設 計 自 動 化 81－12
(1996. 10. 18)

in-order 実行パイプライン CPU の正しさの自動証明例

島谷 肇　森岡澄夫　東野輝夫　谷口健一

大阪大学基礎工学部情報工学科

〒560 豊中市 待兼山町 1-3
Tel: 06-850-6608　　Fax: 06-850-6609
E-mail: {simatani, morioka, higashino, taniguchi}@ics.es.osaka-u.ac.jp

あらまし

我々は，命令の in-order 実行のみを行うパイプライン CPU を設計する一つの手法と，それによって設計した実現の正しさを，代数的手法を用いて自動で証明する手法を提案している．設計では，要求仕様から次のように段階的詳細化を行い RT レベルを得る．まずパイプラインの各ステージの動作を決め，次にフォーワーディングやストールを行いながらそれらを重ね合わせる．実現の正しさの証明では，実現の各ステージごとに検証条件を求め，代数的手法を用いてその恒真性を判定する．実際に，提案した手法に基づく検証支援系を作成した．本支援系を用いて，命令セットが 11 命令からなる 5 ステージ・パイプライン CPU の正しさの証明の一部を行うことができた．

キーワード　パイプライン CPU，in-order 実行，段階的詳細化，正しさの証明，代数的手法，検証支援系

Automatic Correctness Proof of a Pipelined CPU with In-Order Exection

Hajime SHIMATANI, Sumio MORIOKA,
Teruo HIGASHINO and Kenichi TANIGUCHI

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

Machikaneyama 1-3, Toyonaka, Osaka 560
Tel: 06-850-6608　Fax: 06-850-6609
E-mail: {simatani, morioka, higashino, taniguchi}@ics.es.osaka-u.ac.jp

Abstract

We have proposed a design and automatic proof method for pipelined CPUs with in-order execution. In a design, we carry out the following step-wise refinement to obtain a RT level. First we decide an action of each pipeline stage and arrange those stages in an order, and next pile up those actions with forwarding and stalling introduced. At the correctness proof of a realization, we check that for each pipiline stage in a realization, a verification condition holds using our algebraic methods. We made the verification support system based on the above method, and partly carried out the correctness proof of a 5-stage pipelined CPU, whose instruction set is composed of 11 instructions.

key words　Pipelined CPU, In-order execution, Step-wise refinement, Correctness proof, Algebraic method, Verification support system

1 はじめに

従来より，形式的検証を実用規模のCPUに適用可能にするための研究が広く行われている．実用規模のCPUを，実用時間で検証できるようにするためには，証明作業をなるべく自動化することが重要である．このため，どのような仕様記述言語，証明法を用いれば自動化が容易に行えるかの研究[1, 3, 5]や，また，AAMP5など実用規模のCPUに対する検証実験[2]が行われている．

検証自動化のために研究されている方法のほとんどは，仕様を記述能力の比較的低い言語(一階述語論理，時相命題論理，等式論理，など)で記述し，検証条件の成立を，Model Checking Routine，あるいは様々なクラスの論理式に対するDecision Procedureを用いて，自動で示すというものである．例えば，我々の提案する方法では，仕様記述には代数的言語を用い，証明は，項書換え，場合分け，整数上の論理式(プレスブルガー文，以下P文と略す)の真偽判定手続きなどの代数的手法を一定の手順で組み合わせることで行う．我々の方法では，Tamarack，KUE−CHIP2などの非パイプラインCPUの検証が，実際に完全自動かつ短時間で行えることも分かっている[4, 6]．

パイプラインCPUの証明の自動化については，まだ研究がほとんど行われていない．個々のCPU設計例，例えばAAMP5等については，かなり自動化されたベリファイアPVSで検証を行った例がある[2](注：完全に自動で行ったわけではない．かなり人手作業の部分がある)．しかし，どのようなクラスのパイプラインCPUであれば，また，どのような証明法を用いれば完全自動で証明できるのかは，ほとんど明らかにされていない．

そこで，我々は文献[7]で以下のことを行った．(1)証明の対象を，あるクラスのパイプラインCPU(in-order実行パイプラインCPU)に制限し，それを導出する設計法を与えた，(2)そのクラスのCPUに対して，正しさを完全に自動で証明する方法を考案した．

その概要は次のとおりである．設計では，要求仕様を記述したレベル(要求仕様レベル)から次のように段階的詳細化を行いRTレベルを得る．まずパイプラインの各ステージの動作を決め(逐次実行レベル)，次にフォーワーディングやストールを行いながらそれらを重ね合わせる(パイプラインレベル)．実現の正しさの証明では，実現の各ステージごとに検証条件を求め，代数的手法を用いてその恒真性を判定する．

今回実際に，提案した手法に基づく検証支援系を作成した．例題の証明の一部が自動で行えることを確認し，どの程度のCPU時間がかかるかを測定した．

この支援系では，逐次実行レベルとパイプラインレベルの記述および証明の対象とするパイプラインレベルのステージstg'_iとstg'_iの直前命令の実行可能パスの記述を入力とし，次の(1)，(2)の作業を順に自動で行う．(1)stg'_iの直前命令の実行可能パスをたどりながら，stg'_iで値が書き込まれ得る各レジスタ$F_{w(i)}$ごとに，その場合の検証条件を生成する，(2)項書換え，P文の真偽判定手続きをこの順に適用し，生成された検証条件の恒真性の判定を行う．

本支援系を用いて，CPUとして最小限の機能を持った，命令セットが11命令からなる5ステージ・パイプラインCPUの正しさの証明の一部を行った．パイプラインが定常状態付近にあり，複雑なフォーワーディングを行うステージに対して，実行中の命令の演算コードで場合分けを行うことにより，そのいくつかの場合に対して高々CPU時間(Pentium-Pro200MHz)数十分程度で行うことができた．

本実験により，上述のクラスのCPUに対しては，証明作業を完全自動で行える見通しを得た．すなわち，逐次実行レベルの仕様記述とパイプラインレベルの仕様記述だけをベリファイアに与えれば，補題等を与える必要もなく，完全に自動で証明を行うことができる．今後の課題として，場合分けの総数を減らして，完全な証明が行えるか，また，比較のため，[2]で扱われているような3ステージ・パイプラインCPUの完全な証明が可能か調べたい．

以下，2章では，提案した設計法，実現の正しさの定義およびその正しさの証明法の概要について，3章では，検証支援系を用いた検証実験と考察について述べる．

2 提案した設計法，実現の正しさの定義およびその証明法の概要

検証実験で用いたCPUを例に，[7]で提案したin-order実行パイプラインCPUの設計法，実現の正しさの定義およびその証明法の概要について述べる．

2.1 in-order実行パイプラインCPUの設計法

ここでは，まず各レベルの同期式順序回路の仕様を記述する言語と検証実験で用いたCPUの要求仕様に触れる．そのあと，in-order実行パイプラインCPUの設計法の概要について述べる．

2.1.1 記述言語

各レベルの同期式順序回路(EFSM)の仕様記述は，各遷移に対して，レジスタ(状態成分)群の値がどのように変わるかを指定する動作内容の記述，およびどの条件が成り立つときにどの遷移を実行すべきかを指定する実行制御の記述からなる．

一つの遷移Tの動作内容の公理では，EFSMのレジスタの値などのすべての情報を含んでいる抽象状態を表す特別な変数sを用いる．そして，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F_iの値は，抽象状態を引数とする，状態成分名と同名の状態成分関数を用いて，$F_i(s)$のように表す．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，

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

と記述する．ここで，$Func$は整数型の関数である．

種類	ニーモニック	機能
ALU 演算	ADD SRC1, SRC2, DEST	RF[SRC1] の値に RF[SRC2] の値を加算し，その結果を RF[DEST] に格納する．
	NOT SRC1, DEST	RF[SRC1] の各ビットを反転し，その結果を RF[DEST] に格納する．
	AND SRC1, SRC2, DEST	RF[SRC1] の RF[SRC2] の対応するビットどうしの AND をとり，その結果を RF[DEST] に格納する．
	SGT SRC1, SRC2, DEST	RF[SRC1] の値が RF[SRC2] の値より大きければ，RF[DEST] に 1 を格納し，そうでなければ，0 を格納する．
	SEQ SRC1, SRC2, DEST	RF[SRC1] の値が RF[SRC2] の値に等しければ，RF[DEST] に 1 を格納し，そうでなければ，0 を格納する．
	LOADI IMM, DEST	IMM を RF[DEST] に格納する．
ロード	LOAD SRC1, L_A, DEST	MEM[RF[SRC1]+L_A] の値を RF[DEST] に格納する．
ストア	STORE SRC1, S_A, SRC2	MEM[RF[SRC1]+S_A] に RF[SRC2] の値を格納する．
分岐	BEQZ SRC1, B_A	RF[SRC1] の値が 0 なら PC の値に B_A を加えた番地にジャンプ．そうでなければ，PC の値を 4 インクリメントする．
	SPJMP B_A, DEST	PC の値を 4 インクリメントした結果を RF[DEST] に格納し，PC の値に B_A を加えた番地にジャンプ．
	JMP SRC2	RF[SRC2] 番地にジャンプ．

分岐命令は，遅延分岐スロットが 1 スロットである遅延分岐を行う．

表 1: 検証実験で用いた CPU の命令セット

また，実行制御は，制御部の状態 (control state) を表す有限状態名集合を導入して，現在の状態名を表す特別な状態成分関数 (CONTROL) と各遷移 T に対して Tが実行されるべきかの条件を記述する関数 (VALID) を用いて指定する．

遷移の動作内容の公理と実行条件の論理式は，抽象状態の変数 s 以外の変数を用いずに記述する [5]．

また，証明の際に基本関数どうしの補題が不要となるように，共通の基本関数 (状態成分関数，整数上の加減算，メモリの入出力) を用いる[1]．

2.1.2 検証実験で用いた CPU の要求仕様

CPU として最小限の機能を持つように，表 1のように命令セットを定めた．

その他の要求仕様は以下のとおりである．

命令長: 1 語 8 ビット (=1 バイト) とし，各命令とも 4 語の固定長とする．

データの表現形式: 2 の補数表現．

レジスタファイル RF: RF[0]～RF[31] の 32 個のレジスタからなり，それらのビット長はすべて 32 ビットとする．ただし，RF[0] はその値が常に 0 であるような特殊なレジスタとする．

2.1.3 in-order 実行パイプライン CPU の設計法

例として，上で述べた検証実験で用いた CPU の要求仕様をどのように実現するかについて述べる．各ステージで表 2に示した動作を行い，図 2のアーキテクチャで実現することにした[2]．要求仕様レベルからパイプラインレベルまでの設計過程を図 1に示す．また，その記述の一部を図 3に示す．

[要求仕様レベル]

要求仕様レベル (level1) では，ノンパイプライン CPU の場合と同様に，一命令を実行したときレジスタの値がどのように変化すべきかを，一遷移 $cycle$ の動作内容として記述する．この EFSM を M_1 とする．

[逐次実行レベル]

M_1 における遷移 $cycle$ をパイプラインのステージを順番に実行する遷移系列 $stg_1, stg_2, .., stg_n$ に展開 (詳細化) する．このレベル (level2) を逐次実行レベルと呼び，この EFSM を M_2とする．以下に M_2 の設計法を示す (表 2参照)．

(M_2の設計法)•M_2は，一命令ごとに同数のステージ $stg_1, stg_2, .., stg_n$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．

•M_2の各ステージで値が書き込まれ得るレジスタの集合は，次のレベル level3 のパイプライン化した M_3の対応するステージで値が書き込まれ得るレジスタの集合と全く同じである．ただし，次の制約 (C1)，(C2) を満たすようにする．

> **(C1)** 各レジスタごとに，その値が書き込まれ得るステージがただ一つ存在する．
> **(C2)** 各レジスタについて，その値が読み出され得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．

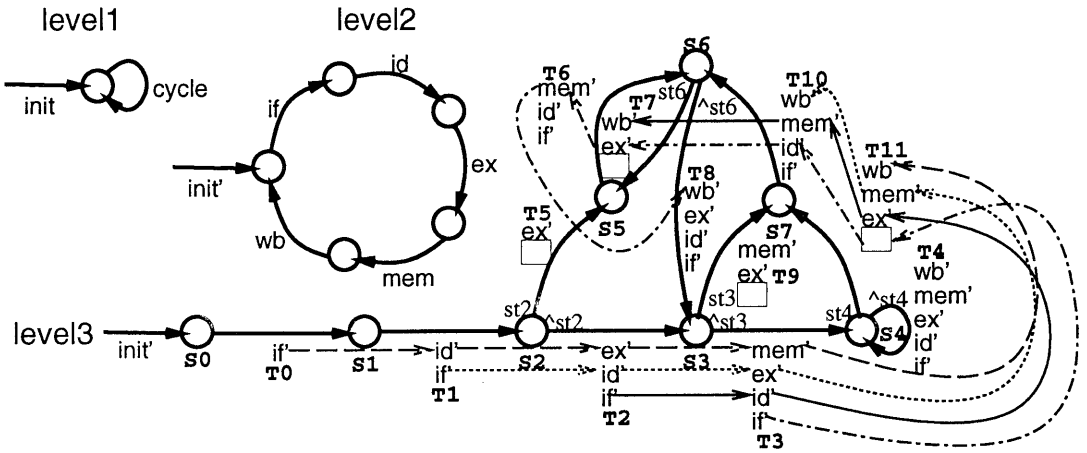
[パイプラインレベル (実現レベル)]

M_2の各ステージの動作を重ね合わせて，一段階の詳細化 (パイプライン化) を行う．このレベル (level3) を，パイプラインレベル (実現レベル) と呼び，この EFSM を M_3とする．以下に M_3の設計法を示す．

(M_3の設計法)•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，これらのステージの動作内容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，ステージの動作内容を重ね合わせたときに RAW ハザー

[1]厳密には，“メモリ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”，というメモリに関する補題のみ，証明の際に必要となる．

[2]NTT CS 研の並列処理アーキテクチャ研究グループによる 47 命令 5 ステージ・パイプライン CPU の設計例における基本アーキテクチャを，採用させていただいた．



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．□印は，ステージid' の実行がストールされていることを示す．また，パス $T_0 - T_1 - T_2 - T_3 - T_{11} - T_{10} - T_7 - T_6 - T_8$ に沿って回路の状態が進行する場合の，各命令の進行をそれぞれ線で示した．

図 1: 設計過程

ステージ stg_i	if	id	ex	mem	wb
ALU 演算 ロード ストア 分岐	命令フェッチ (各命令に共通)	命令 デコード	ALU 演算 アドレス 計算	メモリ アクセス	RF 書き戻し
	分岐アドレス計算 (1 つ前の命令に依存)				
値が書き込まれ得るレジスタ $F_{w(i)}$	IR PC (ストア関係) (演算コード) (デスティネーションレジスタ)	A B MDR_E OP_E OP_E_DIST	RSLT_M MDR_M OP_M OP_M_DIST	RSLT_W (演算結果) MDR_W (ロードデータ) DMEM OP_W OP_W_DIST	RF

表 2: 各ステージの動作内容

ドが生じる場合，これを考慮して，データのフォーワーディングを行うように 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．また，相異なる 2 つの複合遷移において，同じステージ stg'_iが含まれていても，それらのステージの動作内容は異なってもよい．

•ステージの動作内容を重ね合わせるときには，制約 (C3)，(C4) を満たすようにする．

(C3) 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．
(C4) 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．

[RT レベル]

最後に，パイプラインレベルの各遷移のデータ転送を実現するために，部品とデータパスを決定し，RT レベル (level4) を得る．このとき，資源競合による構造的ハザードがなくなるようにする (図2参照)．

2.2 実現の正しさの定義

in-order 実行パイプライン CPU に対して，その実現レベルの正しさを以下のように定義する．すなわち，

命令の実行系列に関する正しさ **(D1)**

実現レベルにおいて，各命令の実行が有限ステップで終了すること，各命令の実行開始・終了順序が in-order であること

および

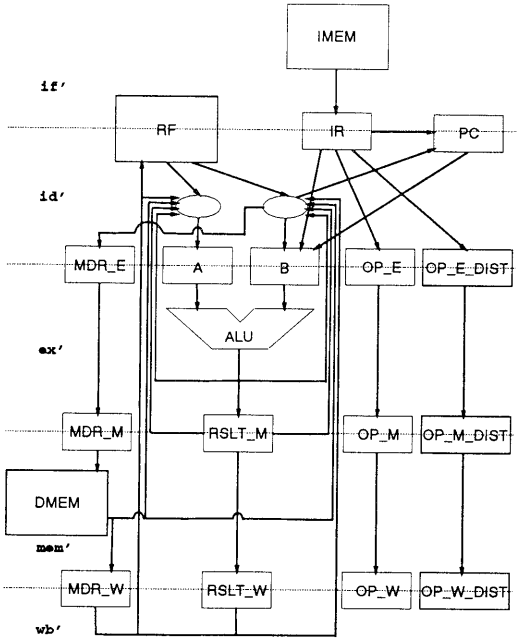
状態成分の値に関する正しさ **(D2)**

要求仕様レベルの各命令の実行後の各レジスタの値が，実現レベルの対応する命令の実行系列中で実現されること

である[3]．

(D2) の対応が *Abs* である [2]．制約 (C1)〜(C4) を満たした，in-order 実行パイプライン CPU に対

[3]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．



□印はレジスタを，○印はセレクタ (あるいはバス) を表す．

図 2: アーキテクチャ

しては，対応 Abs を自動的に与えることができる．すなわち，Tの終点での各状態成分 F_jごとに，その値が t 中の F_jの値が書き込まれ得るステージの実行直後の状態で実現されるとすればよい．

2.3 正しさの証明法

実現レベル M_3が制約 (C3), (C4) を満たすならば，命令の実行系列に関する正しさ (D1) が成り立つのは自明である．また，制約 (C1)〜(C4) を満たす M_3については，対応 Abs が自動で定まる．

正しさ (D2) を証明するためには，" 要求仕様レベル M_1から逐次実行レベル M_2への詳細化の正しさを表す検証条件 V1" と，" 逐次実行レベル M_2から実現レベル M_3への詳細化 (パイプライン化) の正しさを表す検証条件 V2" の二つの検証条件 (十分条件) が成り立つことを証明すればよい．

V1 の証明は，従来と全く同じ手法 [5] で行えるので，本稿では省略する．以下では，V2 の証明法について述べる．

実現レベル M_3が制約 (C1)〜(C4) を満たすならば，各ステージの実行はそのあとに続く命令の実行に全く影響を受けることがない．よって，特定の命令の実行のみ仮想的に進めたり，その実行を無視したりすることができる．このことを利用して，以下の命題 P を，M_2の初期状態からのステージ stg_i の実行回数に関する帰納法によって示す．

命題 P

M_2のステージ stg_iの実行によって値が書き込まれ得る各レジスタ $F_{w(i)}$の値は，それぞれ M_3の対応するステージ stg'_iの実行後の $F_{w(i)}$の値に等しい．

```
(* level1 の遷移 cycle の動作内容 *)
define 'INSTRUCTION' := 'iget(IMEM(s), PC(s))';
define 'PREV_INSTRUCTION' := (1 つ前の命令);
RF(cycle(s)) ==
(if (get_mt(get_op(INSTRUCTION)) = ALU and
     get_nt(get_op(INSTRUCTION)) = ADD) then
    iput(RF(s),
         iget(RF(s), get_src1(INSTRUCTION)) +
         iget(RF(s), get_src2(INSTRUCTION)),
         get_dist(INSTRUCTION))
..
else if (get_mt(get_op(INSTRUCTION)) = LOAD) then
    iput(RF(s),
         iget(DMEM(s),
              iget(RF(s), get_src1(INSTRUCTION))
              + get_la(INSTRUCTION)),
         get_dist(INSTRUCTION))
..
else
    RF(s));
PC(cycle(s)) ==
(if (get_mt(get_op(PREV_INSTRUCTION)) = BRA and
     get_nt(get_op(PREV_INSTRUCTION)) = BEQZ) then
    (if (iget(RF(s), get_src1(PREV_INSTRUCTION)) = 0)
     then PC(s) + get_ba(PREV_INSTRUCTION)
     else PC(s) + 4)
..
else PC(s) + 4);
..
```

```
define 'OPE_CODE' := 'get_op(IR(s))';
(* level2 のステージ id の動作内容 *)
A(id(s)) ==
(if (get_mt(OPE_CODE) = ALU and
 get_nt(OPE_CODE) = LOADI) then 0
 else if (get_mt(OPE_CODE) = BRA and
 get_nt(OPE_CODE) = SPJMP) then 0
 else if (get_src1(IR(s)) = 0) then 0
 else r_get(RF(s), get_src1(IR(s))));
..
OP_E(id(s)) == OPE_CODE;
..
```

```
(* level3 の複合遷移 T4 に含まれるステージ id' の動作内容 *)
A(id(s)) ==
(if (get_mt(OPE_CODE) = ALU and
 get_nt(OPE_CODE) = LOADI) then 0
 else if (get_mt(OPE_CODE) = BRA and
 get_nt(OPE_CODE) = SPJMP) then 0
 else if (get_src1(IR(s)) = 0) then 0
 else if (get_src1(IR(s)) = OP_E_DIST(s) and
 get_mt(OP_E(s)) = ALU and get_nt(OP_E(s)) = ADD)
 then A(s) + B(s)
 ..
 else if (get_src1(IR(s)) = OP_M_DIST(s) and
 get_mt(OP_M(s)) = ALU) then RSLT_M(s)
 ..
 else if (get_src1(IR(s)) = OP_W_DIST(s) and
 get_mt(OP_W(s)) = ALU) then RSLT_W(s)
 ..
 else r_get(RF(s), get_src1(IR(s))));
..
OP_E(id(s)) == OPE_CODE;
..

(* ストールする条件 sti *)
get_mt(OP_E(s)) = LOAD and ((not(get_src1(IR(s))
= 0) and get_src1(IR(s)) = OP_E_DIST(s)) or
(not(get_src2(IR(s)) = 0) and get_src2(IR(s)) =
OP_E_DIST(s)))
```

図 3: 記述の一部

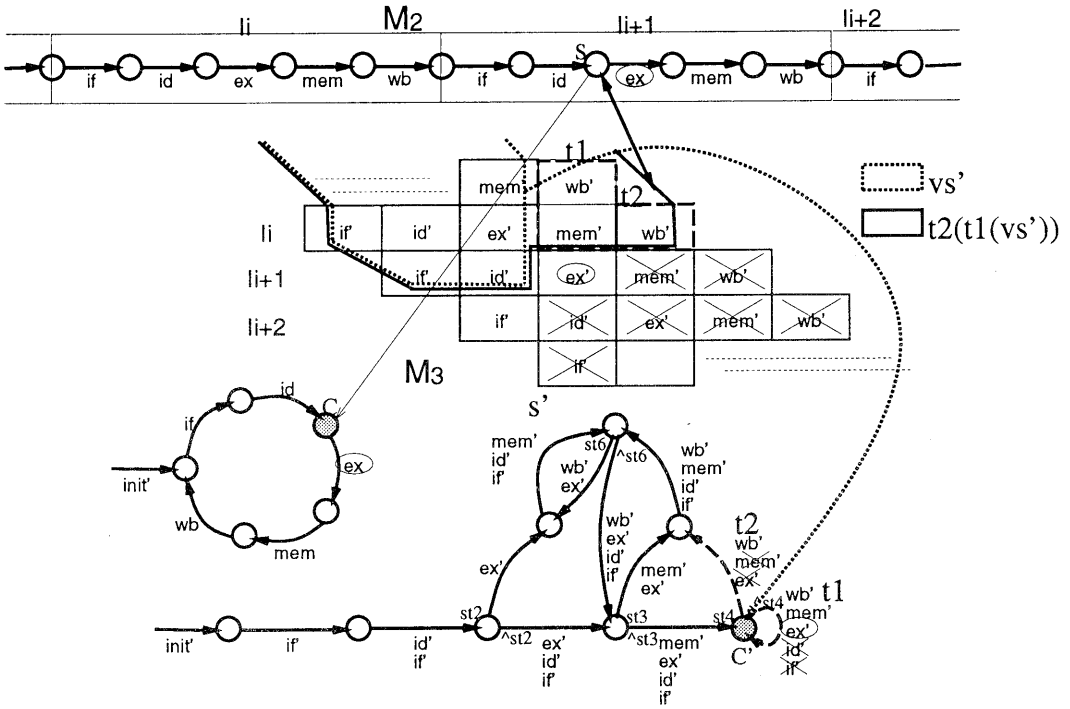


図 4: $M_2 - M_3$間の状態の対応

帰納段階の証明では，以下の検証条件 V2 が成り立つことを示す．

検証条件 **V2**

(V2-1) M_2，M_3の初期状態での各レジスタ $F_j(s)(1 \leq j \leq q)$ が，同じ値をとるものと仮定する．M_2，M_3の初期状態から最初の命令を実行する遷移系列において，対応する stg_i，stg'_iの実行後の各 $F_{w(i)}$の値が等しい．

$\forall s \forall s' [F_{w(i)}(stg_i(s)) = F_{w(i)}(stg'_i(s'))$ (結論部)]

(V2-2) stg_iが実行される有限状態名を C．後行命令を実行せずに到達した，stg'_iが実行される有限状態名を C'とする．C'から直前命令の実行可能パスにそって，先行命令のみ実行する遷移系列の集合をξとする．また，有限状態名が C，C'であるような抽象状態をそれぞれ s，vs'とする．

このとき，各 stg'_i，各$\tau \in \xi$および各 $F_{w(i)}$に対して，以下の条件が成り立つ．

$\forall s \forall vs'$

$[COND$ (τ の実行条件) $\wedge$

$F_1(s) = F_1(\tau(vs')) \wedge F_2(s) = F_2(\tau(vs')) \wedge \cdots \wedge$

$F_q(s) = F_q(\tau(vs'))$ (仮定部)

$\Rightarrow F_{w(i)}(stg_i(s)) = F_{w(i)}(stg'_i(vs'))$ (結論部)$]$

$COND$は stg'_iの実行前の状態からτ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．

実際には，結論部に影響がないので，$F_{w(i)}(stg'_i(vs'))$ は $F_{w(i)}(stg'_i(s'))$ に，仮定部の各 $F_j(\tau(vs'))$ はそれぞれ $F_j(\tau(s'))$ にしてよい．すなわち，vs'の代わりに s'としてよい．

3 検証支援系を用いた検証実験と考察

ここでは，提案した証明法に基づいて作成した検証支援系，その支援系を用いて行った例題の 5 ステージ・パイプライン CPU の検証実験，およびその考察について述べる．

3.1 検証支援系の入出力と特徴的な機能

提案した証明法に基づいて作成した検証支援系の入出力は，以下のとおりである．

(入力) 逐次実行レベル M_2と実現レベル M_3の記述，証明の対象とする M_3のステージ stg'_iと stg'_iの直前命令の実行可能パスおよび場合分け (必要なら)

(出力) 検証条件 V2 の判定結果

場合分けは，ここでは，stg'_iの実行が開始してから直前命令の実行が終了するまでのパイプライン中の命令の実行系列の演算コードのことである．

この支援系では，次の (1), (2) の作業を順に自動で行う．(1)stg'_iの直前命令の実行可能パスをたどりながら，各レジスタ $F_{w(i)}$ ごとに，その場合の V2 を生成する，(2) 項書換え，P 文の真偽判定手続きをこの順に適用し，生成された V2 の恒真性の判定を行う．

支援系内部では，(1), (2) を次のように実現している．

(1) 直前命令の実行可能パスの実行条件 $COND$,

結論部 $F_{w(i)}(stg_i(s)) = F_{w(i)}(stg'_i(s'))$ およびそれらの項書換えに必要な書換え公理の集合 COND_TR_axioms, conc_TR_axioms に分けて，V2 を生成する．実際には，結論部の左辺は，M_2の対応するステージ stg_iの動作内容の公理を書換え公理として項書換えを行い，そのあと仮定部を書換え公理として項書換えを行ったものである．右辺も，stg'_iの動作内容を書換え公理として項書換えを行ったものである．COND_TR_axioms は，直前命令の実行が終了するまでの M_3の複合遷移の動作内容の公理からなり，conc_TR_axioms は，直前命令の実行が終了するまでの M_3の複合遷移の動作内容の公理のうち後行命令の実行に関するものを nop に置き換えたものからなる．

(2) 各レジスタの値を s'で表すよう，COND，結論部の項書換えを，それぞれ書換え公理の集合 COND_TR_axioms, conc_TR_axioms を用いて行う．最後に，P 文の真偽判定手続きを用いて，

$\forall s'$[(項書換え後の $COND$)
⇒ (項書換え後の結論部)]

の恒真性の判定を行う [5].

(1) では，COND_TR_axioms, conc_TR_axioms を生成する際に，検証者が適当に与えた場合分けによって，書換え公理の右辺のうち起こり得ない場合の部分式を取り除き，簡単化を行う．

例えば，例題の CPU の検証の際に，書換え公理として，

```
A(T1(s)) ==
(if (get_mt(get_op(IR(s))) = ALU and
get_nt(get_op(IR(s))) = LOADI) then 0
else if (get_mt(get_op(IR(s))) = BRA and
get_nt(get_op(IR(s))) = SPJMP) then 0
else if (get_src1(IR(s)) = 0) then 0
else r_get(RF(s),get_src1(IR(s))))
```

というものが現れる．if 文をその条件部によって分解するルーチンによって，この右辺の if 文は以下の 4 つの式に分割される．

```
条件1: (get_mt(get_op(IR(s))) = ALU and
       get_nt(get_op(IR(s))) = LOADI)
本体1: 0

条件2: (not(get_mt(get_op(IR(s))) = ALU and
       get_nt(get_op(IR(s))) = LOADI)) and
       (get_mt(get_op(IR(s))) = BRA and
       get_nt(get_op(IR(s))) = SPJMP)
本体2: 0

条件3: (not(get_mt(get_op(IR(s))) = ALU and
       get_nt(get_op(IR(s))) = LOADI)) and
       (not(get_mt(get_op(IR(s))) = BRA and
       get_nt(get_op(IR(s))) = SPJMP)) and
       (get_src1(IR(s)) = 0)
本体3: 0

条件4: (not(get_mt(get_op(IR(s))) = ALU and
       get_nt(get_op(IR(s))) = LOADI)) and
       (not(get_mt(get_op(IR(s))) = BRA and
       get_nt(get_op(IR(s))) = SPJMP)) and
       (not(get_src1(IR(s)) = 0))
本体4: r_get(RF(s),get_src1(IR(s)))
```

get_mt(get_op(IR(s))) = LOAD の場合を考えれば，条件 3 および 4 のみ起こり得る．そこで，右辺は，(if 条件 3 then 本体 3 else if 条件 4 then 本体 4 else PC(s)) に簡略化される．

(2) では，項書換えを一回行うたびに，上述の if 文分解ルーチンを用いて，項書換え後の if 文が含まれた式を分解する．これによって，最終的に P 文真偽判定ルーチンで判定する式の式長が短くなる．

本支援系は，C 言語を用いて作成されており，そのプログラムサイズは約 14,000 行である．

3.2 検証実験および考察

例題の CPU の証明の一部が自動で行えることを確認し，どの程度の CPU 時間がかかるかを測定した．

上述の検証支援系では，M_3のステージ stg'_i, stg'_iの直前命令の実行可能パス，レジスタ $F_{w(i)}$ごとに，設計ミスがわかる．そのデータをもとに記述修正を行った．記述修正を行ったのは，フォーワーディングを行うステージ if', id' の動作内容の記述のみである．フォーワーディングを行うステージでは，現在の命令のソースレジスタのインデックス，先行命令が RF に書き込む順番，先行命令のデスティネーションレジスタのインデックス，演算コードを考慮して，どの値を転送するか決定しなければならないので，記述がかなり複雑になっているためと思われる．ちなみに，フォーワーディングを行わないステージ ex', mem', wb' の動作内容の記述は，各々 M_2の対応するステージの動作内容の記述と全く同じである．

パイプラインが定常状態 S4 付近にあり，複雑なフォーワーディングを行うステージに対する証明が成功したときの CPU 時間 (一部を抜粋) を，表 3に示す．参考のため，フォーワーディングを行わないステージに対するものも示してある．

フォーワーディングを行うステージでは，先行命令の数が多いほど記述が複雑になる．また，直前命令の終了する系列も長くなるため，COND，結論部の項書換えに要する公理の集合も，それぞれかなり大きなものとなる．このため，場合分けを行わざるを得なかった．今回，場合分けは，ステージの実行が開始してから直前命令の実行が終了するまでのパイプライン中の命令の実行系列の演算コードにより行った．実行系列として，例題の CPU に与えるプログラム中に頻繁に現れる系列 (ループ制御等) を選んだ．

かなりの系列に対しては数秒程度で証明が行えたが，一部の系列に対しては数十分程度かかった．これは，演算コードで場合分けを行っても，書換え公理の右辺の部分式がそれほど取り除けないためである．例えば，フォーワーディングを行うステージの動作内容は，その命令の演算コードのみならず，ソースレジスタのインデックスおよび先行命令のデスティネーションレジスタのインデックスにも依存するため，起こり得る部分式がかなりある．そこで，ソースレジスタ，デスティネーションレジスタのインデックスによる場合分けも行ってさらに場合分けを細かくしたところ，場合分けの数は多くなるが，どの場合についても，数秒程度で証明を行うことができた．

演算コードで場合分けを行った場合，11 個の命令を考えているため，完全に証明を行うためには，例えば，ステージ T4-if', 直前命令の実行可能パス T3-T4-T11-T10 に対しては，11^4(=約 15,000) 通り

ステージ パス	場合分け	検証に要した CPU 時間 (秒)		ステージ パス	場合分け	検証に要した CPU 時間 (秒)	
T3-if' T3-T4-T11-T10 • 先行命令は 3 つ • forwarding あり • 場合分けでは, ストールのため, 4 番めは必ず LOAD	LOADI NOT ADD LOAD SEQ	(a) (b) (c) (d)	2.10 1.53 0.02 0.03	T4-id' T4-T4-T11 • 先行命令は 3 つ • forwading あり • 場合分けでは, ストールのため, 5 番めは必ず LOAD	NOT ADD ADD BEQ LOAD	(a) (b) (c) (d)	2.32 2.43 72.70 611.27
	SPJMP STORE LOADI LOAD ADD	(a) (b) (c) (d)	2.00 1.50 0.02 0.02		SPJMP STORE LOADI AND LOAD	(a) (b) (c) (d)	2.15 0.70 18.80 3.28
	SGT NOT BEQ LOAD ADD	(a) (b) (c) (d)	2.33 6.97 1283.82 82.05		SGT NOT BEQ STORE LOAD	(a) (b) (c) (d)	2.15 0.70 9.65 3.40
T4-ex' T4-T4 • 先行命令は 2 つ • forwarding なし	なし	(a) (b) (c) (d)	0.02 0.03 2.22 0.27	Pentium-Pro 200MHz, メモリ 64MB 使用			

"パス" は, ステージの直前命令の実行可能パスを表す." 場合分け" は, ステージの実行が開始してから直前命令の実行が終了するまでのパイプライン中の命令の実行系列の演算コードにより行った. ただし, 最後の if' ステージで実行される命令は, その演算コードに依存した動作を行わないので, 実行系列に含めていない. 検証に要した CPU 時間の内訳 (a), (b), (c), (d) は, それぞれ検証条件の生成, COND の項書換え, 結論の項書換え, P 文の真偽判定に要した CPU 時間を表す.

表 3: 複雑な場合における検証に要した CPU 時間 (一部を抜粋)

の場合分けを行わなければならない. したがって, 完全に証明を行うためには, 実現アーキテクチャに依存して, 検証者がうまく場合分けを行って, 場合分けの総数が多くなく, かつ, それぞれの場合の検証時間が短くなるようにすることが必要である.

4 おわりに

本稿では, [7] で提案した代数的手法を用いて in-order 実行パイプライン CPU の正しさを証明する手法に基づいて, 検証支援系を作成した. 本支援系を用いて, 命令セットが 11 命令からなる 5 ステージ・パイプライン CPU の正しさの証明の一部を自動で行うことができた.

今後の課題として, 場合分けの総数を減らして, 完全な証明が行えるか, また, 比較のため, [2] で扱われているような 3 ステージ・パイプライン CPU の完全な証明が可能か調べたい.

謝辞 御討論を頂いた本学北道淳司助手, 設計例を参考にさせて頂いた NTT CS 研並列処理アーキテクチャ研究グループの皆様に感謝します.

参考文献

[1] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill: "Symbolic Model Checking for Sequential Circuit Verification", IEEE transactions on computer aided design of integrated circuits and systems, Vol 13, No. 4, pp.401–424 (1994-08).

[2] Mandayam K. Srivas, Steven P. Miller: "Applying Formal Verification to a Commercial Microprocessor", Proc. of the 12th IFIP Intl. Conf. on Computer Hardware Description Languages and their Applications (CHDL'95), pp.493-502(1995).

[3] D. Cyrluk, S. Rajan, N. Shankar, M.K. Srivas, "Effective Theorem Proving for Hardware Verification", Proc. of the 1994 Conference on Theorem Provers in Circuit Design (TPCD94), Vol. 901 of Lecture Notes in Computer Science, pp.203-222, Springer Verlag(1994).

[4] J. Kitamichi, S. Morioka, T. Higashino and K. Taniguchi: "Automatic Correctness Proof of the Implementation of Synchronous Sequential Circuits using an Algebraic Approach", Proc. of TPCD94, Vol. 901 of Lecture Notes in Computer Science, pp.165-184, Springer Verlag(1994).

[5] 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).

[6] 北嶋 暁, 森岡澄夫, 島谷 肇, 東野輝夫, 谷口健一: "代数的手法を用いた CPU KUE–CHIP2 の段階的設計の正しさの自動証明", 信学論 D-I, Vol.J79-D-I, No.12(1996) to appear.

[7] 島谷 肇, 森岡澄夫, 北嶋 暁, 東野輝夫, 谷口健一: "代数的手法を用いたパイプライン方式 CPU の設計検証", 情処研報, DA79-02(1996).