計算機アーキテクチャ 116－10
（1996．1．26）

FPGAによる関数型言語向きアーキテクチャを持つプロセッサの実装

柳井啓司　　田中哲朗　　武市正人

関数型言語向きアーキテクチャを持つプロセッサを，1万ゲート相当のFPGAを用いて実現した．本プロセッサは通常命令を実行するノーマルモードと関数型言語実行のためのリダクションモードの2種類の実行モードを持つ．リダクションモードでの実行を使用頻度の高い5つコンビネータにとどめ，他のコンビネータをノーマルモードで実行するという方針で設計をした結果，少量のハードウェアの追加で製作でき，ノーマルモードのみの実行と比較して5倍程度の速度の向上が確認された．

Implementation of a processor for Functional Languages on FPGA

KEIJI YANAI , TETSURO TANAKA and MASATO TAKEICHI

A processor for functional languages was implemented on a Field Programmable Gate Array (FPGA) with 10 thousand gates. This processor has two execution modes, "normal mode" for execution of normal instructions and "reduction mode" for reduction of combinators.

The design of this processor is to execute five frequently used combinators in reduction mode and others in normal mode. Combination of normal mode and reduction mode enables the processor to execute functional programs about five times as fast as that only with normal mode.

1. はじめに

10年程前，関数型言語向きのアーキテクチャを持つプロセッサに関する研究が盛んにおこなわれ，小長谷らによるARM[1]，Clarkeらの SKIM[2]，Berkingらの RED[3]，Darlingtonらの ALICE[4] などいくつかの簡約計算機（リダクションマシン）が提案された．しかし，コストの問題から製作まで到達した研究は少なく，特に国内では実際に製作された例は報告されていない．

本研究では，FPGA(Field Programmable Gate Array) を用い，低コストで簡約計算機を実現した．製作したプロセッサの性能は商用の高性能プロセッサに遠く及ばないが，関数型言語と汎用プロセッサとのセマンティックギャップを埋める上でどのような機能が必要になるかを示唆する点で意義があると考えられる．

2. 関数型言語の実行

2.1 関数型言語

関数型言語はラムダ算法に基礎をおいたプログラム言語である．関数型言語のプログラムは簡約規則にしたがって，与えられた式を簡約（リダクション）することによって実行を進める．例えば，関数型言語ではnの階乗を求めるfacは，

$fac = \lambda$ n. *if* (= 0 n) 1 (× n (it fac (− n 1)))

のように書ける．fac 3 の簡約の過程は以下のようになる．

fac 3 ⇒ 3*fac 2 ⇒ 3*(2*fac 1)
⇒ 3*(2*(1*fac 0)) ⇒ 3*(2*(1*1))
⇒ 3*(2*1) ⇒ 3*2
⇒ 6

簡約順序として，最左最外から簡約する正規順序簡約と最左最内から簡約する作用順序簡約があり，それぞれ遅延評価 (Lazy Evaluation) の関数型言語と先行評価 (Eager Evaluation) の関数型言語に対応する．遅延評価は先行評価よりも計算量のオーダーが落ちる場合もあり，無限長リストを使った自然なプログラミングが可能になるなどの利点を持つが，半面，単純な実装では速度の遅い処理系しか作れないという欠点がある．

2.2 コンビネータ論理

1979年にTurnerがコンビネータ論理に基づく遅延評価型関数型言語の実行モデルを提案した[5]．これは，ラムダ式から束縛変数をすべて除去し S, K の2種類のコンビネータを用いたコンビネータ式に変換して実行するというものである．ただし，S, K だけでは，変換後の式が長くなることが多いので、その他に I, B, C なども使用する。コンビネータ式においては，最左端のコンビネータが必ずリダクション可能となっており，正規順序簡約を自然に実現することができる．

以下に，S, K, I, B, C に関する簡約規則を示す．

† 東京大学大学院工学系研究科
Graduate School of Engineering, University of Tokyo

$S\ x\ y\ z \Rightarrow x\ z\ (y\ z)$

$K\ x\ y \Rightarrow x$

$I\ x \Rightarrow x$

$B\ x\ y\ z \Rightarrow x\ (y\ z)$

$C\ x\ y\ z \Rightarrow x\ z\ y$

数や代数的データ型を持つ通常の関数型言語は定数を持つラムダ算法に対応するため，組込み関数に対応するコンビネータも必要となる．先程のnの階乗を求めるプログラムfacのコンビネータ式は，以下の様になる．

$fac = ((S\ (C\ (B\ if)\ =\ 0)\ 1)(S\ \times)(B\ fac)(C\ -)\ 1)$

2.3 グラフリダクション方式

コンビネータ式はペアになったセルを用いたグラフで表現する．それに簡約規則を適用してグラフを書き換えながら簡約を行なうのがグラフリダクション方式である．例えば，簡約規則

$S\ x\ y\ z \rightarrow x\ z\ (y\ z)$

は，図1に示すようなグラフの書換えを行なう．(左が簡約前，右が簡約後)

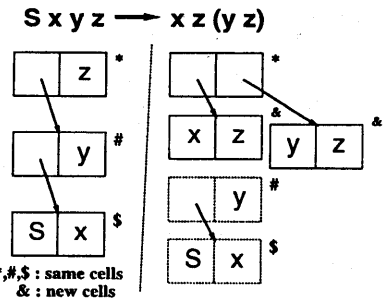

図1 コンビネータ S のリダクション

このように，変換するグラフのルートにあたるセル(*)を上書きして書き換えてしまうのがこの方法の特徴である．ルート以外のセルは他のグラフと共有している可能性があるので新しいセル(&)を使用する．

正規順序による簡約はスタックを用いて以下のように実現される．

(1) 最初に，スタックトップは評価すべきコンビネータ式のルートを指している．

(2) スタックトップが指すセルの左側が他のセルを指すポインタであれば，そのポインタをスタックに積み，グラフを下にたどる．(トラバース)

(3) スタックトップにコンビネータが来たら，そのコンビネータに定義された簡約規則に従い簡約を行なう．

(4) それ以上簡約できなくなったら簡約は終了する．

簡単なコンビネータ式 $C\ I\ 2\ (plus\ 1)$ の簡約前とコンビネータ C による簡約後の様子を図2に示す．

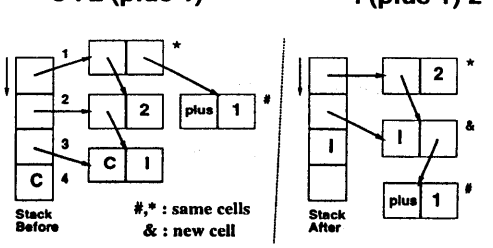

図2 グラフリダクション

3. アーキテクチャ

3.1 アーキテクチャの概要

製作したプロセッサは，グラフリダクションの処理機能を従来型のプロセッサに追加した形のアーキテクチャをもつ．その他のプロセッサの特徴を以下に示す．

- 10000ゲート相当のFPGA(Xilinx XC4010)での実現
- 内部コンビネータと，外部コンビネータ
- リダクション用の内部スタック

本プロセッサ(RBETH)は，汎用のプロセッサ(BETH)に変更を加える形で実現された．BETHは学生実験用に作られたシンプルなプロセッサで以下のような特徴を持つ．

- データバス，アドレスバスは共に16ビット
- ワードアドレッシング
- 16個の汎用レジスタ
- データ転送，算術演算，論理演算，シフト演算，比較，条件分岐の各命令を備える
- 命令長は，1～2ワード
- 水平型マイクロプログラムによる制御
- 6MHzのクロックで動作

16ビットアドレス空間は実用的な関数プログラムの実行には不足しているが，ゲート数の都合で拡張できなかった．

RBETHのインストラクションはBETHのインストラクションに，リダクション実行開始命令，後述する外部コンビネータリターン命令，スタック操作命令，サブルーチンコール命令を追加したものになっている(表1)．また，アドレッシングモードも拡張されている(表2)．

RBETHでは，初期状態ではノーマルモードで通常処理を行なうが，リダクション実行開始命令の実行によって，リダクションモードに移行し，コンビネータ表現に変換された関数型プログラムの簡約を行なう．簡約が終了すると再び，ノーマルモードに戻る．

3.2 内部コンビネータと外部コンビネータ

関数型言語には，いくつもの組み込み関数が存在する

表1 命令セット (上段:BETH 命令, 下段:RBETH 拡張命令)

命令の種類	ニモニック
転送命令	ld, st
算術命令	add, adc, sub, sbb, neg
比較命令	cmp
論理命令	and, or, xor, not
増減命令	inc, dec
シフト命令	sr, sl, rr, rl, ar, al
条件分岐命令	jpa, jpn, juhi, juhe, julo, jne, jeq
	jvc, jvs, jpl, jmi, jshe, jslo, jshi, jsle
リダクション命令	redex
外部コンビネータリターン	ret_comb
スタック命令	push, pop, sread, s_init
サブルーチン命令	jsub, ret

表2 アドレッシングモード (上段:BETH, 下段:RBETH 拡張)

アドレッシングモード	ニモニック
レジスタ直接	Rn
即値	e
レジスタ間接	[Rn]
絶対アドレス	[e]
インデックス付きプログラムカウンタ相対	[PC+Rn]
ディスプレースメント付きレジスタ間接	[Rn+e]
ディスプレースメント付きプログラムカウンタ相対	[PC+e]
レジスタプラス1間接	[Rn++]

が, そのすべてをハードウェア化するのはゲート数の点で困難である. そこで, プロセッサ内部で実現するのは S, K, I, B, C の5種類のコンビネータにとどめ (これらを内部コンビネータと呼ぶ), その他の組み込み関数を外部コンビネータとしてソフト的に追加する方針を採用した.

リダクションモードの実行中にスタックトップに外部コンビネータが来たら一時的にリダクションモードからノーマルモードに移行し, その外部コンビネータを処理するプログラムを呼び出す. 処理が終ると再び特殊なリターン命令 (外部コンビネータリターン命令) で, リダクションモードに復帰する. 外部コンビネータと内部コンビネータは, 処理が機械語レベルでおこなわれるか, マイクロプログラムで行なわれるかの違いのみで, 基本的な動作自体は同じである.

3.3 内部スタック

グラフリダクション方式はスタックへのアクセスを頻繁に行なうので高速なスタックをプロセッサ内部に持つことにより実行速度が上がることが期待される.

本プロセッサでは16ワードの内部スタックを実現した. スタックの長さが16ワード内に収まっていれば, 内部スタックだけがアクセスされる. 内部スタックが一杯のときにプッシュする場合は, 一番古い内容をメモリのスタック領域に書き出して新たな領域を確保するオーバーフロー処理を行なう. 逆に内部スタックが空のときにポップを行なう場合は, メモリのスタック領域から内部スタックへ読み出すアンダーフロー処理を行なう.

通常のメモリアクセスが2サイクル掛かるのに対して内部スタックは1サイクルでのアクセスが可能となる. ただし, スタックのオーバーフロー時には3サイクル, アンダーフロー時には2サイクル掛かる.

3.4 コンビネータ式の表現

コンビネータ式は, 2ワードの大きさのセルのグラフとして表現する. 例えば,

$f\ 1\ 2\ 3$

という式は, f のコードを 6000h とし, メモリ上の 8000h 番地から置くとすると, 図3のように表現される.

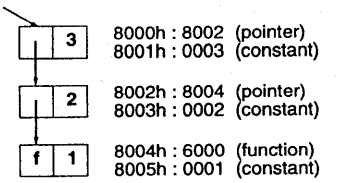

図3 コンビネータ式の内部表現

これをコンビネータ形式と呼ぶ. コンビネータ形式は, Lisp のリスト形式とは引数の順番とグラフの深さの対応が逆になるので注意が必要である.

内部コンビネータ, 外部コンビネータ, ポインタ, 数値, NULL 値は16ビットで, 表3のように表現される. 外部コンビネータの値は, 外部コンビネータの処理ルーチンが書かれているアドレスが, そのまま外部コンビネータの表現値となっている.

表3 コンビネータ等の内部表現

表現対象	上位4ビット				値の範囲
整数値 (定数)	0	0	x	x	0000h→3FFFh
内部コンビネータ	0	1	0	0	4000h→4FFFh
外部コンビネータ	0	1	1	x	6000h→7FFFh
ポインタ	1	x	x	x	8000h→FFFFh
NULL 値 (マーク)	0	1	0	1	5000h

3.5 メモリ空間

リダクションモードでは表4のようにメモリ空間が使用される.

表4 リダクションモードのメモリ空間

0000h→7FFFh	ROM
5000h→	Overflow Routine
6000h↔7FFFh	External Combinator
8000h→FEFFh	RAM
8000h↔FEFFh	Combinator Program
8000h↔FEFFh	Heap Erea (in Free List)
←FEFFh	Stack Erea
FF00h→FFFFh	I/O (KEY,LED etc.)

スタックは, RAM の上限である FEFFh 番地から小

さい番地の方向へ伸びていく．下限を設定でき，それを越えると，スタックオーバーフロー割り込みが発生する．

タグの関係でヒープ領域は8000h番地以降のアドレスに置かれる．ヒープ中の空きセルはフリーリストで管理する．フリーリストの最後のセルには本システムでのNULLコード(5000h)が書かれており，フリーリストがすべて消費されるとヒープオーバーフロー割り込みが発生する．

3.6 リダクションモードの実行

リダクション開始命令(redex)の実行によってリダクションモードに入り，簡約を開始する．簡約するコンビネータ式はR0(汎用レジスタ0番)で指定し，簡約の結果は同じR0で返される．具体的な処理内容は，R0が数値かNULLの時は終了し，それ以外の時は，それぞれの処理を行ってR0を更新し，さらに新しいR0の内容に応じた処理が選択されるというループになっている(表5)．

表5 redex命令の処理

No.	動作内容	動作(分岐条件)
1	redex命令の読み込み	Fetch
2	PCを退避する	PUSH PC
3	スタックに印を付ける	PUSH MARK
4	R0の内容によって分岐[分岐A]	
	(内部コンビネータ処理へ)	R0(上位4ビット)=0100
	(外部コンビネータ処理へ)	011x
	(ポインタ処理へ)	1xxx
	(数値処理へ)	00xx
	(NULL処理へ)	0101
	[内部コンビネータ処理]	(別に説明)
	[外部コンビネータ処理]	(別に説明)
E-1	R0の指す番地へJUMP	R0→PC
	[ポインタ処理]	(トラバース)
P-1	ポインタをPUSH	PUSH R0,R0→MAR
P-2	ポインタの指す内容のREAD	[MAR]→R0,分岐Aへ
	[数値,NULL処理(終了処理)]	(redexの終了処理)
N-1	印をPOPする	POP MAR
N-2	PCを復帰し，終了する	POP PC

内部コンビネータの実現例として，コンビネータBの簡約の動作を図4，その処理を表6に示す．まず，引数を3つ読み込み，フリーリストからセルを1つ獲得する．1つ目の引数は簡約後の式において最左端になるのでR0の値とし(表6,2行目)，トップセル(*)は簡約後もトップセルとなるのでスタックをREAD(POPと異なりスタックポインタを変化させない)して，トップセルへのポインタをそのままスタックに積んでおく(6行目)．トップセルと新しいセルを用いて$x(yz)$を作り，2番目と3番目のセルは他の式に共有されている可能性があるので，そのままにしておく．共有されていない場合はゴミとなる．

外部コンビネータの実現例として，組み込み関数plus(+)の簡約の動作を図5，その処理を表7に示す．+ (+ 3 2) 1という式を簡約する場合，表7の4行目で1番目の引数の(+ 3 2)が再帰的に簡約されて5になり，2番目の引数の1と14行目で加算されて，R0に

表6 内部コンビネータBの処理

No.	動作内容	動作
1	第1引数の読み込み	POP MAR
2		[MAR+1]→W0,R0
3	第2引数の読み込み	POP MAR
4		[MAR+1]→W1
5	第3引数の読み込み	SREAD MAR
6		[MAR+1]→W2
7	新しいセルの獲得	HP0→HP1,[HP0]→HP0
8	トップセルの書き換え	W0→[MAR]
9		HP1→[MAR+1]
10	右のセルへの書き込み	W1→[HP1]
11		W2→[HP1+1]
	処理の終了	分岐Aへ

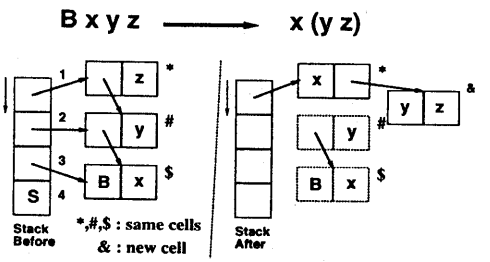

図4 内部コンビネータBの動作

6という結果を得る．トップセル(*)は他の式にも共有されている可能性があるので，I 6で書き換えておく．

表7 外部コンビネータ+の処理

No.	動作内容	動作	
1	第1引数のアドレスをリード	sread	R13
2	引数不足のチェック	jpn	lacks
3	第1引数をロード	ld	R0,[R13+1]
4	第1引数の評価	redex	
5	第1引数のアドレスをポップ	pop	R13
6	第1引数を書き換える	st	R0,[R13+1]
7	第2引数のアドレスをリード	sread	R13
8	引数不足のチェック	jpn	lacks
9	評価済み第1引数の退避	push	R0
10	第2引数をロード	ld	R0,[R13+1]
11	第2引数の評価	redex	
12	評価済み第1引数の復帰	pop	R14
13	第2引数のアドレスをポップ	pop	R13
14	演算	add	R0,R14
15	コンビネータIのコード	ld	R14,0x4027
16	トップセルの左の書き換え(I)	st	R14,[R13]
17	トップセルの右の書き換え(結果)	st	R0,[R13+1]
18	処理の終了，分岐Aへ	ret_comb	
19	lacks: [引数不足の処理]	pop	R13
20	redexの終了，normal modeへ	ret	

3.7 ブロック図

プロセッサ内部のブロック図を図6に示す．
各ブロックの構成と動作を以下に示す．

- ALU(Arithmetic and Logic Unit)
- PC(Program Counter)
- MAR,MAR2(Memory Address Register)
- IR(Instruction Register)
- REGISTER(16 word×2, Work Register×4, Heap Pointer×2)
  2組のレジスタファイルが用意され，同時に2つの

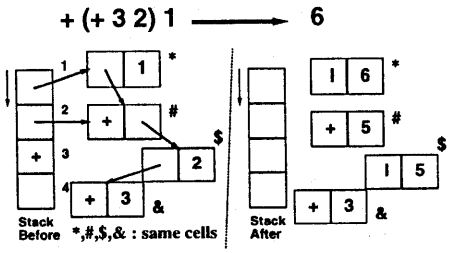

図5 外部コンビネータ＋の動作

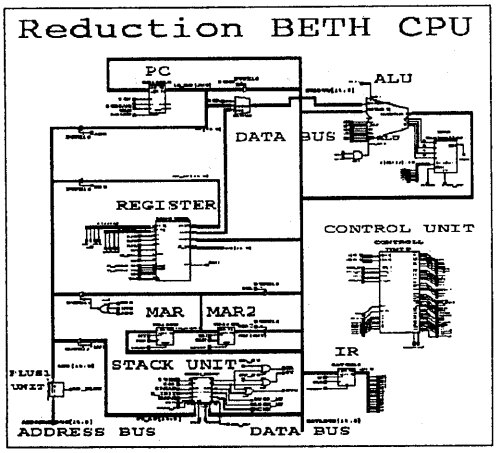

図6 RBETH のハードウェア構成

レジスタにアクセスできる．RBETH では，ワークレジスタ 4 つとヒープポインタ 2 つが追加されている．

- CONTROL UNIT
  32 ビット ×64 ワードの水平型マイクロコードにより，プロセッサ全体のコントロールを行なう．
- JUMP CONTROLER
  分岐の条件を判断する部分．
- STACK UNIT
  内部スタック．16 ワードのメモリ，スタックポインタ，スタックの下限を記憶するレジスタ，スタックの動作を制御するコントローラが含まれる．
- ADDRESS PLUS ONE UNIT
  セルの右側にアクセスし易くするために，ALU を介さずに，アドレスバスの信号に＋1することができる．

4. 評　　価

4.1 ハードウエアコストの見積り

設計したプロセッサ (RBETH) は，XC4010 に用意された CLB function generator 800 個のうち，663 個を使用するだけで実現できた．元にした汎用プロセッサ (BETH) では 336 個を使用したので，2 倍弱にあたる．Flip-Flop, RAM, 3-state buffer 等の使用の比率もほぼ同様だったので，ハードウェアの使用量は 2 倍弱であると見積もることができる．

4.2 リダクションモードによる高速化の検証

実現したプロセッサを実験ボードシステム (図 7) に組み込み，評価を行なった．リダクションモードによるハードウエアでの関数型プログラムの実行時間とノーマルモードで動くインタプリタによるソフトウエアでの実行時間を比較することにより，ハードウェア化による高速化を検証した．

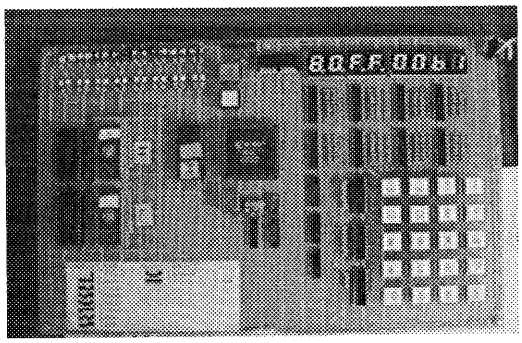

図7 実験ボード

3 つの小さなサンプルプログラムについて，テストを行なった．実用を目指す簡約計算機を評価するには大規模なプログラムを実行する必要があるが，本簡約計算機はあくまでも実験用の CPU で，メモリの制約もあるので，テストには簡単なプログラムを用いた．

プログラムは，1 から n までの和を求める `sigma`, n の階乗を求める `fact`, フィボナッチ数を求める `fib` の 3 つである．`sigma 100,fact 5,fib 10` をそれぞれ 1.5MHz で 100 回実行した時のハードウエアとソフトウエアによる実行時間の実測値を表 8に示す．これより，平均して 5 倍以上の高速化が達成できていることが分かる．

表8 プログラムの実行時間 (実測)　(単位: 秒)

Prog.	software	hardware	ratio
sigma 100	20.23	3.89	5.2
fact 5	12.53	1.98	6.3
fib 10	32.34	5.94	5.4

内部実行時間の内訳を調べるために，ワークステーション上でのシミュレータを用いて，内部コンビネータ，外部コンビネータ，トラバース，リダクション呼び出し (再帰的リダクション実行) の 4 つの処理時間の見積りを

表9 プログラムの実行時間 (シミュレーション) (単位: 秒 (%))

Prog.	Int.C.	Ext.C.	Trav.	Redex	Total
sigma 100	9.29	3.34	4.77	2.76	20.16
(soft)	(46.1)	(16.6)	(23.7)	(13.7)	
	1.03	1.98	0.40	0.21	3.71
(hard)	(27.8)	(53.5)	(10.8)	(5.7)	
fact 5	7.51	1.15	3.20	0.96	12.82
(soft)	(58.6)	(9.0)	(25.0)	(7.5)	
	0.88	0.68	0.27	0.07	1.93
(hard)	(45.6)	(35.2)	(8.9)	(3.6)	
fib 10	15.06	5.17	7.84	4.29	32.36
(soft)	(46.8)	(16.0)	(24.3)	(13.3)	
	1.76	3.08	0.65	0.33	5.86
(hard)	(30.2)	(52.8)	(11.1)	(5.7)	

した (表9). これによると, ノーマルモードでのインタプリタによる実行では, コンビネータの処理に全実行時間の約50%掛かっていることが分かる. さらに, グラフをたどっていく処理であるトラバースに約25%掛かっている.

ハードウエア化によって, 内部コンビネータは約9倍, トラバースは約12倍の高速化を達成し, 全体の実行速度の向上に大きく寄与していると考えられる. 外部コンビネータの処理は, リダクションモードにおいても, ソフトウエア的に実行しているが, 処理ルーチンを呼び出す時のオーバーヘッドが減少し2倍程度, 同様に, リダクション呼び出し処理もオーバーヘッドが減少し13倍程度, 速くなっている.

以上のことから, 使用頻度の高い S, K, I, B, C の5つのコンビネータとトラバース処理を中心としたハードウエア化で, 十分な高速化が実現できているといえる.

4.3 内部スタックの効果の検証

内部スタックの容量を変化させた時の実行時間の変化と, 内部スタックあふれの割合を, 前節と同じサンプルと `fib` の値を15にした場合について, シミュレータを用いて調べてみた (表10). 実行時間は, 内部スタックのあふれと関係し, あふれが少ないほど高速に実行できる.

表10 内部スタックの容量による実行時間の変化と内部スタックあふれの割合 (単位: 秒,%)

Stack size	0	4	8	16	32	64
sigma 100	4.44	3.83	3.71	3.71	3.70	3.70
	100	24.4	9.8	9.6	9.2	8.4
fact 5	2.36	2.03	1.95	1.93	1.93	1.92
	100	27.0	7.7	5.2	3.4	1.8
fib 10	7.13	6.16	5.93	5.86	5.82	5.82
	100	25.7	8.1	2.7	0.3	0.0
fib 15	79.6	68.8	66.2	65.4	65.1	65.0
	100	25.7	8.1	2.7	0.4	0.0

表によると, 内部スタックの容量が8ワードまでは, あふれが減少し, 実行時間も減少しているが, 8ワードより大きくしても, あふれ, 実行時間共にあまり変化がないことが分かる. ただ, `fib` においては16ワードにするとあふれが1/3に減少する. このことより, 内部スタックは本CPUが備えている16ワードあれば, 十分であると考えられる.

5. まとめ

本研究では関数型言語の高速実行のために, 通常の実行モードとリダクションモードを合わせ持つプロセッサを設計, 製作した. リダクションモードでの実行を使用頻度の高い5つコンビネータにとどめ, 他のコンビネータをノーマルモードで実行するという方針で設計をした結果, 少量のハードウェアの追加で製作でき, ノーマルモードのみの実行と比較して平均5倍以上の速度の向上が確認された.

本プロセッサで採用した実行モデルは1979年に提案されたもので, その後, G-machine[6], TIM[7], STG[8] など, 汎用プロセッサ上での実行に適した実行モデルがいくつも提案されている. 実用的なプロセッサの実現には, これらの実行モデルに対応したアーキテクチャの検討も必要となるだろう.

従来のリダクションチップの多くがペーパーマシンに終ったのに対して, 実際に動作するものを作ったことは, おそらく日本では初めての例と思われ, その意味において意義あるものといえる.

参考文献

1) 小長谷, 山本, 北森, 内藤: リダクションマシン"ARM"の基本アーキテクチャ, 信学技報 EC 82-88, pp.69-78 (1982).
2) T.J.W.Clarke,P.J.S.Gladstone,C.D.Maclean and A.C.Norman: SKIM – The S,K,I Reduction Machine, ACM Conference on LISP and Functional Programming, pp.128-135 (1980).
3) K.J.Berkling: Reduction Language for Reduction Machines, Proc. Second Int. Symp. Computer Architechture, pp.133-140 (1975).
4) J.Darlington and M.Reeve: ALICE: A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Language, Proc. Functional Programming Languages and Computer Architechture, pp.65-75 (1981).
5) D.A.Turner: A New Implementation Techniqu e for Applicative Languages, SOFTWARE-PRACTICE AND EXPERIENCE Vol.9, pp.31-49 (1979).
6) L.Augustsson: A Compiler for Lazy ML, Proceedings of LFP 1984, pp.218-227 (1984).
7) J.Fairbairn and S.Wray: TIM – a simple lazy abstract machine to execute supercombinators, functional Programming Languages and Computer Architecture, LNCS 274, Springer Verlag (1987).
8) P.S.L.Jones: Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine, Journal of Functional Programming, Vol.2 No.2 (1992).