Artgraphics

SystemVerilogアラカルト

篠塚一也
アートグラフィックス

Document Revision:1.3,2020.03.15

www.artgraphics.co.jp

注意事項
（Caveat）

SystemVerilogの知識を個人的に習得する目的として本資料を活用して下さい。本資料を通して、業務（実践）で必要となるSystemVerilogに関する知識を習得して頂くのが本来の目的です。

転用目的（本来の目的と違った他の用途に使う事）で本資料を使用する事はご遠慮下さい。また、本資料から学んだ知識を転載する場合等は出典が本資料である事を明記して下さい。但し、他の著者の文書にも書かれている内容は、この限りではありません。本注意事項は現在及び過去に於ける弊社からの全てのフリーダウンロード資料に適用されます。

本注意事項に合意出来ない場合には、本資料を速やかに抹消して下さい。尚、ダウンロード記録は、依然として残ります。

はじめに

- この資料は、SystemVerilogに関する話題をLRMに沿って解説するコースメニューに従わずに、SystemVerilogの持つ特徴的な機能を主観的に選択し解説した技術資料です。主題の多くはVerilog HDLには見られない機能を重点的に取り扱っています。
- 資料の内容は、随時追加する為、何回かの改訂版を得て完成する事になります。
- 本文書中に引用する例は弊社の設計・検証ツールにより確認してありますが、絶対的に正しい記述であるかは保障の限りではありません。それらの例を実践で使用する場合には、使用者の責任において実行して下さい。
- 本資料では、SystemVerilog言語仕様IEEE Std 1800-2017をLRMと略称します。

 Copyright 2020 © Artgraphics. All rights reserved.

Unpackedアレイ

- Unpackedアレイは、不連続な領域として確保されます。

bit state[3];

- 従って、センシティビティリストにはunpackedアレイ全体を指定する事が出来ません。

```
4        always @(state[0],state[1],state[2]) begin
5        ...
6        end
```

```
8        always @(state) begin
9        ...
10       end
```

 Copyright 2020 © Artgraphics. All rights reserved.

プロセスとRNG

- SystemVerilogでは、プロセス毎に固有のRNG（Random Number Generator）を保有しています。
- 従って、並列プロセスに於いて、あるプロセスの乱数発生法を変更しても、他のプロセスの乱数発生には影響をしません。即ち、デバッグが容易になるという利点があります。
- LRMに依れば、親プロセスが子プロセスを生成すると、子プロセスのRNGのseedは親プロセスから引き渡されます。
- 子プロセスに引き渡されるseedはシミュレータの実装方式に依存する可能性があるので、ベンダーに問い合わせる事を勧めます。
- 子プロセスに異なるseedが引き渡されないと、次の例は正しく動作しない事に注意して下さい。

 Copyright 2020 © Artgraphics. All rights reserved.

Seedの重要性

- 以下の記述では、二つのgeneratorプロセスが適度なディレーを置いて、consumerにトランザクションを送信します。
- もし、二つのgeneratorプロセスが全く同じseedを与えられると、二つの子プロセスは全く同じ動作をする為に、並列処理の検証にはなりません。

```
 6         initial begin
 7                 lock = new(1);
 8                 fifo = new(16);
 9
10                 fork
11                         generator(1);
12                         generator(2);
13                         consumer();
14                 join
15         end
16
17 task automatic generator(int identifier);
18         forever begin
19                 repeat( $urandom_range(0,10) ) @(posedge clk);
20                 send(identifier);
21         end
22 endtask
```

 Copyright 2020 © Artgraphics. All rights reserved.

EnumとRTL論理合成

- Enumを利用すると、RTL論理合成に於いて論理の最適化が最大限に活かせます。Verilog HDLの時代には、FSMのステートをparameter文で定義しますが、SystemVerilogではenumを利用してステートを定義する事が出来ます。ステートをenumで定義すると以下の利点が得られます。

- ステートを表現する為に必要なビット数を厳密に算出する事が出来る。
- 全てのenumラベルを列挙したか否かを容易に判断する事が出来る

- 例として、以下の様な記述のFSMを見てみます。

 Copyright 2020 © Artgraphics. All rights reserved.

Enumは論理の最適化を最大限に活用する

- case文には全てのステートS0~S3が記述されている事が分かるので、このcase文は fully specifiedという事が言えます。ケース文にはdefaultが指定されていませんが、結果としてラッチは生成されません。

```
1    typedef enum logic [1:0] { S0=0, S1, S2, S3 } State;
2    ...
3    State   MooreState;
4
5    always @(posedge ClkM)
6            case (MooreState)
7              S0:
8                     begin
9                     Z <= 1;
10                    MooreState <= (!A) ? S0 : S2;
11                    end
12             S1:
13                    begin
14                    Z <= 0;
15                    MooreState <= (!A) ? S0 : S2;
16                    end
17             S2:
18                    begin
19                            Z <= 0;
20                            MooreState <= (!A) ? S2 : S3;
21                    end
22             S3:
23                    begin
24                            Z <= 1;
25                            MooreState <= (!A) ? S1 : S3;
26                    end
27           endcase
```

 Copyright 2020 © Artgraphics. All rights reserved.

仕様の標準化

- タスクやファンクション等を開発する際、開発手法に於いて個人差が出てくるのは当然の事ですが、それらの使用法は統一されていた方が望ましいのは明らかです。仕様の標準化をする為にSystemVerilogのインターフェース・クラスを利用する事が出来ます。
- 辞書を例にしてインターフェースクラスによる仕様の標準化を説明します。周知の様に、以下の様な機能を持つデータ構造を辞書と呼びます。

- 登録
- 検索
- 削除

- そこで、これらの機能の仕様を規格化する事にします。次は、規格例です。

 Copyright 2020 © Artgraphics. All rights reserved.

インターフェースクラスによる仕様の標準化

- 以下の様にして、インターフェースクラスを利用して仕様の標準化をします。

```
1    interface class dictionary_ifc #(type KEY=int,type DATA=int);
2    pure virtual function void put(KEY key,DATA data);
3    pure virtual function DATA get(KEY key);
4    pure virtual function void delete(KEY key);
5    pure virtual function int exists(KEY key);
6    endclass
```

- このインターフェースクラスでは、使用法を標準化しただけです。それぞれの辞書で異なる機能を実装する事が出来ます。
- 実装するには、インターフェースクラスをimplementsしなければなりません。

SystemVerilogアラカルト Copyright 2020 © Artgraphics. All rights reserved.

インターフェースクラスの実装例

```
1     class dictionary_t #(type KEY=int,type DATA=int) implements dictionary_ifc #(KEY,DATA);
2     DATA    value[KEY];
3
4     function void put(KEY key,DATA data);
5             value[key] = data;
6     endfunction
7
8     function DATA get(KEY key);
9             get = value[key];
10    endfunction
11
12    function void delete(KEY key);
13            value.delete(key);
14    endfunction
15
16    function int exists(KEY key);
17            exists = value.exists(key);
18    endfunction
19
20    function void print;
21            foreach(value[i])
22                    $display("%s = %0d",i,value[i]);
23    endfunction
24
25    endclass
```

SystemVerilogアラカルト Copyright 2020 © Artgraphics. All rights reserved.

ダイナミックアレイの利用法

- ダイナミックアレイは、実行時に領域を割り当てる事が出来る利点を持ちますが、その他にも効果的な利用法があります。
- 初期値が決定しているアレイでは、ダイナミックアレイを使用すると、初期値の数合わせをする必要が無くなります。
- 例えば、以下に示すアレイaはダイナミックアレイなので、初期化する数を5から8に変更しても宣言a[]を変更する必要がありません。

```
2    string  week[] = '{ "Sunday", "Monday", "TuesDay", "Wednesday",
3                         "Thursday", "Friday", "Saturday" };
4    int     a[] = '{ 5{10} };
```

 Copyright 2020 © Artgraphics. All rights reserved.

UVMメッセージユーティリティとUVM_DEBUG

- UVM_DEBUGは、比較的大きな値に設定されている為、このVERBOSITYを持つメッセージは通常プリントされません。
- 従って、このVERBOSITYを指定して`uvm_infoマクロでメッセージをプリントする様にしておくと、必要な時だけメッセージがプリントされます。
- 以下の例に於いて、通常は、16行目にあるメッセージのみプリントされます。

```
15    task run_phase(uvm_phase phase);
16        `uvm_info("UVM_NONE","none message",UVM_NONE)
17        `uvm_info("UVM_DEBUG","debug message",UVM_DEBUG)
18    endtask
```

- コマンドラインから以下の様に指定をすると、17行目のメッセージもプリントする事が出来ます。

```
+UVM_VERBOSITY=UVM_DEBUG
```

 Copyright 2020 © Artgraphics. All rights reserved.

wait_orderとシーケンスによるイベント制御

- イベントが一定の順序で発生する事を待つには、wait_orderを使用する事が出来ます。以下の記述は、イベントa、b、cがこの順で発生するのを待ちます。

```
2    event   a, b, c;
3
4        initial begin
5            wait_order(a,b,c)                       // wait events a-b-c
```

- 然し、この方法では、イベントが発生する間隔の指定をする事が出来ません。もう少し柔軟性のあるイベント制御には、シーケンスを利用する事が出来ます。下記の＠(abc)は、a、b、cが1クロック・サイクルの間隔をあけて真になるまで待ちます。

```
2    bit clk, a, b, c;
3
4        sequence abc;
5            @(posedge clk) a ##1 b ##1 c;
6        endsequence
7
8        initial begin
9            @(abc);     // wait until sequence of events a-b-c
```

 Copyright 2020 © Artgraphics. All rights reserved.

インターフェースの利用法

- インターフェースには、DUTに接続されている信号に対応する変数が宣言されています。しかも、多くの場合、クロッキング・ブロックも宣言されています。
- インターフェースをテストベンチとDUTを接続する為だけでなく、DUTからのレスポンスをサンプリングする為にも使用する事が出来ます。例えば、以下の様な回路の出力をサンプリングする事を考えます。

```
1    module swap(input logic clk,ref logic [7:0] a,b);
2
3        always @(posedge clk) begin
4            a <= b;
5            b <= a;
6        end
7
8    endmodule
```

- この回路は、バイトをスワップしますが即座に完了せず、NBA regionでスワップが完了します。従って、出力のサンプリングはそれ以降でなければなりません。

 Copyright 2020 © Artgraphics. All rights reserved.

インターフェースによるサンプリング

- インターフェースにクロッキング・ブロックが定義されていれば、DUTからのレスポンスを安全にサンプリングする事が出来ます。
- 下記の様にインターフェース内に記述したalwaysプロシージャは、DUTの出力を正しいタイミングでサンプリングします。

```
10    module top;
11    bit      clk;
12    logic [7:0] a, b;
13
14    simple_if SIF(.*);
15    test TEST(.*);
16    swap DUT(.*);
17
18        initial forever #10 clk = ~clk;
19        initial #100 $finish;
20
21    endmodule
```

```
1    interface simple_if(input bit clk,input logic [7:0] a,b);
2
3            clocking cb @(posedge clk); endclocking
4
5            always @(cb)
6                    $display("@%3t: a=%h b=%h",$time,a,b);
7
8    endinterface
```

 Copyright 2020 © Artgraphics. All rights reserved.

クラス以外の制約付き乱数発生法

- クラスを使用しなくても、乱数を発生する事が出来ます。しかも、乱数発生に制約を課す事も出来ます。下記の例を参考にして下さい。

```
10    module test(input logic [3:0] sum,logic co,output logic [3:0] a,b);
11    logic [3:0]      ta, tb;
12
13            initial begin
14                    repeat( 5 ) begin
15                            #10 assert( std::randomize(ta,tb) );
16                            a = ta;
17                            b = tb;
18                    end
19
20                    repeat( 5 ) begin
21                            #10 assert( std::randomize(ta,tb) with { ta < tb; } );
22                            a = ta;
23                            b = tb;
24                    end
25            end
26
27            initial
28                    forever @(co,sum)
29                            $display("%3t: a=%2d b=%2d {co,sum}=%2d",$time,a,b,{co,sum});
30
31    endmodule
```

 Copyright 2020 © Artgraphics. All rights reserved.

ランダムダイナミックアレイ

- ダイナミックアレイに乱数を発生する場合、アレイサイズにも制約を設定しなければなりません。制約なしでは、膨大なダイナミックアレイが生成されて、検証が終了しない状況に陥ります。

```
1    class sample_t;
2    rand byte       a[];
3    rand int        size;
4
5            constraint C1 { size inside { [1:10] }; }
6            constraint C2 { a.size == size; }
7            constraint C3 {
8                    foreach(a[i])
9                            i < a.size()-1 -> a[i+1] > a[i]; }
10
11   function void print;
12           $write("a:0..%0d",a.size-1);
13           foreach(a[i])
14                   $write(" %3d",a[i]);
15           $display;
16   endfunction
17   endclass
```

 Copyright 2020 © Artgraphics. All rights reserved.

ランダムダイナミックアレイのサイズ制御

- ランダムダイナミックアレイのサイズを制御する際、補助変数を使用すると、柔軟性を持つトランザクションを定義する事が出来ます。
- 行6：補助変数sizeを利用してランダムアレイaのサイズを制御している。

```
1    class sample_t;
2    rand byte        a[];
3    rand int         size;
4
5            constraint C1 { size inside { [1:10] }; }
6            constraint C2 { a.size == size; }
7            constraint C3 {
8                    foreach(a[i])
9                            i < a.size()-1 -> a[i+1] > a[i]; }
10
11   function void print;
12           $write("a:0..%0d",a.size-1);
13           foreach(a[i])
14                   $write(" %3d",a[i]);
15           $display;
16   endfunction
17   endclass
```

 Copyright 2020 © Artgraphics. All rights reserved.

ランダムダイナミックアレイのサイズ制御
補助変数の有効性

- 行30：実行時、sizeに制約を与えて、アレイのサイズを制御する。

```
19   module test;
20   sample_t        sample;
21
22           initial begin
23                   sample = new;
24                   repeat( 5 ) begin
25                           assert( sample.randomize() );
26                           sample.print();
27                   end
28                   $display("%s",{25{"-"}});
29                   repeat( 5 ) begin
30                           assert( sample.randomize() with { size == 7; } );
31                           sample.print();
32                   end
33           end
34   endmodule
```

 Copyright 2020 © Artgraphics. All rights reserved.

rand_mode()

- クラス内に定義されているランダム変数の制御をする為には、rand_mode()メソッドを使用します。
- 特定のランダム変数だけを有効、又は無効にする事が出来ます。全てのランダム変数を同時に有効、又は無効にする事も出来ます。

 Copyright 2020 © Artgraphics. All rights reserved.

rand_mode()の使用例

- 行２：a及びbをランダム変数として宣言しています。
- 行１９：ランダム変数aを無効にしていますが、変数bはランダム変数です。

```
1     class sample_t;
2     rand bit [7:0]   a, b;
3
4     function void post_randomize;
5             $display("a=%3d b=%3d",a,b);
6     endfunction
7     endclass
8
9     module test;
10    sample_t          sample;
11
12            initial begin
13                    sample = new;
14                    repeat( 5 )
15                            assert( sample.randomize() );
16
17                    $display("%s",{25{"-"}});
18
19                    sample.a.rand_mode(0);  // make a inactive
20                    repeat( 5 )
21                            assert( sample.randomize() );
22            end
```

constraint_mode()

- クラス内に定義されている制約の制御をする為には、constraint_mode()メソッドを使用します。
- 特定の制約だけを有効、又は無効にする事が出来ます。全ての制約を同時に有効、又は無効にする事も出来ます。

 Copyright 2020 © Artgraphics. All rights reserved.

constraint_mode()の使用例

- 行4：制約C1の定義
- 行5：制約C2の定義
- 行22：制約C2を無効にしていますが、制約C1は有効です。

```
1    class sample_t;
2    rand bit [7:0]   a, b;
3
4            constraint C1 { a + b == 32; }
5            constraint C2 { a < b ; }
6
7    function void post_randomize;
8            $display("a=%3d b=%3d",a,b);
9    endfunction
10   endclass
11
12   module test;
13   sample_t        sample;
14
15           initial begin
16                   sample = new;
17                   repeat( 5 )
18                           assert( sample.randomize() );
19
20                   $display("%s",{25{"-"}});
21
22                   sample.C2.constraint_mode(0);   // make C2 inactive
23                   repeat( 5 )
24                           assert( sample.randomize() );
25           end
```

SystemVerilogアラカルト Copyright 2020 © Artgraphics. All rights reserved.

randomize(null)

- クラスにランダム変数と制約が定義されていても、randomize(null)としてメソッドを呼び出すと、乱数は割り当てられずに、制約だけがチェックされます。
- この方法は、制約チェッカーとして知られている便利な使い方です。

Copyright 2020 © Artgraphics. All rights reserved.

自動カバレッジ計算

- カバーグループを定義する際に、カバレッジイベントを一緒に定義すると自動的にカバレッジ計算をする事が出来ます。
- 自動的にする為には、クラスに備わっているpost_randomize()メソッドを利用する方法が最も簡単です。

 Copyright 2020 © Artgraphics. All rights reserved.

自動カバレッジ計算の例

- 行7：カバレッジイベント@evが発生すると、カバレッジ計算が行われます。
- 行20：randomize()メソッドが呼ばれると、post_randomize()がカバレッジイベントを起こさせるので、カバレッジ計算が行われます。

```
1    class sample_t;
2    rand bit [3:0]  port;
3    event           ev;
4
5            constraint C_PORT { port inside { [0:5] }; }
6
7            covergroup cg @ev;
8                    coverpoint      port
9                    {
10                           bins value[] = { [0:5] };
11                   }
12           endgroup
13
14
15   function new;
16           cg = new;
17   endfunction
18
19   function void post_randomize();
20           ->ev;
21   endfunction
22
23   endclass
```

 Copyright 2020 © Artgraphics. All rights reserved.

ProgramとUVM

- SystemVerilogのprogramブロックは、特異性を持つ為UVMでは使用する事は出来ません。
- 特に、programブロックの全てのinitialプロシージャが終了すると、シミュレーションが終了するという特徴は、UVMと合いいれません。

 Copyright 2020 © Artgraphics. All rights reserved.

Programの例

- 行22が終了するとシミュレーションの終了となる為、行１２はスケジューリングはされますが、決して実行されません。従って、行９のイベント待ちが解除される事はありません。

```
1     module top;
2     bit      clk;
3     event    ev;
4
5     test TEST(.*);
6
7             initial begin
8                     $display("@%0t: main started",$time);
9                     @ev $display("@%0t: event released",$time);
10            end
11
12            initial #60 ->ev;
13
14            initial forever #10 clk = ~clk;
15
16    endmodule
17
18    program test(input bit clk);
19
20            initial begin
21                    $display("@%0t: test started",$time);
22                    #40 $display("@%0t: test is about to terminate",$time);
23            end
24
25    endprogram
```

SystemVerilogアラカルト Copyright 2020 © Artgraphics. All rights reserved.

未定義モジュールの宣言

- extern宣言したモジュールは、モジュールの定義が存在しなくてもエラボレーション時にエラーになりません。
- 多くのエンジニアで構成されているプロジェクトの初期段階では、未定義のモジュールが多々存在します。その様な場合、extern 宣言は便利な機能となります。

 Copyright 2020 © Artgraphics. All rights reserved.

未定義モジュールの宣言の例

- 行1で未定義モジュールdutの宣言をしています。
- 下記の記述を持つファイルだけでコンパイルしても、エラーが出ません。

```
1    extern module dut(input clk,d,output q);
2
3    module test;
4    bit      clk, d, q;
5
6    dut DUT(.*);
7
8    endmodule
```

 Copyright 2020 © Artgraphics. All rights reserved.

パッケージのインポート

- 名称に関する矛盾を回避する為に、パッケージをグローバルなスコープでインポートしない方が望ましい。
- モジュールのヘッダでパッケージ要素を参照している場合には、モジュール名称の直後でインポートする。
- 下記のモジュールヘッダ内でローカルにパッケージをインポートしている。

```
4    module ArithLogicUnit import Package::*; #(N=WIDTH)
5              (input [N-1:0] A,B,[OPERATOR_WIDTH-1:0] Select,
6              output CompareOut,[N-1:0] DataOut);
```

SystemVerilogアラカルト Copyright 2020 © Artgraphics. All rights reserved.

conditionalオペレータ ?:

- conditionalオペレータには、若干の注意が必要です。先ず、このオペレータは以下の様なシンタックスを持ちます([1])。

  conditional_expression ::=
  cond_predicate ? { attribute_instance } expr1 : expr2

- このオペレータの意味は、以下の様になっています。

cond_predicate	結果
真	expr1の評価結果を戻します。但し、expr2は評価されません。
偽	expr2の評価結果を戻します。但し、expr1は評価されません。
x又はz	expr1==expr2であれば、何れかの評価結果を戻します。 expr1==expr2が真でない場合には、それぞれの式のデータタイプから複雑なルールに従い計算されます([1],[3],[6])。

SystemVerilogアラカルト Copyright 2020 © Artgraphics. All rights reserved.

組み合わせ回路の検証([3])

- 組み合わせ回路の出力は入力で決定されます。従って、組み合わせ回路を検証する場合、入力値の変化に対してだけ結果を確認すれば良い事になります。
- 組み合わせ回路の入力を(i1,…,in)とすると、テストベンチのセンシティビティリストは、@(i1,…,in)となります。
- 例えば、以下の様な簡単な組み合わせ回路を仮定します。

```
1    module my_or(input a,b,output out);
2    assign out = a | b;
3    endmodule
```

- この回路をテストする場合、結果の確認は入力aとbの組が変化するケースだけを考慮すれば十分です。信号outの変化に配慮する必要はありません。即ち、センシティビティリストは@(a,b)であり、@(a,b,out)ではありません。
- 一般的には、テストベンチを次の様に書きます。

 Copyright 2020 © Artgraphics. All rights reserved.

組み合わせ回路の検証
テストベンチ（その１）

```
1     module my_or(input a,b,output out);
2     assign out = a | b;
3     endmodule
4
5     module top;
6     logic   a, b, out;
7
8     my_or DUT(.*);
9
10        initial begin
11            for( int i = 0; i < 4; i++ )
12                #10 {a,b} = i;
13        end
14
15        initial
16            $monitor("@%2t: a=%b b=%b out=%b",$time,a,b,out);
17
18    endmodule
```

@(a,b,out)

- このテストベンチは、正しいのですが、先程の前提条件に違反しています。即ち、テストベンチで使用している$monitorタスクでは、センシティビティリストが、@(a,b,out)となっているので、DUTの出力outを余分に考慮しています。
- outは、aとbに依存しているので、センシティビティリストに指定する必要がありません。寧ろ、outを指定しているので、正しい動作の検証になっているとは言えません。

 Copyright 2020 © Artgraphics. All rights reserved.

組み合わせ回路の検証
テストベンチ（その2）

- そもそも、$monitorタスクの使用は、幾つかの理由で望ましくない。
- ①$monitorタスクはPostponed領域で実行する。
- ②ただ一つの$monitorタスクしか有効にできない。
- ③検証すべき項目が多い場合、$monitorタスクでは対処できない。
- 等が挙げられます。今の場合、①の理由が当てはまります。例えば、以下の様に組み合わせ回路を記述しても、$monitorタスクは正しいと判断してしまいます。

```
1    module bad_my_or(input a,b,output logic out);
2        always @(a,b)
3            out <= a | b;
4    endmodule
```

組み合わせ回路の記述法としては正しくない。

- 従って、$monitorタスクを避ける事が望ましいと言えます。

 Copyright 2020 © Artgraphics. All rights reserved.

組み合わせ回路の検証
テストベンチ（その3）

- 組み合わせ回路は、本来、Active領域で動作します。従って、組み合わせ回路の検証をInactive領域で行うのが最適です。$monitorタスクによる組み合わせ回路の検証は、Postponed領域で実行するため、検証するタイミングが遅すぎます。
- Inactive領域で検証を行えば、bad_my_orの記述が正しくない事も判断する事ができます。

 Copyright 2020 © Artgraphics. All rights reserved.

組み合わせ回路の検証
テストベンチ（その4）

- 下記の検証部分は、Inactive領域で実行します。
- 従って、このテストベンチは、bad_my_orが正しいRTL記述ではないと判断する事が出来ます。

```
5     module top;
6     logic   a, b, out;
7
8     my_or DUT(.*);
9
10        initial begin
11            for( int i = 0; i < 4; i++ )
12                #10 {a,b} = i;
13        end
14
15        initial forever @(a,b)
16            #0 $display("@%2t: a=%b b=%b out=%b",$time,a,b,out);
17
18    endmodule
```

この文は
Inactive領域
で実行する。

ノンブロッキング代入文

- ノンブロッキング代入文の左辺には、automatic変数を使用する事は出来ません。例えば、下記のaはautomatic変数であってはなりません。

```
a <= b;
```

- 何故だかわかりますか？簡潔な解説を試みると良い演習問題になります。

 Copyright 2020 © Artgraphics. All rights reserved.

論理合成ー背景

- 論理合成の分野は成熟期に至り、今や、EDA分野の一部門というより学問といえるまで成長しました。この為、一般の場所で、もはや、論理合成のアルゴリズムを議論する機会も少なくなってしまっている様です。
- 論理合成ツールの動作を十分に理解している設計者にとっては、その様な動向も気に留める必要はないと思えますが、論理合成結果を予測するに十分な知識と経験を持たない設計者にとっては、自ずから不利な状況になっています。
- そこで、以下では論理合成の仕組みを原始的な方法で解説し、論理合成結果を予測する為の技術を養う事にします。回路は、組み合わせ回路を基本とします。シーケンシャル回路は、論理合成のルールに従い、レジスタを生成するだけなので、合成結果の予測は簡単です。
- 本来は、論理合成のアルゴリズムを基礎から解説するのが望ましいのですが、準備するには多大の時間が必要であるとともに、解説を理解するには、専門的な知識が要求される為、この方法は実際的ではないといえます。

SystemVerilogアラカルト Copyright 2020 © Artgraphics. All rights reserved.

論理合成の基礎ー前提と目標

- 組み合わせ回路の記述がSystemVerilogで記述されているとすると、記述をブーリアン式で表現するのが目標です。
- 回路記述に対応するブーリアン式を$F(x_1, x_2, \ldots, x_n)$とします。ここで、$(x_1, x_2, \ldots, x_n)$は、回路への入力を意味します。
- ブーリアン式には、次の様なBoolの展開定理と呼ばれる有用な定理があります（[7]）。この定理は、Shannonの展開定理とも呼ばれます。

$$F(x_1, x_2, \ldots, x_n) = x_i * F_{x_i} + {x_i}' * F_{{x_i}'}$$

- ここで、F_{x_i}は$F(x_1, x_2, \ldots, x_n)$に於いて、$x_i == 1$を代入した式を意味します。同様に、$F_{{x_i}'}$は$F(x_1, x_2, \ldots, x_n)$に於いて、$x_i == 0$を代入した式を意味します。

 Copyright 2020 © Artgraphics. All rights reserved.

論理合成の基礎ーmux2

- 先ず、以下の様なmultiplexerの記述を仮定します。

```
1    module mux2(input a,b,sel,output logic out);
2
3        always @(a,b,sel)
4            if( sel == 1'b1 )
5                out = a;
6            else
7                out = b;
8
9    endmodule
```

- この記述に対応するブーリアン式は$F(a, b, sel)$で、以下の関係式が得られます。

$$F(a, b, 1) = a$$
$$F(a, b, 0) = b$$

- 従って、Boolの展開定理から、記述は以下の組み合わせ回路に等しくなります。

$$F(a, b, sel) = sel * a + sel' * b$$

- これは、よく知られたmultiplexerの表現式で、mux2の論理合成は完了しました。

SystemVerilogアラカルト Copyright 2020 © Artgraphics. All rights reserved.

論理合成の基礎ーmux4

- mux2の合成手法を確実に理解しておくと、机上で殆ど全ての合成を行う事ができる様になります。例えば、下記の様な記述を仮定します。

```
1    module mux4(input logic a,b,c,d,logic s1,s0,output logic out);
2
3        always @(s1,s0,a,b,c,d)
4            case ({s1,s0})
5            0:  out = a;
6            1:  out = b;
7            2:  out = c;
8            3:  out = d;
9            default:    out = 'x;
10           endcase
11
12   endmodule
```

- この記述が以下の様に合成されるのは、容易に想像が出来る様になります。

$$F(a,b,c,d,s1,s0) = s1' * s0' * a + s1' * s0 * b + s1 * s0' * c + s1 * s0 * d$$

 Copyright 2020 © Artgraphics. All rights reserved.

論理合成の基礎ー等号

- a==bは、a ~^ bと同じなので、次のようなブーリアン式になります。

$$F(a == b) = a * b + a' * b'$$

- 従って、a==1は、以下の様になります。

$$F(a == 1) = a$$

- 同様に、a==0は、以下の様になります。

$$F(a == 0) = a'$$

- これらの式から以下の事が分かります。

$$F(\{s1, s0\} == 0) = s1' * s0'$$
$$F(\{s1, s0\} == 1) = s1' * s0$$
$$F(\{s1, s0\} == 2) = s1 * s0'$$
$$F(\{s1, s0\} == 3) = s1 * s0$$

 Copyright 2020 © Artgraphics. All rights reserved.

論理合成——一般的な記述

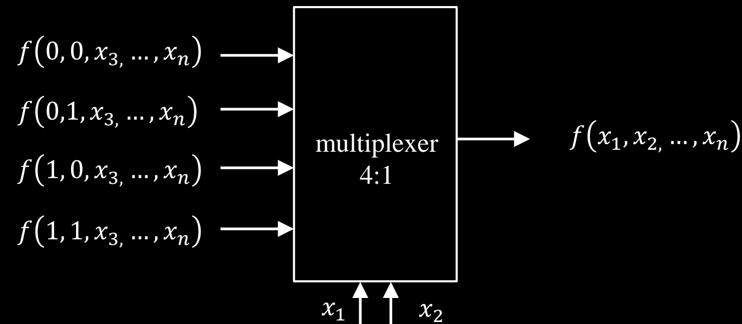
- 一般的な記述でも、同じ手順を適用する事ができます。例えば、if-else if-else if-elseの様な構造を取っていても、(if-else if-else if-)elseの様にまとめて前述の手順を適用する事ができます。
- その際、以下のブール代数の定理を記憶しておくと便利です。

$$a * b' + b = a + a' * b = a + b$$
$$a * b + b * c + a' * c = a * b + a' * c$$

 Copyright 2020 © Artgraphics. All rights reserved.

Boolの展開定理の応用

- 4入力multiplexerを使用すれば、より小さい回路からn変数の回路を構築する事ができます。



 Copyright 2020 © Artgraphics. All rights reserved.

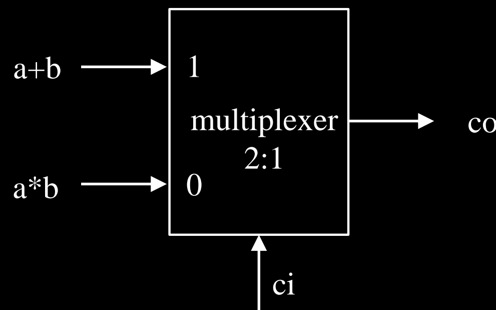
Boolの展開定理とフルアダーのcarry-out

- フルアダーのcarry-outは、majority関数で以下の様に表現されます。

$$co = a * b + b * ci + ci * a$$

- この式に、ciに関してBoolの展開定理を適用すると、以下の様な式を得ます。

$$co = ci * (a + b) + ci' * (a * b)$$



SystemVerilogアラカルト Copyright 2020 © Artgraphics. All rights reserved.

シーケンシャル回路のクロック信号

- シーケンシャル回路の記述で、センシティビティリスト以外には現れない信号は、クロック信号として認識されます。
- センシティビティリストに指定された非同期信号は、必ず、ビヘイビアで使用されなければならない。
- センシティビティリストに指定されたエッジと条件判定は一致しなければならない。

clkはセンシティビティリストで指定されているが、ビヘイビアでは使用されていないので、クロック信号と判断される。

```
5        always @(posedge clk,posedge reset)
6            if( reset == 1'b1 )
7                count <= 0;
8            else if( load == 1'b1 )
9                count <= data_in;
10           else if( up_down )
11               count <= count+1;
12           else
13               count <= count-1;
```

posedge resetと指定しているので、reset == 1’b1と判定する。

 Copyright 2020 © Artgraphics. All rights reserved.

参考文献

文献[1]は、誰もが一度は目を通す言語仕様書です。SystemVerilogは、数年に一度改訂されます。その前に一読する事を薦めます。

文献[2]は、UVMに関するユーザガイドで、一読に値します。然し、予備知識なしでは読破するのは難しいかも知れません。

文献[3]は、文献[1]に従い、SystemVerilogを忠実に解説した国内唯一のSystemVerilog全般に関する包括的な資料です。SystemVerilogを深く学習したい人は読むべき資料です。末永く使用できる文献です。

文献[4]は、UVMを使用する検証技術者の為に書かれた資料です。UVMを検証作業に適用する前に読むべき資料です。

文献[5]は、検証に必要なSystemVerilogの知識をまとめた国内最初の専門書です。この文献を読むと、再発見する事実が多いと思います。

文献[6]は、SystemVerilogへの初心向けの入門書です。検証には深入りしていないので、設計技術者にも薦めます。

文献[5]、又は[6]から学習を進め、知識習得度に従って次のステップを決めるのは、効率的な学習法だと思います。

文献[7]は、論理合成に関する非常に優れた専門書です。難易度が高いので、読破するにはかなりの覚悟が必要です。

[1] IEEE Std 1800-2017: IEEE Standard for SystemVerilog – Unified Hardware Design, Specification and Verification Language.

[2] Universal Verification Methodology (UVM) 1.2 User’s Guide, Accellera, October 8, 2015.

[3] 実践SystemVerilog入門, アートグラフィックス, February 16, 2020.

[4] 実践UVM入門, アートグラフィックス, November 30, 2019.

[5] 篠塚一也、SystemVerilogによる検証の基礎、森北出版 2020.

[6] SystemVerilog入門, アートグラフィックス, February 16, 2020.

[7] Giovanni De Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill 1994.

 Copyright 2020 © Artgraphics. All rights reserved.