計算機アーキテクチャ　108－12
（１９９４．１０．２８）

関数型言語指向プロセッサ アーキテクチャ

田中哲也[1]　荒木啓二郎　福田晃

奈良先端科学技術大学院大学 情報科学研究科

Abstract

リダクション・マシンに用いるプロセッサとして、RISC アーキテクチャをベースにしたアーキテクチャを提案する。本アーキテクチャは、関数プログラムの格納領域としてオンチップ・スタックをプロセッサ内部に持つため、関数プログラムの高速なアクセスができ、複数のスタックとスタックへのポインタを実現する機構により、高速な関数評価を実現する。さらに、オンチップ・スタックを共有し、独立に動作可能なマルチ・パイプライン・ユニットにより、並行評価を行う機構を備えている。これらの特徴により、本プロセッサ・アーキテクチャは関数型言語を高速に実行する。

Processor Architecture for Functional Programs

Tetsuya TANAKA[1], Keijiro ARAKI and Akira FUKUDA

Graduate School of Information Science
Nara Institute of Science and Technology

8916-5, Takayama, Ikoma, Nara 630-01 Japan
{*tetuya-t,araki,fukuda*}*@is.aist-nara.ac.jp*

Abstract

In this paper, we propose a processor architecture for reduction machines which is based on a RISC architecture. The proposed architecture can access functional programs fast, using an on-chip-stack in the processor for storing functional programs. And, the architecture can evaluate functions fast using a mechanism supporting a multi-stack and its pointer-to-stack as the on-chip-stack. Furthermore, the architecture can evaluate functions in parallel using multi-pipeline which can run in independent each other and share an on-chip-stack. By these features, the architecture can execute functional programs fast.

[1]松下電器産業(株)より派遣中

1 はじめに

近年、関数型言語はプログラムの生産性・検証容易性の観点から注目されている。また、並列処理の分野においても、関数型言語はアルゴリズムに内在する並列性を自然に表現でき、参照透明性により関数の評価結果が評価の順番に依存しないことから、注目されてきている。しかしながら、関数型言語に基づく計算モデルは、従来のノイマン型計算機とは必ずしも親和性が高くないため、ノイマン型計算機で高速に実行するのが困難であるという問題がある [1]。

このような問題を解決するため、これまでに関数型言語を指向した非ノイマン型のアーキテクチャの研究がなされ、リダクション・マシンやデータフロー・マシンとして良く知られている。このような非ノイマン型アーキテクチャは古くから検討されているが、現段階ではノイマン型アーキテクチャを置き換えるほどの成果は得られていないと考えられる。

リダクション・マシンとデータフロー・マシンを比較した場合、一般にリダクション・マシンの方が言語モデルとの親和性が高いといわれている [2]。リダクション・マシンは一般に評価すべき関数プログラムを格納する格納領域と、書き換え規則を命令として持つプロセッサで構成される。関数プログラムは関数とデータを区別なく格納領域に格納され、書き換え規則に従って書き換えるステップを繰り返すことで計算が達成される。関数プログラムを格納する時の表現形式はグラフ (木構造) が一般に用いられており、グラフで表現したプログラムの書き換え処理がグラフ・リダクションと呼ばれている。

一方、近年の VLSI 技術の発展は目覚しく、非常に大規模なハードウェアを集積化できるようになった。このため多くの記憶装置や、演算器をワン・チップに集積できるようになり、プロセッサ内部で並列処理ができるようになると考えられる。また、このような LSI 技術の発展は、RISC プロセッサに代表されるノイマン型アーキテクチャの性能を飛躍的に向上させ、その結果、非常に処理能力の高いプロセッサが得られるようになった。

本稿は、リダクション・マシンに用いるプロセッサ・アーキテクチャとして、RISC アーキテクチャをベースに、関数プログラムを格納し高速な書き換えを実現するためのスタック機構と、並行評価を行うためのマルチ・パイプライン機構とを備えたアーキテクチャを提案する。まず、第 2 章でリダクション・マシンの一般的な構成と計算メカニズムを紹介し、第 3 章では本稿で提案するアーキテクチャについて、一般的なリダクションマシンと異なる特徴を示す。第 4 章では本アーキテクチャを実装する上での問題点と解決策を示す。

2 リダクション・マシン

関数型言語の計算は関数プログラムの評価によって達成される。つまり、関数プログラムは目的とする値を表現するものであると考えられ、値を表現している式は、参照透明性によりその一部をそれと等価な表現に置き換えても値 (意味) は変化しないので、式を最も単純で等価な表現に書き換えることで評価を達成する [2][3]。このような書き換えのステップをリダクションといい、すべての計算をリダクションで実現しようとする計算機がリダクション・マシンと呼ばれている。リダクション・マシンは、図 1 に示すように、リダクションの対象となる式を格納する格納領域と、書き換え規則を備えたプロセッサからなる。式の格納領域に関数およびデータの両方を格納されるので、関数とデータは同格に取り扱われる。一般に、リダクション・マシンは次のアルゴリズムに従って動作する [1][4]。

1. 式を走査して、書き換え可能な項 (以下、redex と呼ぶ) を検出する。検出できなければ計算終了となる。
2. 検出した redex から、書き換えを行う redex を選択する。

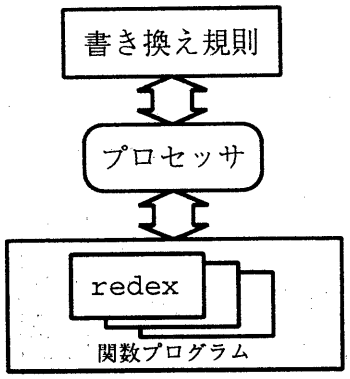

図 1: リダクション・マシンの概念図

3. 選択したredexを書き換え規則に従って書き換え、1に戻る。

式を2分木(グラフ)で表現し、木を書き換えることでリダクションを行う方法をグラフ・リダクションといい、リダクション・マシンにおいて一般的に用いられている。グラフ・リダクションにおいては、式の走査は木の走査に対応し、検出したredexは部分木に相当する。また、redexの書き換えは部分木の書き換えに相当する。[4]

リダクション・マシンの計算メカニズムの例を図2に示す。図2は以下に示す関数プログラムにおいて$f12$を評価するステップを式の書き換えと木の書き換えについて示すものである。

$$g\,a\,b = a + b$$
$$h\,a\,b = a - b$$
$$f\,a\,b = (g\,a\,b) + (h\,a\,b)$$

但し、f, g, hは関数名。+, −は組み込み関数。

図2において、(b)や(c)のようにredexが複数あるときは、関数型言語の参照透明性により、これらのredexを並行に書き換えることができる。

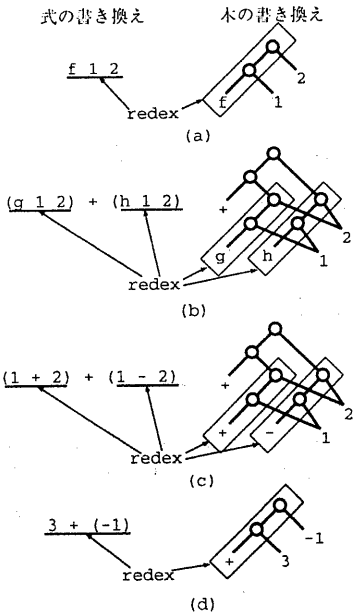

図 2: 計算のステップ

3 プロセッサ・アーキテクチャ

3.1 アーキテクチャの概要

本稿で提案するプロセッサ・アーキテクチャは、RISCアーキテクチャをベースにしており、前述したリダクション・マシンの処理を行う機能を備え、特にredex検出・評価の高速化および並行評価機構に重点を置いて設計されている。RISCアーキテクチャの選択の理由は、非常に高い性能を出すことのできるアーキテクチャであり、ハードウェアが単純でかつ機能拡張が容易であることである。そのため、RISCの特徴を損なうことがないように機能拡張はできるだけ単純にした。また、木の格納領域である大容量のオンチップ・スタックと、並行評価のためのマルチ・パイプライン・ユニットのように多くのハードウェアを利用するための技術的な背景としてVLSIの高集積化技術を前提と

した。

以下に本アーキテクチャの特徴を示す。

- オンチップ・スタック
- 関数評価命令と書き換え規則
- 並行評価機構

これらの機能により本プロセッサ・アーキテクチャは関数型言語を高速に実行する。以下にそれぞれの内容を説明する。

3.2 オンチップ・スタック

本アーキテクチャは、グラフ・リダクションにおける木の格納と、高速な書き換えを行うため、プロセッサ内部にオンチップ・スタックを持つ。図

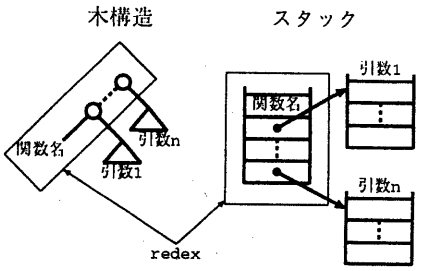

図 3: 木構造とスタックの対応

3は木構造とスタックの対応を示している。通常、redex は関数適用の形をしているので、関数名と引数からなっている。図3の木構造においては、redex は関数名とノード (図中で○印) で表され、ノードの右側には引数としての部分木があり、より内側の redex もしくは値を表す。図3のスタックにおいても、redex は一つのスタックに割り当てられ、引数および関数名が格納される。スタックに格納される関数名は関数に対応する書き換え規則命令列の先頭アドレスであり、引数は他のスタックを指すポインタが格納される。

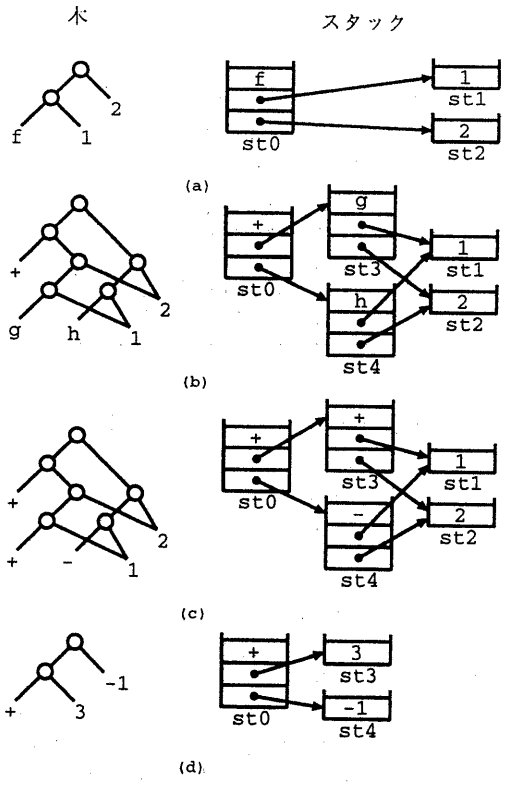

図 4: 木およびスタックの書き換えステップ

図4は木とスタックについての書き換えのステップを示しており、木構造とスタックの対応の具体例を表している。例えば、図4(b) については、スタック st0 に関数名"+" と、引数としてスタック st3 およびスタック st4 のポインタが格納されており、スタック st3 に関数名"g" と、スタック st1 およびスタック st2 のポインタが、スタック st4 に関数名"h" と、スタック st1 およびスタック st2 のポインタがそれぞれ格納されている。スタック st1、スタック st2 には値が格納されている。図4(b) のように引数としてスタック (部分木) を共有することもある。

以上のように複数のスタックを用い、スタックのポインタによる参照を可能にすることで、各redexはそれぞれ対応するスタックに割り当てられるので、引数(部分木へのポインタ)の参照だけでredexの検出ができる。また、このようなスタックをチップ内にハードウェアとして持つので、高速な式へのアクセスができる。

このようなスタックの構造を許すため、本アーキテクチャは複数のスタックを持ち、識別番号(ポインタ)により各スタックを識別・参照できる。さらに、命令により新たな識別番号のスタックの確保や、不要になった識別番号のスタックの開放ができる。

3.3 関数評価命令と書き換え規則

前述したように、関数プログラムの評価を行うためには、木を走査してredexを検出する処理と、redexに対応する部分木を書き換える処理とが必要である。また、木を格納しているスタックには関数とデータが区別なく格納される必要がある。このため、本アーキテクチャは関数評価命令を持つ。

関数評価命令(以下、Eval命令と呼ぶ)はスタックのポインタをオペランドとして要求する。Eval命令が実行されるとオペランドとして与えられたスタックを参照してスタック・トップにある関数名に対応する書き換え規則に制御を移す。その際、それまでに実行中の書き換え規則は中断され、次に実行すべきアドレスがリターン・スタックに積まれる。なお、リターン・スタックはオンチップ・スタックの中のひとつが使われる。このような処理の流れは部分木をオペランドとして与えることにより、部分木の評価を開始することに対応する。

また、Eval命令によりスタックに格納された関数へ制御を移すことができるため、スタックに関数を格納しておくことができる。

書き換え規則は主に関数評価命令・スタック操作命令などを組み合わせて構成される。以下にredex検出および書き換えの動作の概要を示す。

redexの検出 簡約法により処理の順番が異なる。簡約法は命令の並びを変えてやることで選択できる。ここでは、最内簡約法と最外簡約法に関して動作の概要を示す。

1. 最内簡約法—現行スタックのredexの書き換えを行う前に、引数の評価を行う。引数のポインタをオペランドとしてEval命令を発行することで行われる。現行スタックのredexの書き換えは引数の評価が完了し、現行スタックの関数の書き換え規則に制御が戻った後に行われる。

2. 最外簡約法—現行スタックのredexの書き換えを行い、引数はそのまま評価されずに残される。後に引数の値が必要になった時に改めて評価される。

redexの書き換え 書き換えはスタックに積まれている引数をすべて取り出した後、関数の本体をスタック上に構築する処理を行う。必要な場合は新たなスタックを確保し使用する。

3.4 並行評価機構

本プロセッサ・アーキテクチャは、それぞれが独立に動作する複数のパイプライン・ユニット(以下、マルチ・パイプライン・ユニットと呼ぶ)に基づいた並行評価を行なう機能を有する。マルチ・パイプライン・ユニットはプログラム・カウンタ／汎用レジスタ／演算器を個別に持ち、一方ではオンチップ・スタックを共有している。オンチップ・スタックを共有することにより、各パイプライン・ユニットは評価すべき式を共有できる。

並行評価はEvalP命令(並行評価用の関数評価命令)で実現される。あるパイプライン・ユニットでredex検出を行った後、そのredexのスタックをオペランドとしてEvalP命令を発行することにより、他の動作中でないパイプライン・ユニットでの評価が開始される。EvalP命令を発行したパイプライン・ユニットはEvalP命令発行後は、以降の命令を実行し他のredexを評価できる。

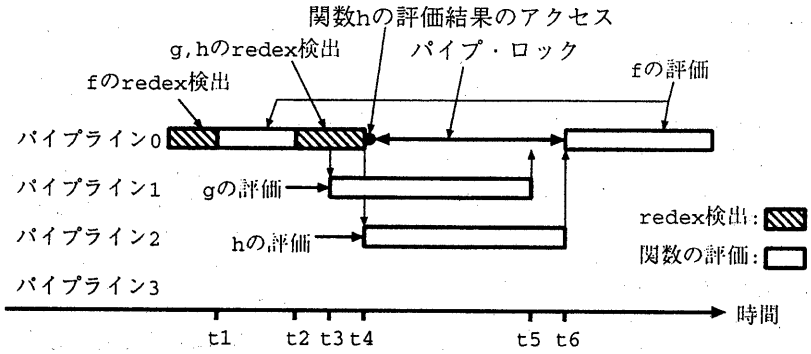

図 5: 並行評価のタイムチャート

オンチップ・スタックはパイプライン・ユニットにより共有されているので、書き換え中のスタックを他のパイプライン・ユニットがアクセスすることのないように、書き換えを行うパイプライン・ユニットを限定する必要がある。本アーキテクチャは各スタックに属性を持たせることで評価中のスタックの排他制御を行なう。スタックの属性の内容を以下に示す。

1. スタックの所有者—どのパイプライン・ユニットが所有しているか。
2. スタックの状態—redex・reducing・normal いずれかの状態をとる。

redex 状態 redex 状態のスタックには redex が格納されており、その評価はどのパイプライン・ユニットでも可能である。どれかのパイプライン・ユニットが EvalP 命令で評価を始めるとスタックの所有者が評価を行うパイプライン・ユニットになり、スタックの状態は reducing になる。

reducing 状態 reducing 状態のスタックにはスタックの所有者による書き換え中の redex が格納されており、所有者でないパイプライン・ユニットからのアクセスができない。所有者でないパイプライン・ユニットがアクセスした場合、そのパイプライン・ユニットはスタックの状態が他の状態になるまでパイプ・ロックされる。

normal 状態 normal 状態のスタックには評価結果 (正規形) が格納されており、そのスタックへのアクセスはどのパイプライン・ユニットでも可能である。ただし、Eval 命令や EvalP 命令がこのスタックに発行された場合は何も行われない。

図5は各パイプラインが独立に動作することで並行評価している様子を示している。横軸は時間であり、縦にそれぞれのパイプライン・ユニットの動作状況を示している。長方形は各パイプライン・ユニットが動作していることを表しており、下方向の矢印は書き換え規則の起動を、上方向の矢印は評価が終了し、スタックの状態を変化させたことを表す。図5において、関数"g" の書き換えはパイプライン1で、関数"h" の書き換えはパイプライン2でそれぞれ並行に行われており、それぞれパイプライン0による redex 検出後、EvalP 命令で起動されたものである。パイプライン0は時間 t4 で関数"h" の結果をアクセスしようとするが、時間 t6

より前では関数"h" の格納されているスタックはパイプライン2により評価中(スタックがreducing状態) であるので、パイプライン0はパイプ・ロック状態になる。時間t6になると、パイプライン2が関数"h" のスタックの内容を評価結果に書き換え、スタックの状態をnormalにするので、パイプライン0は関数"h" の評価結果を取り出すことができ、関数"f" の評価を再開する。

4 実装上の問題点

図6に本アーキテクチャに基づいたプロセッサのブロック図を示す。本構成例では図6に示すようにハードウェア・スタックを共有した独立なパイプラインが4つある。それぞれのパイプラインに汎用レジスタ、プログラム・カウンタ、演算器が個別に存在する。以下に本プロセッサの実装上の問題点と解決策のポイントを述べる。

オンチップ・スタック オンチップ・スタックはチップ上に実装するので容量的な制約がある。関数プログラムは比較的小さな関数を多数使うので、スタックの深さよりはスタックの数の制約が大きな問題となる。この問題を解決するため、本プロセッサのオンチップ・スタックは次のように実現した。

- 各スタックに論理番号を持たせ、オンチップ・スタックの物理番号とスタックの論理番号は連想記憶装置で対応づけられる。
- オンチップ・スタックの物理番号がすべて使用された場合は、例外処理によりオンチップ・スタックの内容をメインメモリに退避し、物理番号を開放する。

以上のような論理的な番号を用いることで非常に多くのスタックを識別でき、使用することができる。

命令の機能と汎用レジスタ 本プロセッサ・アーキテクチャはRISCアーキテクチャをベースにしているので、これまでに述べた拡張機能を複雑な機能で実装するとRISCとしての利点が損なわれ、性能低下の要因となる。この問題を解決するため、本プロセッサの命令は単機能命令とした。

例えば、スタック操作命令の場合、スタック上での演算命令を設けるとスタックのpopおよびpushを一命令で実行する必要がある。そのためパイプライン上の複数のステージにスタック・アクセスが必要になり、パイプラインの複雑化を招くばかりでなく、性能が低下する可能性がある[5]。これを回避するためスタック操作はpush,popなどの単機能に止め、スタック演算命令の代わりにレジスタ間演算命令を設けた。これに伴って、各パイプライン・ユニット内に汎用レジスタを設けた。

間接的スタック操作 スタックに格納されている引数は他のスタックへのポインタである。このポインタは操作するスタックを指定したり、関数評価命令のオペランドとして使用される。このため、ポインタによる間接的にスタックを取り扱う機構が必要である。本プロセッサでは、汎用レジスタにスタックのポインタを格納し、スタック操作命令や関数評価命令には汎用レジスタを指定することで解決した。

5 まとめ

本稿はリダクション・マシンに用いるプロセッサのアーキテクチャを提案した。本アーキテクチャはRISCアーキテクチャをベースにしており、次の特徴で関数型言語を高速に実行する。

- 関数プログラムを木として格納し、高速にアクセスできるオンチップ・スタックを有する。オンチップ・スタックは複数のスタックにより構成されており、各スタックにredexを割り当てることで、高速にredexの検出ができる。
- 関数とデータを同格に取り扱うための関数評価命令を持つ。

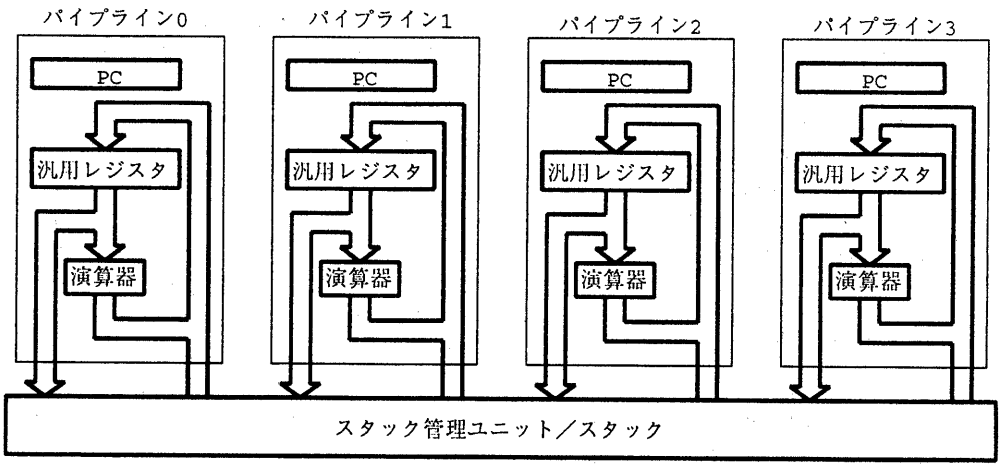

図 6: プロセッサ ブロック図

- マルチ・パイプライン・ユニットによる並行評価機構と、スタックの属性による各パイプライン・ユニットの排他制御機構を持つ。

今後の課題として以下があるが、性能評価については別の機会に報告したい。

- 性能評価
- スタックの管理方法
- ガーベジ・コレクション
- メモリのボトルネック
- 正規形の存在しない場合、または簡約法により正規形に到達しない場合の対処方法
- ストリーム・パイプライン並行評価機構

参考文献

[1] 小長谷明彦, 山本昌弘: " 関数型言語とリダクションマシン", 情報処理学会, Vol.26, No. 7, pp.751-763,July 1985.

[2] 武市正人: " 関数プログラミングの実際", 日本ソフトウェア科学会 コンピュータソフトウェア, Vol. 8, No. 1, pp.3-11, 1991.

[3] Richard Bird, Philip Wadler (武市正人 訳): " 関数プログラミング", 近代科学社, 1991.

[4] Simon L.Peyton Jones, David R.Lester: "Implementing Functional Languages", Prentice Hall, 1992.

[5] Philip J.Koopman Jr.(田中清臣 監訳/藤井敬雄 訳): " スタック コンピュータ", 共立出版, 1994.