパイラ

4.1 中間言語命令

本システムはマイクロプログラムにより制御されているため、機械語命令を本質的には持たない。そのため、コンパイラを構成する場合、機械語命令に対応した中間言語命令を設定し、それをオブジェクトコードとして生成するコンパイラを考えた。オブジェクトコードは1中間言語命令に対し、32ビットの命令コードに変換され、主記憶内のプログラム領域にロードされる。コンパイルされた関数の実行時には、マイクロプログラムで記述された命令フェッチ部と命令実行部(デコーダ)により、中間言語命令が順にフェッチされ、実行される。

これら中間言語命令の設定の際、メモリ参照があまり負担にならない事、パイプライン処理を損わない事などに留意して、図6のように命令コードを構成した。中間言語命令としては、次のような命令体系を設定した。

(1) CALL命令 : operandで示された関数を、その属性に応じた実行の予備操作を行い、その後関数を呼び出す命令である。

(2) PSH命令 : operandで示されたデータをスタックトップに積み上げる命令であり、operandとしては (i) 実行する関数の実引数 (ii) 現在の自由変数 (iii) データそのも

の、の3種類が指定でき、以下の命令においても同様である。

(3) POP、STORE命令 ：現在のスタックの内容をoperandで示されるフィールドにコピーする命令で、SET、SETQに対応している。POP命令はコピー後STPを1つポップアップする。

(4) MKFRM命令 ：関数の実行に先だち、frame-head を作る命令で、operand の指定があるとPSH命令もかねる。

(5) JMP命令 ：条件により、label 先へ分岐するか否かを制御する命令であり、STPをポップアップする事も可能である。条件として、(i) 無条件分岐、(ii) NIL分岐、(iii) T分岐 の3種類が指定できる。

(6) RETURN命令 ：現在のスタックトップの内容を関数の評価結果として、frame を消滅させ、呼び出し元へリターンする。operand の指定があるとその値を評価結果とする。

(7) SHALLOW 命令 ：コンパイル時に自由変数が宣言されていると、その自由変数が束縛を受ける際、shallow-binding を行う命令である。

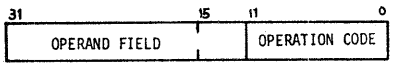

図6．命令コード

```
(DE TARAI (X Y Z)
 (COND
  ((GREATERP X Y)
   (TARAI (TARAI (SUB1 X) Y Z)
          (TARAI (SUB1 Y) Z X)
          (TARAI (SUB1 Z) X Y)))
   (T Y) ))
```

```
(LAP SUBR TARAI 3
     (MKFRM 1 -6)
     (PSHARG -6)
     (RCALL GREATERP)
     (JMPPOPNIL G0)
     (MKFRM 3 -14)
     (RCALL SUB1)
     (PSHARG -10)
     (PSHARG -10)
     (SCALL TARAI)
     (MKFRM 2 -14)
     (RCALL SUB1)
     (PSHARG -10)
     (PSHARG -13)
     (SCALL TARAI)
     (MKFRM 2 -14)
     (RCALL SUB1)
     (PSHARG -13)
     (PSHARG -13)
     (SCALL TARAI)
     (SCALL TARAI)
     (RETURN)
G0   (PSHARG -2)
     (RETURN) )
```

図7．関数定義体とそのコンパイル例

4．2 コンパイラの構造と実行時間

本システムのコンパイラはマイクロプログラムでは書かれず、LISP言語で記述されたEXPR関数であり、モニタレベルから実行可能である。このコンパイラの簡単なM式を図8に示す。図8のM式の中で使用されている処理ルーチン名について、簡単に説明すると、

c-exp	::＝コンパイル処理
catom	::＝アトム処理
cquote	::＝QUOTE処理
cfunc	::＝FUNCTION処理
cprog	::＝PROG処理
ccond	::＝COND処理
csetq	::＝SETQ処理
clambda	::＝LAMBDA処理
clabel	::＝LABEL処理
mkspec	::＝自由変数処理
mkpsh	::＝束縛変数処理

```
c-exp[form] = [
 atom[form] → catom[form]
 eq[car[form];QUOTE] → cquote[cdr[form]]
 eq[car[form];FUNCTION] → cfunc[cdr[form]]
 eq[car[form];PROG] → cprog[cdr[form]]
 eq[car[form];COND] → ccond[cdr[form]]
 atom[car[form]] → [
  eq[car[form];GO] → cpgo[cdr[form]]
  eq[car[form];RETURN] → cret[cdr[form]]
  eq[car[form];SETQ] → csetq[cdr[form]]
  eq[car[form];SET] → cset[cdr[form]]
  T → funcall[form] ]
 eq[caar[form];LAMBDA] → clambda[form]
 eq[caar[form];LABEL] → clabel[form]
 T → c-exp[list[APPLY;
                list[QUOTE;car[form]]
                cons[LIST;cdr[form]]
          ]    ]  ]
```

```
catom[X] = [get[X;APVAL] → mkapval[X]
            numberp[X] → mknum[X]
            special[X] → mkspec[X]
            localp[X] → mkpsh[X]
            T - error[] ]
```

```
clambda[X]=append[complis[cdr[X]]
                  bind[cadar[X]]
                  c-exp[caddar[X]]]
```

図8．コンパイラのM式の例

bind　　::＝bind及びrebind処理
funcall　::＝関数呼び出し手続き処理

ＥＸＰＲ関数をＳＵＢＲ関数にコンパイルすることで実行速度を改善できる点は

(1) 引数値をローカルスタックのロケーションに割当る事により、重複して行われる引数評価、及び変数のbind処理を行わなくてすむこと。

(2) 再帰呼び出しを行わない関数群（マイクロサブルーチンＳＵＢＲ関数）については、frameを作らないですむこと、

(3) COND、PROG、AND、SETQなどのＦＳＵＢＲ関数群は引数の評価手順を示したものであるから、中間言語命令で簡単にオープンに展開することが可能であること。

(1)については、ＥＸＰＲ関数を評価する際、変数のbindのため、ＥＶＡＬからAPPLYを呼び、bindを行い、さらにＥＶＡＬを呼び出す手順が必要であるが、ＳＵＢＲ関数では、変数はスタックのロケーションで表わされるため、そのロケーションをアクセスするだけで済み、大幅なステップ短縮が可能になる。ただし、自由変数の場合、スタックのロケーションに割当ることはできないので、コンパイル時にその宣言を行わなければならない。また(2)、(3)より、関数呼び出しのためのframeの作成、連続したＥＶＡＬの再帰呼び出し、RETURN処理等を省略することが可能になる。上記の3点だけでも、コンパイルすることにより、総実行ステップ数の50%以上の短縮が可能である。図7に関数定義体とそのコンパイルしたオブジェクトの例を示しておく。また、ＬＩＳＰコンテストのプログラムをコンパイルした場合の実行時間を表1に示しておく。

4．3　今後の改良点(フェッチ部のハードウェア化)

表1の実行時間で、インタプリタとコンパイラの場合を比較してみると、コンパイルすることによっても、2～3倍程度の処理速度の向上しか得られていない。これはインタプリタが高速に動作しているためもあるが、中間言語命令のフェッチ、及びデコードをマイクロプログラムによるソフトウェアで構成したためと考えられる。特に命令コードのフェッチを行う部分は関数の実行とは直接関係のない部分である。そこで、このフェッチによるオーバヘッドをなくすために、ハードウェアによる命令コードの先取りを行った場合の実行時間を推定してみた。つまり、プログラム領域の命令コード専用のＰＣ（プログラムカウンタ）を用意し、あらかじめ命令コードを読み出しておき、その命令コードを実行している間にＰＣにより次の命令コードを読み出すという機能をハードウェアで実現した場合の実行時間である。表3は、このようなハードウェア化により、フェッチのオーバヘッドが完全に取り除かれた場合の推定実行時間である。表3より、TARAI、SORTのような組み込み関数をあまり使用しないプログラムでは25%程度の改善率が期待できそうであるが、ＴＰＵのような組み込み関数を多用するプログラムでは15%前後の改善しか望めない。また分岐命令によるパイプライン処理の損失による改善率の低下も考えられるが、プログラム中の分岐命令の使用頻度は、TARAI-3で、1.9%、ＴＰＵ-6で、1.6%なので、これによる影響はさほどないそうである。しかし実現するには、ＷＣＳのビット幅も変更しなければならず、実装上の問題もあり、早急な実現は難しいと考えられる。

表3．命令の先取りを行った場合の実行時間とその改善率　(m sec)

	EXECUTION TIME	ESTIMATED EXECUTION TIME	RATIO (%)
TARAI-3	27	20	25.7
TARAI-4	473	347	26.6
TARAI-5	12,850	9,414	26.7
TARAI-6	472,063	345,857	26.7
SORT-20	21	16	27.1
SORT-40	72	51	29.1
SORT-60	119	84	29.1
SORT-80	179	126	29.3
SORT-100	235	166	29.3
TPU-1	601	547	9.0
TPU-2	1,957	1,708	12.8
TPU-3	805	709	12.0
TPU-4	1,096	973	11.2
TPU-5	125	108	13.9
TPU-6	3,590	3,073	14.4
TPU-7	710	617	13.1
TPU-8	582	488	16.1
TPU-9	411	348	15.4

5. あとがき

本システムにおいて、ＬＩＳＰ言語を高速に処理するための要点をハードウェア、ソフトウェアの両面から紹介してきた。ＬＩＳＰマシンシステムとして本システムを見た場合、まだ入出力関数、組み込み関数の充実が行われておらず、応用プログラムを本システムで実動化することはできないでいる。今後の研究課題としては早急にストリング処理機能、数値演算処理機能を充実させることである。ともあれ、新しいアーキテクチャを持つ計算機が研究室レベルで試作可能な時代を迎えたことは喜ばしいことである。

謝辞

本研究に協力して下さったシステム工学科４講座の三上昭弘氏、高岸英之氏に感謝します。

参考文献

[1] 瀧、金田、前川：ＬＩＳＰマシンの試作、－アーキテクチャとＬＩＳＰ言語の仕様－ 情報処理学会論文誌, Vol.20.No.6 (Nov.1979)

[2] 瀧、金田、前川：ＬＩＳＰマシンの試作、－インタプリタの構造とシステムの評価－ 情報処理学会論文誌、Vol.20.No.6 (Nov. 1979)
瀧、小林、金田、他：試作ＬＩＳＰマシンとその評価、情報処理学会研究会資料、記号処理7-3 (Mar. 1979)

[4] Y.KANEDA,K.TAKI: The Experimental LISP Machine, IJCAI prceedings,(Aug. 1979).

[5] 竹内：第２回ＬＩＳＰコンテスト、情報処理 Vol.20,No.3 1979

[6] 山口、島田：仮想計算機によるＬＩＳＰプログラムの動的特性、信学論文誌D, J61－D,8.(Aug. 1978)

[7] 黒川：ＬＩＳＰのデータ表現、情報処理 Vol.17.No.2 (1976).

[8] J.Allen: Anatomy of LISP, McGrawHill,1978.

[9] W.Teitelman: INTERLISP Reference Manual Xerox,(Feb. 1974).

[10] LISP User'sManual, EPICS-5-ON-4 (Mar. 1978).

[11] 長尾、中村、他：ＬＩＳＰマシンNK3のアーキテクチャとそのマイクロインストラクション 信学 研、EC-77-17 (1977).