一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
VLD2015-104,CPSY2015-136,RECONF2015-86 (2016-01)

レガシーマイグレーションにおける メインフレームアセンブリのC言語への変換

藤原　大輔† 石浦菜岐佐† 酒井　峻† 青木　領†† 小河原隆史††

† 関西学院大学 理工学部　〒669–1337 兵庫県三田市学園 2-1
†† 株式会社システムズ　〒141–0031 東京都品川区西五反田 7-24-5

あらまし　本稿では, レガシーマイグレーションにおいて, メインフレームコンピュータのアセンブリプログラムを C 言語へ自動変換する手法を提案する. レガシーマイグレーションでは, メインフレーム上で動作する業務システムを Linux 等のオープンシステムに移行するが, アセンブリプログラムの変換処理は人手で行われており, 膨大な時間と工数を要する. 本稿で提案する手法は, COBOL 等のサブルーチンとして呼び出される IBM メインフレームのアセンブリを対象に, これを正しく動作し, かつ可読性の高い C プログラムに変換する. 本手法では, アセンブリを SSA 形式の中間表現に変換し, その中間表現上でデータフロー解析, 制御構造の再構築, パターンマッチングによる変換を行うことにより, 可読性の高い C プログラムを生成する. さらに, 本手法では, アーキテクチャ依存コードや自己書き換えコードの変換が完全に行えなかった場合の修正作業を容易化するため, 元アセンブリと変換結果の C プログラムの対応をドキュメントとして生成する. 提案手法に基づくツールを Perl で実装し, 複数のメインフレームアセンブリを正しく動作する C プログラムに変換できることを確認した.
キーワード　レガシーマイグレーション, メインフレームアセンブリ

Mainframe Assembly to C translation in Legacy Migration

Daisuke FUJIWARA†, Nagisa ISHIURA†, Ryo SAKAI†, Ryo AOKI††, and Takashi OGAWARA††

† Kwansei Gakuin University,　2–1 Gakuen, Sanda, Hyogo, 669–1337, Japan
†† SYSTEM'S Co., Ltd.　7–24–5 Nishigotanda, Shinagawa-ku, Tokyo, 141–0031 Japan

Abstract　This article presents a method of translating mainframe assembly programs to C programs. In "legacy migration," where enterprise systems running on mainframe computers are ported to open systems based on Linux, etc., porting of assembly programs needs manual translation and takes enourmous amount of man-hours. In our method, IBM mainframe assembly programs, which are called as subroutines from high-level languages such as COBOL, are automatically translated into C programs which produce the same results and yet have good readability. Assembly programs are converted into intermediate representation with the SSA form on which dataflow analysis, recovering of control structures, and pattern match based transformations are applied to produce highly readable codes. Along with translation, correspondence between source assembly codes and resulting C codes are also generated as documents, which plays an important role in manually correcting incomplete C codes from architecture dependent codes or self morphing codes. A prototype system based on our method successfully translated some assembly codes into C program with function, if, and do-while structures.
Key words　Legacy Migration, Mainframe Assembly

1. はじめに

メインフレームコンピュータは, 高い信頼性や耐障害性を有しており, 多くの企業の基幹システムに採用されている. しかし, 近年, Linux や Windows 等の低価格なオープンシステムの高性能化に伴い, メインフレーム上で稼働するシステム (レガシーシステム) をこれらのオープンシステムに移行する動きが出ている. レガシーシステムには長年渡って業務のノウハウが蓄積されていることが多いため, 新たにシステムを構築し直すよりも, 既存のレガシーシステムをオープンシステムに移植するメリットの方が大きい場合がある. このような移植はレガシーマイングレーション (legacy migration) と呼ばれ, 大規模

なシステムを如何に効率よく移植するかが重要な技術課題となっている.

レガシーマイグレーションでは, COBOL, PL/1, アセンブリ等で記述されたメインフレーム用のプログラムを, 新しいシステムで動作するように変換することが一つの中核技術となる. 高級言語によるプログラムであれば, 再コンパイルで動作するものもあれば, 自動変換ツールも種々開発されている [1]. また, コードが理解しやすいため, 移植に際して必要となる修正も比較的容易に行える. これに対し, アセンブリはそのまま動作させることができないため, 動作を理解した上で人手で高級言語に変換する必要があるが, コードの理解にはアーキテクチャの詳細に関する知識と熟練が必要になる. 移行対象となるプログラムは 1 システムにつき数百から数千本に及ぶため, この変換には膨大な工数が必要となる.

この課題を解決するため, アセンブリコードから高級言語への変換の自動化が試みられている. アセンブリコードのマイグレーションにおいては, 変換したプログラムが正しく動作するということに加え, 変換後のプログラムのデバッグや保守のために可読性が高いことが必要となる.

文献 [5] [6] [7] [8] では, IBM メインフレームのアセンブリを C プログラムに変換する手法を提案している. アセンブリを中間表現に変換し, 中間表現上で冗長なコードの削除や制御構造の抽出, 関数のインターフェースの作成, ローカル変数の型情報の解析等を行い, 可読性の高い C プログラムを生成している.

しかし, 変換対象のアセンブリにはアーキテクチャに強く依存したコードや自己を書き換えるようなコードが含まれていることがあり, 生成された C プログラムは必ずしも正しく動作するとは限らない. このような場合には, 変換されたコードを人手で修正しなければならないが, この際に, アセンブリのソースコードと生成された C プログラムの対応が取れることが非常に重要になる.

そこで本稿では, IBM アセンブリの C 言語への自動変換を行うとともに, 変換過程のドキュメント化を行う手法を提案する. 本手法では, 文献 [5] [6] [7] [8] と同様に, アセンブリを中間表現に変換し, その表現上で関数, if 文, ループ文の再構成を行って, 構造化された C プログラムを出力する. このとき, 変換過程が確認できるよう, 変換元プログラムと生成された C プログラムの対応関係を示す表を同時に生成する.

提案手法に基づくツールを Perl5.18.4 で実装し, いくつかのアセンブリプログラムについて変換を行った結果, 正しく動作する C プログラムを生成することができた.

以下, 本稿では 2 章でアセンブリのマイグレーションについて要点を整理し, 3 章でアセンブリの C 言語への変換について, 4 章でドキュメント化の手法について述べる. 5 章で実装について述べた後, 6 章でまとめと今後の課題を述べる.

2. アセンブリのマイグレーション

2.1 変換の対象

本稿では, 人手で書かれた IBM 370〜390 メインフレームのアセンブリプログラムを C 言語プログラムに変換する問題を扱う. 対象アセンブリは, 図 1 に示すように, COBOL 等で書かれた他のプログラムからサブルーチンとして呼び出されるものを想定する. アセンブリが呼び出すライブラリやマクロに対しては, 新しいシステム上で動作するライブラリが別途準備されているものとする.

このような変換においてまず重要になるのは, 変換結果のプログラムが正しく動作することであり, バイナリトランスレーション [2] と同様の技術が必要になる. また, 変換結果のプログラムを保守する必要性から, プログラムの可読性が高いことも重要になる. これには逆コンパイル [3] [4] と同様の技術が必要になる.

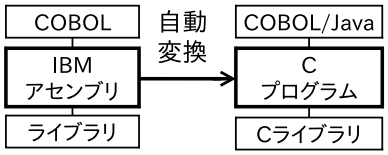


図 1　変換の対象

メインフレームアセンブリを高級言語に変換するツールはいくつか開発されている [5] [6] [7] [8]. このうち, 文献 [5] では HLL, 文献 [6] [7] [8] では WSL と呼ばれる中間言語とそれに対する変換により, IBM アセンブリから可読性の高い C プログラムを生成している.

しかし, アセンブリプログラムには, エンディアンや使用するアドレス空間のマップ, アドレス指定時のビットパターン (サブルーチンコールにおいて最後のパラメタのアドレスの最上位ビットを 1 にセットする) 等, アーキテクチャ (場合によっては特定のモデルのマイクロアーキテクチャ) に依存したコーディングが行われることがある. さらに, 命令の一部または全てを書き換える “self morphing code” や, 手書きアセンブラ特有のコーディング技法が用いられることがあるため, 変換の完全な自動化は必ずしも可能ではない. 変換が失敗した場合には, その原因を突き止めて人手で修正を行う必要があるが, この際には, 元のアセンブリと変換結果の対応関係を把握できることが非常に重要となる.

2.2 IBM アセンブリ

IBM メインフレームは 32bit アーキテクチャであり, 0〜15 と番号付けられた 16 本のレジスタを持つ. IBM アセンブリが扱えるデータ型には 32bit (full word), 16bit (half word), 8bit (character), パック形式 10 進数, ゾーン形式 10 進数等がある. アドレッシングには 24bit モードと 31bit モードがある. 命令は 0〜3 オペランドを持ち, 2 進数, 10 進数, 文字列等に対する命令を含め, 合計 631 個の命令がある.

IBM アセンブリの例を図 2 に示す. このコードは, 呼び出し側から 2 つのパラメタを受け取り, 1 つ目のパラメタで受け取った `YYYYMMDD` 形式の日付データの月の末日を `YYYYMMDD` 形式で 2 つ目のパラメタに保存するものである.

3. メインフレームアセンブリの C 言語への変換

3.1 概　　要

本稿の変換処理の流れを図 3 に示す. アセンブリプログラムを構文解析して各命令をアトミックな演算に分解し, SSA 形式 (静的単一代入形式) の中間表現 (IR) に変換する. これに対してデータフロー解析, 制御構造の抽出, パターンマッチングによる変換等の可読性向上処理を行い, 中間表現から C 言語を生成する.

3.2 中 間 表 現

本稿で扱う中間表現の構造を 図 4 に示す. 変換対象のアセンブリ全体を表す Assembly は 1 つの SymbolTable と 1 つ以上の Section を保持する. SymbolTable はアセンブリの持つ全ての変数, 関数, ラベルの情報を保持する. Section はアセンブリの

```
 1:  EMONTH    CSECT                                  00000000
 2:            USING EMONTH,12                        00000100
 3:            STM   14,12,12(13)                     00000200
 4:            LA    12,0(,15)                        00000300
 5:            LA    15,SAVE                          00000400
 6:            ST    13,4(,15)                        00000500
 7:            ST    15,8(,13)                        00000600
 8:            LR    13,15                            00000700
 9:            LM    3,4,0(1)                         00000800
10:            CLC   0(6,4),=C'00000000'              00001100
11:            BNE   ER_FMT                           00001200
12:            B     FC_EOM                           00001300
13:  ER_FMT    EQU   *                                00001400
14:            ABEND 0999,DUMP                        00001500
15:  FC_EOM    BAL   14,EOM                           00001600
16:  EOM       EQU   *                                00001700
17:            MVC   0(8,4),=CL9' '                   00001800
18:            LA    0,6                              00001900
19:            LR    1,3                              00002000
20:  EOM1      EQU   *                                00002100
21:            CLI   0(1),C'0'                        00002200
22:            BL    RET                              00002300
23:            CLI   0(1),C'9'                        00002400
24:            BH    RET                              00002500
25:            LA    1,1(,1)                          00002600
26:            BCT   0,EOM1                           00002700
27:            CLC   4(2,3),=C'01'                    00003000
28:            BL    RET                              00003100
29:            CLC   4(2,3),=C'12'                    00003200
30:            BH    RET                              00003300
31:            PACK  WORK,4(2,3)                      00003400
32:            CVB   2,WORK                           00003500
33:            BCTR  2,0                              00003600
34:            MH    2,=H'4'                          00003700
35:            LA    5,MLAST                          00003800
36:            AR    2,5                              00003900
37:            LH    0,4(,2)                          00004000
38:            SH    0,0(,2)                          00004100
39:            CVD   0,WORK                           00004200
40:            UNPK  6(2,4),WORK                      00004300
41:            OI    7(4),X'F0'                       00004400
42:            MVC   0(6,4),0(3)                      00004500
43:            B     RET                              00004600
44:  RET       EQU   *                                00004700
45:            L     13,4(,13)                        00004800
46:            LM    14,12,12(13)                     00004900
47:            SLR   15,15                            00005000
48:            BR    14                               00005100
49:  MLAST     DC    H'000',H'000'                    00005200
50:            DC    H'031',H'031'                    00005300
51:            DC    H'059',H'060'                    00005400
52:            DC    H'090',H'091'                    00005500
53:            DC    H'120',H'121'                    00005600
54:            DC    H'151',H'152'                    00005700
55:            DC    H'181',H'182'                    00005800
56:            DC    H'212',H'213'                    00005900
57:            DC    H'243',H'244'                    00006000
58:            DC    H'273',H'274'                    00006100
59:            DC    H'304',H'305'                    00006200
60:            DC    H'334',H'335'                    00006300
61:            DC    H'365',H'366'                    00006400
62:            LTORG                                  00006500
63:  SAVE      DS    18F                              00006600
64:  WORK      DS    D                                00006700
65:            DROP  12                               00006800
66:            LTORG                                  00006900
67:            SPACE                                  00007000
68:            END                                    00007100
```

図 2　IBM メインフレームアセンブリ

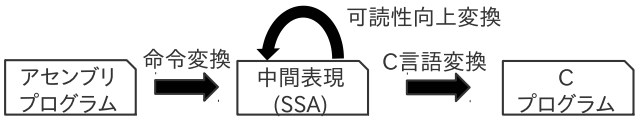


図 3　変換処理の流れ

セクションを表し, Function のリストを持つ. Function は関数 (サブルーチン) を表し, BasicBlock, If, Loop のリストを持つ. If は if 文を表し, 条件, then 部, else 部を持つ. then 部と else 部は BasicBlock, If, Loop のリストを持つ. Loop は do-while 文を表し, 条件と body 部を持ち, body 部は BasicBlock, If, Loop のリストを持つ. BasicBlock は基本ブロックを表し, Operation のリストと次の BasicBlock へのリンクを持つ. Operation は演算 (算術・論理演算, メモリのロード・ストア, 文字列や 10 進数データに対する演算等) を表すものであり, 複数のオペランドに対して演算を適用した結果を変数に格納する.

命令の Operation への変換例を表 1 に示す. ロード命令 `L` は, アドレスを計算する演算 (`addu32`) とメモリを読む演算 (`load32`) に変換する. `AP` 命令等のパック形式 10 進数や `CLC` 命令等の文字列に対する処理は, アトミックな演算として扱う.

命令変換の結果得られる Operation のリストは SSA 形式に変換する. SSA 形式を用いることにより, 意味は同じだが表現が異なるコードの中間表現が標準化される. また, SSA 変換は 3.3 節, 3.6 節で述べる不要コードの削除やパターンマッチング

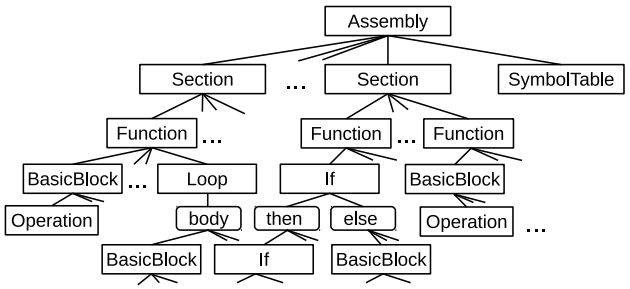


図 4　中間表現のデータ構造

表 1　命令から中間表現への変換例

アセンブリ	中間表現 (IR)
`L 11,4(,13)`	`addr = addu32(r13, 4)` `r11 = load32(addr)`
`AP DAT1,DAT2+5(3)`	`cc=addpack(DAT1, 8, DAT2, 8)`
`CLC 4(2,4),=C'01'`	`addr=addu32(r4, 4)` `cc=compstr(addr, C'01', 2)`

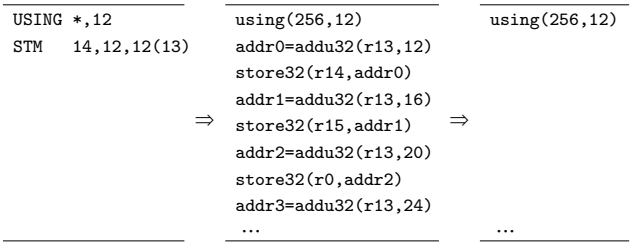


図 5　退避コードの削除

処理を効率的に行う上でも有用である. この後, 最後にデータフロー解析を行って, 変数の定義/参照関係をデータ構造として保持する.

3.3　レジスタ退避/復帰コードと不要コードの削除

データフロー解析の結果に基づき, 高級言語では不要になるレジスタの退避/復帰コードの削除を行う. 次の条件に該当する演算を退避コード, 復帰コードと判定し, 削除する.

- 退避コード

（1） 演算が `store32` (32bit データのメモリ書き込み)
（2） 第 1 オペランドが以降の他の演算で使用されていない
（3） 第 2 オペランドがレジスタ 13 に 0〜68 オフセットを加算したものである.

- 復帰コード

（1） 演算が `load32` (32bit データのメモリ読み込み)
（2） 演算の結果を格納する変数がレジスタ
（3） 第 1 オペランドがレジスタ 13 に 0〜68 オフセットを加算したものである.

退避コードの削除例を図 5 に示す. `STM` (store multiple) 命令は, 全レジスタを退避するためのアドレス計算とストアを行うための中間表現に一旦変換するが, これらは退避コードの判定条件に該当するため, 削除する.

また, アセンブリ中には, 保守の過程で使用されなくなったコードが含まれていることが多い. 本手法では入口点からの可達性解析により, これらのコードに対する中間表現を削除する.

3.4　制御構造の抽出

中間表現に対し, 関数, if 文, do-while 文の制御構造を抽出し, 構造化を行う.

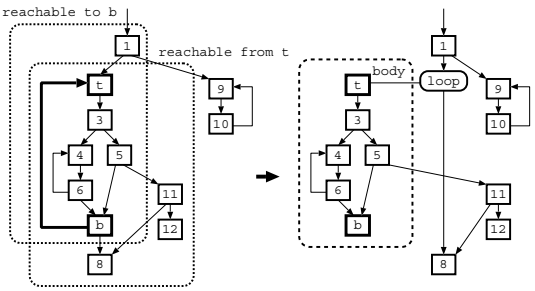


図 6　ループの本体の抽出

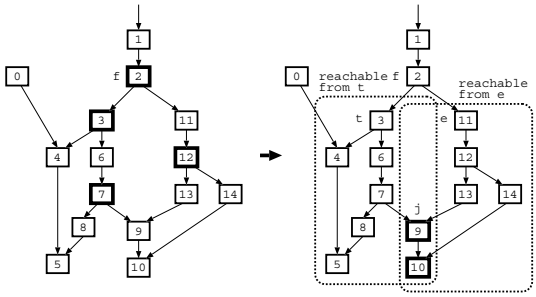


図 7　分岐点と合流点の選択

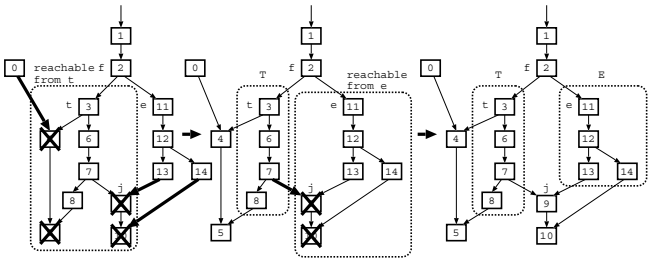


図 8　if 文の then 部と else 部の抽出

表 2　中間表現から C への変換例

中間表現 (IR)	C
r10 = add32(r10, r7)	r10 = r10 + r7
cc = addpack(DAT1, 8, DAT2, 8)	cc = addpack(DAT1, 8, DAT2, 8)
cc = compstr(addr, C'01', 2)	cc = compstr((char*) addr, "01", 2)

3.4.1　関数の抽出

関数の抽出は, 関数の入口からリターンまでの到達可能な全ての基本ブロックを列挙することによって行う. まず, プログラム全体の入口, および BAL, BALR, BAS 等の関数呼び出しの命令のターゲットを関数入口の集合として列挙する. 次に, 各関数の入口から到達可能な基本ブロックをその関数名でマークする. マークされた基本ブロックをその関数の持つ基本ブロック集合とする. ただし, 基本ブロックが複数の関数名でマークされている場合は, その基本ブロックのクローンを作成する.

3.4.2　do-while 文の抽出

do-while 文の認識は次の手順で行う.

（1） 認識の起点 (最初は関数の入口) から基本ブロックをたどり, ループを列挙する.

（2） 処理対象とするループを 1 つ決定する. ループが複数あれば, ループのボトム b が起点から最も遠いものを選び, さらにボトムが b であるループが複数ある場合は, トップ t が起点に最も近いものを選ぶ. これは, 外側のループから先に認識し, ループへの飛び込みが発生するのを避けるためである.

（3） ループ本体を構成する基本ブロックの集合 B を求める. B はトップ t から到達可能かつボトム b へ到達可能な基本ブロックを列挙したものである (図 6 参照).

（4） ボトム b の分岐条件からループの継続条件を抽出し, ループのデータ構造を完成させる.

（5） ループの本体及び残った部分に対して同様の処理を行う.

3.4.3　if 文の抽出

if 文の認識は次の手順で行う.

（1） 処理対象とする if の分岐点 f と合流点 j を求める. 認識の起点 (最初は関数の入口) から基本ブロックをたどり, 起点に最も近い基本ブロックを 1 つ選んで f とする. f の 2 つの分岐先 t, e の両方から到達可能な基本ブロックの集合の中で, f に最も近いものを合流点 j とする (図 7 参照). これは, if 文の then 部, else 部への飛び込みをなくすためである. ただし, 集合が空集合の場合は j は存在しないものとする.

（2） if の then 側に含まれる基本ブロックの集合 T, else 側に含まれる基本ブロックの集合 E を求める. まず t から到達可能な基本ブロックの集合 T' を求める. 次に, T' 以外の基本ブロックから直接到達可能な基本ブロック x と, x から到達可能な基本ブロックを T' から全て削除したものを T とする. E も同様にして求める (図 8 参照).

（3） if の条件部を抽出し, データ構造を完成させる.

（4） T, E, および残った部分に対して同様の処理を行う.

3.5　C 言語への変換と DSECT の扱い

中間表現は, 基本的にほぼそのまま C プログラムに変換できる. レジスタ及びアセンブリ中の変数は C 言語の変数とし, 四則・論理演算やメモリのロード・ストア等の演算は, そのまま C の対応する演算に変換する. また, 文字列や 10 進数に対する演算は, 標準ライブラリまたは別途作成したライブラリの呼び出しに変換する. 表 3.5 に変換の一例を示す. 中間表現中の 10 進演算 (addpack) や文字列演算 (compstr) は, 対応するライブラリの呼び出しに変換する.

ただし, 演算単位で C の文にすると中間変数が多く現れて可読性が低下するため, 可能な限り複数の演算をまとめて一つの文にする. この処理の例を図 9 に示す. C1 のような演算系列は, ツリー構造を式として出力する. ただし, C2 のように同じ中間結果が複数回使用されている場合には, ツリーを分割して出力する.

IBM アセンブリのダミーセクション (DSECT) は, 機械命令も領域も生成しないセクションであり, サブルーチン間で受け渡す大きなデータ領域内のデータ配置を記述するために使われる. 本稿では DSECT によるデータの配置を構造体で表し, USING 命令で定義されるベースレジスタをこの構造体へのポインタとして用いてアクセスする C コードを生成する. DSECT の変換例を図 10 に示す. 変数 A と B を持つ DAT1 というダミーセクションが宣言され, CSECT 直後に現れる USING によって レジスタ 2 を DAT1 に対するベースレジスタとしている. DAT1 は DAT1_t という構造体型に変換し, サブルーチン MAIN を変換した関数 main 内では, r2 を DAT1_t へのポインタとして用いることによりそのメンバにアクセスできるようにする.

3.6　パターンマッチングによる可読性向上

本手法では, パターンマッチングに基づく中間表現の変換により, さらなる可読性の向上を図る. これは SSA 変換された中間表現中の部分木を書き換えるルールを準備することにより実現する. 例えば, 図 11(a) の ASM のような文字列比較による分岐は, 中間表現 IR を経て C のように条件 cc を含む C プ

C1

```
r1 = mul32(r1, X)
r1 = add32(r1, Y)
r1 = mul32(r1, Z)
```

⇒ C1’

```
r1 = ((r1 * X) + Y) * Z;
```

C2

```
r1 = mul32(r1, X)
r2 = load32(r1)
r1 = add32(r1, Y)
r1 = mul32(r1, Z)
r2 = add32(r2, T)
```

⇒ C2’

```
r2 = r1 * X;
r1 = (r2 + Y) * z;
r2 = r2 + T
```

図 9　複数演算の式への変換

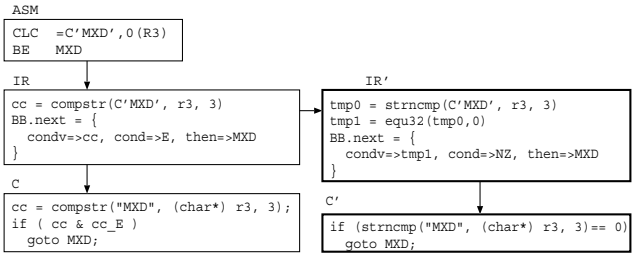
```
DAT1 DSECT
A DS F
B DS 4C
 …
MAIN CSECT
     USING DAT1,2
     MVI   A,123
     MVC   B,=C’abcd’
 …
```

⇒

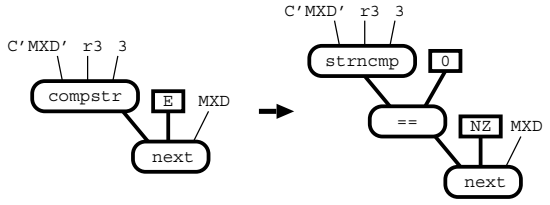
```
typedef struct {
  int A;
  char B[4];
} DAT1_t;

int main(void){
  ((DAT1_t*) r2)->A = 123;
  strncpy( ((DAT1_t*) r2)->B,
                 "abcd", 4);
 …
```

図 10　DSECT 命令の変換



(a) パターンマッチングによる変換



(b) 変換ルール

図 11　パターンマッチングによる可読性向上

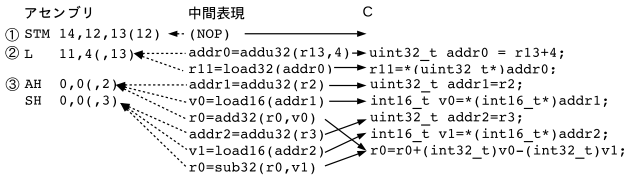
ログラムに変換される．これに対し, 図 11(b) のような中間表現の部分木に対する変換ルールを適用することによって, IR は IR’ に変換され, C’ のような C プログラムが生成される．

4. 変換過程のドキュメント化

生成された C プログラムに修正が必要になった場合には, アセンブリのどの箇所がどのように変換されたかという情報が非常に重要になる．本稿では, アセンブリから C プログラムを生成するとともに, 元ソースコードと生成された C プログラムの対応関係を表の形で生成する手法を提案する．

この目的のために, 中間表現の各演算に対し変換元のアセンブリの命令を記憶しておく (図 12(a) の破線矢印)．アセンブリ命令に対する中間表現がない場合には, 図 12(a)① のように NOP 演算を生成してその演算へのリンクを設定する．

対応表の生成は, 中間表現から C への変換を行う際に, 関連



(a) 対応関係の構築

	アセンブリ	C 言語	備考
①	STM 14,12,12(13)		
②	L 11,4(,13)	uint32_t addr0 = r13+4; r11=*(uint32_t*)addr0;	

③	AH 0,0(,2)
	SH 0,0(,3)
		uint32_t addr1=r2;
		int16_t v0=*(int16_t*)addr1;
		uint32_t addr2=r3;
		int16_t v1=*(int16_t*)addr2;
		r0=r0+(int32_t)v0-(int32_t)v1;
			演算を集約

(b) 対応表の例 (代入文)

アセンブリ	C 言語	備考
CLC =’MXD’,0(R3) BE L1	if (strncmp("MXD", (char*)r3, 3) == 0) {	
L1 SR 1,2	r1 = r1 - r2; }	

L2 AR 1,2
B BB1
	else {
	r1 = r1 + r2;
	}	

BB1 …	BB1: ; …	

(c) 対応表の例 (if 文)

図 12　アセンブリと C 言語の対応の生成

付けられた元プログラムの演算を参照することによって行う．複数の中間表現が同じアセンブリの演算へのリンクを持っている場合 (図 12(a)②参照), その中間表現が全て C に変換された時点でアセンブリと C との対応関係を表のエントリとする (図 12(b)②)．また, 複数の中間表現を集約して C に変換した場合 (図 12(a)③参照), 集約された中間表現とリンクしている全てのアセンブリの演算と C との対応関係を表のエントリとする (図 12(b)③)．

制御構造を再構築した場合の処理も同様である．図 12(c) に if 文を再構築した場合の対応表の例を示す．条件部の演算をまとめ, then 部 と else 部それぞれに対し中間表現からアセンブリの演算を参照することにより, 対応表を作成する．do-while 文の場合も同様の処理を行う．

5. 実　　装

提案手法に基づくシステムを Perl 5.18.4 で実装した．動作環境は Ubuntu 14.04LTS, Windows Cygwin である．

図 2 のアセンブリプログラムを変換した結果を 図 13(a) に示す．アセンブリの 16〜48 行目のサブルーチン EOM が 30〜67 行目の関数に変換されている．また, 34〜42 行目の do-while 文, 36〜45 行目, 48〜70 行目, 82〜89 行目の if-else 文のように, 構造化されたコードが出力されている．54 行目では複数の演算をまとめており, パターンマッチングによって 36, 38, 44, 46 行目などの文字列比較も変換されている．

このプログラムを図 13(b) に示すメインプログラムとリンクして実行した結果, 正しく動作することが確認できた．

本稿で変換した C プログラムには, 幾つかの制限がある．IBM メインフレームは 32bit マシンであり, アドレスの格納領域が 4 バイト前提で確保されているため, 変換した C プログラムも 32bit 環境でしか正しく動作しない．また, メインフレームでは EBCDIC コードを採用しているのに対し, 一般的なオープンシ

ステムでは ASCII を採用しているため, 文字コード依存の処理は完全には変換できない. また, EX (execute) 命令のように命令の一部を修正するものや, MVC (move character) 命令を用いて他の命令の一部または全部を書き換える処理は変換できない.

6. むすび

本稿では, IBM アセンブリの C 言語への変換の自動化を行い, その変換過程のドキュメント化を行う手法を提案した. SSA 形式の中間表現に可読性向上処理を適用することにより, いくつかのアセンブリプログラムを正しく動作する可読性の高い C プログラムに変換することができた. また, アセンブリと生成された C プログラムとの対応関係を示す表を生成し, 変換過程をドキュメント化することにより, アセンブリがどのように変換されたのか確認することが容易になる.

今後の課題としては, 元プログラムと生成された C プログラムとの対応表の生成の実装や, 実行時に命令の意味を修正する EX 命令や MVC 命令によるコードの自己書き換えへの対応, 64bit 環境への対応などが挙げられる.

謝　　辞

本稿の研究に関して多くの御助言をいただきました株式会社システムズの岡本健二氏に感謝いたします. また, 本研究に関してご協力, ご討議頂いた関西学院大学石浦研究室の諸氏に感謝いたします. 本研究の一部は中小企業庁平成 25 年度補正中小企業・小規模事業者ものづくり・商業・サービス革新事業による.

文　　献

[1] 小河原隆史: “情報処理装置、情報処理方法、およびプログラム,” 特開 2014–215938 (Nov. 2014).

[2] C. Cifuentes, M. Van Emmerik, D. Ung, D. Simon, and T. Waddington: “Preliminary Experiences with the Use of the UQBT Binary Translation Framework,” in *Proc. Workshop on Binary Translation*, pp.12–22 (Oct. 1999).

[3] M. Van Emmerik: *Static Single Assignment for Decompilation,* PhD Thesis, University of Queensland (2007).

[4] G. Chen, Z. Wang, R. Zhang, K. Zhou, S. Huang, K. Ni, Z. Qi, K. Chen, and H. Guan: “A Refined Decompiler to Generate C Code with High Readability,” in *Proc. Working Conference on Reverse Engineering (WCRE)*, pp.150–154 (Oct. 2010).

[5] Y. A. Feldman:“Portability by Automatic Translation: A Large-Scale Case Study,” in *Proc. Knowledge-Based Software Engineering Conference*, pp.123–130 (Nov. 1995).

[6] M. P. Ward:“Assembler to C Migration Using the FermaT Transformation System,” in *Proc. IEEE International Conference on Software Maintenance 1999 (ICSM ’ 99)*, pp. 67–76 (Aug.–Sept. 1999).

[7] M. P. Ward:“Reverse Engineering from Assembler to Formal Specification via Program Transformations,” in *Proc. Working Conference on Reverse Engineering*, pp. 11–20 (Nov. 2000).

[8] Martin Ward:“Assembler Restructuring in FermaT,” in *Proc. IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2013)*, pp. 147–156 (Sept. 2013).

```
  1: #include <stdio.h>
  2: #include <stdlib.h>
  3: #include <string.h>
  4: #include <stdint.h>
  5: #include "miglib.h"
  6:
  7: static uint32_t r0, r1, r2, r3, r4, r5, r6, r7;
  8: static uint32_t r8, r9, r10, r11, r12, r13, r14, r15;
  9: static int cc;
 10: static char savearea[72];
 11:
 12: int16_t MLAST[] = {
 13:   0,  0,
 14:   31,  31,
 15:   59,  60,
 16:   90,  91,
 17:   120,  121,
 18:   151,  152,
 19:   181,  182,
 20:   212,  213,
 21:   243,  244,
 22:   273,  274,
 23:   304,  305,
 24:   334,  335,
 25:   365,  366
 26: };
 27: uint32_t SAVE[18];
 28: uint64_t WORK;
 29:
 30: void EOMfunc(void){
 31:   strncpy( (char*) r4,  "          ", 8 );
 32:   r0 = 6;
 33:   r1 =  r3;
 34:   do {
 35:     EOM1: ;
 36:       if ( strncmp( (char*) r1, "0", 1 ) < 0 ) { goto RET; }
 37:       else {
 38:         BB1: ;
 39:           if ( strncmp( (char*) r1, "9", 1 ) > 0 ) { goto RET; }
 40:           else {
 41:             BB2: ;
 42:             r1 =  r1 +  1;
 43:             r0 =  r0 -  1;
 44:           }
 45:       }
 46:   } while ( r0 );
 47: BB3: ;
 48:   if ( strncmp( (char*) r3+4, "01", 2 ) < 0 ) { goto RET; }
 49:   else {
 50:     BB4: ;
 51:       if ( strncmp( (char*) r3+4, "12", 2 ) > 0 ) { goto RET; }
 52:       else {
 53:         BB5: ;
 54:           zone_to_pack( (char*) &WORK, 8, (char*) r3+4, 2 );
 55:           r2 = pack_to_int32( (char*) &WORK, 8) - 1;
 56:         BB6: ;
 57:           r2 = (r2 * 4) + (uint32_t) MLAST;
 58:           uint32_t addr27 = r2 + 4;
 59:           r0 = * (uint16_t* ) addr27;
 60:           int16_t tmpvar29 = * (int16_t*) r2;
 61:           r0 = r0 - (int32_t) tmpvar29;
 62:           int32_to_pack( (char*) &WORK, 8,  r0);
 63:           pack_to_zone( (char*) r4+6, 2, (char*) &WORK, 8 );
 64:           uint32_t addr31 = r4 + 7;
 65:           unsigned char tmp32 = * (unsigned char *) addr31;
 66:           unsigned char tmp33 = tmp32 & 0x0f;
 67:           unsigned char tmp34 = tmp33 | 0x30;
 68:           * (unsigned char*) addr31 = tmp34;
 69:           strncpy( (char*) r4, (char*) r3, 6 );
 70:       }
 71:   }
 72: RET: ;
 73:   r15 =  r15 -  r15;
 74:   return;
 75: }
 76: int32_t EMONTH( void** param ){
 77:   r1 = (uint32_t) param;
 78:   r13 = (uint32_t) savearea;
 79:   r14 = (uint32_t) &&RETURN;
 80: EMONTHbb: ;
 81:   r12 =  r15;
 82:   r15 = (uint32_t) SAVE;
 83:   uint32_t addr15 =  r15 +  4;
 84:   * (uint32_t*) addr15 = r13;
 85:   uint32_t addr16 =  r13 +  8;
 86:   * (uint32_t*) addr16 = r15;
 87:   r13 =  r15;
 88:   r3 = * (uint32_t* ) r1;
 89:   r4 = * (uint32_t* ) (r1 + 4);
 90:   if ( strncmp((char*) r4, "00000000", 6 ) ) {
 91:     ER_FMT: ;
 92:       abend( 999 );
 93:   }
 94:   else {
 95:     FC_EOM: ;
 96:       EOMfunc();
 97:   }
 98: RETURN: ;
 99:   return r15;
100: }
```

(a) 図 2 のアセンブリから生成された C プログラム

```
 1: #include <stdio.h>
 2: #include <stdint.h>
 3: #include <string.h>
 4:
 5: int32_t EMONTH(void **param);
 6:
 7: int main(void)
 8: {
 9:   char gdate[] = "20141215";
10:   char result[] = "00000000";
11:   void* param[2] = {gdate, result};
12:   int32_t rc = EMONTH(param);
13:
14:   printf("rc = %d, result = \"%s\"\n", rc, result);
15:   return 0;
16: }
```

(b) テスト用プログラム

図 13　生成された C プログラム