Vol.2015-EMB-38 No.2
2015/8/28

変則的な命令セットに対応した命令デコーダの自動生成手法

奥田 勝己[1,a)]　竹山 治彦[1]

概要：命令セットシミュレータ（ISS : Instruction Set Simulator）は，組込みシステムの仮想開発環境の構築に不可欠なソフトウェアである．組込みシステムの開発では，新規アーキテクチャの CPU や専用プロセッサを含む多種多様なプロセッサが採用される．このため，組込みシステムの開発に ISS を適用するためには，ISS の構築を効率化する自動生成技術が必要である．ISS の命令デコーダを自動生成する手法は従来から存在するが，変則的な命令セットに対応した手法は確立されておらず，自動生成可能な命令セットは限定的である．そこで，本論文では変則的な命令セットに対応した命令デコーダ生成手法を提案する．一般に命令デコーダ生成手法は，デコードエントリの集合を分割しながら再帰的にデコードツリーを生成する．提案手法では，集合の分割時にオペコード情報のみでなく，除外条件を用いることで変則的な命令セットに対応することを特徴とする．ARM と MIPS64 を対象として提案手法を適用した結果，変則的な命令セットに対しても効率的な命令デコーダを生成可能であることを確認した．

キーワード：命令セットシミュレータ，ISS, エミュレータ，エミュレーション

1. はじめに

近年，組込みソフトウェアの開発環境として，ISS を組み込んだ仮想環境の適用が進んでいる．ISS は，ホスト計算機上でターゲットプロセッサを模擬するソフトウェアである．組込みシステムの開発に ISS を適用することで，実機の開発とソフトウェアの開発を並行に進めることができるため，開発期間を短縮することができる．また，ISS 上では，割込みタイミングの調整や故障注入など，実機では困難な操作を容易に行うことができる．このため，ソフトウェア開発者は ISS を組み込んだ仮想環境上で効率的にソフトウェアを検証することが可能である．

しかし，組込みシステムの開発に仮想環境を適用するためには，ISS 構築のためのコストと期間が問題となる．組込み向けプロセッサの種類は豊富であるため，製品開発では機種ごとに異なるプロセッサを採用する場合がある．また，製品によっては，新しいアーキテクチャのプロセッサや専用プロセッサも適用される．さらには，開発の途中で採用プロセッサが変更となる場合もある．このため，組込みシステムの開発に ISS を適用するためには，機種開発ごとに異なる種類の ISS を低コストかつ短期間で準備することが要求される．したがって，ISS を効率的に開発可能な自動生成技術が重要となる．

命令デコーダは，自動生成技術の適用効果が大きい ISS 構成要素の 1 つである．命令デコーダは，命令語のビット列を入力とし，命令の識別と命令フィールドの抽出を行う．命令の識別や命令フィールドの抽出には，複雑なビット操作を伴うため，命令デコーダのハンドコーディングには多大な労力を要する．

命令デコード手順は，決定木の一種であるデコードツリーを用いて表される．命令デコーダの自動生成では，命令のフォーマット情報を入力として命令デコードツリーを出力する．一般にデコードツリーの生成手法は，オペコードパターンを用いる手法とコストモデルを用いる手法とに大別される．オペコードパターンを用いる手法では，ツリーの深さとメモリ消費量の点で効率の良いツリーを生成することができるが，変則的な命令セットに対応することがでないという問題があった．一方，コストモデルを用いる手法の場合，変則的な命令セットにも対応できるが，同じ命令が複数のサブツリーに繰返し現れるため，デコードツリーに必要なメモリサイズが大きいという問題があった．そこで，本論文では，変則的な命令セットに対してもメモリ効率の良いデコードツリーを生成可能な手法を提案する．

本研究で対象とする変則的な命令セットとは，オペコードのみでは命令の識別ができない命令セットである．オペコードのみでは識別できない命令を区別するのは，オ

[1] 三菱電機株式会社
MELCO, Amagasaki, Hyogo 661–8661, Japan
[a)] Okuda.Katsumi@eb.MitsubishiElectric.co.jp

ペコード以外の命令フィールドの値である．特定の命令フィールドが特定のパターンを持つか否かで，それぞれ異なる命令が割り当てられる．このような命令エンコードは，短い命令長で多くの命令に対応するためや，既存命令セットに対して後から拡張命令を追加するために行われる．たとえば，ARM[1] の場合，32 ビット命令の上位 4 ビットすべてが 1 か否かで同じオペコードに対して，異なる命令が割り当てられる．

提案手法は，デコードツリー生成時におけるデコーディングエントリ集合の分割時に，デコードエントリに含まれる除外条件を用いることで，デコードツリー生成処理を継続できるように従来のオペコードパターンを用いる手法を拡張したものである．また，拡張した手法を用いて ARM と MIPS64 を対象に命令デコードツリーを生成した結果，効率の良いデコードツリーを生成することを確認した．

以下本論文では，まず 2 章で関連研究について示す．続いて，3 章では命令デコーダの定式化と従来手法における課題を示す．4 章では提案手法による変則的な命令セットへの対応のための拡張を示す．5 章では，本手法の有用性の確認を目的とした実験について示す．

2. 関連研究

ISS の自動生成に関する既存研究では，プロセッサ記述言語から ISS を生成する手法が提案されている．ISS を生成可能なプロセッサ記述言語としては，LISA[2]，ISDL[3]，Sim-nML[4]，EXPRESSION[5]，ASIP Meister[6]，HARMLESS[7] が存在する．これらの研究では命令のフォーマット情報をプロセッサ記述言語で記述することができるが，命令デコーダ自動生成の詳細については言及していない．本研究では，上記のプロセッサ記述言語や類似の記述言語による命令フォーマット情報と提案手法とを組合せて使用することを想定している．

命令デコーダを対象とした自動生成手法に関する既存研究としては，文献 [8], [9], [10], [11], [12], [13] などが存在する．これらはいずれも，ISS やディスアセンブラに適用することを想定した命令デコーダを対象としている．以下では命令デコーダの生成に関する既存研究と本研究を比較する．

文献 [8] や [11] が生成対象とする命令デコーダは，入力ビット列に対応する命令を線形探索することで，命令を識別する．一方，提案手法で生成する命令デコーダは，ツリー構造を用いてビット列と対応する命令を探索するため，上記研究による命令デコーダよりも効率が良い．生成対象の命令デコーダをディスアセンブラや静的バイナリトランスレータに用いる場合には，命令デコード効率は問題とならない．しかし，実行時に命令デコードを必要とする ISS では，命令デコード時間がシミュレーション時間全体に影響を与える．このため，本研究が対象とする ISS では，命令デコード効率が重要となる．

本研究と同じくデコードツリーを生成対象としている研究としては，文献 [9], [10], [12], [13] などがある．このうち提案手法と類似の手法は，文献 [9], [12] に記載されており，高さおよびメモリ消費量ともに優れたデコードツリーを生成することができる．しかし，これらの従来手法では変則的な命令セットに対応することができない．提案手法は，文献 [9] の手法を変則的な命令セットに対応可能とするような拡張と位置づけることができる．提案手法では，変則的でない命令セットを入力とする場合，[9] と同じデコードツリーを得る．また，変則的な命令セットを入力とする場合でも変則的でない命令セットを入力とする場合と同程度のメモリ効率のツリーを得ることができる．

文献 [10]，[13] の手法では，デコードツリーのコストモデルを用いて命令デコードツリーを生成する．これらの手法では，同じ命令が複数のサブツリーに繰返し現れるため，デコードツリーのための消費メモリが大きい．文献 [10] では，メモリコストもモデル化しているが，文献 [10] の Table.1 と Figure.4 から，本研究がベースとする文献 [9] 相当の高さ（1.59）のデコードツリー ($\gamma = 1/16$) を得るためには，二分木相当のデコードツリー ($\gamma <= 16$) を生成した場合の数十倍以上のメモリが必要であることが読み取れる．一方，本研究での実験結果では提案手法で生成するツリーは，二分木ツリー相当のメモリ消費である．コストモデルを用いた手法のうち本研究と同様に変則的な命令セットに対応しているのは，[13] の手法である．[13] の手法では，[10] をベースとするコストモデルに命令の出現頻度を加えたモデルを使用してデコードツリーを分割する．一方，本研究では，命令の出現頻度など動的に得られる情報を用いない．新規プロセッサを対象として ISS を開発する場合，事前に命令の出現頻度を取得することはできないため，本研究では出現頻度を用いない方針とした．

3. 変則的な命令セットへの対応における課題

本章では，命令デコーダを定式化し，提案手法のベースとなる従来手法を示した後，変則的な命令セットとその課題を順に示す．

3.1 命令デコーダ

はじめに，n ビット長のビットパターンを $p = \{0, 1, -\}^n$ として定義する．ここで，0 と 1 はそれぞれ論理値の 0 と 1 を表し，$-$ は不定値を表すものとする．集合 $B = \{0, 1\}$ について，ビット列 $s = B^n$ がパターン p とマッチすることは，式 (1) が成り立つことと同値である．

$$\forall i \in \{0, 1, ..., n-1\} : \ s[i] = p[i] \ \vee \ p[i] = - \qquad (1)$$

命令	フォーマット								オペコードパターン
	7	6	5	4	3	2	1	0	
A	0	0	a		b		c		00------
B	0	1	a		b		c		01------
C	1	0	a		b		0	0	10----00
D	1	0	a		b		0	1	10----01

図 1: 命令セットの例

以下本論文では，s が p とマッチすることを $s \in p$ と記す．たとえば，ビット列 0010 はパターン 001- とマッチし，これを $0010 \in 001-$ と記す．

パターン p_o を長さ n のオペコードパターンとし，命令ごとの一意なラベルを $l \in L$ とするとき，各命令は対 (p_o, l) で与えられるデコードエントリ d を持つ．ここでデコードエントリ全体の集合を D と定義する．たとえば，図 1 の命令セットに対応するデコードエントリは，$(00------, A)$，$(01------, B)$，$(10----00, C)$，$(10----01, D)$ である．

命令デコーダは，入力ビット列 s を入力とし，入力ビット列 s とマッチするオペコードパターンを持つエントリ $d \in D$ を出力する．すなわち，命令デコーダは，関数 $B^n \rightarrow D$ である．

3.2 デコードツリー

命令デコーダ $B^n \rightarrow D$ は，$s = B^n$ と対応するデコードエントリ d を探索する処理として実現される．デコードエントリの探索には，線形探索を適用することもできるがデコードエントリの総数 $|D|$ に比例して時間がかかるため効率が悪い．本研究で対象とする命令デコーダは，ツリーを用いてデコードエントリを探索する．

命令デコーダが用いるツリーは，決定木の一種であり，対 (V, E) として定義される．ここで，集合 $V = N \cup L$ であり，N をノードの集合，L をリーフの集合とする．また，E は辺の集合である．ノードは決定関数 $f_v : B^n \rightarrow N$ をラベルとして持つ．一方，リーフはデコードエントリ $d \in D$ をラベルとして持つ．

命令デコーダは，デコードツリーのルートに対して決定関数を適用し，次に参照すべきノードを決定する．次に参照すべきノードがリーフの場合，命令デコーダは，リーフのラベルを結果として返す．一方，次回ノードがノードの場合，命令デコーダはノードラベルの決定関数を入力ビット列に適用し，リーフに到達するまで再帰的に上記の処理を繰り返す．

デコードツリーの例として，図 1 の命令セットに対応するデコードツリーを図 2 に示す．図 2 では，デコードエントリ A，B，C，D は，リーフとして表現される．図 1 の各辺は，ビットを検査するためのビットパターンをラベルに持つ．各ノードの決定関数は，入力ビットパターンに一致するビットパターンをラベルとする辺を選択することで，次に参照すべき子ノードを決定する．たとえば，ビット列

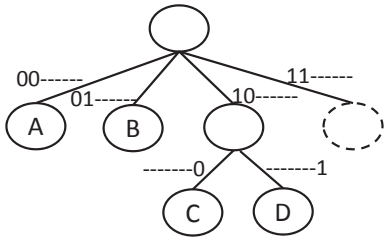

図 2: デコードツリーの例

```
1: function MakeTree(D)
2:     if |D| = 1 then
3:         return MakeLeaf(D)
4:     else
5:         (result, node) ← MakeNode(D)
6:         if result ≠ FAILED then
7:             return node
8:
9: function MakeNode(D)
10:    P ← MakePatterns(D)
11:    if P ≠ ∅ then
12:        return (FAILED, nil)
13:    node ← CreateNode()
14:    for all p ∈ P do
15:        D_m ← MakeMatchingEntries(D, p)
16:        child ← MakeTree(D_m)
17:        node.AddChild(p, child)
18:    return (OK, node)
```

図 3: 従来のデコードツリー生成手順

10000000 を入力とする場合，命令デコーダは，ルートの決定関数を用いて次の子ノードを決定する．ルートの決定関数はビット列の上位 2 ビットを検査し，パターン 10------ をラベルとして持つ辺と結線された左から 3 番目の子ノードを選択する．次に，選択した子ノードの決定関数によって下位 1 ビットを検査し，パターン -------0 を選択する．この結果，命令デコーダは，命令 C のデコードエントリをラベルとして持つリーフに到達する．

各ノードにおける決定関数は，テーブルルックアップとして実装可能である．このため，各ノードの決定関数の計算量は定数オーダである．したがって，命令デコードツリーを用いた命令デコーダの処理効率は，ツリーの高さに依存する．

3.3 従来手法

本節では，本研究のベースとなる従来手法について示す．記法は異なるが本質的には文献 [9] の手法と同様である．従来手法では，デコードエントリの集合 D を重複のないサブ集合 $D_1, D_2, ..., D_n (D_1 \cap D_2 \cap ... \cap D_n = \emptyset)$ に分割し，各サブ集合に対して再帰的にツリー生成手順を適用することで，デコードツリー全体を生成する．

図 3 に従来のデコードツリー生成手順を示す．1-7 行目の関数 *MakeTree* は，入力として与えられたデコードエントリの集合 D に対応したデコードツリーを出力する．

2-3 行目は，入力デコードエントリ集合 D の要素数が 1 の場合に対応した再帰の底である．再帰の底では，D の唯一の集合要素をラベルとするリーフを作成して返す．一方，入力デコードエントリ集合 D の要素数が 1 以上の場合，4-7 行目で後述の関数 *MakeNode* にてノードを作成する．D のエントリ間でオペコードパターンの区別が付かない場合，関数 *MakeNode* は失敗 (*FAILED*) を返す．失敗でない場合，関数 *MakeTree* は作成したノードを返却する．

9-18 行目の関数 *MakeNode* の定義である．関数 *MakeNode* は，デコードエントリ集合 D を入力とし，ノードを作成して出力する．関数 *MakeNode* は，デコードエントリ集合を重複なく分割するためのパターンの集合 P を作成し，各パターン $p \in P$ とマッチするデコードエントリの部分集合に対して再帰的にツリー生成手順を適用する．

10 行目では，デコードエントリ集合 D から式 (2) に示すパターンの集合 P を作成する．

$$P = \{p \mid p \notin \{-\}^n, p[i] \in \begin{cases} \{0,1\} & significant(i) \\ \{-\} & (otherwise) \end{cases}\} \quad (2)$$

ここで，関数 $significant(i)$ は，式 (3) の真偽値を返すものとする．

$$\begin{gathered} \forall (p,l) \in D : p[i] \neq - \\ \wedge \\ \exists (p,l) \in D : p[i] = 0 \\ \wedge \\ \exists (p,l) \in D : p[i] = 1 \end{gathered} \quad (3)$$

すなわち，関数 *significant* は，すべてのエントリのパターン p について i ビット目が - ではなく，i ビット目がすべて同じでない場合にのみ真を返す．関数 $significant(i)$ が真の場合，i 番目のビットはデコードエントリ集合を分割するのに有効であることを意味する．集合 P はデコードエントリ集合の分割に有効なビットがとり得る値のすべての組合せビットパターンである．例えば，$D = \{(0000, X), (0\text{--}1, Y), (1\text{--}0, Z)\}$ の場合，関数 *significant* が真となるのは，$i = 0, i = 3$ の場合である．したがって，0 ビット目と 3 ビット目がとり得るすべての組合せビットパターンの集合 P は $\{0\text{--}0, 0\text{--}1, 1\text{--}0, 1\text{--}1\}$ となる．

任意のデコードエントリ d は，集合 P の要素のいずれか 1 つと必ずマッチする．式 (3) を満たすビットが存在しない場合，集合 P は空集合となる．11-12 行目は集合 P が空集合の場合であり，関数 *MakeNode* は処理失敗を返却する．

13 行目では，関数 *MakeNode* は，返却するノードを作成し，以降の処理で当該ノードに対して子を追加することでツリーを作成する．14-17 行目のループでは，パターン集合 P の要素 p を巡回し，パターン p と一致する集合 D_m に対してツリーを作成する．15 行目の関数 $MakeMatchingEntries$ は，入力としてデコードエントリ集合 D とパターン p を入力とし，パターン p に一致するエントリのみからなる新たなデコードエントリ集合 $D_m = \{(p_o, l) \in D \mid match(p_o, p)\}$ を作成する．ここで，関数 $match$ の定義は，式 (4) のとおりである．

$$\begin{aligned} match(p_x, p_y) = \\ \forall i \in \{0, 1, ..., n-1\} : \\ p_x[i] = p_y[i] \ \vee \ p_x[i] = - \ \vee \ p_y[i] = - \end{aligned} \quad (4)$$

16-17 行目では，部分集合 D_m を対象に再帰的にツリーを生成し，13 行目で作成したノードの子ノードに加える．なお，式 (3) を用いて作成したパターン集合 P では，全エントリが同じパターン p とマッチすることはなく，D_m は必ず D の真部分集合となる．このため，再帰を繰り返すと D_m はいずれ要素数が 1 となり，再帰の底に到達する．関数 *MakeNode* はすべてのパターンを巡回し，ノードの作成が完了すると 18 行目で作成したノードとともに成功を返す．

上記の手順を図 1 の命令セットに対して適用すると，図 2 のデコードツリーが得られる．

3.4 変則的な命令セット

本節では，本研究で生成対象とする変則的な命令セットと従来の生成手法における課題を示す．

既存の命令セットに対して新たな命令を追加する場合や，短い命令長で多数の命令に対応する場合，プロセッサの設計者は，命令のオペコードパターンを再利用して別の命令を定義する．具体的には，命令 I のオペコード以外の命令フィールド f が特定のビット列 s をとりえない場合，命令 I と同じオペコードで命令フィールド f をビット列 s とする新たな命令群 $I'_1, I'_2, ...$ を定義する．

例として，図 1 の命令セットに対して命令 E，F，G を拡張した変則的な命令セットを図 4 に示す．命令 A は，命令フィールド a の値が 00 の場合，命令フィールド c の値として 00 と 11 以外をとらないものとする．図 4 の命令セットでは，これを利用して，別の命令 E, F を定義している．命令 E のオペコードパターン 0000--00 と命令 F のオペコードパターン 0000--01 は，命令 A のオペコード 00------ を一部として含む．同様に，図 1 の命令セットでは，命令 A と命令 B のオペコードフィールド a は値として 11 をとらないものとする．これを利用して定義された命令が G である．命令 G のオペコード 0-11---- は，命令 A のオペー

命令	フォーマット								オペコードパターン	除外条件	
	7	6	5	4	3	2	1	0		一致条件	不一致条件
A	0	0	a		b		c		00------	a=00 a=11	c≠00 && c≠11
B	0	1	a		b		c		01------	a=11	
C	1	0	a		b		0	0	10----00		
D	1	0	a		b		0	1	10----01		
E	0	0	0	0	b		0	1	0000--01		
F	0	0	0	0	b		1	0	0000--10		
G	0	f	1	1	b		c		0-11----		

図 4: 変則的な命令セットの例

ド 00------ と命令 B のオペコード 01------ を拡張したものである．

既存命令のオペコードを利用して命令を追加すると命令セットは変則的になる．命令セットは，式 (5) を満たす場合に変則的である．

$$\exists (p_d, l_d) \in D,\ (p_e, l_e) \in D\ (l_d \neq l_e)\ :\ match(p_d, p_e) \tag{5}$$

式 (5) は，あるビット列 $b \in B$ に対して複数のデコードエントリのオペコードパターンが一致する場合があることを意味する．たとえば，図 4 の命令セットの場合，ビット列 01110000 は，命令 B と命令 G それぞれのオペコードパターンとマッチする．

変則的な命令セットを対象とする場合，命令デコーダは，オペコードパターンのみでは命令を識別できない．このため，命令デコーダは，命令の除外条件も含めてビット列の命令を識別する必要がある．これに対応し，変則的な命令セットのデコードエントリを，パターン p とラベル l に除外条件の集合 C_e を加えた 3 つ組 (p, l, C_e) として拡張する．あるビット列に対してオペコードパターンが一致していても除外条件を満たす場合には，ビット列はデコードエントリの命令ではないことを意味する．除外条件の集合 C_e の要素は，一致パターン p_m と不一致パターン集合 P_u の対 (p_m, P_u) である．たとえば，図 4 の命令セットでは，命令 A のデコードエントリの除外条件集合は，$\{(\text{--00----}, \{(\text{------00}), (\text{------01})\}), (\text{--11----}, \emptyset)\}$ となる．除外条件集合では，一致パターン a=11 は一致パターン --00---- で表され，その他の条件も同様にビットパターンで表される．

デコードエントリ d が除外条件集合 C_e を含む場合，ビット列 b がオペコードパターンと一致していても，式 (6) で表される除外条件が成立する場合，ビット列 b は当該エントリの命令ではない．

$$\bigvee_{(p_m, P_u) \in C_e} (b \in p_m \wedge \bigwedge_{p_u \in P_u} b \notin p_u) \tag{6}$$

従来の手法で，変則的な命令セットを対象としてデコード生成手順を適用した場合，式 (3) を満足するビットが存在せず，デコードエントリを分割するためのパターン集合 P が空集合となって処理を継続できなくなる場合がある．たとえば，図 4 に対するツリーを生成では，はじめにデコードエントリの集合を $\{A, B, E, F, G\}$ と $\{C, D\}$ に分割し，それぞれに対して再帰的にツリー生成手順を適用する．2 回目に選択するときには，式 (3) を満たすビットが存在しないため，パターン集合 P が空集合となり，処理を継続できない．

4. 提案手法

4.1 方針

提案手法によるデコードツリー生成の基本的な方針は従来と同じである．オペコードパターン群から作成したビットパターン集合を用いてデコードエントリ集合を分割し，分割したデコードエントリ集合に対して再帰的にツリー生成手順を適用する．従来の手法では，命令セットが変則的な場合，デコードエントリ集合を分割するためのパターン集合 P が空集合となる場合に処理を継続できないことが問題である．そこで，提案手法では，従来手法でデコードエントリ集合を分割するためのパターン集合を作成できない場合でも処理を継続できるようにツリー生成手順を拡張する．

提案手法では，従来手法でデコードエントリ集合を分割できない場合に，デコードエントリの除外条件を用いてデコードエントリ集合 D を D_m と D_o に分割し，D_m と D_o に対して再度ツリー生成アルゴリズムを適用できるようにする．また，除外条件でデコードエントリ集合を分割するときには，D_m と D_o の 2 つにのみ振り分けられれば良いため，デコードツリーに特殊なノードとして，条件ノードを導入する．条件ノードは，ビットパターンが条件を満たすか否かで子ノードのいずれかを返す決定関数をラベルとする．

4.2 ツリー生成手順

提案手法のツリー生成手順を図 5 に示す．関数 *MakeTree* は，デコードエントリ集合 D を入力とし，デコードツリーを出力する．図 3 の従来手法との相違は，8-9 行目に通常のノード作成に失敗した場合に条件ノードの作成を加えている点である．5 行目の関数 *MakeNode* の手順は，従来と同じであるが，パターンに一致しているデコードエントリを返す関数 *MakeMatchingEntries* は異なる処理を行う．提案手法の関数 *MakeMatchingEntries* はオペコードパターンだけでなく除外条件も用いてデコードエントリを選択する．関数 *MakeMatchingEntries* の詳細については後述する．

10-16 行目は条件ノード作成関数 *MakeConditionNode* の定義である．関数 *MakeCondtionNode* は，11 行目の関数 *SelectPattern* で除外条件集合の中から一致条件のパターン p_m を 1 つ選択する．12 行目では一致エントリ集合作成

```
1: function MakeTree(D)
2:     if |D| = 1 then
3:         return MakeLeaf(D)
4:     else
5:         (result, node) ← MakeNode(D)
6:         if result ≠ FAILED then
7:             return node
8:         else
9:             return MakeConditionNode(D)
10: function MakeConditionNode(D)
11:     p_m ← SelectCondPattern(D)
12:     D_m ← MakeMatchingEntries(D, p_m)
13:     D_o ← MakeOtherEntires(D, p_m)
14:     t_m ← MakeTree(D_m)
15:     t_o ← MakeTree(D_o)
16:     return ← MakeCondNode(t_m, t_o)
```

図 5: 提案手法によるデコードツリー生成手順

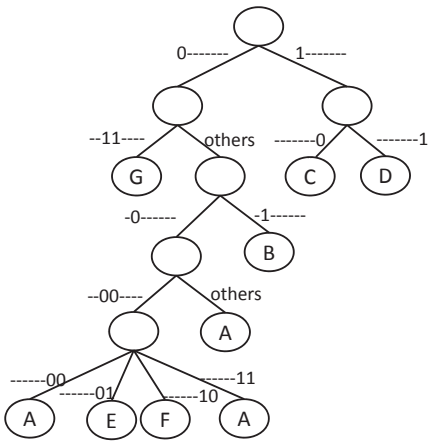

図 6: 提案手法によるデコードツリー

関数 *MakeMatchingEntries* で p_m と一致するエントリの集合 D_m を作成する．また，13 行目では other エントリ集合作成関数で p_m 以外のパターンと一致する可能性があるエントリの集合 D_o を作成する．14-16 行目では，関数 *MakeTree* の再帰的な適用で D_m，D_o からそれぞれサブツリー t_m，t_o を作成し，t_m，t_o を子とする条件ノードを作成して返す．

本処理で図 4 のデコードエントリ集合から生成したツリーを図 6 に示す．

以下本節では，図 5 の関数 *MakeMatchingEntries*，*MakeOtherEntries*，*SelectPattern* の詳細を順に示す．

4.2.1 一致エントリ集合作成 (*MakeMatchingEntries*)

関数 *MakeMatchingEntries* は，デコードエントリの集合とパターンを入力とし，入力パターンと一致する新たなデコードエントリのみからなる集合を出力する．出力集合は，入力集合の単なるサブセットではない．入力パターンによっては除外条件は無効となる場合があり，出力のデコードエントリでは無効な除外条件を含まない．また，一致パターンがすべて無効な場合には，出力デコードエントリでは不一致パターンをオペコードパターンとして展開した形式を持つ．

出力集合の作成手順を以下に示す．

(1) 出力エントリの選択
(2) 不要な除外条件の削除
(3) パターンの更新
(4) 不一致パターンの展開

(1) では，条件一致命令はパターンと適合するデコードエントリを選択する．(2) 以降は (1) で選択されたエントリに対してパターンと除外条件を更新する．(2) では，選択したパターンによって無効となる除外条件を削除する．(3) では，選択したパターンを加味して，残った除外条件を更新する．(4) では，(3) の結果として除外条件が不一致パターンのみとなった場合に不一致パターンをエントリのオペコードパターンとして展開する．以下では，各手順の詳細を示す．

step1. 出力エントリの選択

出力エントリの選択では，D のエントリ (p_o, l, C_e) のうち式 (4) の $match(p_o, p)$ を満たし，かつ下記の式 (7) の条件に該当しないエントリを選択する．

$$\exists (p_m, P_u) \in C_e : contain(p, p_m) \ \wedge \bigwedge_{p_u \in P_u} \neg match(p, p_u) \tag{7}$$

ここで，関数 $contain : \{0, 1, -\}^n, \{0, 1, -\}^n \rightarrow B$ の定義は式 (8) のとおりである．

$$contain(p_x, p_y) = \\ \forall i \in \{0, 1, ..., n-1\} : \ p_x[i] = p_y[i] \ \vee \ p_y[i] = - \tag{8}$$

すなわち，関数 *contain* は，ビットパターン p_x がビットパターン p_y をパターンの一部として含む場合のみ真となる．したがって，式 (7) は，入力パターンが一致パターンを包含しており，かつすべての不一致パターンと一致しない場合に成立する．

step2. 不要な除外条件の削除

step1 で選択した各エントリ (p_o, l, C_e) の除外条件 C_e を式 (9) の C'_e で更新する．

$$C'_e = \{(p_m, P_u) \in C_e | \ match(p, p_m) \bigwedge_{p_u \in P_u} \neg contain(p, p_u)\} \tag{9}$$

これは，入力パターンによって無効となった除外条件を削除することに相当する．また，$(p_m, P_u) \in C'_e$ の不一致パターン集合 P_u を式 (10) の P'_u に更新する．

$$P'_u = \{p_u \in P_u | match(p, p_u)\} \tag{10}$$

これは，今後成立しないことが確定した不一致パターンの削除に相当する．

step3. パターンの更新

入力パターン p で値が確定したビットは以降無効となる．step2 で更新したデコードエントリ集合に含まれるすべての除外条件中の一致パターン，不一致パターンのすべてのパターン p_c の各ビット $p_c[i]$ を式 (11) の $p'_c[i]$ のように更新する．

$$p'_c[i] = \begin{cases} p_c[i] & (p[i] = -) \\ - & (otherwise) \end{cases} \tag{11}$$

step4. 不一致パターンの展開

パターン更新の結果，一致パターンのすべてのビットが - となる場合がある．これは一致パターンの成立が確定していて，不一致パターン群の成立が未確定の場合に相当する．このような場合には，オペコードパターンが不一致パターンを含むようなデコードエントリに展開する．

展開したデコードエントリのオペコードパターンの各ビット $p'_o[i]$ は式 (12) のとおりである．

$$p'_o[i] = \begin{cases} p_o[i] & (p_u[i] = -) \\ p_u[i] & (otherwise) \end{cases} \tag{12}$$

なお，展開したデコードエントリは，展開に使用した除外条件を除き展開前の除外条件集合を引き継ぐ．

4.2.2 other エントリ集合作成 (*MakeOtherEntires*)

関数 *MakeOtherEntires* では，デコードエントリ集合 D とパターン p を入力とし，入力パターン p 以外のパターンと一致する可能性があるデコードエントリの集合 D_o を返す．D_o は D の単なる部分集合ではなく，更新された除外条件を持つ．D_o の作成手順は下記のとおりである．

(1) 出力エントリの選択
(2) 除外条件の削除

(1) では，入力パターン以外とも一致する可能性があるエントリを選択する．また，(2) では無効となる除外条件を削除する．以下に各ステップの詳細を示す．

step1. 出力エントリの選択

入力デコードエントリ集合 D のエントリ (p_o, l, C_e) で，オペコードパターン p_o が入力パターン p を一部として含んでいるエントリは，p 以外のパターンと一致する可能性がないため，出力エントリから除外する．出力エントリに含めるエントリ D_o は式 (13) のとおりである．

$$D_o = \{(p_o, l, C) \in D | \neg contain(p_o, p)\} \tag{13}$$

step2. 除外条件の削除

D_o の各デコードエントリ (p_o, l, C_e) の除外条件 C_e を式 (14) の C'_e で更新する．

$$C'_e = \{(p_m, P_u) \in C_e | \neg contain(p_m, p)\} \tag{14}$$

式 (14) は成立する可能性がなくなった除外条件の削除に相当する．

各デコードエントリのすべての除外条件 $(p_m, P_u) \in C'_e$ の不一致パターン集合 P_u を式 (15) の P'_u に更新する．

$$P'_u = \{p_u \in P_u | \neg contain(p_u, p)\} \tag{15}$$

式 (15) は入力ビット列が不一致となることが確定した不一致パターンの削除に相当する．

4.2.3 一致パターンの選択 (*SelectCondPattern*)

変則的な命令セットでは，特定の命令群の除外条件の場合に別の命令を定義しているため，従来どおりのノード作成手順 *MakeNode* を適用することができない．しがたって，特定の除外条件を持つ命令群と当該命令群の除外条件のパターンで定義された命令群とを分離したデコードエントリの集合が得られれば，分割した集合に対して従来のオペコードパターンを用いたノード作成手順を適用できるようになる．

条件ノードの作成における目標は，デコードエントリ集合に含まれる一致条件を用いてデコードエントリ集合を分割することで，従来のオペコードパターンを用いたノード作成手順を適用できるようにすることである．このためには，デコードエントリ集合に含まれる複数の一致パターンの中から適切な一致条件を選択する必要がある．不適切な例外条件を選択した場合，分割したデコードエントリ集合に対して従来どおりの手順を適用できず，再度条件ノードを作成する必要がある．

不適切な一致パターンを選択した場合には，命令がうまく分割されず，同じ命令が一致エントリ集合 D_m と other エントリ集合 D_o の両方に存在することになる．そこで，提案手法では，一致パターンの選択時に各一致パターンについて一致エントリ集合 D_m と other エントリ集合 D_o とを一旦作成し，$|D_m| + |D_o|$ を最小とする一致パターン p_m を採用する．

5. 実験

提案手法の有用性を確認することを目的とし，提案手法による命令デコーダ生成ツールの実装と生成されるデコードツリーの評価を行なった．本研究と同じく変則的な命令セットを対象とする関連研究の文献 [13] では，ARMv7[1] と MIPS4 の評価結果が記載されている．このため，従来手法と比較できるように，生成対象の命令セットとして ARMv7 と MIPS64[14] を選択した．なお，MIPS64 は MIPS4 の上位命令セットであり，MIPS4 の命令をすべて包含する．

ARMv7 は変則的な命令セットであり，MIPS64 は変則的でない命令セットである．ARMv7 のデコードエントリの除外条件には，ARM のユーザーズマニュアル [1] で命令ごとの説明頁に明示的に記載されている条件を使用した．

表 1: デコードツリー生成結果

ISA	命令数	平均深さ	最小深さ	最大深さ	テーブルエントリ数
ARM（提案）	442	6.15	2	10	978
ARM（EFF）	442	7.89	2	18	−
MIPS64（提案）	265	2.21	1	4	518
MIPS4(EFF)	163	3.66	1	8	−

また，明示的に記載されている条件以外に *cond* フィールドを持つ命令のエントリには，実態に則して $cond \neq 1111$ を除外条件に追加した．

提案手法で生成した命令セットの生成結果を文献 [13] に記載の EFF(Efficient occurrence aware decoding tree algorithm) の生成結果と共に表 1 に示す．深さはデコードツリーのルートからリーフまでの辺の数であり，命令のデコード効率を表す指標である．提案手法で生成したツリーの深さは平均値，最小値，最大値ともに EFF アルゴリズム以下である．これは，EFF ではデコードエントリを分割するときに隣接するビット列しか選択しないのに対し，提案手法では，隣接しない複数のビット列を選択できるためである．

MIPS64 を対象とする場合，提案手法では，平均の深さ 2.21 のデコードツリーを生成した．提案手法で生成したツリーは，平均値，最小値，最大値共に文献 [13] の MIPS4 を対象としたツリー以下の深さのツリーを生成している．MIPS64 は変則的な命令セットではないため，提案手法による結果は，本研究がベースとする文献 [9] の手法で生成した結果と同じになると考えられる．

表 1 のテーブルエントリ数は，各ノードの決定関数をルックアップテーブルで実装した場合におけるテーブルエントリの総数である．コストモデルを用いた手法 [10]，[13] では，浅いデコードツリーを得るためにはメモリを犠牲にする必要があった．文献 [10] の Table.1 と Figure.4 から，コストモデルを用いた手法で本研究がベースとする文献 [9] 相当の高さ（1.59）のデコードツリー ($\gamma = 1/16$) を得るためには，二分木相当のデコードツリー ($\gamma \geq 16$) を作成した場合の数十倍のメモリが必要であることが読み取れる．一方，提案手法で消費するメモリサイズは二分木相当である．n 個のリーフを持つ二分木の実現に必要なテーブルエントリ数は，$2 \times (n-1)$ で見積もると ARM，MIPS64 でそれぞれ 882，554 となり，提案手法のテーブルエントリ数 978，518 と大差がない．

6. おわりに

本論文では，変則的な命令セットに対応した命令デコーダの自動生成手法を提案した．提案手法では，オペコードパターン群でデコードエントリの集合を分割できない場合に，除外条件を用いてデコードエントリを分割し，再度オペコードパターン群による分割を適用できるようにすることを特徴とする．除外条件を用いてデコードエントリ集合を分割する場合，複数の除外条件から適切な除外条件を選ぶ必要があるが，本論文では，適切な除外条件の選択方法についても論じた．ARM と MIPS64 を対象に提案手法でデコードツリーを生成し，生成したツリーの評価を行なった結果，ツリーの深さとメモリ消費量の点で良好なツリーを生成することを確認した．今後の課題は，デコードエントリの効率的な記述を実現する記述言語の設計である．

参考文献

[1] *ARM Architecture Reference Manual ARMRv7-A and ARMRv7-R edition* (2007).
[2] Pees, S., Hoffmann, A., Zivojnovic, V. and Meyr, H.: LISA Machine Description Language for Cycle-accurate Models of Programmable DSP Architectures, DAC '99, ACM, pp. 933–938 (1999).
[3] Hadjiyiannis, G., Hanono, S. and Devadas, S.: ISDL: An Instruction Set Description Language for Retargetability, DAC '97, ACM, pp. 299–302 (1997).
[4] Hartoog, M. R., Rowson, J. A., Reddy, P. D., Desai, S., Dunlop, D. D., Harcourt, E. A. and Khullar, N.: Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign, DAC '97, ACM, pp. 303–306 (1997).
[5] Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N. and Nicolau, A.: EXPRESSION: a language for architecture exploration through compiler/simulator retargetability, DATE '99, ACM (1999).
[6] OKUDA, K.: A Simulator Generator Based on Configurable VLIW Model Considering Synthesizable HW Description and SW Tools Generation, *Proc. Workshop on Synthesis and System Integration of Mixed information Technologies, Apr. 2003* (2003).
[7] Kassem, R., Briday, M., BéChennec, J.-L., Savaton, G. and Trinquet, Y.: Harmless, a hardware architecture description language dedicated to real-time embedded system simulation, *J. Syst. Archit.*, Vol. 58, No. 8, pp. 318–337 (2012).
[8] Jeremiassen, T.: Sleipnir-An Instruction-Level Simulator Generator, ICCD '00, IEEE Computer Society, pp. 23–31 (2000).
[9] Theiling, H.: Generating Decision Trees for Decoding Binaries, LCTES '01, ACM, pp. 112–120 (2001).
[10] Qin, W. and Malik, S.: Automated Synthesis of Efficient Binary Decoders for Retargetable Software Toolkits, DAC '03, ACM, pp. 764–769 (2003).
[11] Reshadi, M., Dutt, N. and Mishra, P.: A retargetable framework for instruction-set architecture simulation, *ACM Trans. Embed. Comput. Syst.*, Vol. 5, No. 2, pp. 431–452 (2006).
[12] Ratsiambahotra, T., Cassé, H. and Sainrat, P.: A versatile generator of instruction set simulators and disassemblers, SPECTS'09, IEEE Press, pp. 65–72 (2009).
[13] Fournel, N., Michel, L. and Pétrot, F.: Automated Generation of Efficient Instruction Decoders for Instruction Set Simulators, ICCAD '13, IEEE Press, pp. 739–746 (2013).
[14] MIPS64 5K. Processor Core Family Software User's Manual (2001).