ICOT Technical Memorandum: TM-0392

TM-0392

遂次型推論マシン：CHI

幅田伸一，中崎良成，新　　淳
小長谷明彦，梅村　護
（日本電気）

October, 1987

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

"Co-operative High performance sequential Inference machine: CHI" (in Japanese)

by Shin-ichi Habata,Ryousei Nakazaki et al. (C&C Systems Research Laboratories,NEC Corporation, 4-1-1 Miyazaki Miyamae-ku,Kawasaki,213,Japan)

A Co-operative High performance sequential Inference (CHI) machine has been developed within the Fifth Generation Computer Systems (FGCS) project in Japan. This paper describes the CHI-II (advanced version CHI) hardware architecture, including design approach, its characteristics and performance estimation.

The CHI-II was designed for personal use and to be as small as a desk-side computer, using two CMOS gate-array chips. In the design, tag architecture, large main memory capacity and Prolog oriented specialized hardware were considered for executing micro operations in parallel and reducing the number of micro steps required in Prolog program execution. In 6MHz machine clock, CHI-II performance is estimated to be at 500 KLIPS.

逐次型推論マシン：ＣＨＩ

畑田　伸一、中崎　良成、小長谷　明彦、新　淳、梅村　護
日本電気（株）　Ｃ＆Ｃシステム研究所

本報告では、通産省の第５世代計算機プロジェクトの一環として、研究開発した逐次型推論マシンＣＨＩの設計思想、高速化手法、性能評価を中心に、小型化版ＣＨＩのハードウェア・アーキテクチャについて述べる。

小型化版ＣＨＩは、ＣＭＯＳゲートアレイによるＬＳＩ化などにより、デスクサイド型マシンを目指している。タグ・アーキテクチャ、大容量主記憶、ＰＲＯＬＯＧの為の専用ハードウェア・ユニットを採用し、マイクロ操作の並列処理化を進め、マイクロステップ数を減らすことに努めた。さらに、新しい命令を追加しハードウェアの機能を発揮させるように努めた。この結果、”ａｐｐｅｎｄ”プログラムの実行において、１要素の処理に必要な命令が３命令、マイクロステップ数が１２ステップとなり、６ＭＨｚのマシン・クロックで５００ＫＬＩＰＳの処理速度を得ることができた。さらに、ＣＨＩのデータと命令共用のキャッシュメモリをデータ専用と命令専用に分割することで、１要素の処理に必要なマイクロステップ数を８ないし９ステップにできることが分かった。

Co-operative High performance sequential Inference machine : CHI

S.Habata,R.Nakazaki,A.Konagaya,J.Atarashi and M.Umemura

C&C Systems Research Laboratories,NEC Corporation
4-1-1 Miyazaki Miyamae-ku,Kawasaki 213,Japan

(発表分野)

A Co-operative High performance sequential Inference (CHI) machine has been developed within the Fifth Generation Computer Systems (FGCS) project in Japan. This paper describes the CHI-II (advanced version CHI) hardware architecture,including design approach, its characteristics and performance estimation.

The CHI-II was designed for personal use and to be as small as a desk-side computer, using two CMOS gate-arrays. In the design, tag architecture, large main memory capacity and Prolog oriented specialized hardware were considered for executing micro operations in parallel and reducing the number of micro steps needed in Prolog program execution. New instructions were introduced, in order to take advantage of CHI-II multiple micro operation execution. In this result, 3 machine instructions and 12 micro steps are required for a logical inference in append program. In 6 MHz machine clock, CHI-II performance is estimated to be at 500 KLIPS. Moreover, using data cache and instruction cache, 8 or 9 micro steps are required for a logical inference in append program.

1．はじめに　近年、知識情報処理の研究及び知識情報処理に基づいたシステムの開発が活発である。知識情報処理のプログラム記述言語としては、ＬＩＳＰ，ＰＲＯＬＯＧ等がある。このなかで、一階述語論理をベースとしたプログラミング言語は通産省の第５世代計算機プロジェクトの核言語として採用されている。したがって、述語論理に基づいたプログラミング言語で記述される知識情報処理システムが多数出現することが期待できる。

ＰＲＯＬＯＧなどの述語論理に基づいたプログラミング言語は、優れた記述機能とバックトラックを使用した解の探索機能を備えているため、従来のプログラミング言語より知識情報処理に適している。その反面、言語処理系の負荷が従来言語より大きいため、従来の汎用計算機では充分な開発環境と実行環境を得ることが難しい。そこで、第５世代計算機プロジェクトの一環として、述語論理に基づいたプログラミング言語で記述した知識情報処理システムを高速処理する専用マシン（ＣＨＩ）の開発を行った。

2．開発の経緯　ＣＨＩは単一ユーザ用の逐次型推論マシンを目標にしている。当プロジェクトでは、ＣＭＬ素子を使用したプロトタイプ（ＣＨＩ－Ｉ）と、ＴＴＬとＣＭＯＳ素子を使用した小型化版（ＣＨＩ－ＩＩ）を開発した。ＣＨＩハードウェアの特徴は、

・タグ・アーキテクチャ
・バックエンド型マシン
・大容量主記憶
・高度に専用化したハードウェア・ユニット

にある。ＣＨＩシステム実現上必要な機能である利用者インタフェース及び周辺装置の制御などは、従来の計算機システムの機能を利用することにした。そのため、バックエンド型マシンの構成を採用した。ＣＨＩアーキテクチャは、Ｄ．Ｈ．Ｄ．Ｗａｒｒｅｎの提案したＷＡＭアーキテクチャ[1]をもとに、改良を加えたものである。プロトタイプは、ＣＨＩアーキテクチャの検証を目的に、１９８５年に完成し、マシン・クロック１０ＭＨｚで最高２８０ＫＬＩＰＳ（ａｐｐｅｎｄ実行時）の性能を達成した。[2] この成果を基に、開発したのが小型化版ＣＨＩ（ＣＨＩ－ＩＩ）である。プロトタイプと小型化版ＣＨＩの諸元を表１に示す。

3．小型化版ＣＨＩの設計思想　小型化版ＣＨＩ開発にあたって、以下の目標を設定した。

(1) デスクサイド型マシン
(2) 処理速度の向上
(3) 主記憶容量の拡張
(4) ホストマシンとの非同期双方向通信
(5) 汎用命令の導入

項(1)は、プロトタイプは、アーキテクチャの検証、高性能の実現を最優先にして開発した。小型化版ＣＨＩは、オフィス等への設置を目指し開発している。項(2)は、尽きることのない知識情報処理研究者の要求に答えるべく、処理速度の向上を目指している。項(3)は、実用的な応用システムを高速実行できる環境を提供するため、大きなメモリ空間を半導体メモリで提供することを目指している。項(4)は、ホスト・マシンが備えている周辺装置の機能をＣＨＩから使用できる環境を目指している。第５は、組込み述語の拡張及び修正が容易な環境を目指している。

4．小型化版ＣＨＩのハードウェア技術　小型化し、且つ、主記憶容量を増加し、且つ、処理速度を向上させるため、以下のハードウェア技術を採用している。

・ＣＭＯＳゲート・アレイによるＬＳＩ化
・６４キロ・ビットＳ－ＲＡＭ
・ＰＡＬ，ＰＬＡ
・１メガ・ビットＤ－ＲＡＭ
・サーフェース・マウントによる両面実装基板

プロセッサの制御系を２種類のＣＭＯＳゲート・アレイでＬＳＩ化している。ＬＳＩ化した部分のゲート規模は、約２万ゲートで、ＣＨＩ－ＩＩプロセッサの約５分に１にあたる。開発したＣＭＯＳゲート・アレイの諸元を表２に示す。機械語命令コードの解読及び、タグ値の解読は、高速大容量Ｓ－ＲＡＭで実現し、機械語命令セットの変更及び、タグ値の変更を容易にしている。さらに、高速大容量Ｓ－ＲＡＭの使用は、ＩＣ数の減少、遅延時間短縮の効果もあった。ＣＭＯＳゲート・アレイによるＬＳＩ化が出来なかった制御系の大部分は、高速ＰＡＬ，ＰＬＡを使用し、ＩＣ数の減少、遅延時間の短縮を試みている。主記憶容量の拡張は、１メガ・ビットＤ－ＲＡＭの使用とサーフェース・マウントによる両面実装基板の使用で実現している。

5．小型化版ＣＨＩシステムの構成　小型化版ＣＨＩシステムは、利用者インタフェース及び、周辺装置の制御を行うホストマシンとＣＨＩ－ＩＩマシンで構成する。（図１）ホ

表１　プロトタイプと小型化版ＣＨＩの諸元

比較項目	CML版CHI	小型化版CHI
使用素子	CML 256 Kbit D-RAM	TTL，CMOS 1 Mbit D-RAM
基板枚数 プロセッサ部 主記憶部	81 枚 約110 枚	4 枚 16 枚
マシン・クロック	10 MHz	6 MHz
データ幅	36 ビット tag 4または，7 value 32または，29	40 ビット tag 8 value 32
主記憶容量	64 MW （256MB）	128 MW （640MB）
キャッシュメモリ容量	8 KW （32KB）	32 KW （160KB）
マイクロ命令長	80 ビット	76 ビット
制御メモリ容量	11 KW （110KB）	16 KW (約160KB)
"append" 処理速度	280 KLIPS	600 KLIPS
ホストマシン	PSI または，	EWS4800

表２　ＣＭＯＳゲートアレイの諸元

	MAB	SOB
ゲート数	約60，000 ゲート	約13，000 ゲート
機能	・キャッシュメモリ制御 ・主記憶アクセス制御 ・命令先取りポインタ制御	・マイクロシーケンス制御

ストマシンは、マルチウィンドウ機能を備えている。小型化版ＣＨＩの利用者は、ホストマシン上のＣＨＩ用ウィンドウを使用し、小型化版ＣＨＩをアクセスする。ホストマシンとＣＨＩ－ＩＩマシンは標準インタフェースの１つであるＳＣＳＩで結合している。ＣＨＩ－ＩＩマシンは、ホストマシンとの通信制御を行うホスト・インタフェース・ユニット、主記憶ユニットとプロセッサからなる。図２にＣＨＩ－ＩＩマシンの構成図を示す。

主記憶ユニットは、論理メモリ空間と物理メモリ空間のマッピングを行うアドレス変換部を含む。図３に小型化版ＣＨＩの論理メモリ空間を示す。１つのプロセスは、８個の独立した領域を使用する。その内の４個の領域は、全プロセスに共通の領域となる。論理アドレスは、プロセス番号、領域番号、領域内アドレスの３フィールドからなる。（図４）

６．処理速度の改善　　ＣＭＬ素子からＴＴＬ及びＣＭＯＳ素子への変更は、消費電力、発熱量を少なくしたが、マシン・クロックを下げる原因となった。マシン・クロック低下の問題を解決し、プロトタイプ以上の処理速度を実現するため、以下の点に重点をおいた設計を行った。

・メモリ・アクセス速度
・マイクロ操作の並列処理
・マイクロ分岐の高機能化
・ハードウェア機能の効果的利用

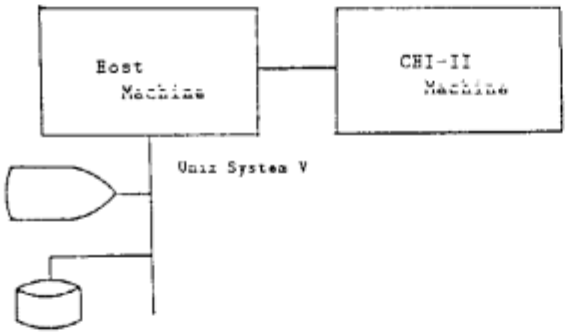


図１　小型化版ＣＨＩシステムの構成

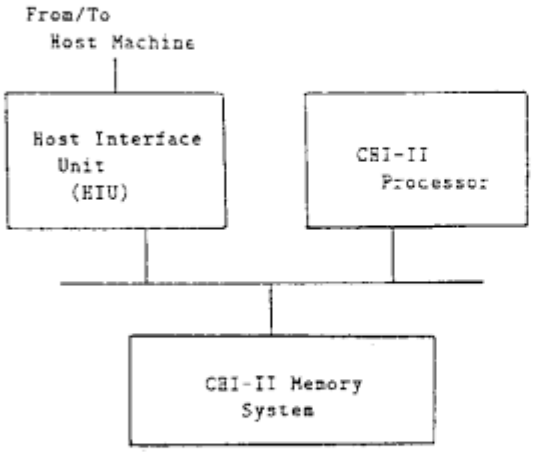


図２　ＣＨＩ－ＩＩマシンの構成

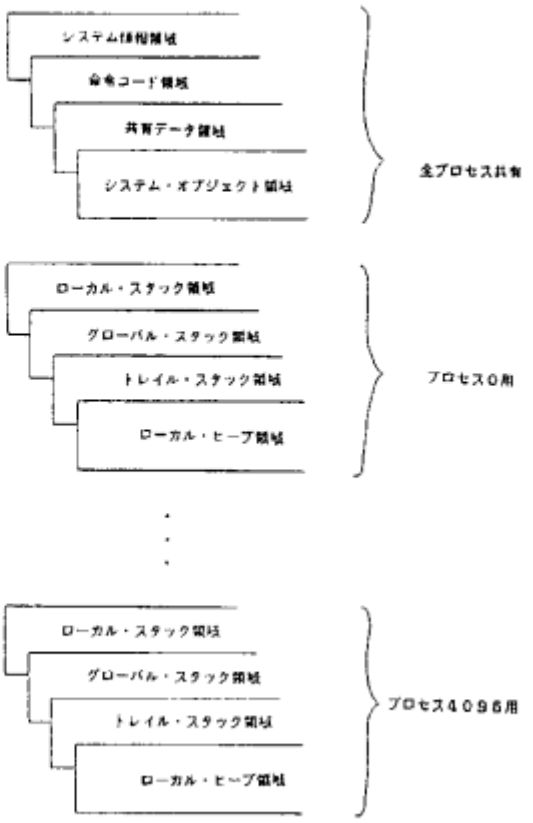


図３　論理メモリ空間の構成

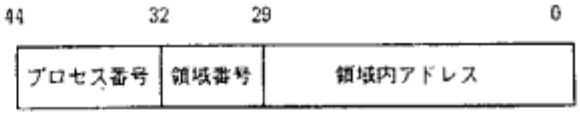


図４　論理アドレス・フォーマット

メモリ・アクセス速度　　メモリ・アクセスの点で、ＰＲＯＬＯＧの言語処理系は２つの特徴がある。第１は、ＰＲＯＬＯＧの言語処理系は、述語読み出し及び、バックトラック処理の為、図５に示すフレームをメモリに生成する。これらのフレームはローカル・スタックと呼ぶ領域に格納する。[1] ＰＲＯＬＯＧプログラムの実行において、述語呼出の頻度は高く、また、バックトラックはＰＲＯＬＯＧの特徴である解探索の基本手段であるから、上記フレームの生成、アクセス、消去の頻度は高い。したがって、ＰＲＯＬＯＧで記述したプログラムの実行では、従来のプログラミング言語で記述したプログラムの実行の場合より、頻繁にメモリ・アクセスが発生する。第２は、ＰＲＯＬＯＧの基本処理であるユニフィケーション処理で変数同士の単一化を行った時に、２つの変数を束縛する手段として参照ポインタを使用する。変数間の単一化が複雑に絡み合うと、データをアクセスするために、上記参照ポインタのチェーンを辿る必要が生じる。参照ポインタ・チェーンを辿る処理は、参照ポインタが指すデータを読

BLC	:	前選択点フレームへのポインタ
BALT	:	次の候補節へのポインタ
BLE	:	選択点の環境フレームへのポインタ
BTTOP	:	選択点のトレイル・スタック・ポインタの値
BGTOP	:	選択点のグローバル・スタック・ポインタの値
BCP	:	選択点の述語呼出から復帰番地
BAN	:	選択点の引数の個数（値をａとする）
A(0)	:	第１引数の値
：		：
A(n-1):		第ｎ引数の値

（ａ）選択点フレーム

ELE	:	前環境フレームへのポインタ
ECP	:	述語呼出から復帰番地
BHC	:	バインドフック・ハンドラへのポインタ
P(0)	:	第１永久変数
：		：
P(m-1):		第ｍ永久変数

（ｂ）環境フレーム

図５　スタック・フレームの構成

み出し、データタイプを調べる操作からなる。読み出したデータが参照ポインタであれば、さらに、参照ポインタを辿る操作を繰り返すため、メモリ・アクセスをパイプライン処理化して得られる効果が小さい。そこで、メモリ・アクセスをアドレス・セットとデータ・アクセスの２段階に分け、マシン・サイクル時間をキャシュメモリのアクセス時間から決定した。

マイクロ操作の並列処理　　マシン・サイクル時間が決まると、１マシン・サイクル内で、可能な限り多くのマイクロ操作を並列実行させるように努めた。このため、容量が６４語のレジスタ・ファイルに４個の読み出しポートと２個の書込ポートを設け、４個のデータ読み出しと２個のデータ書込を１マイクロ・サイクルで処理する。汎用の算術論理演算器の他に、アドレス演算専用の加減算器を用意し、データの演算とアドレスの演算を並列処理する。さらに、メモリ・アドレス・レジスタのインクリメント／デクリメント用加算器、命令先取りアドレス演算用加算器を用意し、データとアドレスの演算の有無に拘らず、メモリ・アドレス・レジスタの更新及び命令先取りアドレスの演算を可能にする。タグ操作機能としては、メモリとレジスタにデータを書込むときに、データのタグを個々に生成することを可能にする。ハードウェア構成設計の目標は、マイクロプログラムの実行時間を、そのマイクロプログラムが行うメモリ（キャッシュメモリ）・アクセスに必要な時間に近づけることである。図６にプロセッサ部の構成を示す。

マイクロ分岐の高機能化　　ＰＲＯＬＯＧ言語処理系では、動的データタイプの検査及び、メモリ・アクセス位置の検査を行う。データタイプの検査機能としては、１データのタグ値検査、２つのデータのタグ値の組合わせの検査がある。１データのタグ値を使用したマイクロプログラム・レベルの多方向分岐機能では、特定方向への分岐を、次の機械語命令の呼出に置き換えている。２つのデータのタグ値の組合わせを使用した多方向分岐では、バリュー部を比較し、一致、不一致の結果も分岐方向の決定に使用する。この場合も、特定方向への分岐を、次の機械語命令の呼出に置き換えることができる。表３にタグ値を使用した多方向分岐の制御の流れを示す。メモリ・アクセス位置の検査は、算術論理演算器とアドレス演算用の加減算器の演算結果を使用する多方向分岐機能である。この時、論理アドレスの領域番号フィールドを比較し、一致、不一致の結果も分岐方向決定に使用する。小型化版ＣＨＩが提供するメモリ・アクセス位置検査機能を図７に示す。

ハードウェア機能の効果的利用　　小型化版ＣＨＩでは、マイクロ操作の並列処理、マイクロ分岐の高機能化を進め、マイクロプログラムの実行に要するマイクロステップ数が減

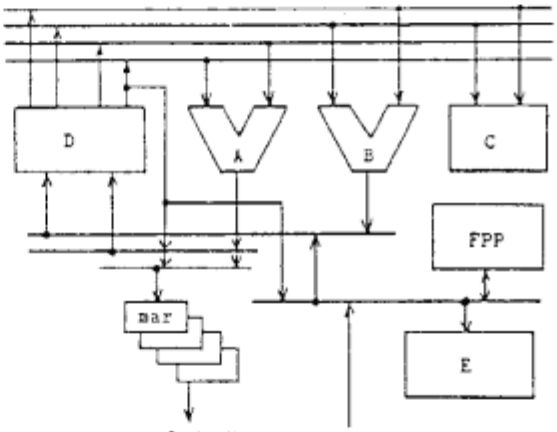
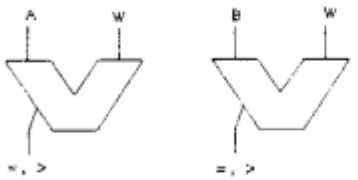


図６　プロセッサ部の構成

表３　タグ値を使用した多方向分岐機能

タグ値の組合わせ	バリュー部の比較結果	分岐機能
alos : alos	Ｘａ ＝ Ｘｂ Ｘａ ≠/≈ Ｘｂ	次の命令呼出 ｆａｉｌ処理
nil : nil	----	次の命令呼出
funb : funb	Ｘａ ＞ Ｘｂ	変数Ｘａから変数Ｘｂへのポインタ再込
	Ｘａ ＜ Ｘｂ	変数Ｘｂから変数Ｘａへのポインタ再込



領域番号の関係	領域内アドレスの関係	分岐オフセット	アドレスの関係
Aa = Ba = Wa	A =< V , B > V	0	A =< V < B
Aa = Ba > Wa	A > V , B =< V	1	B =< V < A
Aa = Ba = Wa	A =< V , B =< V	2	A , B =< V
Aa < Ba > Wa	A > V , B > V	3	V < A , B
Aa = Ba =/= Wa	----	4	A、BとWは別領域
Aa = Wa =/= Ba	A =< V , ---	0	A =< V
Aa > Wa =/= Ba	A > V , ---	1	A > V
Aa =/= Wa < Ba	--- , B =< V	2	B =< V
Aa =/= Wa > Ba	--- , B > V	3	B > V
Aa =/= Wa =/= Ba	--- , ---	4	比較困難

図7　メモリ・アクセス位置検査機能

少するようにハードウェア機能を設計している。特に、マイクロ操作を並列処理するため、多くのデータパス及び演算器を用意している。しかし、WAMで提案された機械語命令セットは、小型化版ＣＨＩのハードウェア機能を充分に発揮できない。そこで、インデクシング関係及びリストのユニフィケーション関係の命令を新たに追加している。”ａｐｐｅｎｄ”プログラムを例に、新命令の効果を説明する。図８の（ａ）がWAMの提案している命令セットで記述した”ａｐｐｅｎｄ”プログラムである。＊印の付いている命令が、

　？－append（［a,b,c］,［d,e,f］,X）.

に対して実行される機械語命令である。１要素の処理に、８命令、２１マイクロステップを必要とする。図８の（ａ）の＃印の付いている命令

```
        switch_on_term $nl,$ls,$fail
  $ls: get_list A0
```

を１命令にした複合命令

```
        switch_on_list
```

を使用した”ａｐｐｅｎｄ”プログラムが図８の（ｂ）である。第１引数に対する get_list 命令が、 switch_on_list 命令に吸収され、１要素の処理に必要な命令数が７命令、１９マイクロステップに減少する。図８の（ｂ）の＃印の付いている命令

```
        get_list A2
        unify_temporary_value A3
        unify_temporary_variable A2
```

を１命令にした

```
      get_list_t_val_t_var A2,A3,A2
```

命令を使用した”ａｐｐｅｎｄ”プログラムが図８の（ｃ）である。第２引数のユニフィケーションのために展開されていた３命令が１命令となり、１要素の処理に必要な命令数が５命令、１７マイクロステップに減少する。図８の（ｃ）の＃印が付いている命令

```
        switch_on_list $nl,$ls,$fail
  $ls: unify_t_variable A3
```

を１命令にした

```
        switch_on_list_t_var A3,$nl,$ls,$fail
```

命令を使用した”ａｐｐｅｎｄ”プログラムが図８の（ｄ）である。第１引数のユニフィケーションを

```
        switch_on_list_t_var A3,$nl,$ls,$fail
  $ls: unify_t_variable A0
```

の２命令で処理する。１要素の処理に必要な命令数は４命令、１５マイクロステップとなる。最後に、図８の（ｄ）で＃印が付いている命令

```
        execute $ap
```

をインデクシング命令

```
        switch_on_list_t_var A3,$nl,$ls,$fail
```

```
**********************************************************
**  append([],X,X).                                     **
**  append([X|A],B,[X|C]):-append(A,B,C).               **
**********************************************************

#*  $ap:  switch_on_term                  $nl,$ls,$fail
    $v0:  try_me_else                     $v1
    $nl:  get_nil                         A0
          get_t_value                     A1,A2
          proceed
    $v1:  trust_me_else_fail
#*  $ls:  get_list                        A0
 *        unify_t_variable               A3
 *        unify_t_variable               A0
 *        get_list                       A2
 *        unify_t_value                  A3
 *        unify_t_variable               A2
 *        execute                        $ap

                (a)

 *  $ap:  switch_on_list                  $nl,$ls,$fail
    $v0:  try_me_else                     $2
          get_nil                         A0
    $nl:  get_t_value                     A1,A2
          proceed
    $v1:  trust_me_else_fail
          get_list                        A0
 *  $ls:  unify_t_variable                A3
 *        unify_t_variable                A0
#*        get_list                        A2
#*        unify_t_value                   A3
#*        unify_t_variable                A2
 *        execute                         $ap

                (b)

#*  $ap:  switch_on_list                  $nl,$ls,$fail
    $v0:  try_me_else                     $v1
          get_nil                         A0
    $nl:  get_t_value                     A1,A2
          proceed
    $v1:  trust_me_else_fail
          get_list                        A0
#*  $ls:  unify_t_variable                A3
 *        unify_t_variable                A0
 *        get_list_t_val_t_var            A2,A3,A2
 *        execute                         $ap

                (c)

 *  $ap:  switch_on_list_t_var            A3,$nl,$ls,$fail
    $v0:  try_me_else                     $v1
          get_nil                         A0
    $nl:  get_t_value                     A1,A2
          proceed
    $v1:  trust_me_else_fail
          get_list                        A0
          unify_t_variable                A3
 *  $ls:  unify_t_variable                A0
 *        get_list_t_val_t_var            A2,A3,A2
#*        execute                         $ap

                (d)

(*) $ap:  switch_on_list_t_var            A3,$nl,$ls,$fail
    $v0:  try_me_else                     $v1
          get_nil                         A0
    $nl:  get_t_value                     A1,A2
          proceed
    $v1:  trust_me_else_fail
          get_list_t_variable             A0,A3
 *  $ls:  unify_t variable                A0
 *        get_list_t_val_t_var            A2,A3,A2
 *        switch_on_list_t_var            A3,$nl,$ls,$fail
          goto                            $v0

                (e)
```

図８　”ａｐｐｅｎｄ”プログラムの展開例

に置き換えることで、１要素の処理に必要な命令数は３命令、１２マイクロステップとなる。１要素の処理に必要な命令数、マイクロステップ数、処理速度を、表４に示す。

新たに追加した命令としては、インデクシング命令では、

switch_on_list
switch_on_list_t_var
switch_on_list_t_val
switch_on_list_p_var
switch_on_list_p_val

ユニフィケーション用命令では、

get_list_t_var_t var
get_list_t_val_t_val
get_list_t_var_t_val
get list t val t var

などがある。新命令追加の為、オペレーション・コードは、１０ビットに拡張している。命令コードのフォーマットを図９に示す。

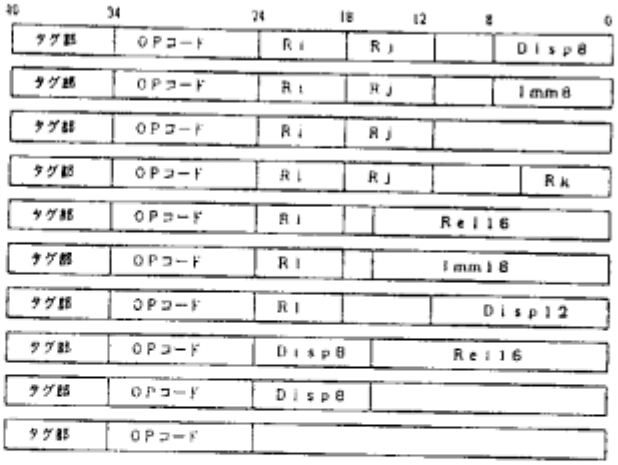


図９　命令コード・フォーマット

７．ホストマシンとの非同期双方向通信　　従来システムが備えている機能を流用し、開発を効率化するため、利用者とのインタフェス制御および周辺装置の制御をホストマシンに任せている。しかし、ＣＨＩ上で応用システムを開発すると、ホストマシン上の周辺装置を使用したり、利用者との会話形式の処理が必要となる。ＣＨＩ上の応用システムがホストマシンのＯＳ機能を利用するため、非同期双方向通信機能を用意している。小型化版ＣＨＩシステムにおけるホストマシンとＣＨＩ－ＩＩマシンの通信方式を図１０に示す。非同期双方向通信機能により、ＣＨＩ上の応用システムがホストマシンの二次記憶上のファイルをアクセスしたり、キー・ボードからの入力を受けたり、ビットマップ・ディスプレイ上のウィンドウを操作することができる。さらに、ＣＨＩ－ＩＩマシン上で異常が発生した場合は、ホストマシンがＣＨＩ－ＩＩマシンの状態のダンプ、再立上げの操作を行う。

８．汎用命令の導入　　ＰＲＯＬＯＧの組込み述語を実現する手段として、汎用命令を用意している。実用的な応用システムをＣＨＩ上で開発するためには、組込み述語が提供する算術演算機能や論理演算機能、ベクターの操作機能が必要となる。これらの機能を、状況に応じて、変更したり、拡張する環境を提供するため、大部分の組込み述語は汎用命令のルーチンで実現する。汎用命令の中には、直接、算術論理演算器などのハードウェア・ユニットにマイクロ制御コードを送る命令がある。これにより、本来ソフトウェア・レベルで操作できないハードウェア・ユニットを使用した多方向分岐命令などを実現している。さらに、マイクロプログラムを格納しているコントロール・ストレージの書き換え命令も用意しており、ＣＨＩ上のシステムがマイクロプログラムの変更を行うこともできる。

表４　小型化版ＣＨＩの" ａｐｐｅｎｄ" 処理速度

使用命令セット	命令数	マイクロステップ数	処理速度
問8の (a)	8 命令／節末	21 step／節末	285 KLIPS
(b)	7 命令／節末	19 step／節末	316 KLIPS
(c)	6 命令／節末	17 step／節末	352 KLIPS
(d)	4 命令／節末	16 step／節末	400 KLIPS
(e)	3 命令／節末	12 step／節末	500 KLIPS

９．性能評価　　機械語命令及び" ａｐｐｅｎｄ" プログラムの実行に必要なマイクロステップ数から、プロトタイプと小型化版ＣＨＩの性能評価を行った。評価は、ＰＲＯＬＯＧのユニフィケーション用命令と" ａｐｐｅｎｄ" プログラムの実行に必要なマイクロステップ数とメモリ・アクセス回数から、マイクロステップ数とメモリ・アクセス回数の比を求め、プロセッサの備えているハードウェア機能とマシン・サイクル時間について、検討してみた。ここで扱うデータは、キャッシュメモリのヒット率を１００％として、机上評価したものである。表５は、WAMで提案されたユニフィケーション用命令の実行に必要なマイクロステップ数とメモリ・アクセス回数及びそのデータを基に求めたマイクロステップ数とメモリ・アクセス回数の比である。ＣＨＩでは、データと命令がキャッシュメモリを共用している為、メモリ・アクセス回数は、データ・アクセス回数と命令アクセス回数の合計となる。プロトタイプを基にハードウェア構成を改善した小型化版ＣＨＩでは、ほとんどの命令の実行が１から２マイクロステップで完了するため、メモリ・アクセスの割合が高いことが分かる。マイクロステップ数減少の結果、表５の命令の平均実行速度に限ると、小型化版ＣＨＩはプロトタイプの約１．１倍高速化されている。また、メモリ・アクセスの割合はプロトタイプの約２倍の０．８８９になっている。表６は、" ａｐｐｅｎｄ" プログラム実行の場合のマイクロステップ数、メモリ・アクセス回数、メモリ・アクセスの割合

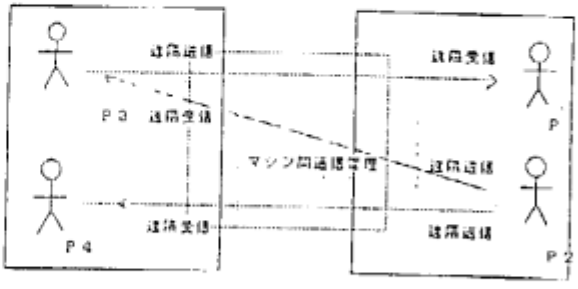


図１０　マシン間通信

を示している。小型化版ＣＨＩでは、WAMの提案した命令セットを使用した場合でも、メモリ・アクセスの割合は、０．８５０と高いことが分かる。さらに、新たに追加した命令を使用した場合は、０．９１７になっている。処理速度では、小型化版ＣＨＩはプロトタイプの１．７５倍高速化されている。

小型化版ＣＨＩのメモリ・アクセスの割合が０．８から０．９くらいの値になっていることは、マイクロプログラムの処理時間がマイクロプログラムが行うメモリ・アクセスに要する時間に近づいていることを示している。小型化版ＣＨＩでは、データと命令共用のキャッシュメモリを使用している為、命令読み出しとデータ・アクセスを別のマイクロステップで実行する必要がある。この為、"ａｐｐｅｎｄ"プログラムの実行に必要な１２ステップのなかに、マイクロ操作の並列処理、命令の先取り処理で省略可能なステップが３ないし４ステップ残っている。

１０．まとめ　小型化版ＣＨＩはデスクサイド型のパーソナルな逐次型推論マシンを目指し、ＣＭＯＳゲート・アレイによるＬＳＩ化などで小型化を実現した。さらに、処理速度を改善するため、汎用演算器の他に専用の演算器を用意するなどのマイクロ操作の並列処理化を進めた。その結果、処理に必要な時間を、その処理が行うメモリ・アクセス時間に非常に近づけることができた。"ａｐｐｅｎｄ"プログラムの例では、メモリ・アクセス時間と処理時間の比が０．９１７となっている。したがって、推論マシンのハードウェア・アーキテクチャを検討する上で、メモリ・アクセスを高速化することが非常に重要だとわかる。

ＣＨＩでは、キャッシュメモリをデータと命令共用としたが、"ａｐｐｅｎｄ"プログラムの場合、メモリ・アクセスは、データ・アクセス６回と命令アクセス５回からなっており、キャッシュメモリをデータ専用と命令専用に分け、マイクロ操作の並列処理度を高めると、"ａｐｐｅｎｄ"プログラムを実行するために必要なマイクロステップ数は８ないし９ステップにすることができることがわかった。すなわち、マシンサイクル時間が１００ｎＳｅｃで、１ＭＬＩＰＳ以上のマシンを実現できると考えられる。

謝辞：　最後に、本研究の機会を与えて下さったＩＣＯＴ第４研究室の内田室長、日本電気　Ｃ＆Ｃシステム研究所の石黒所長、同所　コンピュータ・システム研究部の山本部長に深謝致します。

参考文献：

(1) D.H.D.Warren, "An Abstract Prolog Instruction Set", Tech. report 309, Artificial Intelligence Center, SRI International,1983

(2) R.Nakazaki, et al. "Design of a High-speed Prolog Machine (HPM)", Proc. of the 12th International Symposium on Computer Architecture, June 1985

Table 5: Memory Access Ratio in typical 18 machine instructions

Machine Instruction	Micro Step Count CHI-I	Micro Step Count CHI-II	Memory Access Count	Memory Access Ratio CHI-I	Memory Access Ratio CHI-II
get_permanent_variable	2	2	2	1.000	1.000
get_temporary_variable	2	1	1	0.500	1.000
get_permanent_value	5	3	2	0.600	0.667
get_temporary_value	3	1	1	0.667	1.000
get_list	2	1	1	0.500	1.000
get_structure	6	3	2	0.500	0.667
put_permanent_variable	3	2	2	0.667	1.000
put_temporary_variable	3	2	2	0.667	1.000
put_permanent_value	4	3	2	0.500	0.667
put_temporary_value	3	1	1	0.333	1.000
put_list	3	1	1	0.333	1.000
put_structure	5	2	2	0.400	1.000
unify_permanent_variable	4	3	3	0.750	1.000
unify_temporary_variable	4	2	2	0.500	1.000
unify_permanent_value	5	3	3	0.600	1.000
unify_temporary_value	5	2	2	0.400	1.000
unify_list	2	1	1	0.500	1.000
unify_structure	6	3	2	0.333	0.667
total	67	6	32	—	—
average	3.7	2.0	1.8	0.478	0.889

Table 6: Performance Estimation in append program execution

	Machine Instruction Count/LI	Micro Step Count/LI	Memory Access Ratio	Performance (KLIPS)
CHI-I	9	35	0.457	285
CHI-II without compile optimization	8	20	0.850	294
CHI-II with compile optimization	3	12	0.917	500