社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2006－ARC－166 (9)
2006／1／24

Alphaアーキテクチャ用COINSマシン記述の実装とGCCとの比較

塚本智博† 吉瀬謙二† 弓場敏嗣†

Alphaアーキテクチャ用のマシン記述を実装することによって，コンパイラインフラストラクチャとしてのCOINS(COmpiler INfraStructure)の有効性を検証する．具体的には，開発の時間軸に沿って，Alphaアーキテクチャ用のマシン記述の実装の詳細を示し，容易に新しいターゲットアーキテクチャのマシン記述が生成できることを明らかにする．また，COINSコンパイラとGCCで生成したオブジェクトコードの速度比較により，GCCと比較して，平均で25%の速度低下に抑えたコンパイラを構築できることを確認する．

The COINS TMD Implementation for the Alpha Architecture and Its Performance Comparison with GCC

Tomohiro Tsukamoto,† Kenji Kise† and Toshitsugu Yuba†

We verify the effectiveness of the COINS by implementing the machine description for the Alpha Architecture. For this purpose, we demonstrate the detail process of the Alpha TMD implementation in accordance with time. And we show that machine description of the new target architecture is easily obtained. Then, we compare the performance of the COINS with GCC. We conclude that the COINS is acceptable since it is only 25% slower than GCC on average.

1. はじめに

新しいコンパイラを容易に構築し，評価できる共通インフラストラクチャを提供するプロジェクトとしてCOINS(COmpiler INfraStructure)[1]がある．**図1**に，COINS基盤部の構成を示す．これはCOINSホームページの図を基に作成したもので，我々が開発したAlphaアーキテクチャ用のマシン記述(TMD: Target Machine Description)を灰色で示している．COINSの中間表現は，高水準中間表現(HIR: High level Intermediate Representation)と低水準中間表現(LIR: Low level Intermediate Representation)[2]からなる．我々は，入力ファイルとしてC言語のコードを想定している．Cのコードから HIRへと変換され，その後，低水準のLIRに変換される．COINSにおける，それぞれのプロセッサアーキテクチャのコード生成部は，LIRからマシン語への変換規則を記述することで実現される．この記述がTMDである．TMDには，レジスタ構成，LIRとマシン命令の対応関係，関数呼び出しの方式などを記述する．現在のところ，COINSプロジェクトはSPARC, x86, ARM, MIPS, PowerPC, SH4の6種類のTMDを公開している．

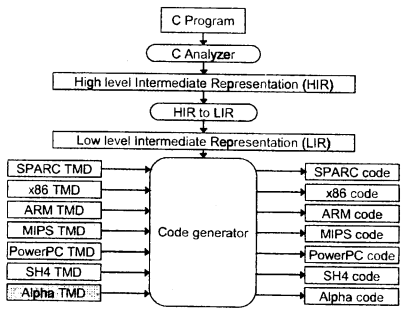

図1 COINS基盤部の構成(COINSホームページの図を基に作成)と新しく追加したAlpha TMD

本研究では，今まで開発されていなかったAlphaアーキテクチャ用のマシン記述を実装することによって，インフラストラクチャとしてのCOINSの有効性を検証する．具体的には，開発の時間軸に沿って，Alphaアーキテクチャ用のマシン記述の実装の詳細を示し，新しいターゲットアーキテクチャのマシン記述が容易に生成できることを明らかにする．また，COINSコンパイラとGCCで生成したオブジェクトコードの速度比較をおこなう．それにより，GCCと比較して，

† 電気通信大学 大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications

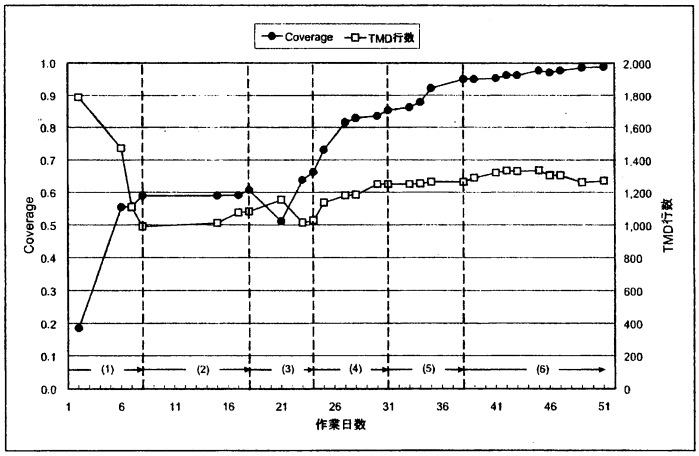

図 2 時間軸に沿った Coverage と TMD 行数の推移

平均で 25%程度の速度低下に抑えたコンパイラを構築できることを確認する．

本稿の構成を示す．2 章では，Alpha アーキテクチャ用 TMD(Alpha TMD と記述) の開発の時間軸に沿った実装の詳細を述べる．3 章では，Alpha アーキテクチャ用の COINS コンパイラと GCC が生成するオブジェクトコードを比較する．4 章では，最適化について議論する．5 章で本稿をまとめる．

2. Alpha アーキテクチャのマシン記述の実装

2.1 開発方針

本研究では，COINS-1.2.1 の 678 個の全テストプログラムに対して，正しい実行結果を得られるプログラムの割合（これを Coverage と定義する）が 95%を超えることを目標として Alpha TMD の実装をおこなった．Alpha アーキテクチャには条件付き移動命令 (CMOV 命令) があるが，記述を簡潔にするために，この命令は利用しないこととした．

本研究の開発には，実装環境と実行環境という 2 つの環境を利用する．実装環境では，TMD の実装作業とテストプログラムの狭義のコンパイルをおこなう．これは，x86 プロセッサを搭載する PC で Java が動作する環境である．実行環境では，COINS コンパイラで生成したアセンブリファイルのアセンブルと，実行テストをおこなう．ここでは，Alpha 21264[3] プロセッサを搭載する計算機を利用する．

TMD の記述に関する概要を付録にまとめる．TMD 記述に関する詳細は，COINS のホームページ[1] を参照のこと．

2.2 Alpha TMD の時間軸に沿った実装

Alpha TMD の実装は，次に示す 6 ステップの開発プロセスでおこなった．

(1) 既存の TMD の単純化
(2) LIR とマシン命令の対応関係の記述
(3) 厳密な関数呼び出しと浮動小数点演算の実装
(4) COINS ソースコードの修正をともなう改良
(5) COINS 開発者による 64 ビット化への対応
(6) Coverage 100%に向けての機能追加

図 2 に Alpha TMD の実装時の時間軸に沿った作業フローを示す．横軸は作業日数（休日を除く），丸が Coverage，四角が TMD 行数を表している．各 Coverage と TMD 行数は，TMD のバージョンを上げた時点の値を示している．括弧は，上で示した各プロセスの作業期間を表わしている．最初のステップの担当者は Alpha アーキテクチャを熟知していたが，それ以降のステップの担当者は Alpha アーキテクチャに対する知識を持っていなかった．いずれの担当者も TMD に対する知識は持っていなかった．各プロセスについて，以下のそれぞれの節で詳細を述べる．

2.2.1 既存の TMD の単純化

最初のステップとして，TMD の記述内容を把握するためと，後の作業をスムーズにおこなうために，既存の TMD をひな形として，四則演算程度の簡単なプログラムをコンパイル可能な単純な Alpha TMD を作成することにした．

既存の TMD として，MIPS アーキテクチャ用の TMD を採用した．MIPS アーキテクチャが，Alpha アーキテクチャに最も近いアーキテクチャと考えたからである．MIPS アーキテクチャ用の TMD から Alpha アーキテクチャに対応しない機能を削除し，四則演算程度の簡単なプログラムをコンパイル可能な Alpha TMD を作成した．プログラムは関数の集合であり，TMD に関数呼び出しの処理を実装しなければコンパイルできない．関数呼び出しの処理はアーキテクチャにより異なるため，LIR にはその処理が埋め込まれていない．つまり，関数呼び出しの処理は，TMD で記述しなければならない．しかし，ABI(Application

Binary Interface) を満たす関数呼び出し処理の実装は複雑な作業となる．そこで，本ステップでは，ABI を考慮しない関数呼び出し処理の実装をおこなった．

本ステップには 8 日を必要とした．これらの多くは，既存の TMD から不要となる部分を削減する処理であり，それほど困難な作業ではない．図 2 から，TMD 行数が 992 行に削減されていることがわかる．また，この作業により，Coverage は 59%に到達した．

2.2.2 LIR とマシン命令の対応関係の記述

論理演算とシフト演算，浮動小数点レジスタに定数を格納する処理と型変換処理の記述をおこなった．ターゲットアーキテクチャに関わらず，入力される LIR はほぼ同一である．既存の TMD を参考に，LIR と Alpha アーキテクチャのマシン命令の対応関係の記述をおこなった．

TMD に慣れるために大半の時間を費やした．このため，本ステップには 10 日を必要とした．これは，50 行程度の追加として実現できたので，それほど困難な作業ではなかった．以降のステップで実装する厳密な関数呼び出しなどが未実装であるため，ここでの実装では，Coverage はほとんど上がっていない．

2.2.3 厳密な関数呼び出しと浮動小数点演算の実装

関数呼び出しの処理はターゲットアーキテクチャにより異なるため，その記述は TMD でおこなう必要がある．既存の TMD と GCC が生成するマシン命令列を参考にして，ABI を考慮した関数呼び出しと浮動小数点演算を実装した．

本ステップには 6 日を必要とした．途中で Coverage が低下しているのは，関数呼び出しの実装にバグがあったためである．その後デバッグをおこない，Coverage は 65%に向上した．浮動小数点演算の実装には多くの行数を必要としていない．加えて，最初のステップで利用した MIPS アーキテクチャ用の TMD における関数呼び出し処理には，多くの行数が使われている．一方，Alpha TMD では単純な記述を心がけたために，関数呼び出しの処理の記述が削減されている．よって，全体として 1,028 行に削減されている．

2.2.4 COINS ソースコードの修正をともなう改良

LIR において一部のアドレスのビット幅が 32 ビットとなっていた．このため，64 ビットアーキテクチャの Alpha に対応させるためには，TMD の記述だけではなく，COINS ソースコードの修正をともなう改良が必要となることが判明した．

調査の結果から，COINS では HIR のレベルでアドレスや大小比較などのテスト命令の演算結果のビット数が決定されていることがわかった．このため，HIR 変換部が参照しているターゲットマシンのアドレスやデータのビット数の定義ファイルを一部変更すること (COINS ソースコードの修正) で 64 ビットへ対応させた．また，これにともない，Alpha TMD の記述を修正した．

本ステップには 7 日を必要とした．今までの作業は TMD の変更により実現していたが，COINS が正式にサポートしていない 64 ビットアーキテクチャの場合には，TMD のみの記述では対応できないという問題点がみつかった．この問題点を解決するために LIR，HIR を参照したり，図 1 の C Analyzer のソースコードを解析することで，その問題が C Analyzer にあることを突き止めた．これらは想定外の作業であり，簡単な作業ではなかった．この作業により，Coverage は 86%に到達した．

2.2.5 COINS 開発者による 64 ビット化への対応

前節の改良のみでは，64 ビットのアーキテクチャに完全に対応することはできなかった．この問題は，LIR のレベルでアドレスのデータ型が 32 ビット整数型であったことから，明らかとなった．原因が不明であったため，COINS の開発者に修正を依頼し，数日の間に，ソースコードが修正された版の COINS を入手することができた．これに対応して，TMD の修正をおこなった．

本ステップには 7 日を必要とした．COINS ソースコードの不都合を見つける必要があったために，この作業も困難なものとなった．この作業により，目標であった Coverage 95%を達成した．ここまでに要した作業日数は 38 日間である．TMD の行数は 1,290 行である．

2.2.6 Coverage 100% に向けての機能追加

Coverage 100% に向けて，ここまでで未実装になっていた機能を実装した．LIR とマシン命令との対応関係の追加，関数呼び出しの実装において戻り値が複合体の場合の関数の実装，浮動小数点命令の修正，レジスタ定義の修正などである．また，COINS を version 1.2.1 から 1.3.1 へと変更した．ただし，先に述べた 64 ビットアーキテクチャのための，COINS のソースプログラムの修正は必要だった．

本ステップには 13 日を必要とした．ただし，目標を達成しているので他の作業と並行して TMD の実装を継続した．最終的に，51 日間の作業 (工数は 2 人月) で Coverage 99% を達成した．また，TMD の行数は 1,271 行である．幾つかのテストプログラムは正しくコンパイルできていない．これは，TMD の記述の問題，COINS の 64 ビット対応の問題などが原因と考えられるが，詳細は調査中である．

3. Alpha アーキテクチャ用の COINS コンパイラの評価

3.1 マシン記述の定量的な比較

COINS-1.3.1 で公開されている既存の TMD と Alpha TMD の記述行数と工数を**表 1** にまとめる．表の 2 列目に，TMD の記述行数を示す．TMD には，Java

表 1　マシン記述の定量的な比較（COINS ホームページの表に基づき Alpha のデータを追加）

機種名	TMD の記述行数		工数
	総行数	Java 行数	
SPARC	1,949	802	-
x86	2,391	985	-
ARM	3,052	2,176	6 人月
MIPS	2,081	1,379	3 人月
SH4	3,568	2,467	6 人月
PowerPC	5,016	2,913	6 人月
Alpha	1,271	736	2 人月

で記述された部分とそれ以外の部分がある．3 列目に Java で記述されたコードの行数，4 列目に工数を示す．

簡単な変換規則では対応できない場合は，Java による動作の記述が可能となっている．Java による記述では，関数呼び出しの処理，TMD の内部で利用するメソッド，アセンブラへのマクロ命令の生成メソッドなどを定義する．

Alpha TMD の記述量は 1,271 行である．これは COINS プロジェクトで公開されている TMD と比べて最小である．この理由として，Alpha TMD では，Java 以外の記述において，同じような形のものを何回も書かなくてすむように積極的にマクロを利用していることが挙げられる．

SPARC と x86 は，COINS 開発と並行して実装されているため，工数は不明となっている．Alpha TMD の工数は，他の TMD と比べて短い．ただし，64 ビット化の問題解決のために余計な工数が必要となっており，これを除けば，40 日人程度で記述できると考えられる．TMD を短期間で記述できた原因として，参考にできる TMD が多いこと，Alpha アーキテクチャが簡潔であることなどを考えている．今回は，64 ビット化の実装の作業が多かった．つまり，32 ビットアーキテクチャであれば，さらに短期間で実装することは可能である．また，ターゲットマシンに精通している開発者であれば，さらに短期間での実装も可能であると考えている．

3.2 COINS と GCC との性能比較

本節では，Alpha アーキテクチャ用の COINS コンパイラの性能評価として，GCC との実行時間の比較をおこなう．そして，機能レベルのプロセッサシミュレータ SimCore[4] を用いて，命令コードの実行回数を計測し，実行性能への影響について議論する．

3.2.1 実験環境

実験での比較対象は，COINS version 1.3.1 と GCC version 3.3.2 である．実行時間の計測は，AlphaStation DS10 (Alpha21264 プロセッサ 600MHz, 256MByte メモリ) 上でおこなった．命令コードの実行回数は，SimCore 上でシミュレーションして計測した．また，SimCore を改変し，それぞれの命令についての実行回数を計測できるようにした．これにより，

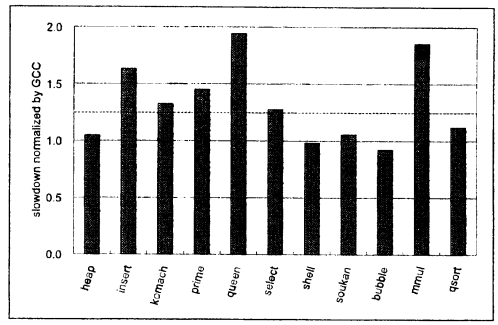

図 3　実行時間の GCC に対する相対比

プログラムにおけるコア計算部分が容易に分かる．ベンチマークプログラムは COINS, GCC ともに最適化オプション O2 を用いてコンパイルした．

ベンチマークには，COINS のテストプログラムより，ヒープソート (heap)，挿入ソート (insert)，小町算 (komach)，素数算出 (prime)，クイーン (queen)，選択ソート (select)，シェルソート (shell)，相関係数 (soukan) の 8 個と，独自に用意したバブルソート (bubble)，行列積 (mmul)，クイックソート (qsort) の 3 個，全 11 個を用いた．

3.2.2 オブジェクトコードの実行時間の比較

GCC と COINS において，それぞれについて 3 回ずつ実行し，その実行時間の算術平均を求めた．**図 3** に，GCC を 1 としたベンチマーク毎の実行時間の相対比を示す．全てのベンチマークの調和平均値 1.25 を図の破線で示す．

図 3 の結果より，最も速度が低下したベンチマークが queen で，この時，94%の性能低下となることがわかる．一方，COINS が GCC に優っているベンチマークは bubble と shell の 2 つで，bubble が最も高い 7%の性能向上を達成する．全てのベンチマークの調和平均では，GCC に対して，COINS は 25%程度の速度低下に抑えることができている．

次節では，図 3 において特に速度低下が著しい insert, queen, mmul の速度低下の原因を，命令コードの実行回数を用いて検討する．

3.2.3 速度低下の原因

図 4 に，insert における COINS と GCC で生成された命令コードの実行回数の分布を示す．Load Address はロードアドレス命令，INT は整数値に対する命令，FLT は浮動小数点に対する命令であることを表している．mem はメモリアクセス，load/store はロード／ストア命令，op は操作命令，Branch は分岐命令である．Others には予約語などが含まれる．

配列アクセスにおいて，COINS ではインデックスの計算に算術命令を使っている．一方，GCC ではロードアドレス命令を使っている．このため，図 4 において，COINS では Load Address の実行回数が少なく，

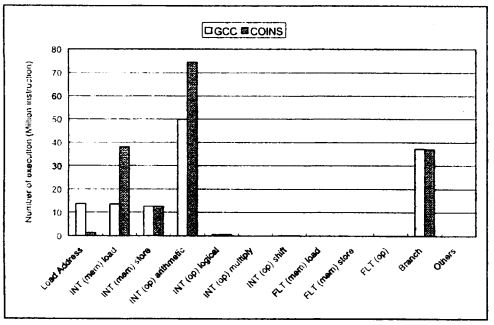

図 4　insert における命令コードの実行回数の分布

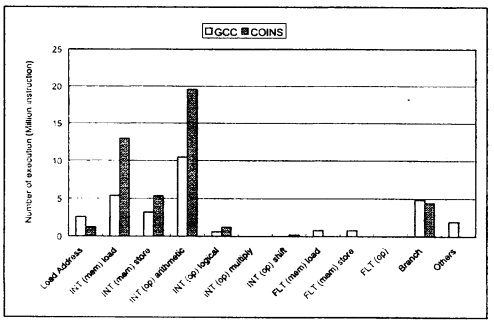

図 5　queen における命令コードの実行回数の分布

GCC では，arithmetic の実行回数が少ない．

COINS は GCC の 2.8 倍のロード命令を実行している．SimCore が出力したログとオブジェクトコードをダンプして得られた出力から，コアループの 1 イタレーションで同じ大域配列へ 2 回のメモリアクセスがおこなわれていることがわかった．一般に，メモリアクセスは，実行コストの高い命令である．COINS では，メモリアクセスの度に大域変数のベースアドレスのロード（これはメモリへアクセスする）をおこなっている．一方，GCC では，最初のアクセス時にロードしたベースアドレスを，2 回目のアクセス時にも使っている．つまり，コアループ内に COINS は 4 回，GCC は 2 回のメモリアクセスがあり，この差が速度低下に影響していると考えられる．また，COINS の総実行命令数は，GCC の約 1.3 倍と多いことも，COINS の速度低下の一因となっている．

図 5 に，queen における COINS と GCC での命令コードの実行命令数の分布を示す．総実行命令数については，COINS は GCC の約 1.5 倍であった．queen においても，insert と同様の傾向が得られた．ただし，queen ソースコードで浮動小数点演算が使われていないにもかかわらず，GCC では浮動小数点命令が実行されている．ログを調査したところ，COINS ではレジスタをメモリへ退避しているところを，GCC では浮動小数点レジスタへ移動していることがわかった．

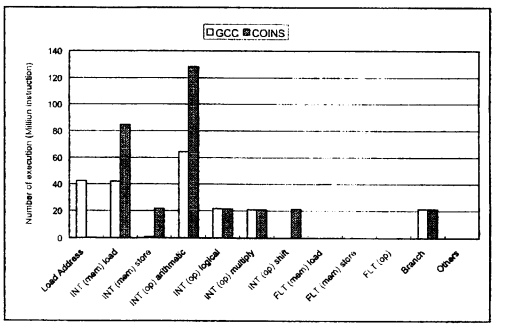

図 6　mmul における命令コードの実行回数の分布

これに伴い，Others に分類されている浮動小数点レジスタと汎用レジスタ間の移動命令が実行されていた．これに伴い，GCC では，多くの汎用レジスタが利用可能となり高速化につながっている可能性がある．

図 6 に，mmul における COINS と GCC での命令コードの実行回数の分布を示す．COINS の総実行命令数は，GCC の約 1.5 倍であった．命令コードの実行回数の分布は，ロードアドレスとメモリのロード命令，算術演算については insert と queen の傾向と同様の傾向が得られた．しかし，メモリストア命令について，COINS は GCC の 26 倍の実行をおこなっていた．COINS では，コアループにおいて，乗算結果をメモリにストアしていることが影響していた．mmul ソースコードでは大域配列に対して行列積の演算が記述されていたことから，大域変数のレジスタ割付が適切になされていないことがわかった．この差が速度低下に影響していると考えられる．

4. 最適化についての議論

本研究で実装した Alpha TMD では，短期間で実用に耐えうる性能のコンパイラを実装することを目的としている．このため，十分な最適化が施されているという訳ではない．本節では，今後の課題として検討すべき最適化を議論する．

COINS では大域的な変数に対するレジスタ割付において不必要なロード命令とストア命令を生成することがある．レジスタ割付の方式を改善することで，ロード命令とストア命令の削減が可能となり，より高性能なコンパイラを構築できる可能性がある．COINS がインフラストラクチャということを考えると，大域的な変数に対するレジスタ割付は COINS の機能として実装されていることが望ましい．

開発方針で示したように，Alpha TMD の実装では，条件付き移動命令 CMOV を利用していない．これらの命令を適切に利用することで条件分岐命令を削減し，性能を向上できることが期待される．CMOV 命令の利用は今後の課題である．

整数系のプログラムでは，浮動小数点レジスタを利用しないものが多い．この様な場合にも整数レジスタの待避先として浮動小数点レジスタを利用することで性能が向上する可能性がある．浮動小数点レジスタを利用した最適化は今後の課題である．

GCC が生成したアセンブリコードを調査した結果，GCC では多くの場合，実行コストの高い命令を 8 バイト境界に配置していることがわかった．一方，Alpha TMD では，8 バイト境界を意識したコード生成をおこなっていない．これらの実装は今後の課題である．

5. おわりに

本研究では，Alpha アーキテクチャ用のマシン記述を実装することによって，インフラストラクチャとしての COINS の有効性を検証した．

開発の時間軸に沿って，Alpha アーキテクチャ用のマシン記述の実装の詳細を示し，51 日間という短期間で，新しいターゲットアーキテクチャのマシン記述が生成できることを示した．COINS を 64 ビットに対応させるという作業を除いて考えると，さらに短期間で，容易に新しいアーキテクチャ用のコンパイラを構築できると考えられる．

COINS コンパイラと GCC で生成したオブジェクトコードの速度比較により，GCC と比較して，平均で 25%の速度低下に抑えたコンパイラを構築できることを確認した．現時点では，COINS コンパイラは GCC より低速という結果が得られているが，本稿では幾つかの未実装の最適化に関して議論した．これらの実装により，COINS は GCC を超える性能を達成する可能性がある．

機能レベルのプロセッサシミュレータ SimCore は，オブジェクトコードの解析に非常に有用であり，コンパイラの最適化ツールとして，有効であった．

付　　録

A.1 Alpha TMD の記述

TMD は，ターゲットマシンの特性を記述するものである．TMD には，(1) データ型とレジスタの定義，(2)LIR とマシン命令との対応関係の記述，(3) 複雑な機能の Java による記述という 3 つの項目を記述する．

A.1.1 データ型とレジスタの定義

最初にアドレスを表現するデータ型と，大小比較などのテスト命令の演算結果のデータ型を定義する．次に，Alpha アーキテクチャが持つ実レジスタ（汎用レジスタ 32 個と浮動小数点レジスタ 32 個）を定義する．このレジスタの中から，レジスタ割付に使うレジスタを定義する．整数 64 ビット変数に割り付けるレジスタは，次のように記述した．I64 は，64 ビット整数型を表わしている．

```
(def *reg-I64* (
  (foreach @n (0 1 2 3 4 5 6 7 8 9 10 11
   12 13 14 16 17 18 19 20 21 22 23 24 28)
  (REG I64 "%@n"))))
 (defregsetvar (I64 *reg-I64*))
 (defregset regq *reg-I64*)
```

ここでは，*reg-I64*という名前のレジスタ集合を定義している．defregsetvar 定義により，整数 64 ビットの変数が*reg-I64*レジスタ集合から割り当てられる．フレームポインタなどの特別な機能を持つレジスタ 5 個は，含めていない．defregset 構文で，次に述べる文法規則の中で使用されるレジスタを表す非終端記号 regq を定義した．

A.1.2 LIR とマシン命令との対応関係の記述

LIR とマシン命令の対応関係は文脈自由文法の形で記述する．例えば，$z = x \mid y$ というソースコードは，**or $16, $17, $0** のようなマシン命令に変換される．この場合の処理は，LIR では次の様に表される．

```
(SET I64 (REG I64 "z.3%_1")
 (BOR I64 (REG I64 "x.1%_1")
          (REG I64 "y.2%_1")))
```

これに対して利用される 2 つの変換規則を次に示す．

```
(defrule regq (BOR I64 regq regq)
 (code (or (_r $1) (_r $2) (_r $0)))
 (cost 1))
(defrule regq (REG I64))
```

defrule の第 1 引数はこの還元規則の還元先である．第 2 引数には LIR 式を記述する．入力 LIR 式が第 2 引数が表す LIR 式とマッチしたとき，code 属性が定義されていれば記述されたマシン命令が出力される．これらの変換規則を利用して，**or $16, $17, $0** という命令が生成される．

謝　　辞

64 ビット化の作業において貴重なご意見を頂きました COINS プロジェクトの皆様，迅速なコード修正をして頂きました渡邊坦先生，鈴木貢先生に感謝いたします．また，本研究をおこなうに際して貴重なご意見を頂いた中西悠氏に感謝いたします．

参　考　文　献

1) A compiler infrastructure project: http://www.coins-project.org/.
2) Abe, S., Hagiya, M. and Nakata, I.: A Retargetable Code Generator for the Generic Intermediate Language in COINS, *IPSJ Transactions of Programming*, No.44, pp.12–29 (2005).
3) R. E. Kessler: The Alpha 21264 Microprocessor, *IEEE Micro*, Vol. 19, No. 2, pp. 25–36 (1999).
4) 吉瀬謙二, 片桐孝洋, 本多弘樹, 弓場敏嗣: SimCore/Alpha Functional Simulator の設計と実装, 電子情報通信学会論文誌, Vol. J88-D-I, No. 2, pp. 143–154 (2005).