8 AUG－'83 TR－021

汎用型マイクロプログラム・アセンブラ

高木茂行

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

Customizable Microprogram Assembler is described.

In the first development stage of computer systems, it is often the case that the firmware system has to be developed before the specification of the hardware is completely fixed. In such cases, the specification of the microprogram is likely to be changed frequently.

In order to cope with such frequent changes, we have developed a customizable microprogram assembler. Its kernel part, which describes machine independent functions of the assembler, is separated from machine dependent parts. Machine dependent descriptions are read in from machine definition files to customize the assembler.

In addition to this customization feature, checking machine dependent errors and/or printing out information other than the object program can be specified in the definition.

In this memo, we describe the specifications and the implementations of this assembler and, as an example, the microprogram assembler for the personal sequential inference machine PSI.

汎用型マイクロプログラム・アセンブラ

第３研究室　高木　茂行

目次

1. まえがき

複数のハードウェアに対応するマイクロプログラムや機械語のアセンブラを容易に作成するため，汎用アセンブラを用いる方法がある。汎用アセンブラを用いればアセンブラ作成の工数を大幅に短縮でき，実用上の価値が大きいと言える。

しかしアセンブラの核部が汎用であるため，ハードウェアに密着したエラー・チェック等は困難になる。特にソース・プログラム上では同じ字面の命令から，同一ステップ内の別のフィールドやそれ以前のステップの影響によって，異なるオブジェクトを生成しなければならないような場合の処理が難しい。

このような周囲に影響を受ける処理に柔軟に対処するため，バックトラックとユニフィケーションの機能を利用した汎用型アセンブラを作成した。これを用いれば従来は困難であった他のフィールドやステップからの影響も機械定義として容易に記述することができる。

本報告の第２章では今回作成したアセンブラの外部仕様と機能について述べる。第３章では外部仕様と機能を実現する方法とバックトラックを用いたエラー・チェックの方法について述べる。第４章では本アセンブラの適用例として，逐次型推論マシンφのマイクロプログラム・アセンブラについて述べる。第５章はむすびである。

2. 汎用型アセンブラ

2.1 はじめに

アセンブラを汎用化する主な目的の１つは，異なる機種あるいは新しい機種に対応するアセンブラを短期間に作成することであり，もう１つは作成したアセンブラの改訂を容易に行なうことである。このような目的の汎用アセンブラは多数開発されているが，ハードウェアに依存した部分が汎用化しきれない場合や，新しいハードウェアの出現に対応できない場合があり，これが決定版というものに至っていない。

今回開発したアセンブラもハードウェア・ファームウェアの仕様変更に柔軟に対応することを目的としており，完全に汎用というものは考えていない。しかし，従来の汎用アセンブラでは非常に難かしかったエラーの検出やメッセージの出力の部分については，ユーザの機械定義部分に大幅に取り入れた。これによって汎用アセンブラでありながら，エラー・チェックをかなりの程度行なうことが可能になり，アセンブリ言語のプログラマに対する負担を減らすことができるようになった。また，バックトラック機能を用いてオブジェクト・プログラムのフィールド形式の変更についても機械記述に吸収することが可能になった。

本章では今回開発した汎用型アセンブラの外部仕様と機能について述べる。

2.2 汎用型アセンブラの構成

汎用型アセンブラは機械定義前処理プログラム，アセンブラ本体，リンカから成る。従来の汎用アセンブラと同様，本アセンブラもアセンブラ本体と機械定義部から専用アセンブラを作成する方式をとっている。ユーザが作成した機械定義部を前処理プログラムを用いてアセンブラ本体と整合させ，前処理後の定義部と本体を合わせることによって専用アセンブラができあがる。

アセンブラは相対アドレス型式のオブジェクトを生成し，リンカによってそれを絶対形式に変換する。この時点でメモリ割付けのベース値を設定する。この方式を概念的に示したのが図 2.1である。

前処理プログラムによって機械定義部はアセンブラ本体と整合したプログラムに変換され，本体部と合せて各々の専用アセンブラができあがる。

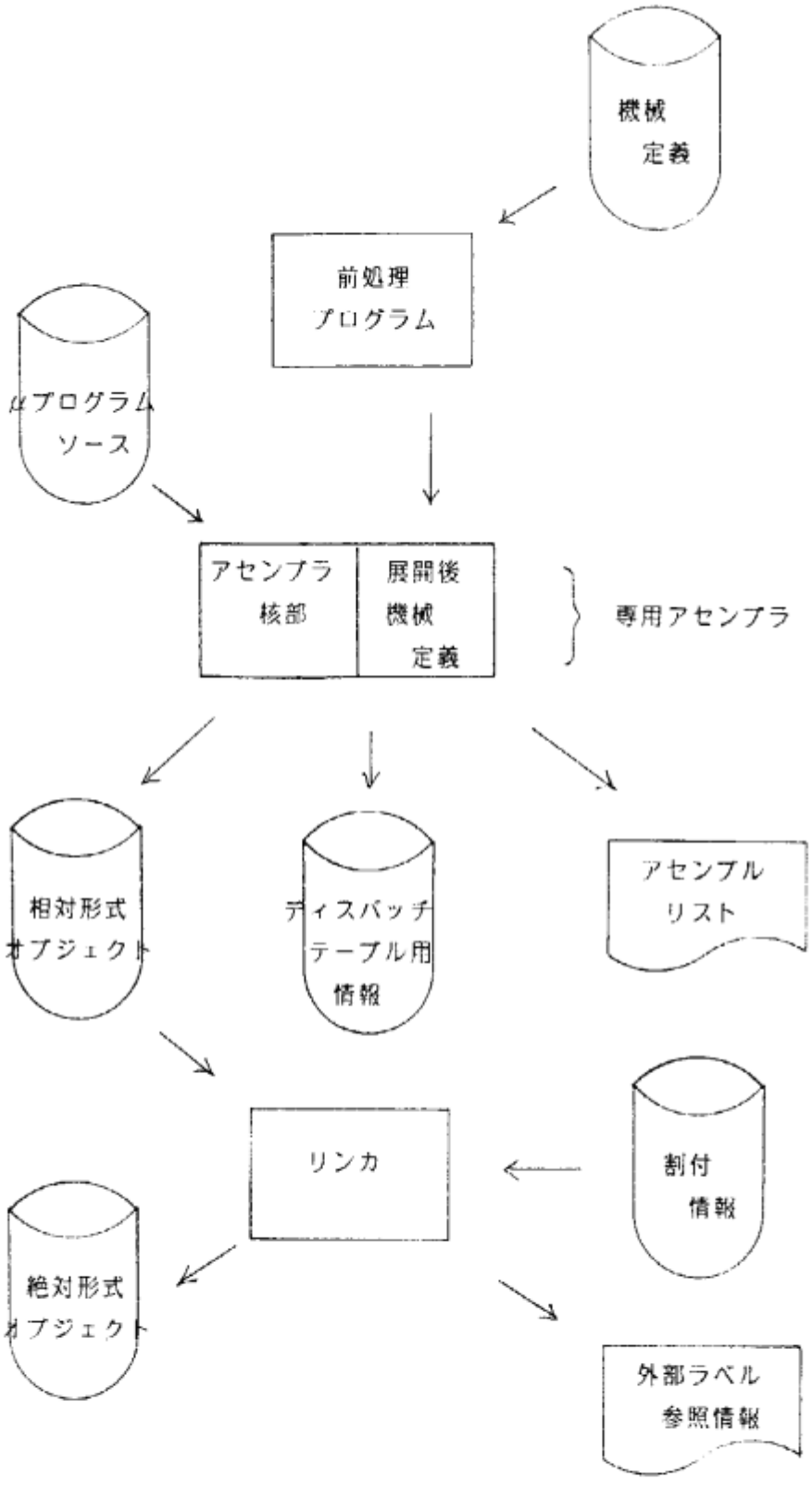

図２．１　汎用型アセンブラの構成

2.3 ターゲットのソース型式

2.3.1 ソース・ステートメント

汎用型アセンブラのターゲットとなるアセンブリ言語のソース・プログラムは，全て次のような型式である。すなわち

```
add  r1, r2.
```

のように，ソースの１ステートメントはピリオドで終了する。アセンブラの入力ルーチンがコメントとして読みとばす文字列（% から行末まで，及び /* から */ まで）はソース・テキスト上の覚え書きとして使用して良いが，リスト上には表示されない。

ソース・ステートメントはオブジェクトを生成するものと，アセンブラに指示を与える等の処理をするためのオブジェクトを生成しないものに二分できる。前者をオブジェクト命令，後者を非オブジェクト命令と呼ぶ。非オブジェクト命令はオブジェクト命令と区別するために：：で始める。例えば割付けアドレスを指定するアセンブラ命令は

```
::  org  100.
```

のように記述する。

2.3.2 ラベル

オブジェクト命令にはラベルを付けることができる。ラベルはステートメントの先頭に書き，コロンで区切る。例えばラベルlab1を持つ命令は

```
lab1 : add r0,r1.
```

のように記述する。ラベルの標準的な名前は英字で始まる英数字とアンダーラインの列である。

ラベルには上記のような標準のラベルと，２個の標準ラベルの間だけにスコープが限定されるローカル・ラベルがある。ローカル・ラベルはラベル名の前にＳを付ける。ラベル名としては数字も使用して良い。

```
          :
lab1 : add r0,r1.
       goto $1.
          :
$1 :   sub r2,r3.
          :
lab2 : compare r4,r2.
       branch__if__equal $1.
          :
$1 :   add r2,r3 .
          :
```

このような場合 goto $1の飛び先はlab1とlab2の間にある$1で，branch__if__equal の

飛び先はlab2の後ろの$1となる。ローカル・ラベルはこのように狭い範囲にラベルのスコープを限定するために用いる。

2.3.3 外部ラベルの参照

別々にアセンブルしたモジュールをリンカで結合する場合，外部モジュールのラベルを参照する必要が生じる場合がある。これをアセンブラが認識するために次のような宣言が必要である。

:: import lab_1 , lab_2 , …, lab_n .
:: export lab_1 , lab_2 , …, lab_n .

importは外部ラベルを参照する宣言(external reference)，exportは外部から参照されるラベルの宣言(global declaration)である。ここで宣言されたラベルについて，アセンブラはリンカが扱うべき情報をオブジェクトに出力する。

2.4 アセンブラ命令

ソース・ステートメント中の非オブジェクト命令のうち，アセンブラの処理に対する指示を与えるものをアセンブラ命令と言う。アセンブラ命令にはアセンブラ本体に組込みのものと，機械定義部で定義するものとがある。アセンブラ本体に組込みのアセンブラ命令は,2.3.3に述べた import,exportの他に次のものがある。

:: a equ b . 名前のequation。
:: int equ 0 .
gr00:=int!gr01. のように定数への名前付けに用いる。
:: list on . リスティングの制御。
:: list off. ソースの最後の文の処理が終了した時点で
on の状態であればラベルの参照リストを出力する。

機械定義で定義するアセンブラ命令は，オブジェクトの操作を行わないように作成しなければならない。

2.5 機械定義

機械依存部分の定義は前処理プログラムの入力となる。これを処理してアセンブラの本体とのインタ・フェースを合わせたものが前処理プログラムから出力される。この出力とアセンブラの本体をまとめてコンパイルすることによって専用アセンブラが得られる。

2.5.1 非オブジェクト操作の定義

オブジェクトを生成しない命令をアセンブラの制御や付加情報の処理に使用することができる。また，注釈のみのステップもこれに属する。

非オブジェクト操作はソース・プログラム上では：：で始まるステップに記述されるものである。定義部では：：の後ろに記述されるパターンに対する処理を記述する。例えばアドレス割付けを制御する org命令はソースでは

```
:: org 100.
```

のように書き，定義部では

```
(org _a) :- change_address( _a).
```

のように書く。

ソース上の org 100はアセンブラの核部で処理され，change_address(100)として実行される。

2.5.2 付加情報のファイルへの出力

オブジェクトとリスト以外の情報を別のファイルに出力するためには，定義の中でファイルを宣言し，専用の手続きを用いてこのファイルへ出力を行なう。アセンブラの核部は，ここで宣言されたファイル名と外部のファイル実体に付ける名前との対応をアセンブル時にユ・ザに問合せ，実際の出力はここで対応付けされたファイル実体に対して行なう。宣言は次のように行なう。

```
report_file file名1 , …, file名n .
```

情報の出力は処理の一部として次の手続き呼出しを行なうことによって行われる。

```
report(File,Info)
```

report_fileで宣言したファイル名を第１引数，出力したい情報を第２引数で指定する。例えばファイルf1に対して a,b,cという情報を出力するには，

```
report(f1,(a,b,c))
```

とする。これによってファイルf1に

```
a,b,c .
```

という出力が得られる。数字またはアトム以外のものを出力するためには上のように情報Infoを括弧でくくること。

2.5.3 オブジェクト生成に関する定義

ハードウェアに依存したフィールドの位置やニモニック，パリティ，個々の操作に対応するオブジェクトのパターンをここに記述する。複数のμ操作の間に相互を束縛するような条件がある場合や，フィールドの構成が変化する場合の処理もここに記述する。フィールド間やステップ間にわたる情報の伝達は後述の大域変数を用いて記述する。

2.5.3.1 フィールドの記述

オブジェクトのフィールド構成をここで記述する。オブジェクトのビットはleast significant bitであるビット0から始まり，フィールドとして定義された最大ビット位置までである。値を設定したい領域は全てフィールドとして定義する必要がある。一つのμ命令の中で同時に用いられない（排他的な）フィールドは，ビット位置が重複していても良い。フィールドの定義は次のように行なう。

field(フィールド名，開始ビット位置，終了ビット位置，デフォルト値)．

または

field(フィールド名，開始ビット位置，終了ビット位置，X) :-
X の値を計算するための処理，エラー・チェックの処理等．

X には定数を書いておき，その値が有効である条件を：－の右辺に書いても良い。

処理系の都合でフィールドの幅が17ビットを越えると処理が正しく行なえなくなる。従って幅の広いフィールドは２つ以上に分割する等の処理が必要である。

2.5.3.2 各フィールドの操作の記述

個々の操作に対して，オブジェクトのどのフィールドにどんな値を設定するかを記述する。個々の定義はソース・プログラムのパターンに対する処理を記述したものとして解釈実行される。従ってサブルーチンを作成したり，ソース・プログラムをもとに演算を行なってその結果をオブジェクトに出力することも可能である。ソースのパターンに対する処理は

ソース・パターン　=>　処理．

のように記述する。一方サブルーチンは

ヘッド　:－　ボディ．

のように記述する。オブジェクトに対する値の設定は，

フィールド名　<－　値

のようにフィールドに値をセットするという記述をボディの処理の一部に書く。

2.5.3.3　フィールド間の影響関係

あるフィールドに特定のニモニックが指定されることによって，オブジェクトのフィールド構成が影響を受けたり，他のフィールドの記述の意味が変わったり，オブジェクトに出すべき値が変わったりする場合がある。このような処理を定義部で記述するには，複数の操作の間で大域的に設定・参照され，かつバックトラック時には未定義に戻されるような変数を用いる。ここで言う変数とはアセンブラを記述するプログラミング言語の意味での変数ではなく，アセンブラが認識する名前としての変数である。この変数の宣言は次のように行なう。

declare　変数名$_1$，…，変数名$_n$．

ここで宣言された変数名を，操作の処理を記述する中に

var_i$ ＝値　あるいは　値＝ $ var$_i$

という形で使用できる。これを書くことによって，宣言した変数に値をユニファイして使用できる。使える演算子は＝のみである。

例えばａというニモニックは１と２の２つの型のオブジェクト形式で使用できるが，ｂは２という型のみであるとすれば，

```
declare type.
a => ($type=1; $type=2), field<-3 .
b => $type=2, field<-4.
```

のように定義する。

変数の宣言は定義ファイルの先頭のオペレータ宣言の次に書かねばならない。また宣言は１回しか行なってはならない。複数の変数を宣言するには，

declare　v_1 ,v_2 , …,v_n .

のようにコンマで区切って書く。１個の宣言が複数の行に渡るのはかまわない。この制限に反した場合，前処理プログラムの処理が正しく行なえないため，期待したようにアセンブラが動かないことがある。

2.5.3.4 ソース・ステップ間にまたがる情報の伝達

1ステップ中のフィールドにまたがる情報の伝達は前述の変数でできる。しかし複数のソース・ステップ間にまたがる情報の伝達のためには，1ステップの処理が完了した後も保存されるようなデータが必要である。このために次の3つの手続きを用意してある。

clear_ inter_step_var(変数名)　　削除
set_ inter_step_var(変数名，値)　　設定
inter_step_var(変数名，値)　　参照

ここで用いる変数名は宣言しなくて良い。この変数を用いる場合，一度設定された値は処理が途中で失敗しても設定されたまま残る。従って値の再設定や回復は全て定義の中に記述されている必要がある。

前節及び本節に述べた情報の伝達は全て定義の中に記述を行なうものである。ソース・プログラムでこれらの機能を用いたい場合には，ここで述べた機能を用いるようなニモニックを定義することが必要である。

2.5.3.5 パリティの記述

パリティは次のように指定する。複数個のパリティがあっても良い。これで指定されたパリティ・ビットは独立した1つのフィールドであり，値はリンカが設定する。従ってパリティ・ビットとビット位置が重複するようなフィールドを定義した場合のオブジェクトの内容は保証されない。

parity(S,E,P,M) ,

ここで　S …パリティを計算する開始ビット位置
　　E …パリティを計算する終了ビット位置
　　P …パリティ・ビットの位置
　　M …パリティのモード(zero,one,odd,even)

である。　S,E,Pは全て整数値であること。（16進表現等は使用不可）

2.5.3.6 分岐の記述

操作のうち分岐に関する命令は，ラベルの参照を解決するための特別の処理が必要となる。そのため，分岐命令に記述されたラベルからアドレスを求め，特殊な処理を行なうためのルーチンが用意してある。この処理は次のように書く。

branch_process(Mode,Field,Label)

第1引数は相対分岐の場合relative，絶対分岐の場合absolute
第2引数はアドレス・データを設定するフィールド名
第3引数は分岐命令に現われるラベル

現在のところ，相対分岐は処理中の命令のアドレスを基準にして，正方向も逆方向も可であるものをサポートしている。

2.5.4 注釈

注釈は：：の後ろに，オブジェクトを生成せず，かつアセンブラへの指示も行なわない命令を書けば良い。例えば次の定義

```
/ _x .
```

は /の後ろが何であっても良いという定義である。従ってオブジェクト生成命令の中に

```
r2:=r1+r2 , /'add r1 to r2' ,
```

のように注釈として使用できる。注釈のみから成るステップは全フィールドがデフォルトであるような命令をオブジェクトとして生成する。オブジェクトを生成しないためにはステップの頭に：：を付けることが必要である。

その他に処理系が注釈として受付ける /* から */ までや，% から行末までをコメントに用いても良いが，これはリストには表示できない。

2.6 処理系で用意しているサブルーチン

機械定義を作成するために処理系で用意している組込みサブルーチンの一覧を以下に示す。引数についている＋－？はそれぞれ次のような引数のモードを示す。

＋　値が入って呼ばれる。呼出し側は値の入っていない変数であってはならない。

－　サブルーチンが値を返す。呼出し側の引数は値が未設定の変数に限る。

？　どちらの場合もある。

```
append(+List1,+List2,-List3)
```

List1とlist2 を結合した結果を List3に返す。（引数は全てリスト）

```
branch_process(+Mode,+Field,+Label)
```

分岐処理を行なう。(2.5.3.6参照)

```
change_address(+Address)
```

オブジェクトの割付けアドレスを Addressにする。

Address として次のものが許される。

整数	その値を番地とする。
* 整数	語境界を整数に合わせる。
Label ±整数	Label±整数番地とする。 ただし Labelはこれ以前に定義されていること。

```
conv_to_int(+N,-Int)
```

N を整数に変換して Intに返す。16進→10進,8進→10進等の変換に用いる。

入力の形式は n#strの形で基数nの表現の文字列 strである。

nを省略すると16とみなす。

```
current_address(-Address)
```

現在処理中のステップのアドレスを返す。

error(+Message1,+Message2)

メッセージをリストと端末に出力する。

get_label_address(+Label,-Address)

ラベルが定義されているアドレスを返す。

外部ラベルの場合は-1，未定義の場合はエラー・メッセージを出して0を返す。

inter_step_var(+Variable,-Value)

ステップ間に渡される情報の参照。

clear_inter_step_var(+Variable)

ステップ間に渡される変数名の削除。

max(+N1,-N2,-N)

N1とN2の大きい方の値をN に返す。 N1,N2は整数であること。

member(+Elem,+List)

要素がリストの中に含まれれば成功，含まれなければ失敗。

out_message(+Message)

エラーメッセージのリストを出力する。

Message は最後が変数であるようなリストである。

report(+File,+Info)

ファイルに付加情報を出力する。

reverse(+List1,-List)

リストを反転したものを返す。

set_inter_step_var(+Variable,+Value)

ステップ間に渡される情報を設定する。

+Field <- +Value

フィールドに値を設定する。

2.7 制限事項

処理系の都合により以下の制限がある。

・ファイルの最後の行も復帰改行で終了しなければならない。

・機械定義の右辺には=>の左辺のニモニックをサブルーチンとしては使用できない。

・処理系で用意しているサブルーチン及び

pt_,asm_,sub_,portray,asm,def,equ,go,word_lengthで始まる名称，

さらに処理系が提供している predicateと同じ名称は

ニモニックやサブルーチンとして使用してはならない。

・フィールドの幅は17ビット以下。

・ニモニックの定義に付随してオペレータ宣言が必要。

(if tag(cdr) then goto lab. 等と書くには，

ifとgotoをprefix，thenを infixでかつ適当な優先順位を持つオペレータに

宣言する。)

・：，：：，Ｓ，※，＝，＜－，＝＝＞はアセンブラ本体に組込みなので，

ユーザが機械定義に使用してはならない。

2.8 リンカ

リンカは複数のオブジェクト・ファイルをまとめて１個の絶対型式ファイルにすることができる。アセンブラの出力は相対型式であるため，外部ラベル参照の解決を行って分岐アドレス・フィールドを設定する必要がある。そのためには各々のオブジェクトの絶対アドレスを決定しなければならない。リンカはこれを行なうための基準点となる各オブジェクト・ファイルの先頭のオブジェクトの割付けアドレスを制御ファイルから読込む。

ユーザは割付け情報を持つファイルのファイル名を端末から与える。このファイルは各行が

アドレス，ファイル名．

の型式である。アドレスは10進表現，16進表現のどちらでも良い。ファイル名はアセンブラの場合と同様，エクステンションがある場合にはクォートでくくる。外部参照に関する割付け情報はlinker.outという名前のファイルに出力される。

3. 実現方式

従来のアセンブラでほとんど使用されていなかった機能として，引数のパターン照合，ユニフィケーション，バックトラックの３機能がある。

3.1 パターン照合

機械定義に記述されているニモニックの定義は，入力されるソース・プログラムのコンマで区切られた個々の項に対する処理を記述するものである。例えば

```
f(a),g(b) .
```

という入力に対しては，f(a)に対する処理と，g(b)に対する処理が定義されている必要がある。この定義は

```
f(a)==>f<-0.      field(f,0,3,0).
g(b)==>g<-1.      field(g,4,4,0).
```

のように記述される。この例では引数が定数の場合であったが，

```
f(g(_x))==>_y  is _x+4, f<-_y.
f(_x)==>f<-_x.
```

のように，引数のパターンが異なる場合や，引数に応じた演算を行なう場合を定義してもかまわない。もちろん引数が０個でも，２個以上でも良いし，個々の引数がもっと複雑な形でもかまわない。

```
(_d:= _s1//_s2)==> dest(_d),s1(_s1), s2(_s2),operation(('//')) .
```

のように，一般の演算式風の形のものも定義することができる。この場合:=や//は機械定義の中でオペレータとして宣言する。また，dest,s1,s2,operation等はそれぞれサブルーチンとして定義する。

ニモニックに対しf(g(_x)) とf(_x)の２つを定義するような場合，複数の選択枝を定義すると言う。定義された選択枝と一致しない場合はその項の処理は失敗し，エラー・メッセージが出力される。例えばf(g(_x)) とf(h(_x)) に関する定義のみがある場合の入力として f(a(0))が与えられると，これは定義のどちらとも異なるパターンであるから照合が失敗する。またf(_x)に対する定義は引数１個のｆの呼出しには全て照合が成功するが，f(0,1)といった２個以上の引数に対しては失敗する。同様にf(_x+_y)と言う定義はf(3+5)に対しては照合が成功するが，f(0)に対しては照合は失敗する。

3.2 ユニフィケーションとバックトラック

ソース・プログラムに記述された複数の項のそれぞれは，生成されるオブジェクトの一部に対して値を設定する。全ての項の処理が終了後に暗黙に設定されるフィールドの処理を行ない，それでも未定義のフィールドはデフォルト値を設定する。

これらの値の設定に矛盾が生じる場合その処理は誤りであり，失敗する。例えば同一フィールドに異なる値を設定しようとしたり，フィールドに重複が生じる場合は誤りである。本アセンブラはこの種の誤りは自動的に検出する。それは値の設定を代入でなくユニフィケーションによって行なっているためである。処理の開始時にはオブジェクトの全ビットは未定義となり，どんな値でもユニファイできる。一度ユニフィケーションによって値が確定した場合は，同一ビットに対するユニフィケーションは代入ではなく比較として働き，全く同一の値以外をユニファイしようとすると失敗する。

ユニフィケーションが失敗すると項の処理はバックトラックし，別の選択枝でパターンの照合を行ない，成功したものを実行しようとする。その実行が失敗するとさらに別の選択枝をさがす。

成功する選択枝がなくなるとそのパターンに合致する処理に失敗したことになり，さらにその前の項へバックトラックする。前の項に選択枝がある場合にはそのパターン照合を行ない，成功した場合にはそれを実行する。そしてあらためて次の項の処理に移る。

このように次々にバックトラックを行ない，最終的に全部の項の処理に成功した時にそのステップの処理は成功し，オブジェクトを出力する。

例えば定義が次のような場合，

```
t==>f(0),g(0).
f(0):-f<-0.        f(0):-f<-1.
g(0):-f<-1.        g(0):-f<-2.
```

tという項はf(0)とg(0)の呼出しとして処理される。この時まずf(0)が実行され，フィールドfに0がユニファイされる。次にg(0)を処理しようとすると，フィールドfには既に値0がユニファイされているので値1とのユニファイも値2とのユニファイも失敗する。そこでf(0)の次の選択枝が選ばれ，フィールドfは1にユニファイされる。次にあらためてg(0)が実行され，fを1にユニファイして実行が終了する。

この例は同一フィールドに異なる値を入れるだけの極端なものであるが，フィールドへの値の設定だけでなく，項の間に伝達できる変数(2.5.3.3で述べたステップ内の大域変数)に対する値のユニフィケーションを条件チェックに用いることができる。

3.3 バックトラックによる条件チェック

3.2で述べたユニフィケーションとバックトラックの機能を用いると，オブジェクトの生成条件をチェックすることが可能となる。これにはステップ内の大域変数を用いる。例えば

```
:-op(300,fx,(add)).
declare cond.
add==> $cond=1,f<-0 .
add==> $cond=2,f<-1 .
reg==> $cond=1,g<-7 .
reg==> $cond=2,g<-10.
```

という定義をし，呼出しとしては

```
…, add reg , ….
```

を行なうと，他の項の条件によって大域変数condの値が１ならばフィールドｆは0に，ｇは７になる。condの値が２ならばｆは１にｇは10になる。他の項がcondの値についての条件を持たないならば，最も単純な解として先頭に定義したcond=1の場合が選ばれる。

１個以上の条件を同時に指定するには，それらのチェックを並べてコンマで区切れば良い。 andの条件ではなくorの条件の場合にはセミコロンで区切る。例えば

ａかつｂ	→	ａ，ｂ
ａまたはｂ	→	ａ；ｂ
ａかつ（ｂまたはｃ）	→	ａ，（ｂ；ｃ）

従って定義には

```
add==>(a;b),f<-0.
```

のように書く。 and条件の方がor条件より強いので「ａ，ｂ；ｃ」は「（ａかつｂ）またはｃ」と解釈される。「ａかつ（ｂまたはｃ）」とするには例のように「ａ，（ｂ；ｃ）」とする必要がある。

これを用いると，フィールドの値を決めるための条件をいろいろ組合せることができる。また，ある条件下でのみエラー・メッセージを出すことも可能となる。

3.4 機械定義の展開

機械定義ファイルは前処理プログラムで処理され，アセンブラ核部とのインタフェースが合わされる。ここでの前処理は次のものがある。

1) オペレータ宣言はそのまま実行し，出力にもコピーする。
2) 大域変数の宣言は変数の個数だけの引数を持つような構造体を出力する。
3) 付加情報ファイルの宣言は出力にコピーする。
4) パブリック宣言は引数の数を２個増やす。
5) モード宣言は引数に +,+を追加する。
6) ==> で定義されるニモニックは，
   ヘッド部を Hとすると def(H,OW,G)の形のヘッドに置き換える。
   ボディ部は項毎の処理をする。
7) その他の定義はヘッド部に２個の引数(OW,G)を増やし，
   ボディ部は各項毎に処理する。
8) 項毎の処理はヘッドと同様に引数を追加する。
9) 言語組込の手続呼出しはそのままとする。
10) $var=Valueの形のものは，declare 文で宣言された変数名の順番を番号ｎとして，
    ヘッドに追加した引数 Gに対するarg(n,G,Value)と言う呼出しに変換する。
11) Field<-Valueの形のものは，ヘッドに追加したOW,Gに対して，
    asm_ set_object(OW,Field,Value,G)の形とする。

こうして前処理したプログラムは全て解釈実行されるプログラムとなっている。アセンブラの本体はソース・プログラムの各ステップに対して6)と9)に従った変換を行ない，変換結果を呼び出せば良い。

4. 適用例：ψ用マイクロプログラム・アセンブラ

4.1 目的

ψ用マイクロプログラム・アセンブラは，逐次型推論マシンψのマイクロプログラムを処理するためのものである。ψのマイクロプログラムは1ワード64ビットから成り，タイプ1～3と呼ばれる3種のフィールド形式が存在する。また，使用するレジスタの特性に依存する制限がある。形式のちがいと制限事項のうち，動的実行パスに依存しないもののかなりの部分をアセンブラのチェックで吸収することが必要となる。また，フィールド形式によってデフォルト値を変更することも要求される。これらに対し充分応じられるようなアセンブラが必要となった。なお，ここで述べるハードウェアに関する詳細は逐次型推論マシンψ開発の関連文書を参照。

4.2 ソース・プログラム

ソース・プログラムはオブジェクトを生成しないアセンブラ命令と，オブジェクトを生成する命令とに分かれる。アセンブラ命令は：：で始まる。：：のない命令はオブジェクトを生成する命令である。それぞれの命令は全てピリオドで終わる。入力ルーチンが読みとばす%から行末までと /* から */ まではコメントに使用しても良いが，リストには出力されない。

アセンブラ命令の形式は

```
:: / 'comment string'.
:: import lab1,lab2.
:: org *8.
```

のように：：で始まり，ピリオドで終わる。一方オブジェクトを生成する命令は

```
lab : wf00:=wf01+wf02, if tag(wf01)-0 goto lab2 .
```

のように：：以外で始まり，ピリオドで終る。

オブジェクト生成命令の頭にはラベルを付けることができる。ラベルと処理本体の間は：で区切る。ラベルが複数個ある場合にはラベル前に：を付ける。例えば

```
lab1: lab2: gr00:=wfar.
```

のように書く。アセンブラ命令にはラベルは付かないので注意のこと。

4.3 アセンブラ命令

μ用アセンブラには，アセンブラ本体で用意してあるアセンブラ命令の他に機械定義で作成した命令がある。

本体で用意されているアセンブラ命令は次のものである。

:: A equ B.

A をB で置き換える。定数に名前を付けたりするのに用いる。

:: export lab_1 , lab_2 , ….

lab_1 , lab_2 , …を別にアセンブルされたファイルから参照を許す。

:: import lab_1 , lab_2 , ….

lab_1 , lab_2 , …は別にアセンブルされたファイルにある外部ラベルである。

:: list on.　:: list off .

リスティングを「出す／出さない」の切り換え。

機械定義で作成したアセンブラ命令は次のものである。

:: org Address.

次の命令からの割付け番地を Addressに変更する。

Addressが定数の時はその値の番地を次の割付け番地にする。

label±n の形の時は labelの番地±n 番地を次の割付け番地にする。

*n の時は割付け番地をn語境界にする。

:: begin_case Offset step Step .

caseブロックの始まりを示す。

ブロックの先頭に":: org Offset."があったと同様に番地を合わせる。

ブロックの各エントリはStep語ずつあると見なして，

次の:: -->命令によって番地の設定を行なう。

:: --> Info .

caseブロックの各エントリを示す。

begin _case文によって与えられたStepによって番地を合わせるとともに，

ここで合わせた番地の値とInfoを付加情報ファイルdispatchへ出力する。

:: end_case.

caseブロックの終わりを示す。

4.4 オブジェクト命令のニモニック

オブジェクト命令のソース・プログラムには，ハードウェアのフィールド構成に対応して複数のフィールドが記述できる。複数のフィールドは一般にはコマンドで区切って記述するが，演算系と条件分岐系の部分は高級言語風の記述となっている。以下それぞれについて述べる。

4.4.1 data命令

data命令は64ビットの定数データを wcs内に置くために用いる。64ビットのフィールドは16ビット毎の４フィールドに分割して記述する。 LSB側のフィールドを省略すると０を設定する。 HSB側や途中のフィールドを省略することはできないので０と書くこと。

```
data  #'AAAA', #'BBB', #'CC', #'D'.      →  #'AAAA0BBB00CC000D'
data  #'0123', #'4567', #'89AB' .        →  #'0123456789AB0000'
data  1, 2, 3 .                          →  #'0001000200030000'
data  0 .                                →  #'0000000000000000'
```

4.4.2 Debug フィールド

gevc1++	カウンタ gevc1のインクリメント
gevc2++	カウンタ gevc2のインクリメント
break_point	マイクロプログラムのブレークポイント（実行がここで中断される）

4.4.3 ALU オペレーション

ALU オペレーションでは，destination(DSTF),source1(SC1F),source2(SC2F), multi_destination(MDF),shift_mask(SMF),byte_rotate(BYRF),bit_rotate(BIRF),operation(ALF,EALF),tag_immediate(IEI,IVI),source2_immediate(SI1F,SI2F)等のフィールドを指定する。基本形式は

```
destination := source1 operation source2
```

である。これらの一部は省略できる。記述できる組合せは

```
d:=s1 op s2     d:= & s2    d:=s1
 :=s1 op s2      := & s2     :=s1   (tagのみは不可)
```

の６通りである。 source2の省略と区別するため,source1を省略する場合にはオペレーションには＆を書く（実行されるオペレーションは source2 throughである）。

4.4.3.1 destination

destination は，destination registerのみを記述する場合,multi_destination を合せて指定する場合，省略する場合の３通りある。レジスタのみの場合はそのレジスタ名を書き,multi_destination も指定するためには，

(reg,multi_dest)

のように括弧でくくる。省略した場合はzeroが仮定される。 regに指定できるレジスタは表 4.1参照。 multi_destには plar,clar,pdr,cdrの４個のみが書ける。

4.4.3.2 source1

source1はレジスタを指定する他にタグの即値を書くことができる。レジスタを省略してタグ即値のみを書くこともできる。この場合は source2は省略してはならず，かつオペレーションは&でなければならない。記述形式は次の通り。

```
tag_immediate ! register
tag_immediate !
register
```

source1を省略した場合は c00を仮定する。 source1のレジスタは表 4.2参照。

4.4.3.3 source2

source2 は，レジスタの指定，シフトの指定，シフト・マスクの指定，即値の指定ができる。形式は次のようになる。

```
s2_reg>>shift_no/mask      または    s2_reg<<shift_no/mask
```

マスクを省略した場合は

```
s2_reg>>shift_no           または    s2_reg<<shift_no
```

シフトしない場合は

```
s2_reg/mask
```

両方とも省略する場合は

```
s2_reg
```

となる。

シフトは<<が左へ，>>が右へのローテート・シフトである。シフト量はビット数で与える。なおハードウェアが右シフトしかサポートしていないため，<<は右シフトで同じ結果が出るように変換される。

シフト・マスクは16ビット,8ビット,5ビットのみが許される。

レジスタが32ビット未満の場合の上位ビットの値は不定である。従ってシフトとマスクによって不定値を除くようにプログラムを書く必要がある。

s2_reg の部分は即値を書くことができる。即値は 0～63の値であり，上位に０がパディングされて32ビット幅の値として用いられる。

source2のレジスタは表 4.3参照。

source2を省略した場合，タイプ１と３では source2 immediateの０を仮定し，オペレーションは32ビット加算とする。タイプ２では wfar1>>10/5を仮定し，オペレーションはorとする。いずれの場合も source2の値は０となり,destinationへは source1のデータが送出される。 tagやflagについては他に指定された項による。

4.4.3.4 オペレーション

オペレーションには算術演算と論理演算がある。算術演算はタイプ１と３，論理演算はタイプ２と３で使用できる。32ビット幅の加減算にはキャリー付きのものとなしのものがあり，その他に主にアドレス計算に用いられる24ビット幅のキャリーなしの加減算がある。source2 throughの命令は論理演算である。詳細は表 4.4参照。

4.4.3.5 キャッシュ・コントロール

キャッシュ・メモリのアクセス内容の他に，どのアドレス・レジスタとどのデータ・レジスタを用いるかの指定が必要となるものがある。readと writeには両レジスタが必要であり,clear等にはアドレス・レジスタが必要である。

キャッシュ・コントロールに用いられるアドレス・レジスタはplarまたはclarであり，データ・レジスタは pdrまたは cdrである。

キャッシュ・コントロールの命令は表 4.5参照。

4.4.3.6 アドレス・レジスタ

アドレス・レジスタとしてplarまたはclarが用いられる。これらのレジスタはキャッシュ・コントロールの命令で指定される他に独立にインクリメントさせることができる。これは

```
plar ++   または   clar ++
```

と書かれる。

4.4.3.7 フリップ・フロップ

フラグ・ビットのon/offとロードの機能がある。

これに用いることのできるフラグについては表 4.6参照。これらのうち一部はタイプ３でのみ用いることができる。

4.4.3.8 マルチバス・コントロール

マルチバスのコントロールには，バイト単位のread,writeと16ビットワード単位のread, write，及びwait_ioの計５種がある。表 4.7参照。

4.4.3.9 ページ・マップのアクセス

ページ・マップは pmm,pmhm,pmsmを用いてアクセスされる。このためにはまず３種のうちのアクセスしたいレジスタを destinationに指定して値をセットし，次に実行されるステップで write_pmm 等の命令を出す。例えば

```
write_pmbm(plar)
```

となる。この命令の引数はplarまたはclarである。

4.4.3.10　JRデクリメント

JRのデクリメントを特に指定することができる。これはJR--と書く。この命令は multi_destination のフィールドを用いているので multi_destination とは同時には指定できない。

4.3.3.11　サブルーチン呼出し

マイクロプログラム・サブルーチンの呼出しは分岐命令の一種である。これはタイプ１でのみ使用できる。従って相対分岐であり，現在の番地から-256～+255番地の範囲しか呼び出せない。

呼出し　　gosub label

復帰　　return

4.3.3.12　無条件分岐

無条件分岐には次の３種がある。

goto @jr　　または　　goto label　　または　　load_jr

goto @jrはタイプ１または３で使用でき，jrの間接分岐となる。goto Labelはラベルに対しての分岐であり，タイプ１では現在の番地から-256～+255番地の自己相対アドレスによる分岐，タイプ２では絶対アドレスによる分岐となる。load_jrはアドレスをjrにロードするものでタイプ１でのみ使用できる。

4.4.3.13　２方向分岐

２方向分岐はタイプ１でのみ使用でき，tagまたはflagによって分岐するものである。形式は

if Cond goto label

であり，現在の番地から-256～+255の範囲で分岐する。　Condは，

tag(S2_reg)=Value　　または　　not Flag　　または　　Flag

の３形式が書ける。第１の形式のS2_reg は ALUオペレーションの source2フィールドと統一がとれていなければならない。また使用できるレジスタはgr00～gr0fと pdr,cdrの18個のみである。Flagとして使用できるものは表 4.8参照。

4.4.3.14　case分岐

case分岐はタイプ１で使用できる。形式は

case ir_opcode　と　case Cond base label

の２種ある。第１のものはインストラクション・レジスタのオペレーション・コード部をもとに分岐するものであり，第２のものは Labelで指定されるcaseブロックに対してディスパッチ・メモリを用いた分岐を行なう。Condとしては

tag(Bank,DR)　と　ir0　または　ir1　または　ir2　または　ir3

の２形式ある。 ir0～ir3 はインストラクション・レジスタのオペランド部による分岐を指定するものである。tag(Bank,DR)の形式はDRで与えられる pdrまたは cdrのタグ値とディスパッチ・メモリのBankから分岐先を求めるものである。

4.4.3.15　アセンブラが自動設定するフィールド

前節までに述べた各々の処理はμプログラムのソース上に記述されるものである。プログラムで指定されたこれらの情報の他にアセンブラは次のものを自動設定する。

1) オブジェクトのタイプを指定するフィールド。
2) キャッシュ・メモリのデータ・レジスタ(pdr, cdr)に対して必要なwait操作。
3) タイプ１または２で分岐が指定されていない場合に次の番地へ進む分岐命令。

4.4.3.16　条件チェック

アセンブラでは１ステップ内の制限のチェックを行なう。これらの条件はハードウェアに依存するものであり，その詳細はμ関連のハードウェア・ファームウェアの仕様を参照されたい。条件チェックによるメッセージの一覧を表 4.9に示す。

4.5 Dispatchテーブル用情報

case分岐用のディスパッチ・テーブルの生成にはcase命令が発行されるステップとcaseブロックの各エントリに関する情報が必要である。この情報は内部名dispatchのファイルに出力される。アセンブラはこのファイルは実体をどのファイルにするかをアセンブル開始時に端末に問合せる。このファイルに出力する情報は次のものである。

```
case tag(Bank,DR) base Label
    →  現在の番地,case,Bank,Labelの番地.
case ir0 base Label      (ir1～ir3 も同様)
    →  現在の番地,case,ir,Labelの番地.
:: begin_case Offset step Step .
    →  現在の番地,begin_case,Offset,Step.
::  --> 情報.      →  エントリの番地, 情報.
:: end_case.       →  現在の番地,end_case.
```

この情報をもとにテーブル生成プログラムがテーブルにロードするべきデータのファイルを生成する。

4.6 使用法

アセンブラは 7月19日現在、US1:<SIM-H>ASSY.EXEにある。リンカは同じディレクトリのLINKER.EXEである。これらを呼出すことによってアセンブルとリンクがなされる。アセンブラとリンカの問合せに対してはファイル名を答える。書き方は

'ass.src'.　あるいは　source.

等となる。ファイル名がエクステンション部（上例ではsrc)を持つ場合はクォートでくくる。終わりはピリオドである。ピリオドを打ち忘れてreturnキーを打った場合はそのままピリオドを打ってからreturnキーを打てば良い。

4.7 実行速度

現在の処理速度はアセンブルが150ms/step，リンクが 80ms/step程度である。ソース・プログラムの書き方によって処理速度が変わる場合があるので次のようなスタイルを推奨する。

1) まずタイプに無関係のフィールドを書く。
2) 次にタイプを確定するフィールド
   （特定のタイプでのみ使用できるフィールド）を書く。
3) その次に複数のタイプで指定できるフィールドを書く。

エラーがある場合，そのステップの処理が数倍かかる場合もありうる。これはバックトラックによって極力エラー回復を試みているためである。上のようなスタイルによれば，無用なバックトラックを少なくするのに効果がある。

Table 4.1 Destination registers

a) Work File type

bit	mnemonics	remarks
0XX XXXX	wf00-wf3f	Work File 00-3F
10X XXXX	@wfcbr00-1f	WF write by WFCBR#DSTF<4:0>
110 XXXX	tag(gr/wf00-0f)	write tag part only from ALU output<7:0>
111 000?	@pdr	Work File write by WFBR#PDR<4:0>
111 001?	@cdr	Work File write by WFBR#CDR<4:0>
111 0100	@wfar1	WF write by WFAR1
111 0101	@++wfar1	increment WFAR1 & WF write by WFAR1
111 0110	@wfar2	WF write by WFAR2
111 0111	@++wfar2	increment WFAR2 & WF write by WFAR2
111 1???	(nop)	no operation

Logical wf-register mnemonics

WF regs	logical name	remarks
wf00-wf0f	gr00-gr0f	
wf10	pcbr	Parent Control Base Register
wf11	plbr	Parent Local Base Register
wf12	pgbr	Parent Global Base Register
wf13	pfbr	Parent Frame Base Register
wf14	psbr	Parent Skelton Base Register
wf15	pibr	Parent Instruction Base Register
wf16	ccbr	Current Control Base Register
wf17	clbr	Current Local Base Register
wf18	cgbr	Current Global Base Register
wf19	cfbr	Current Frame Base Register
wf1a	csbr	Current Skelton Base Register
wf1b	cibr	Current Instruction Base Register
wf1c	ctr	Control Top Register
wf1d	ltr	Local Top Register
wf1e	gtr	Global Top Register
wf1f	ttr	Trail Top Register
wf20	htr	Heap Top Register
wf21	bcbr	Backtrack Control Base Register
wf22	blbr	Backtrack Local Base Register
wf23	bgbr	Backtrack Global Base Register
wf24	rcbr	Return Control Base Register
wf25	pbr	Procedure Base Register
wf26	nibr	Next Instruction Base Register
wf27	acbr	Alternative Clause Base Register
wf28	obcbr	Old BCBR
wf29	orcbr	Old RCBR
wf2a	ottr	Old TTR
wf2b	ecr	Exception Control Register
wf2c	mcr	Mark Control Register
wf2d	scnt	Symbol Counter
wf2e		
wf2f		

b) Register type

bit	registers	remarks
000 0000	pdr	Parent Data Register
000 0001	tag(pdr)	tag part of PDR
000 0010	cdr	Current Data Register
000 0011	tag(cdr)	tag part of CDR
000 0100	wfar1	Work File Address Register 1
000 0101	wfar2	Work File Address Register 2
000 0110	wfbr	Work File Base Register
000 0111	wfcbr	Work File Constant Base Register
000 1000	ir	Instruction Register
000 1001	mstr	Micro Status Register
000 1010	mstkar	Micro Stack Address Register
000 1011	jr	Jump Register
000 1100	@ir	Dispatch Memory access by IR<9:0>
000 1101		
000 1110	@mstkar	Micro Stack(MSTK) write by MSTKAR
000 1111	@++mstkar	inc. MSTKAR & MSTK write by MSTKAR
001 0000	isetr	Interrupt Set Register
001 0001	irstr	Interrupt Reset Register
001 0010	imskr	Interrupt Mask Register
001 0011		
001 0100	bar	multiBus Address Register
001 0101	bwdr	multiBus Write Data Register
001 0110		
001 0111		
001 1000	plar	Parent Logical Address Register
001 1001	clar	Current Logical Address Register
001 1010		
001 1011		
001 1100	pmbm	Page Map Base Memory
001 1101	pmsm	Page Map Size Memory
001 1110	pmm	Page Map Memory
001 1111		
010 0000	wcsar	WCS Address Register
010 0001		
010 0010	@wcsar	WCS write by WCSAR
010 0011	@wcsar++	WCS write by WCSAR & inc. WCSAR
010 0100		
010 0101		
010 0110		
010 0111	qr	Q Register
010 1000	gevc1	General Evaluation Counter 1
010 1001	gevc2	General evaluation Counter 2
010 1010	stopr	Stop Register
010 1011		
010 11??		
011 0XXX	fwdr0-7	Floating Point interface 0-7
011 1XXX	opwdr0-7	Option interface 0-7
1?? ????		

** "?" is a "don't care bit"

Table 4.2 Source1 registers

bit	register	remarks	
00XX XXXX	wf00-3f	Work File 00-3F	*1
01XX XXXX	c00-3f	WF Constant read by 1111#XXXXXX	
100X XXXX	@wfcbr00-1f	WF read by WFCBR#XXXXX	
101? 00??	@pdr	Work File read by WFBR#PDR<4:0>	
101? 01??	@cdr	Work File read by WFBR#CDR<4:0>	
101? 100?	@wfar1	WF read by WFAR1	
101? 1010	@wfar1--	WF read by WFAR1 & dec. WFAR1	
101? 1011	@wfar1++	WF read by WFAR1 & inc. WFAR1	
101? 110?	@wfar2	WF read by WFAR2	
101? 1110	@wfar2--	WF read by WFAR2 & dec. WFAR2	
101? 1111	@wfar2++	WF read by WFAR2 & inc. WFAR2	
110? ???0	pdr	Parent Data Register	
110? ???1	cdr	Current Data Register	
1110 0000	mstkar	Micro Stack Address Register	*2
1110 0001	mar	Micro Address Register	*2
1110 001?	jr	Jump Register	*2
1110 010?	@ir	Dispatch Memory read by IR<9:0>	*2
1110 0110	@mstkar	Micro Stack(MSTK) read by MSTKAR	*2
1110 0111	@mstkar--	MSTK read by MSTKAR & dec. MSTKAR	*2
1110 1000	wcsar	WCS Address Register	*2
1110 1001	@trcar--	Tracer read by TRCAR & dec. TRCAR	*2
1110 1010	@wcsar	WCS read by WCSAR	*2
1110 1011	@wcsar++	WCS read by WCSAR & inc. WCSAR	*2
1110 1100	pmbm(LAR)	Page Map Base Memory	*2,*3
1110 1101	pmsm(LAR)	Page Map Size Memory	*2,*3
1110 1110	pmm(LAR)	Page Map Memory	*2,*3
1110 1111		(floating)	
1111 0000	gevc1	General purpose Evaluation Counter1	*2
1111 0001	gevc2	General purpose Evaluation Counter2	*2
1111 0010	csrdr	Console Read Data Register	*2
1111 0011		(floating)	
1111 0100	elar	Error Logical Address Register	*2
1111 0101	epar	Error Physical Address Register	*2
1111 011X	(csrdr)		
1111 1XXX	(csrdr)		

*1 Logical Register Mnemonics for wf00-3f are the same
as those for destination
*2 these registers are passed direct to Destination-bus
*3 " LAR " ; plar / plar++ / clar / clar++

Table 4.3 Source2 registers

bit	register	remarks
00 XXXX	gr00-0f	General Register 00-0F
01 XXXX	tag(gr00-0f)	tag part of GR 00-0F
10 0000	pdr	Parent Data Register
10 0001	tag(pdr)	tag part of PDR
10 0010	cdr	Current Data Register
10 0011	tag(cdr)	tag part of CDR
10 0100	wfar1	Work File Address Register 1
10 0101	wfar2	Work File Address Register 2
10 0110	wfbr	Work File Base Register
10 0111	wfcbr	Work File Constant Base Register
10 1000	ir	Instruction Register
10 1001	mstr	Micro Status Register
10 1010	istr	Interrupt Status Register
10 1011	brdr	multiBus Read Data Register
10 1100	plar	Parent Logical Address Register
10 1101	clar	Current Logical Address Register
10 1110	qr	Q Register for multiply/divide
10 1111		(floating)
11 0XXX	frdr0-7	Floating Point interface 0-7
11 1XXX	oprdr0-7	Option interface 0-7

Table 4.4 Operations

a) arithmetic operation [Type 1 / Type 3 & EALF=0]

bit	operation	remarks
000	s1 + s2	add 32 bits
001	s1 + s2 with_c32	add 32 bits with carry
010	s1 @+ s2	add 24 bits
011	s1 *+ s2	shift 64 bits and trough or add 32 bits by S0 for multiply
100	s1 - s2	subtract 32 bits
101	s1 - s2 with_c32	subtract 32 bits with borough
110	s1 @- s2	subtract 24 bits
111	s1 /- s2	shift 64 bits for divide

b) logical operation [Type 2 / Type 3 & EALF=1]

bit	operation	remarks
000	s1 and s2	s1 and s2
001	s1 or s2	s1 or s2
010	not(s1) and s2	(invert s1) and s2
011	not(s1) or s2	(invert s1) or s2
100	s1 xor s2	s1 exclusive_or s2
101	s1 through s2	s2 through
110	s1 and_swap s2	s1 and s2 with swap
111	s1 or_swap s2	s1 or s2 with swap

Table 4.5 Cache control

bit	operation	remarks
0000	(nop)	no operation
0001	wait_pdr	wait PDR data
0010	wait_cdr	wait CDR data
0011	wait_dr	wait PDR or CDR data
0100	read_internal(LAR , DR)	read internal registers
0101		
0110	force_directory_error(LAR)	diagnose
0111	clear(LAR)	invalidate block
1000	clear_with_save(LAR)	invalidate block with write back
1001	read(LAR , DR)	read in normal mode
1010	read_bypass(LAR , DR)	read in cache bypass mode
1011	read_physical(LAR , DR)	read with physical address
1100	write_stack(LAR , DR)	write without read in
1101	write(LAR , DR)	write in normal mode
1110	write_bypass(LAR , DR)	write in cache bypass mode
1111	write_physical(LAR , DR)	write with physical address

** " LAR " ; plar / plar++ / clar / clar++
** " DR " ; pdr / cdr

Table 4.6 Flag operations

a) normal flag operation [Type 1 / Type 2 / Type 3 & FF2F=0]

bit	flag	remarks
0000	(nop)	no operation
0001	load_alu_flags	load arithmetic flags *1
0010	load_s32	load sign32
0011	load_z32	load zero32
0100		
0101		
0110		
0111		
1000	set_sw0	set program switch 0
1001	set_sw1	set program switch 1
1010	set_sw2	set program switch 2
1011	set_sw3	set program switch 3
1100	reset_sw0	reset program switch 0
1101	reset_sw1	reset program switch 1
1110	reset_sw2	reset program switch 2
1111	reset_sw3	reset program switch 3

*1 "arithmetic flags" are as follows ;
same_tag, c10, z24, s24, c24, v32, c32, p_area

b) extended flag operation [Type 3 & FF2F=1]

bit	flag	remarks
0000		
0001	load_str	load DST<31:24> to MSTR<31:24>
0010	move_so_to_s32	SO is shift out bit of 64 bits
0011		
0100		
0101		
0111		
1000	set_sw4	set program switch 4
1001	set_sw5	set program switch 5
1010	set_sw6	set program switch 6
1011	set_sw7	set program switch 7
1100	reset_sw4	reset program switch 4
1101	reset_sw5	reset program switch 5
1110	reset_sw6	reset program switch 6
1111	reset_sw7	reset program switch 7

Table 4.7 Multibus control

bit	mode	remarks
00?	(nop)	no operation
010	put_word	write 2 bytes
011	put_byte	write 1 byte
100	get_word	read 2 bytes
101	get_byte	read 1 byte
11?	wait_io	synchronize CPU access to multibus

Table 4.8 Condition flags

bit	condition	remarks	
0XXXXX	mstr0-31	test MSTR bit(details follow)	
000000	p_area	privileged area	(MSTR<0>)
000001	same_tag	tag of SRC1 = tag of SRC2	(MSTR<1>)
000010	c10	carry from bit<9>	(MSTR<2>)
000011	c24	carry from bit<23>	(MSTR<3>)
000100	z24	ALU output<23:0> is zero	(MSTR<4>)
000101	s24	ALU output<23>	(MSTR<5>)
000110	v32	overflow in 32 bit operation	(MSTR<6>)
000111	c32	carry from bit<31>	(MSTR<7>)
001000	z32	ALU output<31:0> is zero	(MSTR<8>)
001001	s32	ALU output<31>	(MSTR<9>)
001010	mstr10		(MSTR<10>)
001011	mstr11		(MSTR<11>)
001100	mstr12		(MSTR<12>)
001101	mstr13		(MSTR<13>)
001110	mstr14		(MSTR<14>)
001111	mstr15		(MSTR<15>)
010XXX	sw0-7	program switch 0-7	(MSTR<16-23>)
011XXX	str0-7	Status Register<0-7>	(MSTR<24-31>)
100XXX	tag2_0-7	test tag_bit<0-7> of source 2 register	
101000	interrupt0	interrupt request 0	
101001	jr_zero	JR is zero	
101010	wfar1_1f	WFAR1=B'XX XXX1 1111'	
101011	wfar2_1f	WFAR2=B'XX XXX1 1111'	
101100	wfar1_ff	WFAR1=B'XX 1111 1111'	
101101	wfar2_ff	WFAR2=B'XX 1111 1111'	
101110	bus_release	request for bus release (common bus request from other device)	
101111	bus_timeout	timeout for multibus access	
1100??	memory_busy	memory controller busy	
1101??	interrupt1	interrupt request 1	
111XXX	opst0-7	for optional board status 0-7	

Table 4.9 Warning messages

@trcar-- is console mode only
data constant range over : Value
dptm access conflict
flag operation ignored
inrement-decrement conflict : Register
jr decrement and multi destination conflict
may not reach external label : Label_name
mstk access conflict
mstkar inrement-decrement conflict
mstkar inrement-decrement conflict
pdr/cdr is used for ALU operation
s2_immediate cause unexpected branch
shift mask ignored
shift result of qr is lost
short register direct to source...unexpected tag
source2 and flag conflict
source2 immediate value too large : Value
source2 specified though source1 direct to destination
tag immediate ignored
undefined destination : Resister
undefined source1 name : @Resister
undefined source1 name : Resister
unpredictable data from shifter
unpredictable data from shifter mask
wcs access conflict
wcsar will be incremented at the next cycle
wfar1 inrement-decrement conflict
wfar1 inrement-decrement conflict

5. むすび

パターン・マッチング，ユニフィケーション，バックトラックの機能を利用し，エラー・チェックを行なうことのできる汎用型アセンブラを開発した。このアセンブラは逐次型推論マシンψにおいて，マイクロプログラムの開発用として使用される予定である。機械定義ファイルの大きさは1000行程度であり，作成の手間はアセンブラの新規開発と比べての充分小さいと考えられる。今後はこのようなエラー・チェックの可能な汎用アセンブラが有用となろう。

6. 参考文献

1) 内田，棟上：汎用マイクロアセンブラ AMLについて
情報処理学会計算機アーキテクチャ研究会資料 34-2(1979-5-23)

2) 川口，高橋，加藤：マイクロプログラムの設計自動化
昭和48年度情報処理学会第14回全国大会予稿

3) 佐々木，高橋，宮原：ファームウェアジェネレーション支援システム
昭和53年度電子通信学会総合全国大会予稿

4) 牧之内他：
MLTG－マイクロプログラミング ランゲージ トランスレータジェネレーター
：LALRパーサの一つの応用例
情報処理学会論文誌 (1979-1)

5) 清尾，上田：汎用水平型マイクロプログラム開発支援システム－MMPS－II－
電子通信学会研究会資料 EC79-27(1979-10-19)

6) 迫田，平岡，太田，中西：
汎用マイクロプログラム・トランスレータ MARTRANの処理方式
昭和57年度後期情報処理学会第25回全国大会予稿

7) 平岡，坂東，迫田，中西：
汎用マイクロプログラム・トランスレータ MARTRANの言語仕様
昭和57年度後期情報処理学会第25回全国大会予稿