6Q-8 時相論理型言語Ｔｏｋｉｏを利用したハードウェア機能設計

藤田　昌宏*　　西田　俊和**　　川戸　信明*

(* 富士通研究所，** イリノイ大学)

1．はじめに

従来，ハードウェア記述言語は，機能設計以降に重点が置かれており，アルゴリズム・レベルの記述から扱えるものは少なかった．Ｔｏｋｉｏは，時相論理に基づくハードウェア記述言語であり，Ｐｒｏｌｏｇの拡張として定義されているため，アルゴリズム・レベルから，回路レベルにいたるまで，円滑に記述することができる［1］．ここでは，Ｔｏｋｉｏを用いて，２万ゲート程度の論理式簡単化用マイクロプロセッサの機能設計を実際に行ったので，その経緯について報告する．

2．論理式簡単化用マイクロプロセッサ

設計したのは，ＰＬＡ設計等に使われる，積和形の論理式の簡単化アルゴリズムをファームウェアで行うマイクロプロセッサであり，論理式の恒真判定を行う演算ユニットを含んでいる［2］．また，簡単化アルゴリズムには，ESPRESSOⅡを用いており，文献［3］のアルゴリズム記述を出発点とし，機能設計を行っていった．

3．設計の流れ

文献［3］で与えられるアルゴリズムをそのままＴｏｋｉｏのプログラムに直し，その後プログラムの改版を重ねる形でアーキテクチャを決定していった．最終的には，ハードウェアとマイクロプログラムにより，実現された．設計の流れを下に示す．

(1) 文献［3］で与えられるアルゴリズムをＴｏｋｉｏに直す．
(2) chopオペレータを使って並列動作を考慮しながら，単純なレジスタ転送命令まで詳細化する．
(3) (2)の動作記述を実行するのに最適と考えるデータパスを決める
(4) そのデータパス上で(2)の動作が行えるか調べる．
(5) (2)の記述をマイクロプログラムとハードウェアに分けて記述し動作をチェックする．
(6) 本ハードウェアに特有の演算ユニットのみ回路設計し，シミュレーションからその遅延時間を調べ，全体のクロックスピードの概算値を求め，もとのアルゴリズムの実行速度を評価する．

以上において，(6)のみＥＷＳ上のツールを用い，他は，Ｔｏｋｉｏの処理系を用いて行った．

4．Ｔｏｋｉｏによる記述

Ｔｏｋｉｏでは，並列動作は積（Ｔｏｋｉｏでは，','）で，順序動作はchopオペレータ（Ｔｏｋｉｏでは，'&&'）で表現する．また

```
Procedure TAUTOLOGY(F);
/*  F contains formula          */
/*  returns 1 for tautology,    */
/*          0 for nontautology  */
Begin
  TAU := SPECIAL_CASE(F);
  if (TAU =1 or TAU = 0) Return(TAU);
/*  determine case splitting variable */
  I := BINATE_SELECT(F);
/*  check one case    */
  F1 := LEFT_COFACTOR(F,I);
  TAU := TAUTOLOGY(F1);
  if (TAU = 0) Return(0);
/*  check the other case    */
  F2 := RIGHT_COFACTOR(F,I);
  TAU := TAUTOLOGY(F2);
  if (TAU = 0) Return(0);
  Return(1);
End;
```

図1　恒真判定プロシジャ：tautology［3］

```
tautology:-
% check formula falls into special cases
% *tau = 1 (tautology)
%        0 (notautology)
%       -1 (undicided)
  specialcases &&
  if (*tau = -1)
% if not, then case split
    then (binate_select && leftcofact &&
% and recursive call
          tautology &&
          if (*tau = 0)
            then empty
            else (rightcofact &&
                  tautology))
% if special case happens, return
    else empty.
```

図2　図1をＴｏｋｉｏに直したもの

```
specialcase:-
% *n register has number of variables
  if (*n < 16)
    then special_hardware
    else
      (check_tautology_product
       if (*tau = 1)
         then empty
         else
           (check_var_appear_all_product
            &&
              if (*tau = 0)
                then empty
                else check_enough_minterms)).
```

図3　special-cases の詳細化

Functional Design Using Temporal Logic Based Language: Tokio

Masahiro FUJITA*, Toshikazu NISHIDA**, Nobuaki KAWATO*

* FUJITSU LABORATORIES LTD., ** University of Illinois

通常のレジスタトランスファ・レベルの記述は次の形で表現する.

```
state-1 :- actions && state-2.
```

これは, state-1 では, actions を実行し, 次にstate-2 を実行することを示している. 例えば, FORKして並列に動作し, またJOINして処理を進めるような記述は, 次のように記述できる.

```
state-1 :- ((action-1, length(n) && state-2)
            (action-2, length(m) && state-3)) && state-4.
```

これは, nクロックでaction-1を実行した後state-2 を実行する動作と, mクロックでaction-2を実行した後state-3 を実行する動作を並列に行い, その後state-4 を実行することを表している.

本章では, アルゴリズム・レベルから詳細化法について, 恒真判定を行うプロシジャtautology を例にとって説明する. 文献［2］に示されている恒真判定アルゴリズムを図1に示す. これは, 与えられた論理式が容易に恒真判定できるか否か調べ(special-cases), そうでなければ, 0, 1の2つの場合に分ける変数をヒューリスティックで決め(binate-select), 場合分けの各々を計算し(left-cofact, rightcofact), それぞれについて自分を再帰的に呼ぶ. 図1をＴｏｋｉｏの記述に直したものを図2に示す. chopオペレータにより, ほぼそのまま対応する形で記述することができる. 次に図2に現れる各述語をchopオペレータを用いて, 単純なレジスタ転送命令のみになるまで, 詳細化していった. specialcase のアルゴリズムをＴｏｋｉｏで記述すると図3のようになる. これは, 恒真判定用の演算ユニットを使うか否か, 恒真であることがすぐ分る積項があるか否か, 肯定のみか否定のみでしか現れない変数があるか否か, それに恒真に十分な最小項をもつか否か, という4つの特別な場合に当てはまるか否かを順に調べている. 図で, *tau, *nは, それぞれ恒真判定の途中結果, 現在扱っている論理式の変数の数を表す変数であり, レジスタに対応する. また, empty は長さ0のインターバルを表し, 処理の終了を示す. 例えば,

```
if(*tau=1)then empty else ...
```

の部分から, もし*tauが1ならempty つまり, special-cases が終了することが分る（そうでなければ, ... の部分を実行する）.
また, 記述の初期の段階では, 加算やビット処理等は, Ｔｏｋｉｏのオペレータを用いて直接記述するが, データパス等を意識し, 演算ユニットとして記述する時には, これらの演算ユニットは組合せ回路であるため, 時相オペレータを用いずにＰｒｏｌｏｇのcluaseとして, 記述する. 例えば, 「レジスタａとｂの和をレジスタｃへ転送する」という記述は, 初期の段階では,

```
＊c  <－  ＊a  ＋  ＊b
```

と記述するが, adder という演算ユニットを意識する場合には,

```
ＴＡ＝＊a, ＴＢ＝＊b, ａｄｄ（ＴＡ, ＴＢ, ＴＣ）,
＊c  <－  ＴＣ
```

（ただし, ａｄｄはＰｒｏｌｏｇのclauseとして定義しておく）

というように, ＴＡ, ＴＢ, ＴＣの中間変数を導入し, 演算ユニットａｄｄとの接続を表現する形で記述する.
このようにして, 図2のアルゴリズムをＴｏｋｉｏを用いて詳細化していくことができる. なお, この段階での設計の確認つまり, 一度詳細化する前と後での同一性の検証は, シミュレーションで機能を確認することで行うが, 次章で述べるように, データパスに関するいくつかの支援ツールは用意されている. なお, 定理証明法等を用いた検証の自動化についても検討しているが別の機会に譲る.

5. データパス設計支援

記述が十分詳細化された時点で, その動作記述から最適と考えるデータパスを決定する. このデータパス決定に関連して次のような支援が行われている.

①レジスタ転送のクロスリファレンス
②レジスタ転送の衝突のチェックを行うシミュレーション
③データパス上のデータの衝突のチェックを行うシミュレーション

Ｔｏｋｉｏでは, ＊で始まる変数は, スタティック変数であり, 値が再度代入されない限り前の値を保つ. これは, ハードウェア記述ではレジスタやメモリとして使われる. ①はこのスタティック変数間のデータ転送の関係を示すものであり, 転送元, 転送先, 途中の計算に使用される演算, 及び転送回数が表として出力される. 静的に全ての場合を出力するものと, 一回のシミュレーションにおけるデータを出力する2つのモードがある. これらの情報はデータパス決定の際に利用される.
②は, 同一レジスタに同時に2つ以上のところからデータ転送されないかをシミュレーション実行時に調べる. また, ③は, 与えられたデータパス上で実行するとした時に, 同一パス上で2つ以上のデータが転送されることがないかを同じくシミュレーション実行時に調べる. この際, データパスは, Ｐｒｏｌｏｇのclauseとして表現する. これら2つの手段により, データパスの実現可能性を容易にチェックすることができる.
①の静的なクロスリファレンスは, 簡単なＰｒｏｌｏｇのプログラムで実現されており, それ以外は, 全てＴｏｋｉｏのコンパイラ, インタプリタを一部拡張することで実現されている.

6. おわりに

現在機能設計が終了した段階であるが, CMOSゲートアレイを想定した場合, 遅延時間の評価の結果, 超大型機以上の速度が得られることが判明している.
3. の(1)～(3)に関しては, Ｔｏｋｉｏ処理系の拡張により支援ができているが, (4)は現在支援できていないため, マイクロプログラムと動作記述, データパス間の検証が今後の課題である. また, アルゴリズム・レベルからの自動合成についても考えていきたい.

参考文献

［1］M. Fujita et al.: Tokio, Logic Programming Language based on Temporal Logic and its Compilation to Prolog, 3rd ICLP, London, 1986.

［2］T. Sasao: HART, A Hardware for Logic Minimization and Verification, ICCD '85, PP. 713-718.

［3］R. Brayton et al.: Logic Minimization Algorithms for VLSI Synthesis, Kluwer Academic Publishers, 1984.