設計自動化　40－18
(1987. 12. 18)

Ｔｏｋｉｏによる論理設計支援システム

A Logic Design Assistance System Based On A Temporal Logic Based Language: Tokio

†藤田 昌宏　†藤沢 久典　‡中村 宏　‡田中 英彦
Masahiro FUJITA　Hisanori FUJISAWA　Hirosi NAKAMURA　Hidehiko TANAKA

(† 富士通研究所)　(‡ 東京大学 工学部)

FUJITSU LABORATORIES LTD.　*Faculty of Engineering, University of Tokyo*

あらまし　時相論理型言語Ｔｏｋｉｏを中心とした論理設計支援システム（第一次）について述べる．動作だけでなく，データパス等の構造情報も利用することで，高品質な合成を行うことを目標とする．Ｔｏｋｉｏをそのまま用いて自由に記述すると，ハードウェアとの対応が取りにくくなるため，レジスタ転送レベルに制限したＲＴＬ－Ｔｏｋｉｏを設定し，それによる動作記述とデータパス記述の間の実現可能性を解析し，パスの同時二重使用のチェックを行う．

Abstract A logic design assistance system based on a temporal logic based language: Tokio is presented. It uses both behavioral and structural (datapath) description in Tokio, in order to accomplish high quality synthesis. It is difficult to make a correspondence between behavioral descriptions with Tokio and real hardware components, and so a subset of Tokio, which expresses register transfer level design description and is called RTL-Tokio, is defined and used as a design description language. The assistance system checks whether the behavior in RTL-Tokio can be realized by the given datapath by analyzing path usage.

１．はじめに

本稿では，時相論理型言語Ｔｏｋｉｏ［１］を中心としたハードウェア論理設計支援システムについて述べる．ここでの基本的な考え方は，設計者は設計対象ハードウェアの動作だけでなく，データパス（構造）についても設計のかなり初期の段階から意識しており，この構造情報をうまく利用することが高品質な合成への鍵だということである．

ハードウェア設計の流れとしては，図１に示すものを考えている．まず，設計者は設計したいハードウェアの動作アルゴリズムを記述する．次に，シミュレーション等で動作の確認を行いながら記述を徐々に詳細化し，レジスタトランスファレベルの動作記述とする．これらは，時相論理型言語Ｔｏｋｉｏを用いて記述されるが，レジスタトランスファレベルの記述は，ハードウェアとの対応が比較的容易に取れ，自動合成や検証が可能なようにＴｏｋｉｏを制限したＲＴＬ－Ｔｏｋｉｏ（３章で定義する）を用いて行う．設計者は同時に必要なら，データパス等の構造の指定をＰｒｏｌｏｇで記述する．従来から，動作記述からデータパスを自動生成する研究が行われているが，結果の品質は必ずしも十分ではない．反面，設計者自身は，過去の経験により，設計の初期の段階からデータパスをある程度意識していることが多い．そこで，例えばＤＤＬ［２］を用いた場合に高品質な自動合成を行うためには，設計者が TERMINAL 変数を使って構造を十分意識した記述を行うという方法が取られる．しかし，これでは動作記述の中に構造記述を埋め込む形になっており，分かりにくいだけでなく，設計変更も容易ではなくなる．そこでここでは，動作記述と構造記述は別に与え，支援システムが自動的に対応を取るようにする．このようにすれば，設計者の考えを自然に表現でき，また，設計の再利用も容易になる．

動作記述は，おもに制御系の検証や，構造記述との間で実現可能性のチェックが行われ，そののち必要ならシステムレベルのシミュレーションによる性能の総合評価が行われ，回路合成へと進む．

ここでは，第一次システムとして，動作記述とデータパス記述の実現可能性解析に重点を置く．２章で支援ツールの構成を述べ，３章でＲＴＬ－Ｔｏｋｉｏを定義し，４章以降で中心となるツールについて説明する．なお，以下ではＰｒｏｌｏｇは既知とする．

2．論理設計支援環境の構成

第一次支援システムの構成を図２に示す．アルゴリズムの記述からレジスタトランスファレベルの記述への変換は人手で行い，シミュレータのみ用意する．このシミュレータはＴｏｋｉｏをＰｒｏｌｏｇへコンパイルして実行するものであり，様々なＰｒｏｌｏｇ上で稼働している［１］．データパスの記述はＰｒｏｌｏｇで行うが，ＴｏｋｉｏはＰｒｏｌｏｇを拡張したものであり，Ｐｒｏｌｏｇを含んでいるので，動作記述も構造記述もＴｏｋｉｏを用いていることになる．

レジスタトランスファレベルの動作記述とデータパスの記述はトランスレータによりインターバル遷移表／ファシリティ使用表に変換される．これらの表を用いてデータパスの各パスの使用をインターバル遷移（状態遷移）に基づいて解析することで，動作と構造記述の実現可能性をチェックする．また，論理回路レベルの高速シミュレータと結合するため，回路への単純変換（最適化なし）も行う．さらに，制御系については，すでに開発済である時相論理検証プログラム［３］と接続する．以上の構成により図１の設計の流れを支援する．

3．ＲＴＬ－Ｔｏｋｉｏ

3．1　時相論理とＲＴＬ－Ｔｏｋｉｏ

Ｔｏｋｉｏは簡単には，時相論理に基づくＰｒｏｌｏｇと

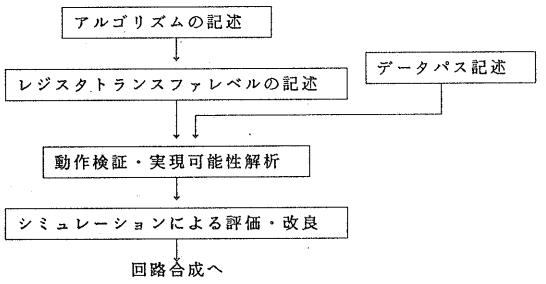


図１　論理設計の流れ

いうことができ，一般のプログラミング言語のような記述も行える．このため，Ｔｏｋｉｏで自由に記述されたものから自動合成するのは，簡単ではないし，また，合成結果の品質も高いものとはならない．そこで，ハードウェア記述として比較的容易に合成や検証の処理が行え，かつ十分柔軟に記述できるレベルとして，レジスタトランスファレベルを考え，ＲＴＬ－Ｔｏｋｉｏを設定する．以下の論理設計支援システムはこのＲＴＬ－Ｔｏｋｉｏの記述を対象とする．設計者は，必要なら始め通常のＴｏｋｉｏでアルゴリズム中心の記述を行い，シミュレーション等で設計を確認しながら，以下に示すＲＴＬ－Ｔｏｋｉｏに変換していく．ここでは，紙面の都合上，もとのＴｏｋｉｏを詳しく紹介できないので，文献［１］も合わせて参照願いたい．

Ｔｏｋｉｏのシンタックスは Ｐｒｏｌｏｇのそれと同じであるが，セマンティックスは時間の概念が入っているため大きく異なる．Ｐｒｏｌｏｇの変数は一度値が決まるとそれを保持し続けるのに対し，Ｔｏｋｉｏの変数は，各時刻内ではＰｒｏｌｏｇと同じように扱われる一方，時刻が変われば値を変えてよい．組合せ回路の仕様を記述する際には，時間の概念は必要ないのでＰｒｏｌｏｇで自然に記述できるが，順序回路を記述する際には，時間と共に値を変えていく動作を表現しなければならず，Ｔｏｋｉｏが有効になる．Ｔｏｋｉｏの時間は離散時間であり，最小時間単位が存在する．この最小時間単位をクロックと考えれば，同期回路にそのまま対応し，絶対時間と結び付ければ，ハードウェアの細かい動作（１クロック内の非同期動作等）の記述も行える．さらに，組合せ回路でも非同期な動作の処理手順を指定したい時には，処理順序を陽に指定できるＴｏｋｉｏの方が有利である．現状では，大規模な組合せ回路を一度に品質よく自動合成するのは極めて困難であり［４］，意味を考えた回路構成等を設計者が指定できる方が実際的である．このような場合，Ｔｏｋｉｏを利用すると，後述するようにインターバルを区切るという考え方で遅延情報も含めて自然に表現できる．

上で述べたような，最小時間単位を仮定して動作を記述すること自体は従来のハードウェア記述言語（例えば，同期回

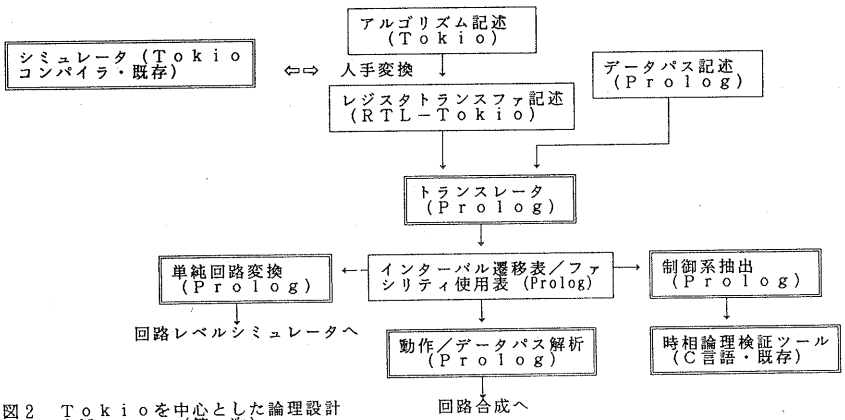


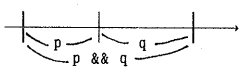
図２　Ｔｏｋｉｏを中心とした論理設計支援システム（第一次）

路ならＤＤＬ）でも行われていたことである．これに対し，Ｔｏｋｉｏはより柔軟な記述が行えるように各時刻単位のみでなく，連続した時刻の集まり（要するに時区間であり，インターバルと呼ばれる）を単位としても扱えるようになっている．各インターバルの切れ目をクロックとすれば同期回路であり，クロックに関係なく各種の動作の終了を切れ目とすれば非同期回路を表現することになる．

Ｐｒｏｌｏｇが通常の論理に基づいているように，Ｔｏｋｉｏもインターバルにより拡張された論理である区間時相論理(Interval Temporal Logic) ［５］を基礎としている．区間時相論理では，インターバルを取り扱うための様々な演算子(Temporal Operator) が定義されている．最も基本的な演算子として，任意のインターバルを前半と後半の２つに区切る chop がある．Ｔｏｋｉｏでは「&&」で表現し，

```
p && q
```

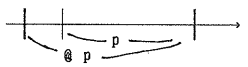
は，現在のインターバルを前半と後半の２つに分け，前半で pを実行し，後半で qを実行することを示す．



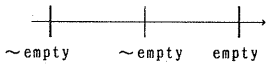
もう１つの基本的な演算子として next がある．Ｔｏｋｉｏでは，「@」で表現し，

```
@ p
```

は，次のインターバル（次の時刻から始まり，現在のインターバルと同じ時刻に終了する）で pが成り立つことを示す．ここで，次とは，同期動作ならば「次のクロック」のことであり，非同期動作ならば最小時間単位後の時刻である．



さらに， chop と next で定義可能であるが，重要な演算子として，empty がある．これは，現在のインターバルが終了する時刻のみで真となるものである．



これらを用いて， ♯, keep, stop, more, <- 等の様々なインターバルに関する演算子を定義できる［５］．

♯ p:現在のインターバルの全ての時刻で pが成り立つ

keep(p):現在のインターバルの最後の時刻を除く全ての時刻で pが成り立つ

stop(p): pが成り立つと現在のインターバルを終了する

more(p): pが成り立つと現在のインターバルの終了を少なくとも１時刻延期する

p <- q: 現在のインターバルの最初の時刻の qの値を最後の時刻の pの値とする（ユニフィケーション）．レジスタ転送の一般形と考えられる

何も演算子がないものは「現在に」そのことが成り立つと解釈し，それらの条件は localであると呼ばれる．Ｐｒｏｌｏｇでは時刻は１つしかないと考えられ，全て localな条件で構成されていると言える．これに対しＴｏｋｉｏでは，個々の時刻内ではＰｒｏｌｏｇと同じように変数等を扱うが，複数の時刻間は時相演算子により制御するようになっている．言い換えると，Ｔｏｋｉｏを全て localな条件で記述したものがＰｒｏｌｏｇである．Ｔｏｋｉｏでは，インターバルとプレディケイトを対応させる．例えば，

```
p :- q && r.
```

において， pを実行することは，インターバル pを生成し，それをさらにインターバル qとインターバル rに分けてそれぞれで，プレディケイト q, r を実行することに対応する．このように，プレディケイト名をインターバル名と考える．

さらに，Ｔｏｋｉｏではハードウェア記述を簡潔に行えるように，どのプレディケイトからでも参照できるグローバル変数も用意されている．グローバル変数は先頭が「＊」で始まるものであり，通常はレジスタ，メモリ，外部端子等に対応する．グローバル変数に対する代入は「＝」に対応するものとして「：＝」（現在時刻での代入）が，「＜－」に対応するものとして「＜＝」（インターバルの最初での値をと最後での値として代入）がある．

このようにＴｏｋｉｏはＰｒｏｌｏｇを含んでいる訳であり，プログラミング言語としても強力である．これは，Ｔｏｋｉｏによる自由な記述とハードウェアとの対応をとることは極めて難しいことを意味する．そこで，ハードウェアとの対応が比較的簡単に取れるＲＴＬ－Ｔｏｋｉｏを設定し，設計者の責任により，そこまで変形してもらうことにする．

ＲＴＬ－Ｔｏｋｉｏとは，Ｔｏｋｉｏを次の形に制限したものである．

```
head(Vars):- localConds, !, actions.
head(Vars):- localConds, !, actions && actions &&
                ・・・ && predCall.
```

localCondsは，Ｐｒｏｌｏｇの範囲で表現されたものであり，現在時刻のみで判断できるものであり，なくても構わない．後にカット「！」があるので，各クローズが起動される条件を表現することになる．同じクローズが複数ある場合には，上から順に評価し，後のクローズは前のクローズの条件の否定が入っているとみなす．例えば，

```
p:- cond1, !, action1.
p:- cond2, !, action2.
```

は，次と同じである．ただし，notcond1はcond1 の否定である．

```
p:- cond1, !, action1.
p:- notcond1, cond2, !, action2.
```

グローバル変数の内，特に制御信号（１ビットの信号）については，条件判定や現在時刻に対する代入として，通常の記述法以外に，論理式の形で書くことも認める．カット「！」の前は条件として，後は代入として解釈される．代入については，記述された論理式が真になるように各変数の値が設定される．例えば，*aと書けば，*a:=1 の意味とし，not *aと書けば*a:=0 の意味とする．

また，actions には，上で示した各種演算子やＰｒｏｌｏｇのクローズが０個以上「，」で区切って入るが，次の形に限る．ただし，「@」は以下の各々に対し，前にいくつついてもよい．

```
stop(localConds), more(localConds),
keep(localActionWithNext),
#(localActions),
variables <- localActions
globalVariables <= localActions
localActions
```

(以上の他に線型時相論理(Linear Time Temporal Logic)の演算子も認めるが，本稿の例では使用しないので，混乱を避けるために省略する．ＲＴＬ－Ｔｏｋｉｏの時相論理による正式な定義は別途発表することとしたい) stop や more は現在のインターバルの終了を決定するものなので，現在時刻内で真偽が決まる条件以外は用いられない．反対に keep や #は実行を表すので，代入や演算等を用いる．

ＲＴＬ－Ｔｏｋｉｏの制限の中心は，各クローズの起動条件に local なものしか認めないことである．これは，ハードウェアが未来の条件を使って現在の計算を行うことはないので，極めて自然なものである．これ以外の制限，例えば時相演算子のネスティングを制限している点は，処理系の作り易さからのものであり，本質的ではないが，ハードウェアを記述する場合には，これで十分であると考えている．

もう１つの制限は，ＲＴＬ－Ｔｏｋｉｏでは，長さの指定が陽にはないインターバルの長さを次のように決めることである．まず，stop，moreがある場合には，終了条件が指定された不定長のインターバルとする．ない場合には，他のインターバルを起動しないインターバルは全て長さを１とし，他のインターバルを起動するものは，その起動されたインターバルの長さと合うように自身の長さを決める（５章の例参照）．これは，<-や，<=等の基本動作を時間１で行うことを仮定することに対応し，レジスタトランスファレベルとしては自然な制限である．

なお，ＲＴＬ－Ｔｏｋｉｏの厳密な論理的定義は Wolper のExtended Temporal Logic［６］をさらに拡張したもので行う予定である．

３．２　８ビットマイクロプロセッサの記述

ＲＴＬ－Ｔｏｋｉｏの範囲内で，８ビットマイクロプロセッサ MC6502 を記述した．結果はコメントも含めて約 600 行となった．その一部を図３に示す．ほとんどレジスタ転送のみで記述されている．

さて，この記述のシミュレーションは，アッカーマン関数 ackerman(1,1)を計算する機械語をつくり，それをメモリにロードして実行することで行った．通常のハードウェア記述言語では，シミュレーション環境の記述を行い難い場合も多いが，ＴｏｋｉｏではＰｒｏｌｏｇとして利用することで，極めて柔軟な記述が行える．今の場合には，図４のようにしてシミュレーションを行っている．sim(Filename) を実行すると，最初の時刻にファイルFilenameからインテルフォーマットの機械語を読み込みメモリにセットし，各種レジスタの初期設定を行った後，CPU 時間の計測とともに，マイクロプロセッサ MCS6502のハードウェア記述である mcs6502を実行している．このように，ハードウェアの記述とそれ以外の記述をクローズではっきり分けておけば，混同することなく柔軟なシミュレーションを実行できる．なお，レジスタの値の設定は，図４のようにプログラムに組み込むことはせず，シミュレーションの実行時に行うこともできる．

実行は，東京大学・田中研究室のSUN3/260上のSICSTUS PROLOG（スウェーデンのＳＩＣＳで開発された）上で行った．現在，ＴｏｋｉｏはＰｒｏｌｏｇにコンパイルされ，さらにそれがSICSTUS PROLOG上でコンパイルされて実行される．ackerman(1,1)を計算するには，MCS6502 のクロックで 323クロック必要であるが，そのシミュレーション時間は以下のようである．

Tokio →Prolog	Prolog compile	実行
５０秒	２３４秒	4.0 秒

(使用計算機：SUN3/260 8MByte メモリ)

上では，コンパイル時間が支配的になっているが，クローズごとにコンパイルできるので，設計修正があっても一度コンパイルしてあれば，修正された部分だけの再コンパイルで済む（数秒程度）．なお，Ｐｒｏｌｏｇのコンパイラを通さずにインタプリタで実行すると２０秒程度かかり，５倍ぐらい遅くなる．

```
mcs6502 :-
    *reset,!,
    int  &&      % Initial startup
    run1.
mcs6502 :-
    !,
    true  &&
    run1.
run1 :-
    *ready,!, true.
run1 :-
    !,
    *ifsync <= 1 &&      % Instruction fetch
    read(*pc,*ir) &&
    *pc <= *pc + 1 &&
    *ifsync <= 0 &&      % Execute
    run_decode(*ir/\binary("11")) &&
    int &&
    run2.
run2:-
    *so,!,
    v <= 1, fin(status_report) &&
    run.

run_decode(I):- I = binary("01"),!,
    I1 = (*ir>>5)/\binary("111"),group1(I1).
run_decode(I):- I = binary("10"),!,
    I1 = (*ir>>5)/\binary("111"),group2(I1).
run_decode(I):- I = binary("00"),!,group3(*ir).
run_decode(I):- I = binary("11"),!,opex.
```

図３　８ビットマイクロプロセッサ MCS6502の記述（一部）

```
sim(Filename) :-
    fname(Filename,".hex",Filename1),
    ld0(Filename1),!,
    *p := 0,
    *reset := 0,
    *s := 0,
    *b := 0,
    *so := 0,
    *y := 0,
    *x := 0,
    *dl := 0,
    *a := 0,
    *mem(hex("FFFC")) := 0,
    *mem(hex("FFFD")) := 0,
    *mem(hex("FFFE")) := 0,
    *mem(hex("FFFF")) := 0
&&
    X <- cputime,
    mcs6502,
    fin((X1 = cputime - X,nl,write(X1),nl)).

fname(Output,Option,Loutput) :-
    name(Output,L),
    append(L,Option,LC),
    name(Loutput,LC).

ld0(File) :-
    seeing(O),ld1(File,Lines),seen,see(O).

ld1(File,Lines) :-
    see(File), get0(Ch),lines(Ch),!,seen.
ld1(File,Lines) :- seen,
    !,fail.
```

図4　シミュレーション環境の設定

4．データパスの記述

先に述べたように，本システムの入力はＲＴＬ－Ｔｏｋｉｏによる動作記述とそれを実現するデータパス記述である．本章では，データパスの記述法について述べる．

ここで用いる形式では，パスは以下のようにＰｒｏｌｏｇのファクトの形で記述され，階層化された構造も記述できる．Ｐｒｏｌｏｇは前に述べたようにＴｏｋｉｏに含まれるので，結局データパスもＴｏｋｉｏで記述されることになる．

4．1　ｔｙｐｅ文

type(タイプ名，ファシリティ名，モジュール名).

これは，あるモジュール内で用いられているファシリティのタイプを宣言する．例えば，「CCというモジュールの中でBBという名前をもつファシリティのタイプはAAである」ということは，type(AA,BB,CC). と記述される．タイプ名としては，以下のものがある．

・入出力を表す input, output
・階層化された記述に対してサブモジュールを宣言するのに用いる module
　例えば，type(module,sub1,top). という記述は topというモジュールの中にsub1というモジュールがあることを表す
・レジスタを表す register （加算等の機能付レジスタは register-inc 等で表現する）
・論理ゲートはその論理名＋入力数で，and2, and4, multiplexer2のように表す

ここで，input, output, module の３つについては予約語であり，それ自身意味をもつが，他のタイプ名はそのタイプ名のもつ機能を後で述べるfunc文で宣言することにより，自分で定義する．ファシリティ名は任意であり，異なるモジュール内で同名のファシリティを用いてもよい．モジュール名に関しては，トップレベルは必ず top という名前を用いるが，それより下のレベルでは任意の名前を用いてよい．また，記述全体がトップレベルしかない（階層になっていない）時は，モジュール名 top を省略できる．

4．2　ｐａｔｈ文

path(パス名，入力ポート，出力ポート，モジュール名).

これは，ファシリティ間のパスの接続情報を記述する．パス名は任意であり，入出力ポート名は［ファシリティ名，端子名］で表す．また，モジュール名のところには，宣言するパスが存在するモジュール名（type文で宣言したもの）を入れる．ただし，type文と同様，トップレベルしかない時はモジュール名top を省略できる．

4．3　ｆｕｎｃ文

func(機能名，出力名，入力名，信号線名).

これは，type文で宣言したタイプのもつ機能を宣言する．機能名として，以下のものを用意してある．

set:　出力と入力を接続してデータを伝える
incr,decr:　値のインクリメント，ディクリメント
and,or,xor,etc.:　論理演算

また，出力名には出力ポート，レジスタの内部変数，入力名には入力ポート，レジスタの内部変数そして信号名には信号線ポート名がくる．例えば，

func(set,reg1,［reg1,in］,［reg1,set］).

は，reg1のset 端子が１になると，reg1の入力inの値を内部に取り込む（内部変数reg1にセットする）ことを意味する．

4．4　ビット幅の指定

データパスを記述する際，入出力ポートおよびレジスタのビット幅を指定する必要がある．ここでは，例えば，［reg1,［in,［0,7］］］とする．また，最初に，bit-width(0,7). と宣言すれば，それ以降の記述に関して特に記述のないものは８ビット幅と解釈する．

図５に示すデータパスを記述したものを図６に示す．

5．インターバル遷移表／ファシリティ使用表への変換

本システムでは，ＲＴＬ－Ｔｏｋｉｏの動作記述が与えられたデータパス上で実行可能かを検証する．その検証に際し，ＲＴＬ－Ｔｏｋｉｏの記述はインターバル遷移表へ，データパス記述は各状態におけるファシリティ使用表に展開される．本章では，この２つの表，およびそれらへの展開法を簡単な例を用いながら説明する．

ここで用いる例は，パイプラインマージソータ［７］の１段目であり，ＲＴＬ－Ｔｏｋｉｏによる記述は図７であり，

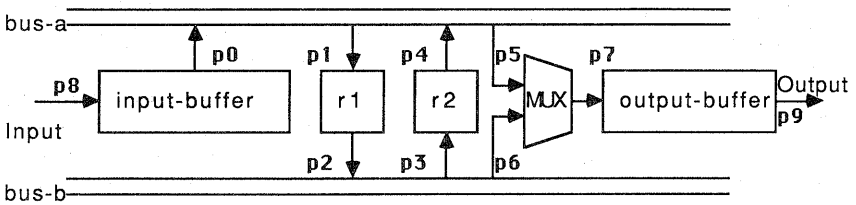


図5　パイプラインマージソータのデータパス

```
type(input,'Input').
type(output,'Output').
type(bus,bus_a).
type(bus,bus_b).
type(register,r1).
type(register,r2).
type(register,input_buffer).
type(register,output_buffer).
type(multiplexer,'MUX').

path(p0,[[input_buffer,out],[bus_b,in]]).
path(p1,[[bus_a,out],[r1,in]]).
path(p2,[[r1,out],[bus_b,in]]).
path(p3,[[bus_b,out],[r2,in]]).
path(p4,[[r2,out],[bus_a,in]]).
path(p5,[[bus_a,out],['MUX',in1]]).
path(p6,[[bus_b,out],['MUX',in2]]).
path(p7,[['MUX',out],[output_buffer,in]]).
path(p8,[['Input'],[input_buffer,in]]).
path(p9,[[output_buffer,out],['Output']]).

func(set,register_incr,
   [register_incr,in],[register_incr,set]).
func(incr,[register_incr,out],
     register_incr,[register_incr,incr]).
func(set,register,
     [register,in],[register,set]).
func(conn,[register,out],
     register,[register,conn]).
func(conn,[bus,out],[bus,in],[bus,cnt]).
func(conn,[multiplexer,out],
     [multiplexer,in1],[multiplexer,sel1]).
func(conn,[multiplexer,out],
     [multiplexer,in2],[multiplexer,sel2]).
```

図6　図5の記述

データパスは図5，6である．

5．1　インターバル遷移表

インターバルは従来の状態とほぼ同じように扱えるが，長さが1とは限らない点のみが異なる．インターバルの遷移は次のように記述する．

int-trans([呼ぶインターバル，インターバルの長さ]，
　　[呼ばれるインターバル，インターバルの長さ]，
　　遷移条件).

これは，あるインターバルが終了した後に続くインターバルを示す．ＲＴＬ－Ｔｏｋｉｏにおける&&に対応する．

int-call([呼ぶインターバル，インターバルの長さ]，
　　[呼ばれるインターバル，インターバルの長さ]，
　　遷移条件).

これは，あるインターバルの中で別のインターバルを生起させることを表す．もし，keepの中で他のインターバルを生起している時には，

int-call([呼ぶインターバル，インターバルの長さ]，
[keep, [呼ばれるインターバル，インターバルの長さ]]，
　　遷移条件).

となる．

int-stop([停止するインターバル，インターバルの長さ]，
　　停止条件).

これは，あるインターバルが終了する条件を表す．もし終了するべきインターバルが他のインターバルをint-callで生起させている時は，生起されたインターバルもその条件で終了する．

ＲＴＬ－Ｔｏｋｉｏをこのインターバル遷移表に展開するフローは次の通りである．

(1)　各インターバルに固有の名前を付ける．この名前がインターバル遷移表におけるインターバル名になる．

(2)　他のプレディケイトを呼ばず，かつkeep演算子を含まないインターバルの長さを1にする．

(3)　chop演算子はint-trans の記述に変換し，インターバルの中で他のプレディケイトを呼んでいるところをint-callに変換する．ここで，インターバルの長さは，

・「，」で結ばれたインターバル同士の長さは同じ
・「&&」で結ばれたインターバルの長さは単純に足算してボトムアップに決めていく．

例えば，

a:- b && c.　　c:- d,e.

の場合，プレディケイトe の長さが1に決まっていると，プレディケイトd の名がも1になる．もし，プレディケイトb の長さが2に決まっていると，プレディケイトa の長さは3になる．このようにすると，keep演算子を含むインターバル，およびkeep演算子を含むインターバルを呼ぶインターバル以外の長さは全て決定できる．もし，矛盾が生じた時には，インターバルの長さを延ばす方向で修正していく．

例えば，図7のＲＴＬ－Ｔｏｋｉｏの記述は図8のように展開される．

5．2　ファシリティ使用表

5．1で展開された各インターバルでの各ファシリティの使用状況を次のように記述する．

```
main :-
        %% ==== main0 ==== %%
    *req,!,
    *ack<= 0,
    *count,
    keep(if(*req)
          then (@*count = (not *count), load)),
    stop(*pipeline_Stop).

main :-
        %% ==== main1 ==== %%
    !,
    true,length(1)   %% the same as skip
&&
        %% ==== main2 ==== %%
    main.

load :-
        %% ==== load0 ==== %%
    !,
    *r1 <= *input_buffer,
    *r2 <= *r1
&&
        %% ==== load1 ==== %%
    if (*count) then sort.

sort :-
    %% ==== sort0 ==== %%
    !,
    *ack <= 1,
    if (*r1 < *r2)
     then (*output_buffer <= *r1)
     else (*output_buffer <= *r2, *r2 <= *r1)
&&
    %% ==== sort1 ==== %%
    *ack <= 1,
    *output_buffer <= *r2.
```

図7　パイプラインマージソータのＲＴＬ－Ｔｏｋｉｏによる記述

```
int_call([start,_],[main0,_],[req]).
int_call([start,_],[main1,1],[not,req]).
int_trans([main1,1],[main2,_],[]).
int_call([main2,_],[main0,_],[req]).
int_call([main2,_],[main1,1],[not,req]).
int_call([main0,_],[keep,[load0,1]],[req]).
int_trans([load0,1],[load1,2],[]).
int_call([load1,2],[sort0,1],[count]).
int_trans([sort0,1],[sort1,1],[]).
int_stop([main0,_],[pipeline_stop]).
```

図8　図7に対するインターバル遷移表

f-trans(ファシリティの名前，入力パス，条件，インターバル名，id-number).

ファシリティ使用表への変換のフローは次のようになる．

(1) ＲＴＬ－Ｔｏｋｉｏの記述中のデータ転送の部分でsourceRegister,destinationRegister，およびその間に行われるデータ操作を見つける．ここで，RegisterはＲＴＬ－Ｔｏｋｉｏの記述では，グローバル変数であるか，<-によってのみ値を代入される変数である．

(2) そのデータ操作を実現するファシリティ（あるいはその集合）をデータパス記述のfunc文から捜す．

(3) sourceRegisterからそのファシリティ（の集合）へのパス，およびそのファシリティ（の集合）からdestination-Registerへのパスを見つける．

id-number はファシリティの使用にいくつかの候補がある時に用いる．例えば，図9のようなデータパス上でmainという長さが１のインターバルで，*b <= *aというデータ転送が条件load-ok で実行される時には，

f-trans(*b,p1,［load-ok］，［main,1］,1).

f-trans(bus-a,p2,［load-ok］，［main,1］,2).

f-trans(*b,p3,［load-ok］，［main,1］,5).

と記述し，実際にはid-number の１または（２かつ５）が成り立つことを管理しておく．

以上から，図５と図７から図１０のファシリティ使用表が得られる．

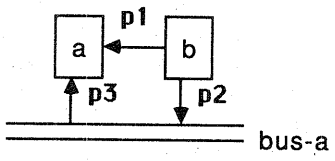


図9　２つのパスが選択できるデータパスの例

```
f_trans(*r1,p1,[],[load0,1],1).
f_trans(bus_a,p0,[],[load0,1],2).
f_trans(bus_b,p2,[],[load0,1],3).
f_trans(*r2,p3,[],[load0,1],4).
f_trans(*output_buffer,p7,[],[sort0,1],5).
f_trans('MUX',p5,[],[sort0,1],6).
f_trans('MUX',p6,[],[sort0,1],7).
f_trans(bus_b,p2,[],[sort0,1],8).
f_trans(*r2,p3,[>=,*r1,*r2],[sort0,1],9).
f_trans(bus_a,p4,[],[sort0,1],10).
f_trans(bus_a,p4,[],[sort1,1],11).
f_trans('MUX',p5,[],[sort1,1],12).
f_trans(*output_buffer,p7,[],[sort1,1],13).
```

図１０　図６，７に対するファシリティ使用表

6，動作／データパス解析

本章では，与えられたデータパス上でＲＴＬ－Ｔｏｋｉｏの記述が実行できるか否かを検証する手法について述べる．

問題となるのは，あるファシリティに対し複数のデータ転送がある場合である．それは次の４通りに分類される．

	インターバル	入力パス	実現
①	同じ	同じ	可能
②	同じ	異なる	不可能
③	異なる	同じ	可能
④	異なる	異なる	？？？

①，③の場合は，入力されるデータが同じなので問題はないが，②の場合は明らかにデータが衝突してしまう．一方，④の場合は，その異なるインターバルが同時に起こり得るかを調べる必要がある．そこで，まずファシリティ使用表をみて②と④の場合を捜し，④の場合はさらにインターバル遷移表を用いて調べればよい．②の場合はファシリティ使用表から明らかに分かるので，以降は④の場合について話を進める．

異なるインターバルが同時刻に存在するか否かは，それぞれのインターバルから同じ長さだけ時刻を遡った時に，同じ状態にたどりつくか否かで判定できる．この判定は，次の３つの操作から構成される．

(1) 時刻の遡り

現在のインターバルから１時刻前に起こり得る全てのインターバルを捜し出す．もし，同じインターバルにたどりついたら(2)へいき，そうでなければ(3)へいく．１時刻前のインターバルを得ることができなくなれば終了する（問題の２つのインターバルは同時には起こり得ない）．

(2) 排他的遷移条件の検出

同一インターバルにたどりついても，そのインターバルから問題の２つのインターバルに遷移する条件が排他的であれば，やはりその２つのインターバルは同時に起こり得ないことになるので，これを調べる．排他的であれば(3)にいき，排他的でなければ「与えられたデータパス上で実行不可能」という結論を出して終了する．

(3) ループの検出

現在のインターバルから１時刻前のインターバルを捜し出す(1)の操作が実は以前にした(1)の操作と全く同じになる場合がある．これはループなので，終了する（問題の２つのインターバルは同時には起こり得ない）．そうでなければ１時刻前のインターバルを現在のインターバルとして，(1)にもどる．

また，(1)の与えられたインターバルから時刻を遡る時には，同じインターバルを全て見つけてから，１時刻前のインターバルに遡るようにし，次のアルゴリズムを用いる．

・与えられたインターバルと同時刻のインターバルを見つける

(a) int-tran文から見つける．この時，得られた（遡った）インターバルの長さだけ，前の時刻へそのインターバルは続くので，さらに時刻を遡る時にはそのように処理する．

(b) keep演算子を含むインターバルの長さは決定できないので，そのインターバルの１時刻前も同じインターバルの場合と前のインターバルの場合の両方を調べる．

・与えられたインターバルから１時刻前のインターバルを見つける

(c) state-call文から

state-call([???,__], [与えらえたインターバル, __], [__]).

を見つけ，その??? のが捜すべきインターバルになる．この時与えられたインターバルと得られたインターバルの生起時刻がずれている場合は（state-call文において@演算子が入っている場合），時刻を遡る時に(a)と同じ処理をする．

例題として，図７に対して，load0 とsort0 が同時に起こり得るかを調べたトレースを図１１に示す．この図より，１時刻遡れば，main0 というインターバルになることが分かる．もし，条件が排他的であれば，さらに遡ることになるが，その時も４時刻遡れば，今度は(3)で述べたループになるので，そこで処理を終了する．

７．おわりに

レジスタトランスファレベルを記述するＴｏｋｉｏとして，ＲＴＬ－Ｔｏｋｉｏを設定し，それによる動作記述とデータパス記述を入力とする論理設計支援について，動作／データパス解析を中心に述べた．現在，８ビット・マイクロプロセッサ MCS6502を例題として実行中である．

参考文献

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

[2] J.R. Juley et al.: A Digital System Design Language (DDL), IEEE Trans. on Computer, Vol.C-17, No.9, 1968.

[3] H. Nakamura et al.: Temporal Logic Based Fast Verification System Using Cover Expressions, VLSI '87, 1987.

[4] R. Brayton et al.: Multi-Level Logic Optimization System, Proc. ICCAD-86, 1986.

[5] B. Moszkowski: A Temporal Logic for Multi-Level Reasoning about Hardware, IFIP 6th CHDL, 1983.

[6] P. Wolper: Temporal Logic Can Be More Expressive, 22nd Annual Symposium on Foundation of Computer Science, 1981.

［７］喜連川他： パイプラインマージソータの構成，電子情報通信学会論文誌，J66-D，1983.

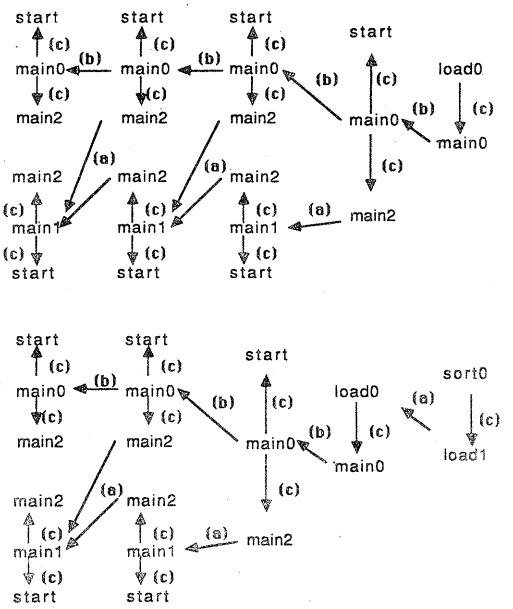


図１１ 動作／データパス解析におけるトレース例