パイプライン動作する演算モジュールのRTL記述から動作記述への抽象化

高木 彬[1,a)]　久我 守弘[1,b)]　飯田 全広[1,c)]　伊藤 寛人[2,d)]　井戸 大介[2,e)]

概要：過去に設計された RTL 記述を再利用する際，その回路はサイクル精度で設計されているため再利用が困難である．一方で高位合成によるアーキテクチャ探索を行うことで，動作記述から様々な設計空間を持つ RTL 記述を短時間で設計することが可能である．そこで，本研究では RTL 記述から動作記述への抽象化を行い，高位合成によるアーキテクチャ探索を行えるようにすることで，RTL 記述の再利用を容易にすることを目指す．本稿では，パイプライン動作する演算モジュールを対象とした RTL 記述を，転送表を用いて高位合成可能な動作記述へと抽象化する手法について報告する．検証結果より，一部制約はあるものの，パイプライン動作する演算モジュールの RTL 記述から動作記述へ抽象化が行えることを確認した．

An Abstraction Method from RTL to Behavior Level Description for Pipelined Modules

Akira Takaki[1,a)]　Morihiro KUGA[1,b)]　Masahiro IIDA[1,c)]　Hiroto ITO[2,d)]　Daisuke IDO[2,e)]

Abstract: It is difficult to reuse RTL descriptions because they are designed with the cycle accurate model. On the other hand, the high-level synthesis can generate RTL descriptions with various design spaces from behavioral descriptions by architectural exploration. In this research, we aim to facilitate the reuse of RTL by abstracting RTL to behavioral descriptions and enabling architectural exploration by high-level synthesis. We report a method for abstracting RTL descriptions of pipelined modules into high-level synthesizable behavioral descriptions using transfer tables. By the verification results, we confirm that the abstraction from RTL descriptions to behavioral descriptions of pipelined modules is possible, although there are some restrictions.

1. はじめに

集積回路の集積度の向上に伴い，ディジタルシステムを SoC（System on a Chip）としてひとつの集積回路として実現できる時代においては，いかに大規模なシステムを短期間に設計・開発できるかが重要である．集積回路の設計開発手法は従来の机上でのゲートレベル設計に取って代わり，1990 年代からは Verilog HDL や VHDL 等のハードウェア記述言語を用いてレジスタトランスファレベル（RTL：Register Transfer Level）による設計が主流となった．近年では RTL よりもさらに抽象度の高いビヘビアレベル（Behavioral Level）を用い，C 言語ライクな記述から高位合成（High-Level Synthesis）により回路設計・開発を行えるようになってきた．より抽象度の高いレベルで設計できることから，設計期間や検証期間の短縮を図ることができるようになってきた．一方，従来の RTL やゲートレベルで開発されてきた設計資産は多くの集積回路設計関連企業等において蓄積されている．これらの設計資産を IP（Intellectual property）として再利用することも，集積回路の集積度向上に伴う設計期間や検証期間の増大に対処できる方法として有用である．

しかし，RTL やゲートレベルによる設計資産は再利用が難しいという問題点がある．ゲートレベルの論理回路は，実装デバイスに依存したセルライブラリを用いて最適化さ

1　熊本大学，Chuo-ku，Kumamoto-shi 860–8555，Japan
2　三菱電機エンジニアリング，Higashi-ku，Nagoya-shi 461–0048，Japan
a)　takaki@st.cs.kumamto-u.ac.jp
b)　kuga@cs.kumamoto-u.ac.jp
c)　iida@cs.kumamoto-u.ac.jp
d)　Ito.Hiroto@ma.mee.co.jp
e)　Ido.Daisuke@ma.mee.co.jp

© 2022 Information Processing Society of Japan

れているため，他の実装デバイス向けにはそのまま利用することが難しい．また，ゲートレベルよりも抽象度の高い RTL 記述の論理回路においても，その動作はクロックに同期し決められた状態遷移に従って動作するサイクル精度レベル（Cycle Accurate Level）で実現されているため，再利用の際に状態遷移や機能ユニット数等をターゲットシステムに合わせて最適化することが困難である．

一方，近年実用化されてきた高位合成を用いると，時間概念のない Untimed Functional レベルで記述された動作記述レベルのコードから，アーキテクチャ探索機能により同一サイクル下における演算器等の機能ユニット数の異なるものや状態遷移数やその構造が異なる様々な論理回路を自動生成することが可能であり，開発するアプリケーションに適応した論理回路の設計空間を広げることができるメリットがある．

そこで，本研究では RTL で開発された過去の設計資産を一度 Untimed Functional モデルの動作記述へと抽象化し，高位合成を利用することで設計資産の広範囲な再利用を可能にする手法について研究を行う．具体的には，高位合成が行っている処理を逆に辿るアプローチであり，サイクル精度レベルで記述された RTL から CDFG（Control Data Flow Graph）を生成し，データパスと状態遷移に着目して Untimed Functional モデルの動作記述を抽出する方法である．我々の先行論文 [1] では，組み合わせ回路のみの RTL 記述を対象として抽象化をしていた．本稿では，パイプライン動作する演算モジュールの RTL 記述を対象とし，転送表を用いて抽象化する手法について述べる．

なお，関連研究として，verilator[2]，東芝の特許 [3]，v2c[4]，VeriIntel2C[5] がある．verilator および東芝の特許は，論理 RTL 記述をほぼ同じ記述レベルのまま C コードへ変換し，機能レベルでの動的シミュレーションが高速に行えるようにすることを目的としている．また，v2C および VeriIntel2C は RTL 記述の再利用を目的として，抽象化を行っている．これらの研究では構文解析器として商用パーサである Verific Parser Platform を用いているが，我々はオープンソースで提供されているツール群を利用することで，広く利用できるツールとしての開発を目指している．

以下，2 章において提案する転送表を用いた制御フローを有する RTL 記述の抽象化手法について述べる．3 章において，変換前のコードと逆変換後のコードの等価性について議論する．最後に 4 章を本稿のまとめとする．

2. RTL から動作記述への抽象化

本章では提案する抽象化手法について説明する．なお，本稿で対象とする RTL 記述は，パイプライン動作する演算モジュールであり，以下のような特徴を持つものとする．

- 制御フローは分岐せず，小ループを持たない直線的な FSM である．
- 抽象化した際の数式が，出力=f（入力）の形式を取る RTL を対象とする．

2.1 抽象化フローの概要

我々が提案する抽象化ツールの処理フローを図 1 に示す．一般的な言語処理システムと同様に，Verilog HDL により記述された RTL 記述に対して構文解析を行い抽象構文木（AST:Abstract Syntax Tree）を作成する必要がある．構文解析にはオープンソースの Verilog 用構文解析器である Pyverilog[6] を用いた．Verilog HDL コードに対して，Pyverilog の制御フロー解析およびデータフロー解析を用いて，制御フローとデータフローを入手可能である．

抽象化ツールの開発にあたり，以下の方針で開発を行う．RTL 記述から動作記述へ抽象化する際には，先ず制御フローが存在するか否かで処理を分ける．組み合わせ回路でデータフローのみ存在する RTL は，基本的に output の変数から input の変数までデータフローを逆にたどることで数式の抽出が行える．本稿では，制御フローが存在する RTL のうち，パイプライン動作する演算モジュールを対象としてた抽象化手法について説明する．

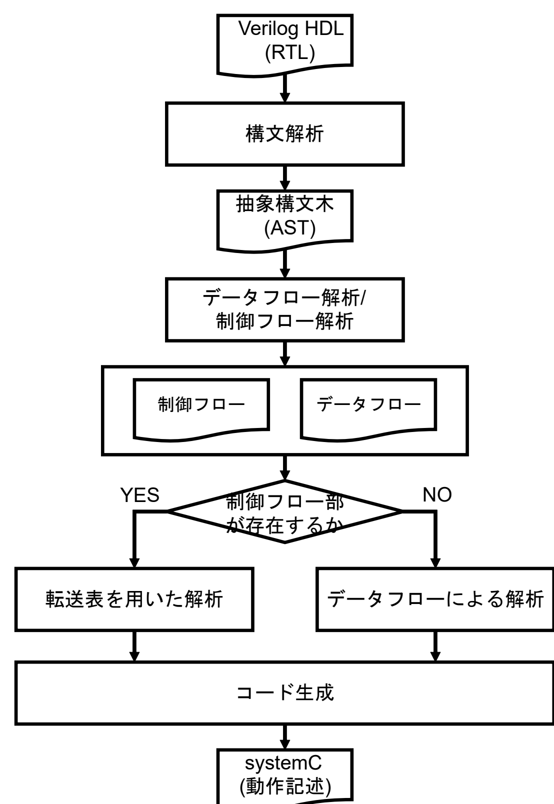

図 1 抽象化の処理フロー

© 2022 Information Processing Society of Japan

また，抽象化の詳細を説明するために，NEC 社製である Cyber Work Bench（CWB）[7] の高位合成によって作成したソースコード 3（付録に掲載）を使用する．この RTL 記述は Y=A+B+C の計算を行うもので，制御フローが S0 および S1 の 2 つの状態を遷移する FSM（Finite State Machine）で構成され，演算器制約により 1 サイクル中で加算器を 1 つしか使えない状況を想定し，2 サイクルかけて加算を行う設計である．1 サイクル目に A および B が入力され，2 サイクル目に C が入力される回路となっている．

2.2 制御フローがある RTL の抽象化

RTL においては時間概念が存在し，特に制御フローがある RTL においては FSM を有し，時間概念を持つ．抽象化においてはその時間概念を取り払う必要がある．即ち，TF（Timed Functional）レベルから UTF（UnTimed Functional）レベルへの変換を目指すのが抽象化といえる．制御フローがある RTL における時間概念とは，制御フローの制御により状態毎に処理が変わる場合を想定しており，時系列で異なる処理を行うことになり，そこに時間の概念があることを指す．例えば，2 度加算を行うが演算器制約により 1 サイクル間では 1 つの加算ブロックしか行えない場合を考える．状態遷移の状態レジスタが S0 のときに実行される処理と状態レジスタが S1 のときに実行される処理が異なることになり，そこには時間概念が存在するといえる．

本研究では転送表を用いて時間概念を取り払う．制御フローの解析を行った後，制御フローとデータフローを用いて転送表を作成し，対象の RTL の処理を解析，時系列を考慮して出力から入力まで転送表内のデータフローをつなぐことにより抽象化を行う．

2.3 制御フロー解析

制御フロー解析では，RTL 記述内における状態遷移を行いデータフローに対して制御を行う変数を特定し，その変数がどの状態を取り，どのような状態遷移を行うのかについて解析を行う．解析には Pyverilog 内の機能を用いている．

付録ソースコード 3 の 17〜42 行が制御フローの状態遷移を表し，これは解析により図 2 中の制御フローとして得られる．

2.4 データフロー解析

データフロー解析では，RTL 記述内における全てのデータフローを，出力先-木構造という形式で構築する．解析には Pyverilog 内の機能を用いている．

ソースコード 3 を例として，データフロー構造の一部を図 2 中のデータフローに示す．例として，Y_r と add32s1ot について記述した．全てのデータフローが図中の例のような木構造で表現される．

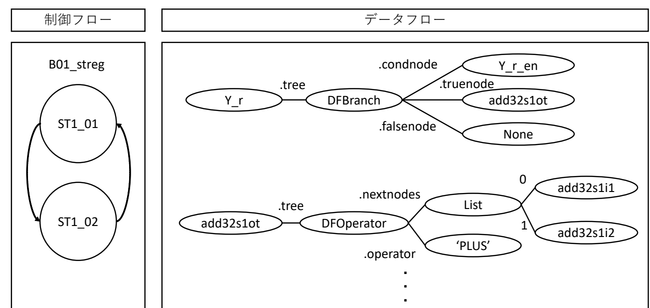

図 2 制御フロー解析による状態遷移とデータフロー解析によるデータフローの例

2.5 転送表

本研究で用いる転送表は，文献 [8] の高位合成における「制御回路とデータパスの生成」内の転送表を参考に作成した．この文献における転送表は，演算器数制約を受ける場合の制御回路とデータパスの関係を状態毎のレジスタの接続関係や演算器の使用状況，出力の接続関係で表したもので，レジスタ転送表，演算器入力端子転送表および出力端子転送表という 3 種の転送表を用いる．高位合成における転送表の考え方を利用し抽象化を実現した．本研究における転送表関連の処理は，2.5.1 項の転送表の作成のフェーズと 2.5.2 項の転送表内のデータフローの連結のフェーズに分かれる．

2.5.1 転送表の作成

データフローは制御フローから受け取る信号により処理を変えるため，各状態毎のデータフローを切り出すことができる．ここでは計算の順序や時系列を考える必要はない．各状態 S0，S1 で切り分けたデータフローを図 3 に示す．また，四角枠はレジスタ変数，丸枠はワイヤ変数や入力変数，出力変数を表している．+は加算器を表す．

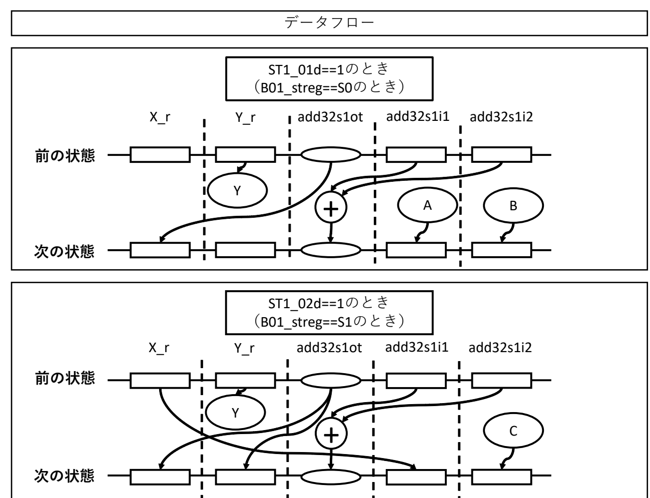

図 3 S0 および S1 の各状態で切り出したデータフロー

各出力変数に対して各状態毎のデータフローを作成し転送表とする．それらを制御フローの状態遷移に対応した順

ⓒ 2022 Information Processing Society of Japan

序でデータフローを接続させることで，RTL の数式・論理式を抽出することが可能となる．

ソースコード 3 を例として，作成した転送表を表 1 に示す．表内のデータフローは簡易化のため数式的な表記を用いているが，プログラムの内部表現としては木構造データ表現となっている．なお，表内の Keep は，レジスタにおいて前状態の値を保持することを意味する表現として使用している．

表 1　転送表作成の例

出力先の変数	状態	データフロー
X_r	S0	add32s1ot
	S1	add32s1ot
Y_r	S0	Keep
	S1	add32s1ot
Y	S0	Y_r
	S1	Y_r
add32s1ot	S0	add32s1i1 + add32s1i2
	S1	add32s1i1 + add32s1i2
add32s1i1	S0	A
	S1	X_r
add32s1i2	S0	B
	S1	C

2.5.2　転送表内のデータフローの連結

転送表と制御フローの状態遷移の遷移順序を用いて RTL 内の数式・論理式を抽出する．本ツールでは出力から入力までたどる形式になっており，転送表内のデータフローをたどることで実現する．たどる際のルールは以下の 3 つである．

(1) 参照しているデータフローの値が Keep のとき，同レジスタの前の状態の値を参照する．
(2) 参照しているレジスタがクロックの立ち上がりによって値が変わるレジスタの場合，たどる先の前の状態の値を参照する．
(3) 参照しているレジスタがクロック以外の立ち上がりによって値が変わるレジスタまたは assign 文の場合，たどる先の同状態の値を参照する．

状態遷移は S0 の次状態が S1 になるため，逆の順序を辿り，S1 の前状態は S0 と解析している．図 1 内の転送表を例にして，上記のルールにしたがってデータフローをたどる．Y が S0 のときの例を図 4 に，Y が S1 のときの例を図 5 に示す．

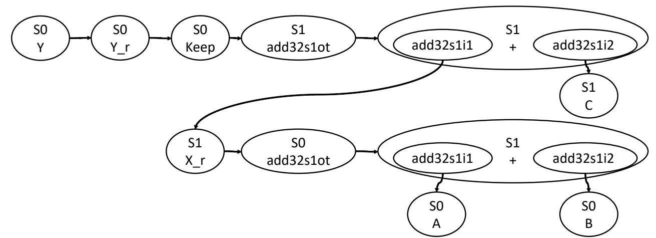

図 4　Y が S0 のとき，転送表内のデータフローを連結する例

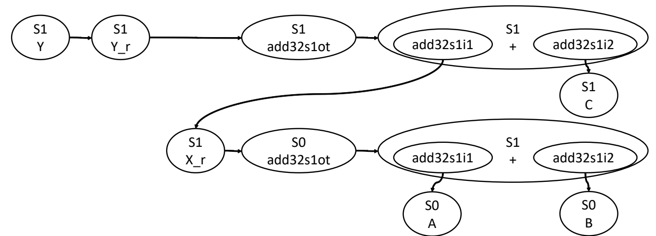

図 5　Y が S1 のとき，転送表内のデータフローを連結する例

どちらの例においても，Y=A+B+C の数式が抽出される．これらの処理を行い，制御フローのある RTL 記述を転送表を用いて抽象化することができる．

3.　検証および考察

3.1　検証

本節では，本研究により作成した抽象化ツールが生成する動作記述が，「元のコードと同じ機能を持ち」かつ「高位合成可能であること」を検証により示す．

2 章で述べた抽象化方法を用いて抽象化を行う．入力は RTL 記述で記述された Verilog HDL コードであるが，現状では高位合成によって生成された Verilog HDL コードのみを対象としている．加算と乗算のみで構成された動作記述に対して CWB による高位合成を行い生成した 4 つの RTL 記述を抽象化の対象とした．加算と乗算で構成された元の動作記述をソースコード 1 に示す．

ソースコード 1　元のソースコード（SystemC）

```
1  SC_MODULE(kasan_zyouzan)
2  {
3      sc_in_clk clk;
4      sc_in<int> a, b, c, d, e, f, g;
5      sc_out<int> x, y, z;
6      void kz(void)
7      {
8          int t;
9          wait();
10         while (1)
11         {
12             t = (a + b) * c;
13             x = t * e;
14             y = (b + c) + (t + f);
15             z = (c * d) * g;
16             wait();
17         }
```

```
18|     }
19|     SC_CTOR( kasan_zyouzan )
20|     {
21|         SC_THREAD( kz );
22|         sensitive << clk.pos();
23|     }
24| };
```

ソースコード 1 に対する，高位合成時の合成制約を表 2 に示す．状態数やレジスタの数が変化させるために表 2 の値を合成制約として選択した．

表 2　加算，乗算の合成制約

No.	加算器数 / 乗算器数	CLK 周期
1.	1 / 1	5ns
2.	2 / 2	5ns
3.	2 / 2	2.5ns
4.	2 / 2	3.7ns

また，4 つの RTL 記述の仕様と抽象化の合否を示したものを表 3 にそれぞれ示す．これら全ての RTL 記述内の制御フロー FSM は直線的な構造であり，分岐を含まない．

表 3　4 つの RTL 記述の仕様と抽象化の可否

No.	状態数	レジスタ数	RTL 記述行数	抽象化可否
1.	5	15	219	○
2.	4	15	224	○
3.	6	19	248	○
4.	4	13	204	○

加算と乗算の混ざった 4 つの RTL 記述を全て抽象化することができた．抽象化の結果のうち，No.1 の RTL 記述を抽象化して得られた SystemC ソースコードをソースコード 2 に示す．

ソースコード 2　抽象化後の動作記述（SystemC）

```
1 | SC_MODULE( s1_1_5 ){
2 |     sc_in_clk clk;
3 |     sc_in<int> a,b,c,d,e,f,g ;
4 |     sc_out<int> x,y,z ;
5 |
6 |     void p_s1_1_5(void){
7 |         wait();
8 |         while(1)
9 |         {
10|             x=(e*((a+b)*c));
11|             y=(((b+c)+f)+((a+b)*c));
12|             z=((c*d)*g);
13|         wait();
14|         }
15|     }
16|
17|     SC_CTOR( s1_1_5 ){
18|         SC_THREAD( p_s1_1_5 );
19|         sensitive << clk.pos();
20|     }
21| };
```

ソースコード 1 とソースコード 2 を比較すると，中間変数 t がなくなっていること，式の演算の順序は同じだが表記が異なることなどが変化した点として挙げられる．しかし，計算結果 (処理内容) は同じであるため抽象化には成功したと判断した．なお現状では，元の動作記述と抽象化後の動作記述が有する処理内容が同じかどうかの判断は目視で確認を行っている．

また，抽象化後の動作記述を再度高位合成にかけて新たな RTL 記述を作成することも可能であった．

3.2　考察

以上の検証結果から，パイプライン動作する演算モジュールの RTL 記述を抽象化することが可能であることが確認できた．しかしながら，現状では制約事項が多く残されている．例えば，FSM が分岐を持つ場合や抽象化の結果において残すべき中間変数がある場合への対応はできていない．本稿で示した手法は，FSM が直線的な場合に成り立つため，分岐がある場合の RTL 記述を複数の直線構造として解析を行うことができれば，分岐のある制御フローにおいても同様に抽象化できる可能性がある．さらに，抽象化できる Verilog コードとしては CWB で高位合成したものにのみ対応しているため，ハンドコーディングされた RTL への対応も検討する必要がある．今後はより汎用性のある逆変換ツールへの拡張を図る予定である．

4.　おわりに

本稿では，RTL で開発された過去の設計資産を動作記述へと抽象化し高位合成を利用することで再利用を可能にする手法について述べた．本稿では制御フローを有する RTL 記述を転送表を用いて抽象化する手法を示した．今後は作成した抽象化ツールに対して分岐を含むような複雑な FSM から構成される制御フローの場合にも対応できるようにして，抽象化可能な RTL 記述の広いツールの完成を目指す予定である．

謝辞　本研究を進めるに当たり，研究室 OB である新玲於奈氏・安楽亮大郎氏には多大なるご協力をいただいた．ここに感謝致します．

参考文献

[1] 安楽 亮大郎, 久我 守弘, 伊藤 寛人, 井戸 大介, 飯田 全広：“RTL 記述から動作記述への抽象化における組合せ回路の解析フロー，” 火の国情報シンポジウム 2021，情報処理学会九州支部，2021 年 3 月.
[2] W. Snyder, P. Wasson and D. Galbi : Verilator, https://www.veripool.org/projects/verilator/wiki/Intro，2017.
[3] 秋葉剛史，五十嵐真悟：ハードウェア動作記述変換方法及びそのためのプログラム，特許 4153732，株式会社東芝，2008 年 9 月 24 日.
[4] Rajdeep Mukherjee，Michael Tautschnig and Daniel Kroen-

ing : v2c – A Verilog to C Translator Tool, `http://www.cprover.org/hardware/v2c/`.

[5] Anushree Mahapatra and Benjamin Carrion Schafer : "VeriIntel2C: Abstracting RTL to C to maximize High-Level Synthesis Design Space Exploration, " Integration, the VLSI Journal 64, pp.1–12, Elsevier, 2019.

[6] 高前田（山崎）伸也 : Pyverilog, `https://github.com/PyHDI/Pyverilog/`

[7] NEC Corporation : CyberWorkBench, `https://jpn.nec.com/cyberworkbench/index.html`

[8] 若林一敏 : "ソフトウェアプログラムからハードウェア記述を合成する高位合成技術—プロセッサ以外の汎用プログラム実行機構—, " IEICE Fundamentals Review Vol. 6, No. 1, pp.37-50, 2012 月 7 日.

付録

ソースコード 3　Y=A+B+C の RTL 記述 (Verilig HDL)

```
1  module SAMPLE ( clk ,A ,B ,C ,Y ,RESET );
2  input  clk ;
3  input [31:0] A ;
4  input [31:0] B ;
5  input [31:0] C ;
6  output [31:0] Y ;
7  input  RESET ;
8  wire  ST1_01d ;
9  wire  ST1_02d ;
10
11 SAMPLE_fsm INST_fsm ( .clk(clk) ,.RESET(
       RESET) ,.ST1_02d(ST1_02d) ,.ST1_01d(
       ST1_01d) );
12 SAMPLE_dat INST_dat ( .clk(clk) ,.A(A) ,.B(
       B) ,.C(C) ,.Y(Y) ,.ST1_02d(ST1_02d) ,.
       ST1_01d(ST1_01d) );
13
14 endmodule
15
16 module SAMPLE_fsm ( clk ,RESET ,ST1_02d ,
       ST1_01d );
17 input  clk ;
18 input  RESET ;
19 output  ST1_02d ;
20 output  ST1_01d ;
21 reg B01_streg ;
22
23 parameter ST1_01 = 1'h0 ;
24 parameter ST1_02 = 1'h1 ;
25
26 assign ST1_01d = ( ( B01_streg == ST1_01 )
       ? 1'h1 : 1'h0 ) ;
27 assign ST1_02d = ( ( B01_streg == ST1_02 )
       ? 1'h1 : 1'h0 ) ;
28 always @ ( posedge clk or posedge RESET )
29  if ( RESET )
30   B01_streg <= ST1_01 ;
31  else
32   case ( B01_streg )
33   ST1_01 :
34    B01_streg <= ST1_02 ;
35   ST1_02 :
36    B01_streg <= ST1_01 ;
37   default :
38    B01_streg <= ST1_01 ;
39   endcase
40
41 endmodule
42
43 module SAMPLE_dat ( clk ,A ,B ,C ,Y ,
       ST1_02d ,ST1_01d );
44 input  clk ;
45 input [31:0] A ;
46 input [31:0] B ;
47 input [31:0] C ;
48 output [31:0] Y ;
49 input  ST1_02d ;
50 input  ST1_01d ;
51 wire [31:0] add32s1ot ;
52 wire  Y_r_en ;
53 reg [31:0] X_r ;
54 reg [31:0] Y_r ;
55 reg [31:0] add32s1i2 ;
56 reg [31:0] add32s1i1 ;
57
58 always @ ( C or ST1_02d or B or ST1_01d )
59  add32s1i2 = ( ( { 32{ ST1_01d } } & B ) |
       ( { 32{ ST1_02d } } & C ) ) ;
60 always @ ( X_r or ST1_02d or A or ST1_01d )
61  add32s1i1 = ( ( { 32{ ST1_01d } } & A ) |
       ( { 32{ ST1_02d } } & X_r ) ) ;
62 assign Y_r_en = ST1_02d ;
63 always @ ( posedge clk )
64  if ( Y_r_en )
65   Y_r <= add32s1ot ;
66 SAMPLE_add32s INST_add32s_1 ( .i1(add32s1i1
       ) ,.i2(add32s1i2) ,.o1(add32s1ot) );
67 assign Y = Y_r ;
68 always @ ( posedge clk )
69  X_r <= add32s1ot ;
70
71 endmodule
72
73 module SAMPLE_add32s ( i1 ,i2 ,o1 );
74 input [31:0] i1 ;
75 input [31:0] i2 ;
76 output [31:0] o1 ;
77
78 assign o1 = ( i1 + i2 ) ;
79
80 endmodule
```

© 2022 Information Processing Society of Japan