設 計 自 動 化 79－2
（1996．2．9）

代数的手法を用いたパイプライン方式CPUの設計検証

島谷 肇　森岡澄夫　北嶋 暁　東野輝夫　谷口健一

大阪大学基礎工学部情報工学科

out-of-order 実行がない基本的な命令パイプライン方式をとる CPU を設計する一つの手法と，それによって実現した CPU が各命令を正しく実行することを，代数的手法を用いて証明する手法を提案する．設計では，まず，パイプラインの各ステージの動作を決め，次に，パイプラインハザードが起きないように，必要なら各ステージの動作を変更して，引き続く CPU 命令がなるべく 1 ステージ遅延で実行できるように重ね合わせる (ストールサイクルを挿入してもよい)．正しさの証明は，設計上の制約を有効に利用したため，項書き換えや整数上の論理式 (プレスブルガー文) の恒真性判定手続きなどの代数的手法を用い，ほぼ自動で行うことができる．

A Method of Designing Pipelined CPUs and Proving its Correctness using an Algebraic Approach

Hajime SHIMATANI, Sumio MORIOKA, Akira KITAJIMA,
Teruo HIGASHINO and Kenichi TANIGUCHI

Department of Information and Computer Sciences,
Faculty of Engineering Science, Osaka University

We propose a method of designing pipelined CPUs, without out-of-order executions, and of proving its correctness using an algebraic approach. First in the design, we decide the action of each pipeline stage. Next we pile those actions up so that the next instruction can be executed with 1-stage delay. At this time we can modify those actions or insert stall cycles so that pipeline hazards may not happen. Because the design constraints are effectively used, the proof of its correctness can be done almost automatically with algebraic methods, such as term rewriting or the decision procedure for Presburger sentences.

1 はじめに

従来より，我々の研究グループは，同期式順序回路を対象とし，その正しさの形式的な証明を実際上可能とするための，代数的手法を用いた仕様記述法，設計法，検証法について研究してきた．我々の提案してきた方法では，要求仕様から部品とデータパスが定まった実現まで段階的に詳細化し，各段階(レベル)ごとに詳細化が正しいことを証明する．各段階の仕様は，我々の代数的言語 ASL/ASM を用いて記述する．詳細化では，順序機械の抽象的な一動作(遷移)を，より具体的な動作の系列で局所的に展開して下位レベルの仕様を得る[1]．証明では，展開に用いた動作系列を下位レベルで実行した後に，展開した遷移の動作内容が満たされることを示す[1]．

この証明を，項書き換え，場合分け，整数上の論理式(プレスブルガー文)の恒真性判定手続きなどを用いて[2]，ほぼ自動で行う検証支援系も開発されている[3]．この支援系を用いて，一命令ごとに逐次実行するノン・パイプライン CPU の検証も行われてきた[5]．しかし，一般に CPU の要求仕様レベルでは，一命令を実行したらレジスタの値がどのように変わるかを一遷移の動作内容として記述するので，このような一遷移を展開するという設計法では，複数の命令をステージ単位でずらしながら並列に実行するパイプライン CPU の設計は不可能である．

そこで今回，out-of-order 実行がない基本的な命令パイプライン方式をとる CPU を対象とした設計法と実現レベル(部品とデータパスが定まった RT レベル)の正しさの証明法を提案する．提案する設計法は，要求仕様レベルから次の二段階の詳細化を経て実現レベルを得る，という自然なものである．(1) 要求仕様レベルを各命令に対するパイプラインのステージの動作が定まった中間レベルに展開(詳細化)する．(2) 引き続く CPU 命令がなるべく 1 ステージ遅延で実行できるように，中間レベルの各ステージの動作内容を重ね合わせて，実現レベルへの詳細化(パイプライン化)を行う．このとき，パイプラインハザードが起きないように，各ステージの動作内容を変更したりストールサイクルを挿入したりしてもよい．

文献 [6]，[7] では，要求仕様レベルと実現レベルの間で直接，実現レベルの正しさを証明していた．提案する証明法は，我々が定義した実現レベルの正しさ(各レジスタ値が，どこかのステージの実行直後の状態から取り出せる)を，設計上の制約，特に out-of-order 実行がないなどの条件を有効に利用したため，要求仕様レベルと実現レベルの間での各レジスタ値の比較を，二段階に分けて簡単に行うことができる．また，その証明は，項書き換えやプレスブルガー文の恒真性判定手続きなどの代数的手法を用いて，ほぼ自動で行うことができる．

以下，2 章では，従来より提案してきた同期式順序回路の記述言語とスタイル制限および提案するパイプライン CPU の設計法について，3 章では，実現レベルの正しさの定義について，4 章では，正しさの証明法について述べる．

[1] 上位レベルの各動作はそれぞれ独立して下位レベルの動作系列で展開されるので，無駄な動作を含むかもしれない．そこで，回路性能を向上させるために，無駄な動作の削除や動作の併合などを，等価性を保証しながら行うための支援系も開発されている[4]．

2 提案するパイプライン CPU の設計法

ここでは，まず各レベルの同期式順序回路を記述する言語とスタイル制限に触れたあと，パイプライン CPU を段階的に設計する手法を提案する．

2.1 記述言語とスタイル制限

設計の過程での各レベルの同期式順序回路の記述には，従来より我々の研究グループが提案してきた代数的言語 ASL/ASM を用いる[1]．この言語は，一般に，有限個のレジスタ(状態成分ともいう，データ型は任意)と有限状態制御部を持つレジスタ付き状態機械(EFSM)を代数的に記述できる．

具体的には，各遷移の動作内容と，各遷移の実行順序・実行条件を公理と呼ばれる等式を用いて指定する．公理は，変数を用いて $p(v_1, v_2, ..) == q(v_1, v_2, ..)$ の形で書かれた等式のことで，述語論理での $\forall v_1 \forall v_2 ..[p(v_1, v_2, ..) = q(v_1, v_2, ..)]$ という式に相当する．

一つの遷移 T の動作内容の公理では，抽象状態を表す特別な変数 s を用いて，任意の状態 s に対して，T の実行前の状態 s での各状態成分の値(状態成分 F_i の値は $F_i(s)$ のように表す)によって，実行後の状態 T(s) での各状態成分の値をどのように定めるべきかを記述する(状態成分の個数だけ公理が必要，これらを合わせて公理集合と呼ぶ)．例えば，いま，あるレベルの記述において p 個の状態成分が使われているとする．遷移 T の実行後の状態成分 $F_i(1 \leq i \leq p)$ の値は，実行前の各状態成分 $F_j(1 \leq j \leq p)$ で定まるように，

$$F_i(T(s)) == C_i(F_1(s), F_2(s), .., F_p(s))$$

と記述する．

ただし，遷移の動作内容の公理と実行条件の論理式を，抽象状態の変数 s 以外の変数を用いずに記述する．このように記述スタイルを制限するによって，後で述べる実現レベルの正しさの証明を，項書き換え，場合分け，プレスブルガー文の恒真性判定手続きなどを用いて，ほぼ自動で行うことができる[3]．

特に，実現レベルにおいて，各遷移の動作内容の公理集合のうち，その遷移の実行によって値が変わるレジスタに関する公理は，パイプラインの各ステージの動作内容に応じて分類することができる．また，中間レベルでは，このような各遷移の実行によって値が変わるレジスタに関する公理集合全体が，それぞれパイプラインのステージの動作内容に対応する．以下，これらをパイプラインのステージの動作内容と同一視して，ステージと呼ぶ．

2.2 パイプライン CPU の設計法

2.2.1 設計の概要

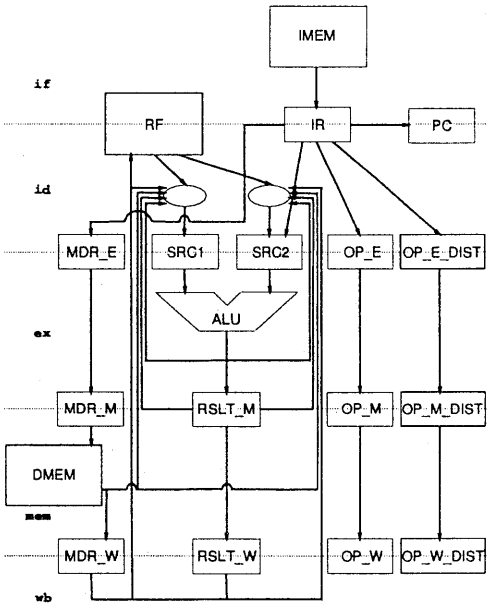
一遷移によって，一命令を実行したらレジスタの値がどのように変わるかという要求仕様を記述したレベル(level1，要求仕様レベル)から，その遷

	if	id	ex	mem	wb
ALU 演算命令 ロード命令 ストア命令 分岐命令	命令 フェッチ	命令 デコード	ALU 演算 アドレス 計算	メモリ アクセス	レジスタファイル 書き戻し

各命令はその動作内容によって，上の代表的な 4 種類の命令に分類される．

表 1: 各ステージの動作内容

移をパイプラインのステージ数だけの遷移系列に展開することによって，level2(中間レベル) の設計を行う．level2 の各ステージの動作内容を重ね合わせることによって，level3(実現レベル) の設計を行う．表 1に示した各ステージの動作を，図 1のアーキテクチャで実現する 5 ステージ・パイプライン CPU の設計の概要を図 3に示す．その記述の一部を図 2 に示す．



□印はレジスタを，○印はセレクタ (あるいはパス) を表す．

図 1: アーキテクチャ

2.2.2 要求仕様レベル

要求仕様レベル (level1) では，一命令を実行したらレジスタの値がどのように変わるかを，一遷移 cycle の動作内容として記述する (図 3の level1 参照，図 2参照)．

2.2.3 中間レベル

要求仕様レベルにおける遷移 cycle を展開 (詳細化) して，中間レベル (level2) において，以下のような $EFSM$ M_2を得るものとする (図 3の level2，図 2参照)．

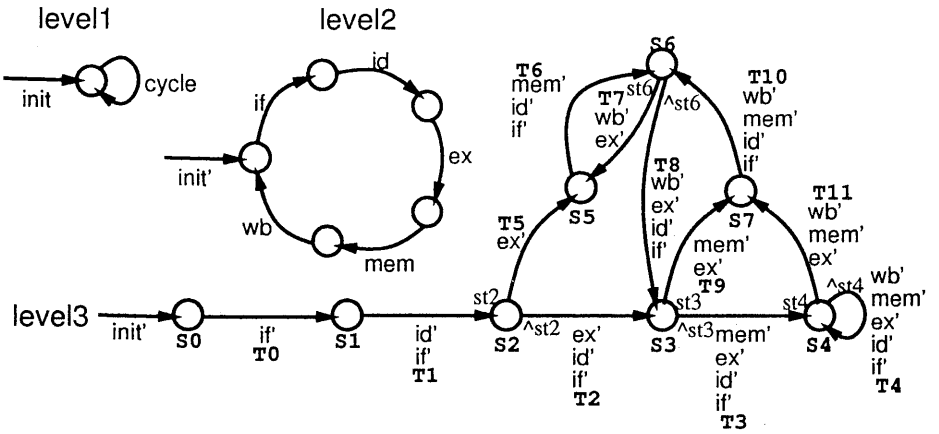
- M_2は，一命令ごとにステージ $stg_1, stg_2, .., stg_n$ をこの順に経て実行するものとする．各ステージでは，どの命令に対してどのような動作を行うかをすでに決めておくものとする (表 1参照)．
- M_2のレジスタの集合は，次のレベル level3 のパイプライン化した M_3と全く同じである．ただし，各レジスタごとに，次の (1)，(2) が成り立つものとする．(1) その値が書き込まれるステージがただ一つ存在する．(2) その値が読み出されるステージの範囲は，最初のステージ stg_1からその値が書き込まれるステージの次のステージまでである．

```
(* level1 の遷移 cycle の動作内容 *)
define 'INSTRUCTION' := 'm_get(IMEM(s), PC(s))';
RF(cycle(s)) ==
if (get_op(INSTRUCTION) = ADD) then
    r_put(RF(s),
          r_get(RF(s), get_src1(INSTRUCTION)) +
          r_get(RF(s), get_src2(INSTRUCTION)),
          get_dist(INSTRUCTION))
else if (get_op(INSTRUCTION) = LOAD) then
    r_put(RF(s),
          m_get(DMEM(s),
                r_get(RF(s), get_src1(INSTRUCTION)) +
                r_get(RF(s), get_src2(INSTRUCTION))),
          get_dist(INSTRUCTION))
else
    RF(s);
PC(cycle(s)) ==
if (get_op(INSTRUCTION) = BRA) then
    BRANCH_ADDRESS
else inc(PC(s));
DMEM(cycle(s)) == ..

(* level2 のステージ id の動作内容 *)
SRC1(id(s)) == r_get(RF(s), get_src1(IR(s)));
SRC2(id(s)) == r_get(RF(s), get_src2(IR(s)));
OP_E(id(s)) == get_op(IR(s));
OP_E_DIST(id(s)) == get_dist(IR(s));
MDR_E(id(s)) == get_m_data(IR(s));

(* level3 の複合遷移 T4 に含まれるステージ id' の動作内容 *)
SRC1(id(s)) ==
if (get_src1(IR(s)) = OP_E_DIST(s)) then
    SRC1(s) + SRC2(s)
else if (get_src1(IR(s)) = OP_M_DIST(s)) then
    (if (OP_M(s) = ADD) then RSLT_M(s)
    else m_get(DMEM(s), RSLT_M(s)))
else if (get_src1(IR(s)) = OP_W_DIST(s)) then
    (if (OP_W(s) = ADD) then RSLT_W(s)
    else MDR_W(s))
else
    r_get(RF(s), get_src1(IR(s)));
SRC2(id(s)) == ..
OP_E(id(s)) == get_op(IR(s));
               ..

(* ストールする条件 sti *)
OP_E(s) = LOAD and (get_src1(IR(s)) = OP_E_DIST(s)
or get_src2(IR(s)) = OP_E_DIST(s))
```

図 2: 記述の一部

2.2.4 実現レベル

M_2のステージの動作内容を重ね合わせることに



level2, level3 における init' を除く各遷移には，ステージ名が付けられている．また，level3 における各sti, ^sti はそれぞれ，ストールする，ストールしない条件を表す．

図 3: 設計の概要

より，中間レベルから一段階の詳細化 (パイプライン化) を行って，実現レベル (level3) において以下のような $EFSM$ M_3 を得るものとする (図3の level3, 図2参照).

- M_2の各ステージ stg_1, stg_2, .., stg_n と動作内容の公理の左辺集合が同じ (対応するということがある) M_3のステージを，それぞれ stg'_1, stg'_2, .., stg'_n とする．M_3の各遷移の動作内容は，引き続く CPU 命令がなるべく 1 ステージ遅延で実行できるように，これらのステージを重ね合わせたものとする (これを複合遷移と呼ぶことがある)．ただし，ステージを重ね合わせたときに RAW ハザード[2] が生じる場合，これを解決するために，データのフォーワーディング[3] を行うように M_2のステージの動作内容を変更したり，ストールサイクルを挿入したりしてもよい．
- ステージの動作内容を重ね合わせるときには，次の (1), (2) が成り立つようにする．(1) 各命令は，ステージ stg'_1, stg'_2, .., stg'_n をこの順に経て実行される (ステージ stg'_i が含まれる複合遷移の次の遷移に，ステージ stg'_{i+1} が含まれるとは限らない)．(2) どの複合遷移においても，各命令のステージ stg'_i を実行するとき，その直前の命令のステージ stg'_{i+1} を同時に実行するか，あるいは，その実行はすでに終了している．すなわち，各命令がその直前の命令に追い付いたり追い越したりすることはない．

なお，このように設計を行うことによって，M_3 において，WAR ハザード[4] と WAW ハザード[5] が生じないことが保証される．

[2] read after write hazard. パイプラインハザードの一つで，あるレジスタへ値を書き込んだ後，その値を読み出すように命令が並んでいるとき，読み出しが先に起こることによって生ずるハザード．

[3] forwarding. バイパスを設けて，先行命令が最終的にレジスタへ書き込む値に相当する中間出力を入力へフィードバックすること．

[4] write after read hazard

[5] write after write hazard

3 実現レベルの正しさの定義

3.1 一般的な正しさの定義

CPU の要求仕様レベルも実現レベルも，命令を解釈する機械 (インタープリタ) とみなすことができる．一般に，要求仕様レベルの抽象状態 (以下，EFSM の状態と混同のない限り単に状態と呼ぶ) は，実現レベルの状態での状態成分をすべて含むわけではなく，実現レベルの動作も，必ずしも要求仕様レベルと同じ順序やステップ数で起こるわけではない．そこでまず，実現レベルの状態系列と要求仕様レベルの状態系列との対応 A を定義する．その上で，out-of-order 実行がある場合も考慮した，一般的な CPU の実現レベルの正しさを，以下のように定義する．これは，要求仕様レベルのインタープリタの動作が，実現レベルのインタプリタの動作に従うというものである[6].

まず，要求仕様レベルのインタープリタ Iを次のように定義する．

$$I[ss] \equiv \forall t : time.\ ss_{t+1} = N(ss_t)$$

ここで，ss_tは，時刻 t における状態系列の値 (すなわち，状態) である．Nは，時刻 t における状態を引数として，時刻 $t+1$ における状態を返す関数である．実現レベルのインタープリタ I'も，同様に定義される．

$$I'[ss'] \Rightarrow I[A(ss')]$$

が成り立つとき，対応 A のもとで，I'は Iの正しい実現であるという．

3.2 提案する設計法のもとでの正しさ

提案する設計法に基づいて設計した，out-of-order 実行がない基本的な命令パイプライン方式をとる CPU に対しては，その実現レベルの正しさにおける対応 A を具体的に与えることができる．すなわち，A を次の (1)〜(3) を満たすように，実現レベルの状態と要求仕様レベルの状態との対応に制限できる (要求仕様レベルの状態での各状態成分ごとに

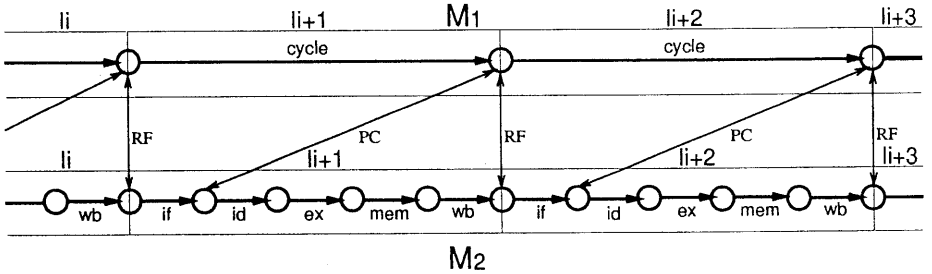


図 4: M_1から M_2への詳細化の正しさ

対応がある)[6].

(1) 要求仕様レベルの任意の状態の一遷移 (以下, EFSM の遷移と混同のない限り単に遷移と呼ぶ) に対して, 実現レベルの状態系列が始点と終点が定まったただ一つの部分系列に区切ることができる.

(2) 要求仕様レベルの任意の一遷移に対応する部分系列の始点は, その直後の一遷移に対応する部分系列の始点より前にある. 終点についても同様である.

(3) 要求仕様レベルの任意の一遷移の終点での各状態成分に対して, その遷移に対応する部分系列に含まれる状態の中に, その状態成分の値を取り出せる状態が少なくとも一つ存在する (このような実現レベルの状態のうちの一つとの対応が A であり, 以下の証明では, 検証者が適当に定めることになる).

提案した設計法に基づいて設計した場合, 各部分系列はそれぞれ一命令のステージ stg'_1, stg'_2, .., stg'_n が含まれる複合遷移系列に相当し, (1), (2) はすでに成り立っている[7]. よって, (3) のみ証明すればよい. これは, 対応 A を用いずに以下のように表すことができる.

証明すべき実現レベル M_3の正しさ (a)

> 要求仕様レベル M_1と実現レベル M_3の状態系列をそれぞれ ss, ss''とする. $F_i(s)$, $F''_i(s'')$ を, それぞれ M_1, M_3の状態 s, s''から状態成分 F_i の値を取り出す関数とする. ss の任意の状態 s での各状態成分 F_iに対して, 次が成り立つ.
> $$\forall s \in ss.\ F_i(s) = F''_i(s'')$$
> ここで, s''は, 検証者が適当に定めた, s を終点とする ss の一遷移 cycle に対応する M_3の部分系列に含まれる状態である.

4 正しさの証明法

先に述べた正しさ (a) を証明するためには, 以下に示す," 要求仕様レベル M_1 から中間レベル M_2への詳細化の正しさを表す性質 P1" と" 中間レベル M_2から実現レベル M_3への詳細化 (パイプライン化) の正しさを表す性質 P2" の二つの性質が成り立つことを証明すればよい. それぞれの証明は, 対象とする引き続く二つのレベルの EFSM の記述および対応[8] を与えれば, 項書き換え, 場合分け, プレスブルガー文の恒真性判定手続きなどの代数的手法を用いて, ほぼ自動で行うことができる[3].

4.1 M_1から M_2への詳細化の正しさを表す性質 P1 の証明

要求仕様レベル M_1から中間レベル M_2への詳細化の正しさを表す性質 P1 を以下に示す.

M_1から M_2への詳細化の正しさを表す性質 **P1**

> M_1の任意の状態 s での各状態成分 $F_i(1 \leq i \leq p)$ に対して, 次が成り立つ.
> $$\forall s.\ F_1(s) = F'_1(s') \wedge F_2(s) = F'_2(s') \wedge \\ .. \wedge F_p(s) = F'_p(s') \\ \Rightarrow F_i(cycle(s)) = F'_i(s'_i)$$
> ここで, s'は, s を始点とする遷移 $cycle$ に対応する M_2の部分系列の始点であり, s'_iは, その部分系列 (ステージ stg_1, stg_2, .., stg_n を実行する遷移系列) に含まれ, F_iの値が変わるステージの実行直後の状態である. なお, $F'_i(s')$ は, M_2の状態 s'から状態成分 F_iの値を取り出す関数である.

図 4では, M_2のある状態系列のいくつかの状態 s での, 状態成分 PC と RF の対応を示した.

性質 P1 の証明は, 次のように行う. 結論 $F_i(cycle(s)) = F'_i(s'_i)$ を, 次の公理と前提 $F_i(s) = F'_i(s')(1 \leq i \leq p)$ を用いて項書き換え[9] を行った後, プレスブルガー文の恒真性判定手続きを用いてそれが真であるかどうか判定し, 結果が真であれば, 性質 P1 が成り立つと結論する.

- M_1の $cycle$ と, M_2の stg_1の始点 s'から stg_1, stg_2, .., stg'_n を含む遷移系列中の指定された s'_iまでの遷移系列の動作内容

4.2 M_2から M_3への詳細化の正しさを表す性質 P2 の証明

中間レベル M_2から実現レベル M_3への詳細化 (パイプライン化) の正しさを表す性質 P2 を以下に

[6] 要求仕様レベルと実現レベルの初期状態における対応する状態成分の値がすべて同じであると仮定する.

[7] 厳密には, 各命令がステージ順に実行されること, 有限ステップで終了すること, 命令の追い付き・追い越しがないことなどを, 構造的帰納法や記号実行などを用いて証明しなければならないが, 本稿では, 陽に議論しない.

[8] 設計上の制約から, 要求仕様レベルの状態と実現レベルの状態との対応は, 検証者が陽に与えなくても暗黙のうちにつく. 以下の二つの性質の中では, その対応を用いている.

[9] 項書き換えの際には, 適宜, 基本関数・述語に関する補題 (例として, "配列のある箇所にデータを書いた後, 同じ場所を読むと, 書いたデータが読める", というものがある) を用いる.

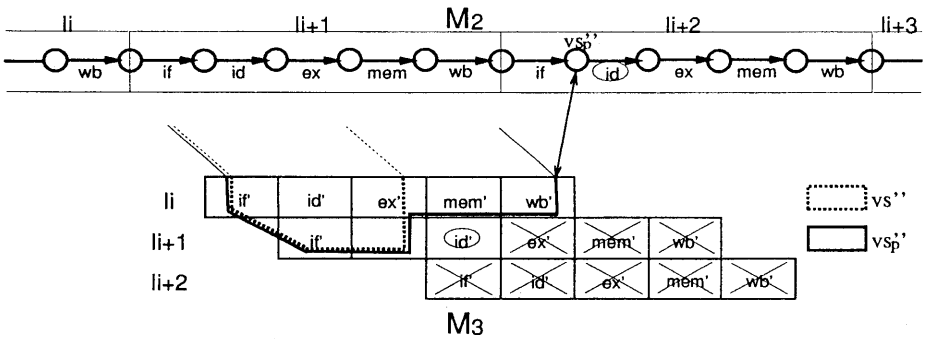


図 5: M_2から M_3へのパイプライン化の正しさ

示す.

M_2から M_3への詳細化の正しさを表す性質 **P2**

M_2の任意の状態 s'(状態成分 $F_i'(1 \leq i \leq q)$ を含む) での，それを始点とするステージ stg_i(対応する M_3のステージを stg_i'とする) の実行によって値が変わる各状態成分 F_i'に対して，次が成り立つ.

$$\forall s'.\ COND \wedge$$
$$F_1'(s') = F_1''(vs_p'') \wedge F_2'(s') = F_2''(vs_p'') \wedge$$
$$.. \wedge F_q'(s') = F_q''(vs_p'')$$
$$\Rightarrow F_i'(stg_i(s')) = F_i''(stg_i'(vs''))$$

ここで，vs''は，stg_i'の実行前の，後行命令の動作内容を完全に無視した，(仮想的な) 状態である．vs_p''は，stg_i'の先行命令の実行直後の，後行命令 (現在の命令の stg_i'以降の動作内容を含む) の動作内容を完全に無視した，(仮想的な) 状態である．また，$COND$は，stg_i'の実行前の状態から stg_i' の先行命令の実行が終了するまでの，複合遷移系列の実行条件である.

性質 P2 を，各場合について証明すれば，次の (1), (2) を示したことになる.

(1)M_3の先行命令がない (初期状態から始まる) 命令 I1 のステージ stg_i'の実行によって値が変わる各状態成分の値が，それぞれ M_2の対応するステージ stg_iの実行によって値が変わる状態成分の値に等しい.

(2) 次の Ii(i は任意定数) を先行命令とする命令 Ii+1 のステージ stg_i'の実行によって値が変わる各状態成分の値が，それぞれ M_2の対応するステージ stg_iの実行によって値が変わる状態成分の値に等しい．証明の際には，Ii までの命令のステージで証明した事柄とステージの出現順序に関する設計上の制約を用いる.

図 5では，M_2の○印を付けたステージ id の実行直後の状態と，M_3の対応するステージ id' の実行直後の，後行命令の動作内容を完全に無視した，(仮想的な) 状態とで，それらのステージの実行によって値が変わる各状態成分の値が等しいことをどのように示すかを示した．事前に証明した事柄と設計上の制約から，M_3において，M_2の id の実行前の状態に相当する (仮想的な) 状態 vs_p''(実線で囲んだ命令のステージを仮想的にすべて実行した状態) を設けることができる.

性質 P2 の証明も，性質 P1 の証明と同様に，項書き換えとプレスブルガー文の恒真性判定手続きを用いて行うことができる.

5 おわりに

本稿では，out-of-order 実行がない基本的な命令パイプライン方式をとる CPU の設計法と，代数的手法を用いた実現レベルの正しさの証明法を提案した．現在，ここで提案した正しさの証明法に基づく検証支援系を試作中であり，今後，実用規模の設計例の検証が可能かどうか，検証にどの程度の手間がかかるのかなどを調べたい.

参考文献

[1] 大蘆雅弘, 杉山裕二, 谷口健一: "代数的言語 ASL における抽象的順序機械型プログラムとその処理系", 信学論 (DI), Vol.J73−D−I, No.12, pp.971-978(1990).

[2] 東野輝夫, 関浩之, 谷口健一: "代数的仕様から関数型プログラムの導出とその実行", 情報処理, Vol.29, No.8, pp.881-896(1988).

[3] 森岡澄夫, 北道淳司, 東野輝夫, 谷口健一: "代数的言語で記述した抽象的順序機械型プログラムの設計検証の自動化", 情報処理学会論文誌, Vol.36, No.10, pp.2409-2421(1995).

[4] 森岡澄夫, 北道淳司, 東野輝夫, 谷口健一: "代数的手法を用いた順序回路の段階的設計支援システムにおける状態図変形機能", 第 8 回回路とシステム軽井沢ワークショップ論文集, E2-2(1995).

[5] Kitamichi,J., Morioka,S., Higashino,T. and Taniguchi,K.: "Automatic Correctness Proof of the Implementation of Synchronous Sequential Circuits using an Algebraic Approach", (Kropf,T. and Kumar,R. eds.), Vol. 901 of Lecture Notes in Computer Science, pp.165-184, Springer Verlag(1995).

[6] Phillip J. Windley: "Verifying Pipelined Microprocessors", Proc. of the 12th IFIP Intl. Conf. on Computer Hardware Description Languages and their Applications(CHDL'95), pp.503-511(1995).

[7] Mandayam K. Srivas, Steven P. Miller: "Applying Formal Verification to a Commercial Microprocessor ", Proc. of CHDL'95, pp.493-502(1995).