計算機アーキテクチャ 121－9
設 計 自 動 化 82－9
（1996. 12. 12）

マイクロプロセッサ記述言語 PDL に基づく設計支援システム

鶴田 三敏, 阿部 公輝

電気通信大学 情報工学科

〒182 東京都 調布市 調布ヶ丘 1-5-1

E-mail: tsurut-m@acorn.cs.uec.ac.jp, abe@cs.uec.ac.jp

あらまし

クロックを意識した振舞い記述, 命令セット・マシンコードとその命令に対応する動作の記述からプロセッサの制御とデータパスの Verilog-HDL の RTL 記述および、アセンブラのソースプログラムを生成することができるプロセッサ記述言語 PDL を考案した。PDL は、逐次実行動作やパイプライン動作を指定することのできるタスクを単位にタスクから別のタスクを呼び出すことで複雑な動作を記述できる。

本稿では、さらに、逐次実行動作やパイプライン動作の PDL 記述を Verilog-HDL の RTL 記述に変換する方法の検討, これらのタスクから別のタスクの呼び出しのための回路の検討, PDL 記述からアセンブラのソースプログラムの生成方法の検討を行ない、PDL 記述が Verilog-HDL の RTL 記述とアセンブラのソースプログラムに変換が可能であることを確認した。

和文キーワード: マイクロプロセッサ, Verilog-HDL, RTL 記述, 振舞い記述, アセンブラ

A Design-Aid System Based on a Microprocessor Description Language PDL

TSURUTA Mitsutoshi, ABE Kóki

Department of Computer Science

The University of Electro-Communications

1-5-1, Chofugaoka, Chofu-Shi, Tokyo 182 JAPAN

E-mail: tsurut-m@acorn.cs.uec.ac.jp, abe@cs.uec.ac.jp

Abstract

A microprocessor description language called PDL is described. The language allows behavioral descriptions being aware of clocks, from which an RTL description in Verilog-HDL of the processor's control and datapath as well as an assembler source program are generated using a table describing correspondence between mnemonic instructions, machine codes, and operations. The PDL supports task invocations from other task, where task is a construct specifying a set of sequential and/or pipelined operations, permitting to describe complex operations.

We examine in detail various ways of translating behavioral PDL descriptions into RTL counterparts, including ways of converting sequential and/or pipelined operations to actual circuit organizations and of realizing task invocation. A way of generating an assembler source program is also described. These investigations show the feasibility of implementing a design-aid system comprising the PDL compiler and assembler generator.

英文 key words: Microprocessor, Verilog-HDL, RTL description, behavioral description, assembler.

1 はじめに

最近、HDL による LSI 設計も一般化しつつあり、さらに、FPGA のゲート数も大容量化し [1]、マイクロプロセッサを FPGA 上に実装することも可能になっている [2][3]。これにより、気軽にプロセッサを製作できる環境が整いつつある。しかし、従来の HDL の RTL での記述によるマイクロプロセッサの設計では、データパスモジュールやコントロールモジュールなどの各機能モジュールを接続するためのトップモジュールの記述や、命令デコーダの記述等は非常に機械的で人が設計を行なうと間違えやすく、他人からは読みづらいという問題がある。さらに、マイクロプロセッサの RTL 記述を見ただけでは、どのような命令セットで、どのような動作をするプロセッサか分りにくい。また、オリジナルのプロセッサを設計すると、少なくとも、そのプロセッサ用のアセンブラを作成する必要がある。

本稿では、クロックを意識したプロセッサの振舞い、特に、命令セット・マシンコードとその命令に対応するプロセッサの動作を記述することのできるプロセッサ記述言語 PDL の考案を行ない、PDL 記述からプロセッサの Verilog-HDL[4] の RTL 記述と、アセンブラを生成する設計支援システムを提案する。

2 設計方針

2.1 システム設計の目的

本システム設計の目的は、マイクロプロセッサを設計するときの、設計者の負担軽減である。したがって、本システムを利用することにより、設計者に以下のような利点が得られるように、本システムを設計する。

- **設計期間の短縮**
  クロックを意識したプロセッサの振舞いを書くだけで論理合成に通る Verilog-HDL の RTL 記述とアセンブラを得ることができ、すぐに Verilog シミュレータにより機能シミュレーションを行なうことができる。さらにこの段階でアセンブラプログラムの開発も可能になる。これにより、ラウンドトリップ型の開発が可能になる。
- **ステートマシンやパイプラインの記述が簡単**
  ステートマシンは手続き型言語のような制御命令が利用可能。また、パイプラインは既存の HDL では RTL に近い記述で書かなくてはならなかったが、PDL では、振舞い記述に近い形で簡潔に表現できる。
- **設計資産の再利用が可能**
  Verilog-HDL 記述で書かれた組合せ回路は、PDL 上で対応する演算子を定義するとこにより利用可能になる。また、PDL は、タスクごとに部分コンパイルすることもでき、コンパイル結果も Verilog-HDL 記述の module であるのでこの module を Verilog-HDL による他の設計にも利用できる。
- **トップダウン設計**
  手続き型言語と同じようにモジュール化できるので、トップダウン設計が行ないやすい。また、自分で演算を定義できるのでボトムアップ的な設計をあまり行なわなくて済む。

2.2 システムの概要

PDL によるシステム構成とこれによるプロセッサの設計フローを図 1 に示す。図の点線の四角形で囲まれている部分が PDL によるシステムで、網掛けの部分が設計者が記述する部分になっている。設計者が書いた PDL 記述をこのシステムでコンパイルを行ない、プロセッサの Verilog-HDL の RTL 記述と、アセンブラを得る。この Verilog-HDL 記述を EDA ツールを使用して、シミュレーションを行なう。このときに、出力されたアセンブラも使用することができる。シミュレーションで、所望の結果が得られたら、論理合成を行ない、配置配線をし、FPGA 等のハードウェアに実装を行ない目的のプロセッサを得る。

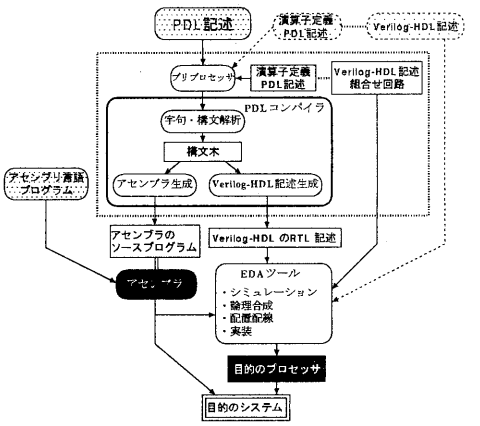

図 1. PDL による設計フローと PDL の処理系の構成

2.3 記述言語 PDL の文法

PDL の文法の概要を以下に示す。

- **変数**
  変数の型は、*wire*, *register*, *clock*, *memory* の 4 種類がある。*clock* 型以外は、ビット幅の指定ができる。*memory* 型および *register* 型は配列も利用可能である。変数宣言時に型名の前に *extern* を記述することにより、資源がプロセッサ外部に存在することを明示できる。また、宣言時にビット幅を指定しない変数は、その変数に代入されるビット幅が適用される。代入されるビット幅が全て一致していない場合はエラーとなる。*wire* 型の変数は宣言しなくても使用できる。この場合、局所変数扱いになる。
- **演算子の定義**
  function 文により演算子の定義を行なう。これは、PDL 上での演算子と実際に使用する組合せ回路 (Verilog-HDL の module または、既に定義されている演算の組合せ) との対応付けを行なう。
- **レジスタグループ宣言**
  group register 文により、アセンブラ上で使用するレジスタ名と PDL 上で使用するレジスタ名とそのレジスタコードの対応を宣言する。
- **制御文**
  処理の流れを変えるために、*if* 文, *while* 文, *goto* 文, *case* 文を利用することができる。
- **タスク**
  behavior 文によりタスクを定義できる。タスクは一連の処理をするための基本単位となる。手続き型言語の手続きに相当する。タスクから別のタスクを呼び出すことで、複雑な動きを実現できる。*main* タスクが一番

はじめに実行される。振舞い動作の指定として以下の文が使用できる。

serial 文：逐次処理を行なう

pipeline 文：パイプライン処理を行なう。制御文は *if* 文と *case* 文のみ使用可能である。

- **@ラベル**
  @ラベルは、*serial* 文や *pipeline* 文中で 1 クロックで実行する文のブロックに付けられる名前である。また、このブロックのことを@ラベルブロックという。@ラベルは *serial* 文ではステート名，*pipeline* 文ではステージ名になっている。
- **ラベルグループ宣言**
  group clock 文により、@ラベルブロックでの処理が次の@ラベルブロックへ処理を移る時に使用するクロックとそのタイミングの対応を宣言する。*clock* 変数が 1 つしかない場合、宣言の行なわれていない "@ラベル" は正エッジとなる。
- **命令動作テーブル**
  アセンブリ言語命令とそれに対応する動作を記述する。アセンブリ言語命令記述部, マシンコード記述部, 動作記述部の 3 つの記述部からなる。
- **プリプロセッサ命令**
  #define 文, *#include* 文等の C 言語のプリプロセッサ命令に加えて、置き換えを行なう見出し (項目名) と置き換える内容の表を記述するエイリアス・テーブル文がある。展開命令 # 項目名 # を記述すると、その行は、この表にしたがって、複数行に展開される。展開例を図 2 に示す。

```
alias table [op, vmodu] begin
  ["+", adder16]
  ["-", sub16  ]
  ["&", and16  ]
  ["|", or16   ]
end

function [y:16 = a:16 #op# b:16] is module #vmodu#(y, a, b);
a[#1:4#] = ##;
```

↓展開

```
function [y:16 = a:16 + b:16] is module adder16(y, a, b);
function [y:16 = a:16 - b:16] is module sub16(y, a, b);
function [y:16 = a:16 & b:16] is module and16(y, a, b);
function [y:16 = a:16 | b:16] is module or16(y, a, b);
a[1] = 1;
a[2] = 2;
a[3] = 3;
a[4] = 4;
```

図 2. エイリアス・テーブルの展開例

PDL によるマイクロプロセッサの記述例として、単純にパイプライン化した MISP のサブセット [5] を図 3 に示す。

2.4 PDL の処理系

PDL の処理系の構成は図 2 の点線で囲まれた四角形の内部のようになっている。設計者が書いた PDL 記述をコンパイルする場合、まず、プリプロセッサでエイリアステーブル文の実行、マクロの展開等のプリプロセッサ命令の実行を行なう。また、通常、ここで標準の演算子定義のヘッダファイルも読み込む。次に、PDL 記述を構文解析し構文木を作成する。この構文木を用いて目的のプロセッサ用のアセンブラのソースプログラム及び、プロセッサの Verilog-HDL の RTL 記述を生成する。

```
// 標準ライブラリのインクルード
#include <stdfunc.h>

// 変数の宣言
clock clk;
extern memory imem<31:0>[0:0xffffffff],
              dmem<31:0>[0:0xffffffff];
register regfile:32[1:31];
register pc:32;
const register zero:32 = 0;   // For "$0"

// レジスタグループ指定
group register r begin
  $0          zero          0:5;
  $#1:31#     regfile[##]   ##:5;
end

// ラベルグループ指定
group clock posedge clk = [IF, ID, EX, MEM, WB];

// メインタスク
behavior main
  pipeline begin
  @IF:  ir = imem[pc]; pc = pc + 4;
  @ID:  dec_exec(ir);
  end

// 命令動作テーブル
table dec_exec begin      [ ]
  nop                          [0x00000000]
                               { }
  lw  r.t, #.ofs "(" r.s ")"   [0b100011 s t ofs:16]
                               {@EX:  addr = s + ofs:32;
                                @MEM: dat = dmem[addr];
                                @WB:  t = dat;}
  sw  r.t, #.ofs "(" r.s ")"   [0b101011 s t ofs:16]
                               {@EX:  addr = s + ofs:32;
                                @MEM: dmem[addr] = t;}
  beq r.t, r.s, #.ofs          [0b000100 s t ofs:16]
                               {@EX: if (t == s)
                                       pc = pc + {ofs:30,0:2};}
  add r.d, r.s, r.t            [0b000000 s t d 0b010000]
                               {@EX:  res = s + t;
                                @MEM:
                                @WB:  d = res; }
  sub r.d, r.s, r.t            [0b000000 s t d 0b010000]
                               {@EX:  res = s + t;
                                @MEM:
                                @WB:  d = res; }
  slt r.d, r.s, r.t            [0b000000 s t d 0b011010]
                               {@EX:  res = (s < t);
                                @MEM:
                                @WB:  d = res;}
end
```

図 3. PDL による MISP のサブセットの記述例

3 PDL から Verilog-HDL 記述の生成方法の検討

PDL 記述から、Verilog-HDL の RTL 記述を生成する方法を検討する。まず、@ラベルブロックの演算を組み合わせ回路として処理する。次に、この組み合わせ回路生成の結果を用いて、レジスタとつなぎ合わせて、*serial* 文や *pipeline* 文の動きを実現する順序回路を生成する。

3.1 組み合わせ回路の配置と接続

3.1.1 回路生成に使用する選択回路

PDL 記述上では、組み合わせ回路になる部分は、代入文と *if* 文, *case* 文である。*if* 文や *case* 文は、演算を実行する・しないことで実現するのではなく、各演算回路は演算を必ず行ない、その演算結果を使用する・しないことで実現する。この演算結果を使用する・しないを選択するために PDL コンパイラでは、目的に応じて以下の 3 つの選択回路を使用する。

- **普通の選択回路**
  case ラベルが全て定数の場合の *case* 文や、*if* 文で使用する。選択信号をバイナリで指定する普通の選択回路である。

- **簡略化選択回路**
  serial 文のコントロールによるデータの選択等のように、複数の選択信号が同時にアサートされることがない場合に使用される。n 入力の場合、n 個の 2 入力の AND と n 入力の OR で実現される。
- **優先順位つき選択回路**
  入力の調停の必要な部分や、「*if* ～ *else if* … *else* ～」の形の *if* 文等で使用される。n 入力の場合、$n-1$ 本の選択信号が必要で、選択信号の添字の小さいものほど優先順位が高い設定になっている。n 番目のデータが出力される場合は、選択信号がどれもアサートされていない場合である。

3.1.2 wire 型変数への代入

はじめに、代入のある変数名とそれ以降に読み出されている同じ名前の変数に@ラベル名を添字として付加する。これは、別の@ラベルブロックの演算も同時に実行するので、変数名が衝突しないようにするためである。次に、同じ変数名に異なる場所で代入がある場合は、たとえ *if* ～ *else* ～ 文中で同時におこることがない場合でも、図 4 のように「変数の名前替え」を行なう。これにより、逆依存と出力依存をなくすことができ、*if* ～ *else* ～ 文中の演算は演算結果を使用する・しないにかかわらず求めることができる。最後に、*if* 文・*case* 文の処理を行なう。これは、図 5 のように、選択回路で、それぞれの演算結果を *if* の条件にしたがって、選択することで実現する。さらに、「*if* ～ *else if* … *else* ～」 の形の *if* 文には、優先順位つき選択回路を使用する。

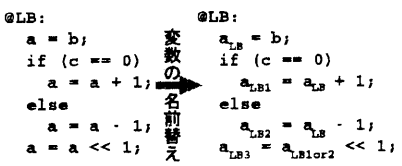

図 4. 変数の名前替えの例
a_{LB1or2} は *if* 文用選択回路の出力である。

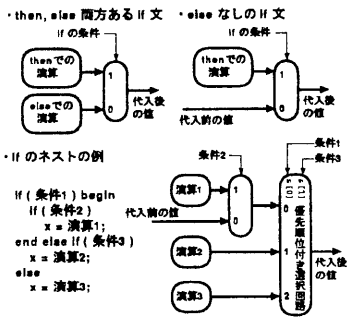

図 5. 選択回路による if 文の実現

3.1.3 register 型変数への代入

register 型変数への代入では、演算結果を一時的にワイヤーで受け、そのワイヤーをレジスタの入力にする。したがって、処理は、*wire* 型変数への代入のときと同じで、そのワイヤーの先にレジスタがあるだけである。ただし、register 型変数への代入の場合は、代入前の値としてレジスタの現在の出力値を用いる。

3.2 serial 文の回路化

serial 文の回路化は、コントロール部とデータパス部の 2 つに分け、コントロール部は、ステートマシンで実現する。データパス部は、各@ラベルブロック毎の組み合わせ回路の出力をコントロールからの信号にしたがって選択することで実現する。ここでは、図 6(a) のような乗算の演算をおこなう簡単な *serial* 文の例を使用して生成過程を説明する。

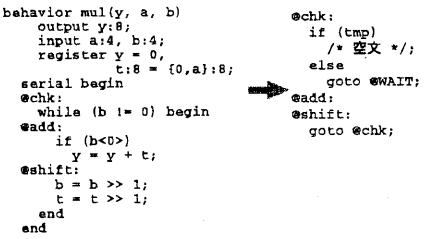

図 6. serial 文による乗算の例 (a) と制御命令の取り出し (b)
(b) の tmp は、tmp = (b != 0) として外部で演算する。

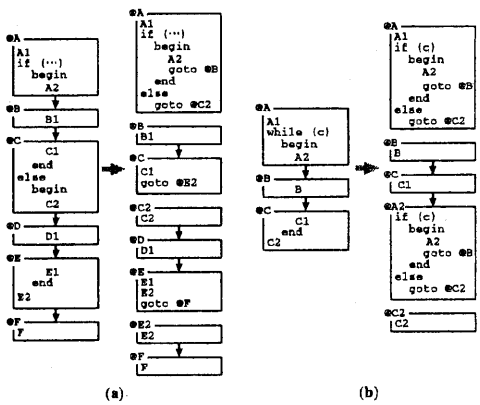

図 7. @ラベルブロックの組み替え
(a) if 文の場合, (b) while 文の場合。

3.2.1 コントロール部の生成

はじめに、制御命令の入っている *case* 文を「*if* ～ *else if* … *else* ～」の形にする。次に、複数の@ラベルブロックを含む *while* 文、*if* 文は図 7 のように、@ラベルブロックの組み替えとを行なう。次に、*goto* とそれに関係のある *if* 文のみを取り出す(図 6(b))。そして、@ラベルをステートとして符号化する。本システムではデフォルトは one-hot 符号化を行なう。そして、次のようにして Verilog-HDL でステートマシンの記述を生成する。

- ステートの保持に state というレジスタを用意し、ステートを移るという動作は、この state に新しい値を書き込むということで実現する。*goto* の記述のあるものは次のステートを指定されているステートにし、*goto* の記述のないものは次のステートを下のステートとする。
- WAIT というステートをあらかじめ用意する。WAIT は、*serial* 文が実行を開始するのを待つステートで、開始信号がアサートされると *serial* 文の一番先頭の@ラ

ベルブロックのステートに移る。*serial* 文の実行が完了した時は、開始信号がアサートされていれば、*serial* 文の一番先頭の@ラベルブロックのステートへ移り、アサートされていなければ、WAIT へ移るようにする。

図 6 の乗算のステートマシンの生成結果を図 8 に示す。

```
'define ST_BIT   [2:0]

'define ST_WAIT  3'b000
'define ST_chk   3'b001
'define ST_add   3'b010
'define ST_shift 3'b100

wire tmp;
reg 'ST_BIT state;

is_neq tmp000(tmp, b, 0);

always @(posedge clk or posedge reset) begin
  if (reset) begin
    state <= 'ST_WAIT;
  end else begin
    case (state)
      'ST_WAIT:
        if (start) begin
          state <= 'ST_chk;
        end else begin
          state <= 'ST_WAIT;
        end
      'ST_chk:
        if (tmp)
          state <= 'ST_add;
        else if (start) begin
          state <= 'ST_chk;
        end else begin
          state <= 'ST_WAIT;
        end
      'ST_add:
        state <= 'ST_shift;
      'ST_shift:
        state <= 'ST_chk;
      default:
        state <= 'ST_WAIT;
    endcase
end

assign busy = !((state == 'ST_WAIT)
              || ((state == 'ST_chk) && !tmp));
```

図 8. ステートマシンの生成結果
モジュール 'is_neq' は、ヘッダファイルで
function [y:1 = a:4 != b:4] is module is_neq(y, a, b);
として定義してあると仮定。

3.2.2 データパス部の生成

データパス部の生成は以下のような手順で行なう。

1. 引数の受け取り・変数の初期化は、INIT という仮のステートを作り処理を行なう。
2. @ラベルブロックの組み替え処理を行なった PDL 記述に 3.1 項での処理を行ない、各@ラベルブロック毎の組合せ回路を生成する (図 9)。
3. コントロールから出力されるステートの値をデコードし、今、どのステートを実行中なのかを示す信号を作る回路を配置する。
4. 各変数毎に代入のあるステートを調査する。
5. この調査結果をもとにレジスタの書き込み信号出力回路と入力データの選択回路を配置する。このときの選択回路は同時に複数のステート実行することはないので簡略化選択回路でよい。
6. レジスタをインスタンス化し、出力は変数名、入力データと書き込み信号は 5 での選択回路の出力を使用する。

図 6 の乗算のデータパスの生成結果を図 10 に示す。

最後に、処理中に新たに作成した wire を定義し、コントロール部とデータパス部の記述をつなげ、*serial* 文を Verilog-HDL の 1 つのモジュールにする。

```
INIT
  assign a_INIT = _a;
  assign b_INIT = _b;
  assign y_INIT = 0;
  assign t_INIT = {0, _a};
@chk
@add
  adder add1(tmp_add, y, t);
  assign y_add = sel8_2(y, tmp_add, b[0]);
@shift
  shift_r shift1(b_shift, b);
  shift_l shift2(t_shift, t);
```

図 9. @ラベルブロック毎の組合せ回路

```
// ステート値のデコード
assign DO_chk = (state == 'ST_chk);
assign DO_add = (state == 'ST_add);
assign DO_shift = (state == 'ST_shift);
assign INIT = start && !busy;
// INIT
assign a_INIT = _a;
assign b_INIT = _b;
assign y_INIT = 0;
assign t_INIT = {0, _a};
// @add
adder add1(tmp_add, y, t);
assign y_add = sel8_2(y, tmp_add, b[0]);
// @shift
shift_r shift1(b_shift, b);
shift_l shift2(t_shift, t);
// レジスタ書き込み信号の生成
assign ld_y = DO_add || INIT;
assign ld_b = DO_shift || INIT;
assign ld_t = DO_shift || INIT;
// レジスタ入力データの選択
assign yin = ssel8_2(0, y_add, {INIT, DO_add});
assign bin = ssel4_2(_b, b_shift, {INIT, DO_shift});
assign tin = ssel8_2(_a, t_shift, {INIT, DO_shift});
// レジスタのインスタンス化
register8 mul_y(y, yin, ld_y, clk, rst);
register4 mul_b(b, bin, ld_b, clk, rst);
register4 mul_t(t, tin, ld_t, clk, rst);
```

図 10. データパス部の生成結果

3.3 pipeline 文の回路化

pipeline 文では、各@ラベルブロックがパイプラインステージに対応している。したがって、各@ラベルブロック毎に組合せ回路を用意し、これらの入出力に、パイプラインレジスタを挿入することで、パイプラインの実現ができる。ここでも、図 11(a) のような乗算を行なう簡単な *pipeline* 文の例を使用して生成過程を説明する。

```
behavior mul (y, a, b)
    output y:8;
    input a:4, b:4;
    wire y;
  pipeline begin
  @s1:
    y = 0;
    if (b<3>)
      y = {0, a}:8;
    y = y << 1;
  @s2:
    if (b<2>)
      y = y + a;
    y = y << 1;
  @s3:
    if (b<1>)
      y = y + a;
    y = y << 1;
  @s4:
    if (b<0>)
      y = y + a;
  end
```

(a)

```
@s1
  assign y_s1_1 = 0;
  assign y_s1_2 = {4'h0,a};
  assign y_s1_if1 = sel8_2(y_s1_1, y_s1_2, b[3]);
  shift_l s11(y_s1, y_s1_if1);
@s2
  adder8 s21(y_s2_1, y, {4'h0,a});
  assign y_s2_if1 = sel8_2(y, y_s2_1, b[2]);
  shift_l s22(y_s2, y_s1_if1);
@s3
  adder8 s31(y_s3_1, y, {4'h0,a});
  assign y_s3_if1 = sel8_2(y, y_s3_1, b[1]);
  shift_l s32(y_s3, y_s3_if1);
@s4
  adder8 s41(y_s4_1, y, {4'h0,a});
  assign y_s4 = sel8_2(y, y_s4_1, b[0]);
```

(b)

図 11. pipeline 文による乗算の例 (a) と各ステージ毎の組合せ回路 (b)

3.3.1 各ステージ毎の演算の処理

@ラベルブロック毎の組合せ回路を生成する (図 11(b))。生成された回路で、読み出されている信号線名に@ラベル名の添字のないもの (つまり、パイプラインステージの入力) に、@ラベル名と記号 "IN" を添字として付加する (図 12)。

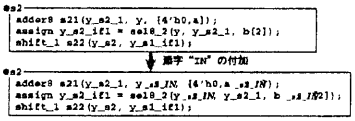

図 12. 添字 "IN" の付加の例

3.3.2 パイプラインレジスタの配置

パイプラインレジスタは、パイプラインの後段の方から配置していく。一番最後のステージの後にはパイプラインレジスタは必要ないので配置しない。以下のことを先頭のステージまで繰り返す。

1. 添字 "IN" のついた記号を出力するパイプラインレジスタを当該ステージの前に配置する。
2. 当該ステージの後ろのパイプラインレジスタの対応する入力と当該ステージの組合せ回路の出力を接続する。
3. 後ろのパイプラインレジスタの入力に、当該ステージの出力とつながっていないものがあれば、当該ステージの前にパイプラインレジスタを配置し、このレジスタ出力と後ろのパイプラインレジスタの対応する入力を接続する。

最後に、先頭のパイプラインレジスタの入力に仮引数と初期値が入るようにする。図 11 のパイプラインの生成結果を図 13 に示す。

```
assign y_s1_1 = 0;
assign y_s1_2 = {4'h0,a_s1_IN};
assign y_s1_if1 = sel8_2(y_s1_1, y_s1_2, b_s1_IN[3]);
shift_1 s11(y_s1, y_s1_if1);

pipereg8 s1y(y_s2_IN, y_s1, ld_s1, rst_s1, clk);
pipereg4 s1a(a_s2_IN, y_s1_IN, ld_s1, rst_s1, clk);
pipereg4 s1b(b_s2_IN, b_s1, ld_s1, rst_s1, clk);

adder8 s21(y_s2_1, y_s2_IN, {4'h0,a_s2_IN});
assign y_s2_if1 = sel8_2(y_s2_IN, y_s2_1, b_s2_IN[2]);
shift_1 s22(y_s2, y_s1_if1);

pipereg8 s2y(y_s3_IN, y_s2, ld_s2, rst_s2, clk);
pipereg4 s2a(a_s3_IN, y_s2_IN, ld_s2, rst_s2, clk);
pipereg4 s2b(b_s3_IN, b_s2, ld_s2, rst_s2, clk);

adder8 s31(y_s3_1, y_s3_IN, {4'h0,a_s3_IN});
assign y_s3_if1 = sel8_2(y_s3_IN, y_s3_1, b_s3_IN[1]);
shift_1 s32(y_s3, y_s3_if1);

pipereg8 s3y(y_s4_IN, y_s3, ld_s3, rst_s3, clk);
pipereg4 s3a(a_s4_IN, y_s3_IN, ld_s3, rst_s3, clk);
pipereg4 s3b(b_s4_IN, b_s3, ld_s3, rst_s3, clk);

adder8 s41(y_s4_1, y_s4_IN, {4'h0,a_s4_IN});
assign y_s4 = sel8_2(y_s4_IN, y_s4_1, b_s4_IN[0]);
```

図 13. パイプラインの生成結果

3.3.3 パイプラインの流れの制御

パイプラインの流れの制御はパイプラインレジスタを制御することにより可能である。ステージが n 段のパイプラインにおいて、ステージ $i(i = 1, \cdots, n)$ での組み合わせ回路を S_i とし、その演算結果を受け取るパイプラインレジスタを r_i、 r_i のロード信号とリセット信号をそれぞれ ld_i, rst_i とする。通常の状態では、$ld_{1\sim n-1}$ はアサートされ、$rst_{1\sim n-1}$ はネゲートされている。

- **ステージ i の破棄**
  rst_i をアサートする。
- **全体の破棄**
  $rst_{1\sim n-1}$ をアサートする。
- **ステージ i のストール**
  rst_i をアサートし、 $ld_{1\sim i-1}$ をネゲートする。
- **全体のストール**
  $ld_{1\sim n-1}$ をネゲートする。

3.4 サブタスクの起動・呼び出し・調停

PDL では、あるタスクから別のタスク (サブタスク) を呼び出すことが可能である。以下に、この「呼び出し」を実現するための回路を検討する。

3.4.1 serial 文の起動

3.2.1 で述べたように、serial 文は、実行開始信号 start がアサートされるまで、WAIT というステートに留まり、start がアサートされると実際の処理を開始する。処理の実行中は、実行中を示す busy という信号がアサートされる。処理の最後のステートに来ると、busy がネゲートされ、このとき、start がアサートされていれば、次にすぐ新しく処理を開始し、アサートされていなければ、再び WAIT に入り、start 信号を待つ。

3.4.2 pipeline 文の起動

パイプラインは常にデータが流れているので、別のタスクからの処理の依頼がない場合は、バブルをパイプライン中に流す。処理の依頼が来たら、そのデータをパイプラインに流し処理をする。またこのデータと一緒にどこから呼び出したかを示す「呼び出し識別子」も流し、処理の終了をこの呼び出し識別子を使って確認する。呼び出し識別子は、PDL のコンパイル時に静的に割り付けられる。

3.4.3 serial 文からの呼出し

serial 文からの呼出しを行なうには、呼び出し命令のあるステートで、渡す引数を出力し、処理依頼のための call という信号をアサートする。この call 信号は相手の start 信号につながっている。相手が処理中かを相手の busy 信号を見て確認し、busy がアサートされていれば、呼び出しが可能になるまで、呼び出し待ちのステートに移り待機する。相手が依頼した処理を開始してから busy がアサートされている間、つまり、処理中は、別のステートに移り待機する。ここで、busy がネゲートされたら、処理が終了したので、相手から戻り値を受け次のステートへ進む。

pipeline 文を呼び出した場合は、パイプラインから出力されている呼び出し識別子と自分の呼び出し識別子が同じ値かどうかで終了を確認する。

3.4.4 pipeline 文からの呼出し

- **順序タグ**
  パイプラインからサブタスクを呼び出す場合、パイプラインが乱れ、実行形式によってパイプラインの動作が変わる。
  in-order 実行: パイプラインに処理を依頼した順番で実行する。
  out of order 実行: 先に依頼したものを後に依頼したものが追い越して実行することがある。
  したがって、パイプラインに処理を依頼した順序を知る必要があり、データと共にこの順序を示すためのタグ (順序タグ) もパイプラインに流す。この順序タグはパイプラインに処理が依頼されるたびに生成される動的なものである。サブタスクも含めたパイプラインの総ステージ数が N の場合、前後 N の範囲で順序を一意に定めるためには、順序タグは、$2N$ 個識別できればよい。また、順序タグが、0 の場合はパイプライン

レジスタ中にバブルが存在することを示すことにする。したがって、n を $2N < 2^n$ を満たす整数として、順序タグを 0 を出力しないモジュロ $2^n - 1$ カウンタで生成する。PDL コンパイラは、デフォルトでは、n ビット LFSR カウンタを使用する。また、順序を調べるための比較器は、0 を最大の数とみなし、このカウンタ出力の前後 $2^{n-1} - 1$ 個を正しく比較できなければならない。たとえば、$n = 3$ で、カウンタに 3bit LFSR カウンタを使用したときの a, b の順序、$a <_{tag} b$ を判定する比較器は、図 14 のような出力が得られればよい。この比較器は、パイプラインから呼び出され分岐したサブタスクが再びパイプラインに合流するときにどちらの命令を先に実行するかを決定するときに使用する。

a \ b	0(000)	1(001)	2(100)	3(010)	4(101)	5(110)	6(111)	7(011)
0(000)	0	0	0	0	0	0	0	0
1(001)	1	0	1	1	1	0	0	0
2(100)	1	0	0	1	1	1	0	0
3(010)	1	0	0	0	1	1	1	0
4(101)	1	0	0	0	0	1	1	1
5(110)	1	1	0	0	0	0	1	1
6(111)	1	1	1	0	0	0	0	1
7(011)	1	1	1	1	0	0	0	0

図 14. 3bit LFSR カウンタ用順序タグ比較器
$a <_{tag} b$ が成り立つとき 1 を出力する。

- **パイプラインの分岐**

パイプラインから他のサブタスクを呼び出すということは、パイプラインの流れを分岐させるということである。呼び出す側のパイプラインを本流といい、呼び出される側のタスクを支流ということにする。支流へ分岐する場合、本流側の次ステージへのパイプラインレジスタををリセットする。もし、支流がすぐに処理を行なえない状況の場合、本流の実行形式に応じて次の方法をとる。

in-order 実行: 支流が処理可能になるまで該当ステージをストールさせる。

out of order 実行: 支流が処理可能になるまで該当ステージのデータを退避するための退避レジスタを用意し、そこにデータを退避させ、後続データの処理を続行する。もし、退避レジスタにデータが既に存在するときは、該当ステージをストールさせる。また、*serial* 文への分岐の場合は、*serial* 文処理中のデータ保存のために必ずデータを退避させる。

- **パイプラインの合流**

支流が *serial* 文の場合、処理の終了を合流点で待たなければならない。本流の実行形式に応じて次のように処理する。

in-order 実行: 処理終了まで該当ステージをストールさせる。

out of order 実行: *serial* 文の実行中は順序タグ 0 を出力する。処理終了時に本来の順序タグを出力し、戻り値と分岐時に退避していたデータを本流に合流させる。

支流の処理が終了し、本流に合流する場合、順序タグを見て先に依頼された命令を優先して実行するが、片方の順序タグが 0 の場合、実行形式に応じて対応が以下のように異なる。

in-order 実行: 先に依頼された命令を優先して実行する。ストール等により順序タグが 0 になっている場合は、そのステージより前のステージにある順序タグをバイパスして比較に用いる。もし、そのステージより前のステージにある順序タグすべてが 0 である場合は、順序タグが 0 でない方を実行する (図 15)。

out of order 実行: 順序タグが 0 でない方を実行する。

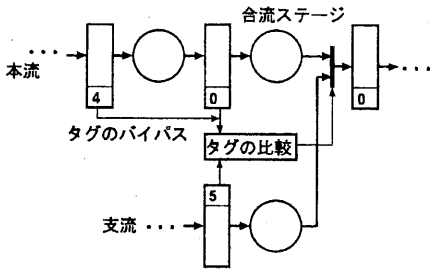

図 15. in-order 実行時の合流例
この場合タグ値 '4' と '5' を比較し、本流パイプラインのタグ値 '4' のデータが処理されるまで、支流はストールする。

3.4.5 同一タスクへの複数の呼出しの調停

あるタスクへの呼び出しが複数ある場合、同時にそのタスクへの呼び出しが起こってしまう可能性がある。したがって、呼び出しに優先順位を付け、その順位にしたがって調停を行なうようにする。デフォルトでは、図 16 のような簡単な回路で調停を行なう。この回路では、優先順位の低い呼び出しは、飢餓状態になってしまうので、これが良くない場合は、設計者が独自に調停を行なう必要がある。

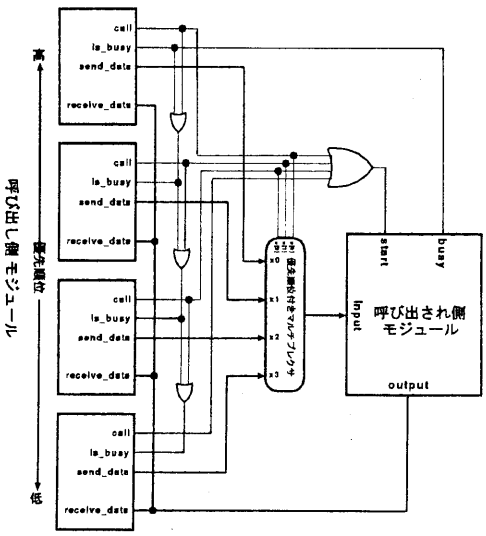

図 16. 複数呼び出しの調停回路

4 PDL からアセンブラソースプログラムの生成方法の検討

4.1 アセンブラの概要

PDL 記述のレジスタグループ宣言と命令動作テーブルの部分からアセンブラのデータベースを作成し、このデータベース利用して動作するアセンブラ・コアと組み

合わせれば、PDL で記述されているプロセッサ専用のアセンブラを得ることができる。アセンブラとして、命令のマシンコードの変換以外には、以下の機能は最低限必要と思われる。

- ラベル
- 相対アドレスの計算
- 数値の変換 (2 進, 8 進, 10 進, 16 進)
- 簡単な式の計算
- マクロ命令 (定義と展開)
- 疑似命令

org	処理アドレスの設定。
eq (=)	ラベルに式の値を設定。
db, dw, dd	後メモリ上に式の値を配置。

4.2 アセンブラ用データベース

PDL 記述から、アセンブラの動作に必要な部分を取り出し、データベースを作成する。データベースは以下の 2 種類のものを作成する。

1. **レジスタコード表**
   アセンブラで使用されるレジスタ名とそれに対応するレジスタコードの表である。ニーモニックにより使用できるレジスタやレジスタ名とレジスタコードの対応が異なることがあるので、複数作成されることもある。
2. **マシンコード表**
   ニーモニック, オペランドとそれに対応するマシンコードの表である。ニーモニックの覧には文字列が入る。マシンコードの覧には、対応するマシンコードの鋳型が入る。オペランドは複数指定でき、オペランドの覧にはそれぞれ以下のようなものが入る。
   - レジスタ (どのレジスタコード表を使用するかをポインタ)
   - 文字列 (メモリ参照用の'(',')' 等の表記用文字列)
   - 絶対アドレス型定数 (使用するビット幅)
   - 相対アドレス型定数 (使用するビット幅とオフセット値)

4.3 アセンブラ・コアの設計の検討

生成するアセンブラ・コアは、対応するニーモニックをマシンコード表から引き、マシンコードの鋳型を用意し、オペランドがレジスタならレジスタコード表を引いてレジスタコードを、定数ならその定数を鋳型にセットし、マシンコードを得るという点を除き標準的な 2 パスのアセンブラである。
生成されるアセンブラの処理系の構成を図 17 に示す。

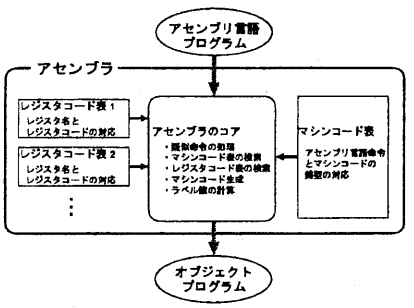

図 17. アセンブラの処理系の構成

5 今後の課題・拡張

今後の課題と拡張すべき点を以下に示す。

- **並列実行文の実現**
  現在、逐次実行とパイプラインのみの対応であるので、PDL を使用して、スーパースカラプロセッサを設計するのは困難である。並列実行文が実現できれば、スーパースカラプロセッサの設計も容易になる。
- **処理系の実際の製作**
  設計の検討を行なったが、実際に処理系は製作していない。処理系を製作し、実際に使用し、さらに PDL に足りない機能を拡張する。
- **マシンコードの自動割り付け**
  アセンブラを自動生成できるので、マシンコードを設計者が指定しなくても自動割り付けを行なう機能があるとよい。自動割り付けにより、ハードウェアがデコードしやすいコードを得られれば、その分プロセッサの処理速度も向上する。
- **ハードウェア・スケジューラの作成**
  現在、@ラベルにより、設計者が 1 クロックで実行できる範囲を定めているが、@ラベルを記述しなくても、自動的に@ラベルを適切な位置に挿入することのできるハードウェア・スケジューラがあると、一層、設計者の負担が軽減できる。

6 まとめ

本稿では、マイクロプロセッサを短期間に簡単に作成できる記述言語 PDL を提案した。*serial* 文の PDL 記述からステートマシンによる制御回路とデータパス の Verilog-HDL の RTL 記述の生成方法と *pipeline* 文の PDL 記述からパイプラインの Verilog-HDL の RTL 記述の生成方法を具体例を用いて検討を行なった。さらに、*serial* 文, *pipeline* 文によるタスクから、別の *serial* 文, *pipeline* 文によるタスク (サブタスク) を自由に呼び出すための回路の検討も行なった。これらにより、クロックは意識するが、手続き型言語のような記述スタイルでプロセッサが設計できるようになる。また、PDL 記述からアセンブラのソースプログラムの生成方法の検討も行なった。これらの検討により、PDL 記述から Verilog-HDL の RTL 記述 と アセンブラのソースプログラムへの自動変換が可能であることを確認した。

参考文献

[1] "The Programmable Logic Data Book", Xilinx Inc, (1996)

[2] 末吉, 井上, 奥村, 久我: 「教育用 32 ビット RISC マイクロプロセッサ DLX-FPGA と教材ボードの開発」, 第 3 回 FPGA/PLD Design Conference & Exhibit 応用技術論文集, p579-588, (1995)

[3] 鵜田, 阿部: 「16bit RISC プロセッサ peco の設計と評価」, 第 3 回 FPGA/PLD Design Conference & Exhibit 応用技術論文集, p597-608, (1995)

[4] Donald E. Thomas, Philip R. Moorby: "THE VERILOG HARDWARE DESCRIPTION LANGUAGE", Kluwer Academic Publishers (1995) (邦訳: 飯塚, 浅田 : 「設計言語 Verilog-HDL 入門」, 培風館 (1995))

[5] D.A.Patterson, J.L.Hennessy: "Computer Organization & Design: The Hardware/Software Interface", Morgan Kaufmann Publishers,Inc (1994) (邦訳: 成田: 「コンピュータの構成と設計 (上・下)」, 日経 BP 社 (1996))