IEEE Std.1800-2005
（SystemVerilog)
テストベンチ
チュートリアル

JEITA　SystemVerilog Task Group

Agenda

- SystemVerilog タスク・グループ紹介
- 検証テクノロジの現状
  - 検証テクノロジの乱立
  - 基本的なテストベンチの役割
  - 最近の検証トレンド
  - IEEE Std. 1800-2005（SystemVerilog）概要
- テストベンチの説明
  - コンストレイント・ランダム・スティミュラス
  - カバレッジ
  - レスポンス
  - テストベンチ、検証環境の再利用
  - program & clocking block
  - クラス

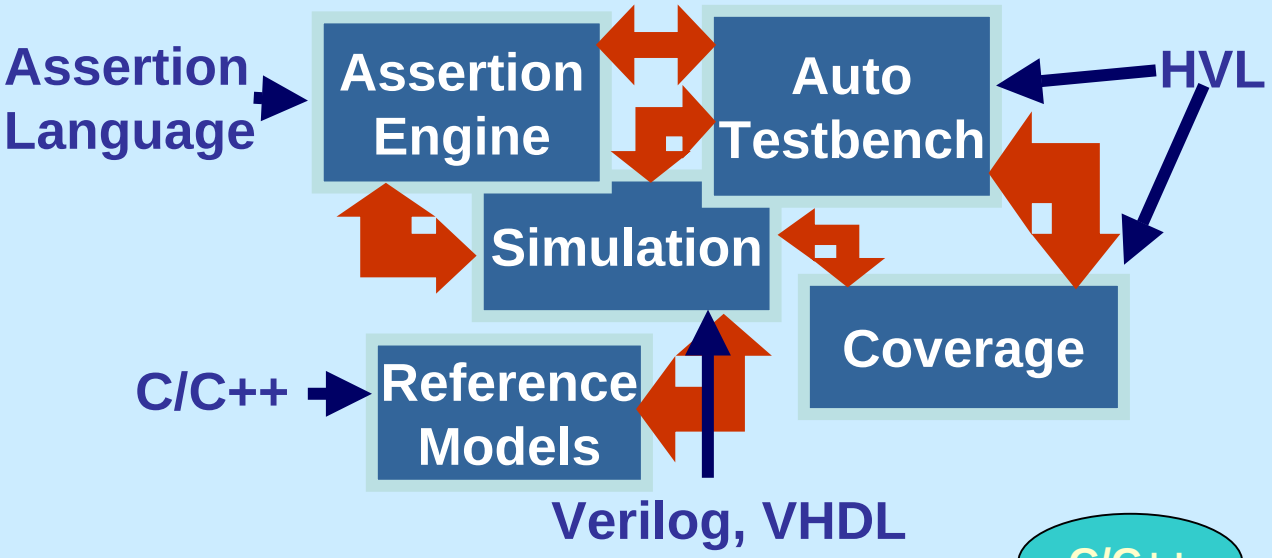
SystemVerilogタスク･グループ

- JEITA：社団法人 電子情報技術産業協会
- 「JEITA EDA技術専門委員会／標準化小委員会傘下の SystemVerilog Task Group (SV-TG)
  - SystemVerilogの国際標準化活動に日本から参画
  - メンバー企業 10社

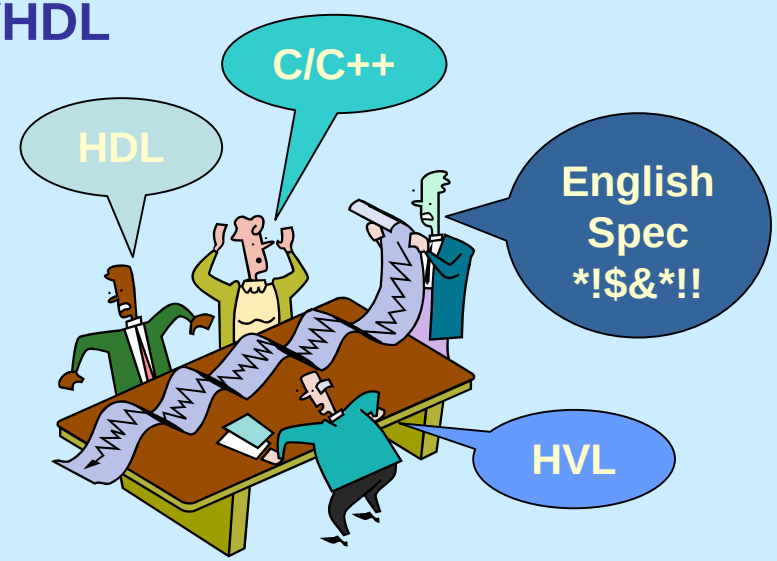
沖ネットワークエルエスアイ	日本シノプシス
三洋電機	松下電器産業
図研	メンター･グラフィックス･ジャパン
東芝	富士通
日本ケイデンス･デザイン･システムズ社	ルネサステクノロジ

(注)五十音順, 敬称略

検証テクノロジの乱立



- 検証の早期完了のための新しい手法
- ツール専用の検証言語の乱立
  - 習得が困難
  - ポータビリティの欠如
    - 技術革新の妨げ
    - ツール価格の上昇

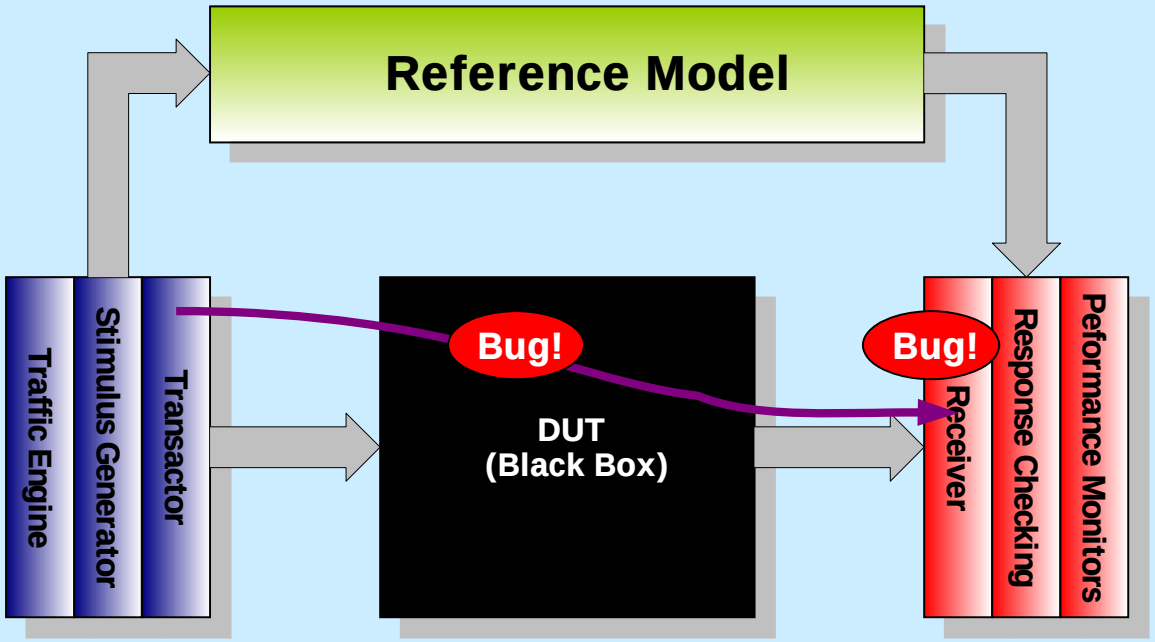


JEITA

SystemVerilog

基本的なテストベンチの役割

- バグの発見のためには. . .
  1. スティミュラスを加える
  2. 出力で結果を観測する
  3. 不一致があれば、デバッグする
- 検証を完了させるためには. . .
  - 全てのテスト項目について 1.～3. を実施する



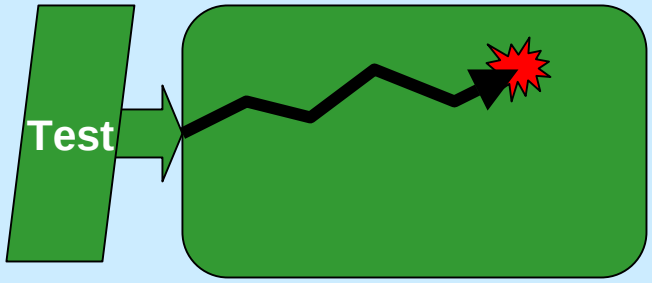
最近の検証トレンド

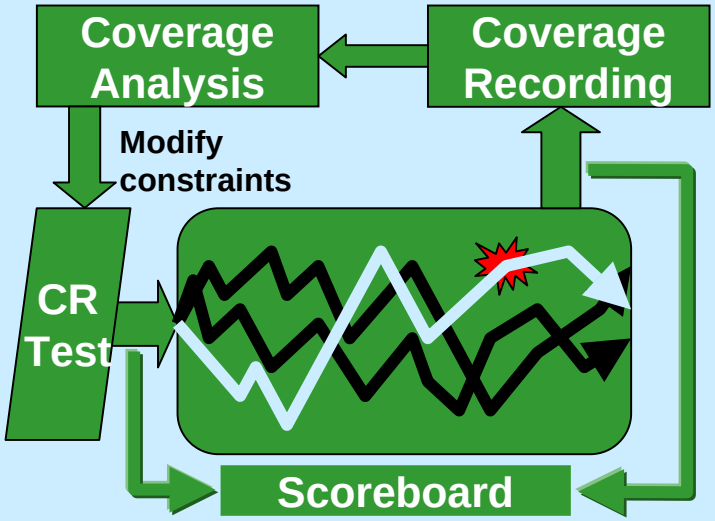
- カバレッジ・ドリブン検証
  - コンストレイント・ランダムスティミュラス生成
  - カバレッジ covergroup/cover
  - レスポンス（目視チェックではありません）
    - アサーション assert
    - スコアボード
- テストベンチ、検証環境の再利用
  - 部品化
  - 再利用

カバレッジ・ドリブン検証

- ダイレクト・テスト
  - テスト作成者による手作業
    - 意図した事象のシナリオを1つ1つコーディング
  - 制御性の問題
    - 入力スティミュラスのみで、深いサイクルでの回路動作を予測しなければならない
  - 調整がきかない
    - 条件の異なるテストの生成にはさらに手作業が必要

- カバレッジ・ドリブン検証
  - コンストレイント・ランダムスティミュラス生成
    - 制約の修正 ＝ テスト生成
  - ダイレクト・スティミュラス
    - 到達しにくいコーナーケース
  - ファンクショナルカバレッジ
    - デザインの動作シナリオは予測不可
    - 意図した事象を捉える仕組みが必要
  - スコアボード
    - 予測できないシナリオの結果をチェック可能



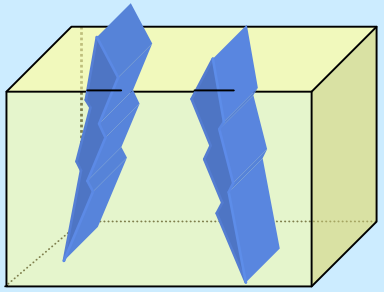


JEITA

SystemVerilog

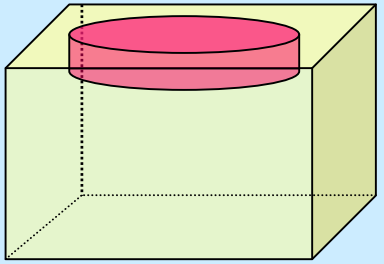
ランダムスティミュラスの導入

- 直接指定（ダイレクト検証）
  - 全て手作業による記述
  - 到達しにくいコーナーケースのカバーが可能
  - 人手によるテストケースの作成には限界がある



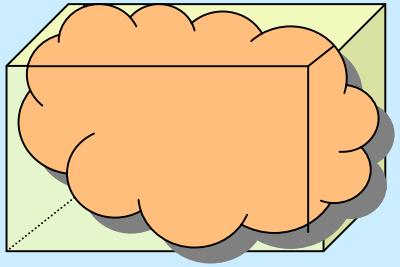
設計機能領域

- ランダムスティミュラス
  - テストパターン生成の自動化（記述は簡単）
  - 広く浅いカバレッジ
  - 冗長なテストを行う可能性がある



設計機能領域

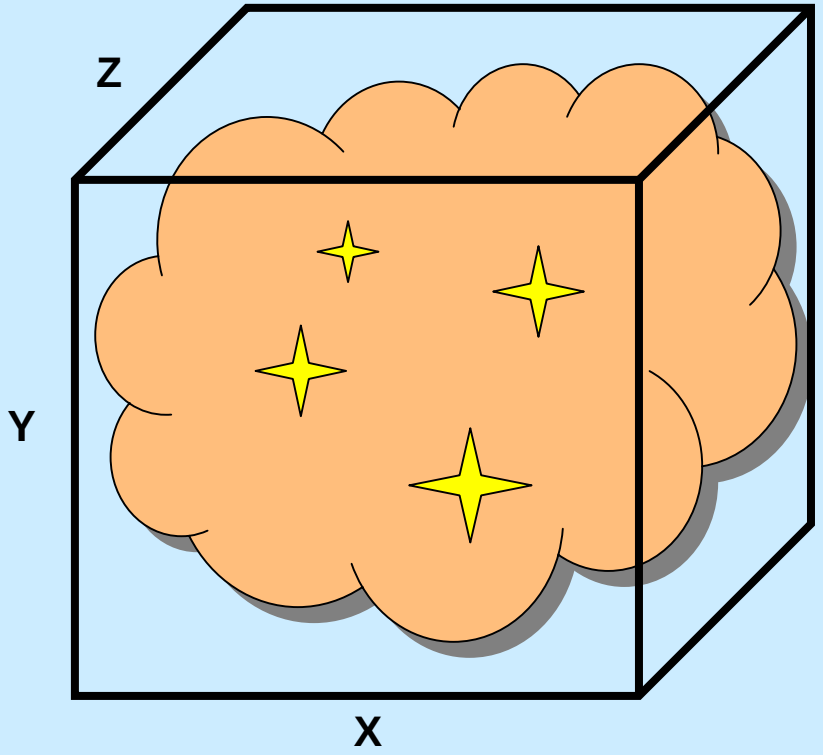
- コンストレイント・ランダムスティミュラス
  - テストパターン生成の自動化
    （記述は比較的簡単、制約を数式で記述）
  - 広く深いカバレッジ
  - 冗長性を排除し、多くの機能をカバー
  - 人手による規則性（偏り）を排除



設計機能領域

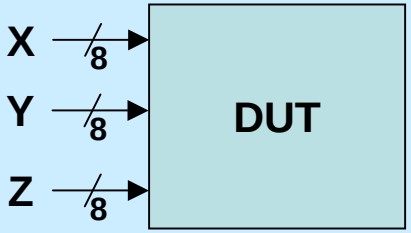
Constraint Solver ― ランダム制約の解決

- ランダム変数の宣言 – X, Y, Z
- 制約の宣言 – Y < 42, X ≤ Y ≤ Z
- 与えられた制約を満たす変数値の集合を求める
- 解空間からランダムに値を取出す

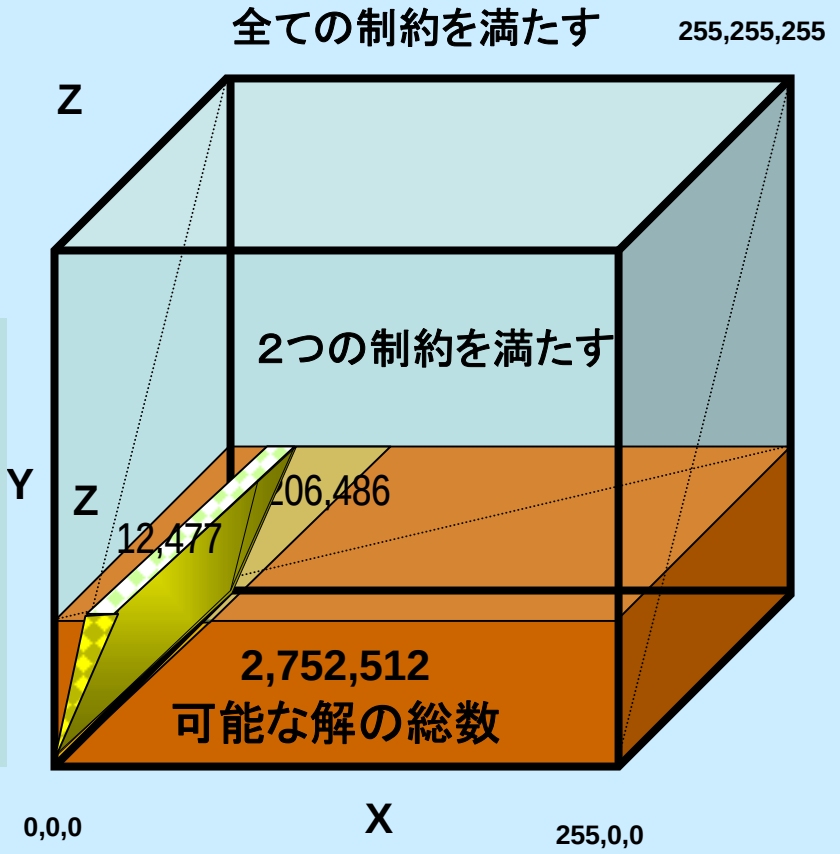


解空間の計算

```
# 21 26 32
# 1d 22 91
# 04 0e d0
# 0f 28 60
# 14 20 d1
# 07 1f 80
```

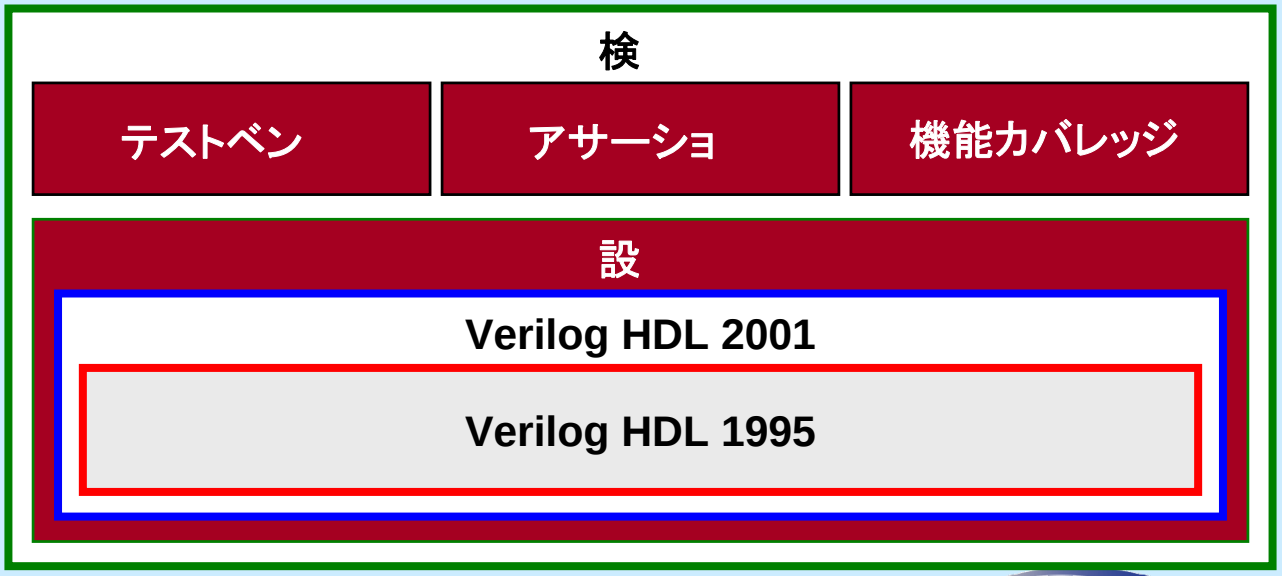


制約	可能な解の総数
制約なし	$2^{8+8+8} = 2^{24} = 16,777,216$
Y < 42	$42 \times 2^{8+8} = 2,752,512$
X ≤ Y ≤ Z	$\sum_{k=1}^{256}(\sum_{n=1}^{k} n) = 2,829,056$
X[7:4]=Z[3:0]	$2^4 \times 2^{8+4+4} = 1,048,576$



SystemVerilog の概要

- IEEE Std. 1364-2001 (Verilog HDL)の拡張
- IEEE Std. 1800-2005 標準
- デザインと検証のために言語を統合
  - HDVL (Hardware Description and Verification Language)
- SystemVerilog はVerilog HDL の新バージョン!



スティミュラスのランダム生成

randomize()メソッド、rand、randc宣言でランダム値を生成する

キーワード	機能
randomize()	ランダム値を生成する
rand	変数が通常のランダム値の生成することを宣言する
randc	変数がcyclicなランダム値の生成することを宣言する "cyclic"とは、例えば2ビットの信号なら4サイクルで0,1,2,3を1回づつ発生させること

```
class Transaction;
  rand  bit [31:0]  Addr;
  randc bit [1:0]   BurstType;
  rand  bit [4:0]   DataSize;
endclass
Transaction t1 = new();
initial begin
  repeat(100) begin
    t1.randomize();
  end
end
```

```
Addr=7337dd17 BurstType=0 DataSize=0e
Addr=30d4b41b BurstType=1 DataSize=15
Addr=cac8ccaa BurstType=3 DataSize=14
Addr=05e23536 BurstType=2 DataSize=02
 .
 .
```

0～3を1回ずつ発

ランダム生成の制約

constraintブロックでランダム生成に制約条件をつける

キーワード	機能
>、>=、<、<=、==	値の大小関係、等しいことの指定
inside	生成値の範囲の制約
->	条件判定

```
class Transaction;
  rand  bit [31:0]  Addr;
  randc bit [1:0]   BurstType;
  rand  bit [4:0]   DataSize;
  constraint c0  {
    Addr inside { [32'h0 : 32'hFFF] };
    Addr[1:0] == 2'b00;
    (BurstType != 2'b01) -> (DataSize == 5'h00);
    (BurstType == 2'b01) -> (DataSize >= 5'h10);
  }
endclass
```

```
Addr=00000b98 BurstType=1 DataSize=1f
Addr=00000024 BurstType=3 DataSize=00
Addr=00000e40 BurstType=2 DataSize=00
Addr=00000840 BurstType=1 DataSize=12
Addr=0000057c BurstType=0 DataSize=00
 ・
 ・
```

・アドレス範囲が0～0xFFF
・アドレス下位2ビットが0

BurstTypeが
1の時のみ
データサイズを
0x10以上

発生確率の重み付け

キーワードdistでランダム発生確率に重み付けを行う

キーワード	機能
dist	生成値の発生確率への重み付けを行う

```
class Transaction;
  rand  bit [31:0]  Addr;
  randc bit [1:0]   BurstType;
  rand  bit [4:0]   DataSize;
  rand  bit         Lock;
  constraint c0  {
    Addr inside { [32'h0 : 32'hFFF] };
    Addr[1:0] == 2'b0;
    (BurstType != 2'b01) -> (DataSize == 5'h00);
    (BurstType == 2'b01) -> (DataSize >= 5'h10);
    Lock dist {0:=80, 1:=20};
  }
endclass
```

```
Lock=0 Addr=0000057c BurstType=0 DataSize=00
Lock=0 Addr=00000b98 BurstType=1 DataSize=1f
Lock=0 Addr=00000024 BurstType=3 DataSize=00
Lock=1 Addr=00000e40 BurstType=2 DataSize=00
Lock=0 Addr=0000092c BurstType=0 DataSize=00
 .
 .
 .
```

Lock=1となる確率が20%

ランダムシーケンスの生成

randsequenceでランダムシーケンスを生成する

キーワード	機能
:=	リスト内プロダクション(シーケンス要素)の選択確率を重み付ける
if...else/case	プロダクションの選択条件を指定する
repeat(n)	プロダクションをn回繰り返す
rand join	選択したプロダクションの順序を変更しない

基本の記述例

```
randsequence( main )
  main : first second done ;
  first : add := 3 | dec := 2 ;
  second : pop | push ;
  done : { $display("done"); } ;
  add : { $display("add"); } ;
  dec : { $display("dec"); } ;
  pop : { $display("pop"); } ;
  push : { $display("push"); } ;
endsequence
```

3シーケンスが
ランダムに発生

```
add pop done
add push done
dec push done
dec pop done
```

repeatの例

```
randsequence()
  ...
  PUSH_OPER : repeat( 2 ) PUSH ;
  PUSH : ...
endsequence
```

rand joinの例

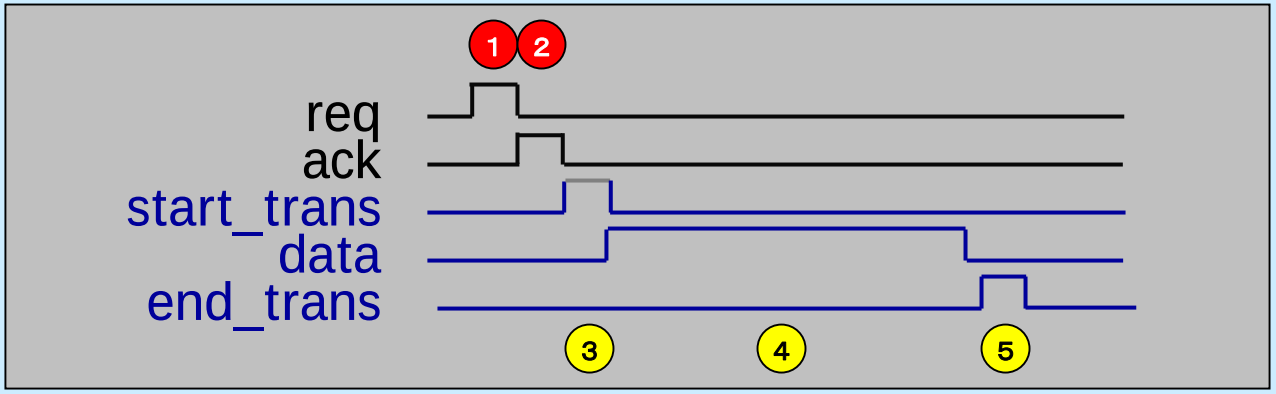
```
randsequence( TOP )
  TOP : rand join S1 S2 ;
  S1 : A B ;
  S2 : C D ;
endsequence
```

```
A B C D
A C B D
A C D B
C D A B
C A B D
C A D B
```

"AB"、"CD"の
順序はそのまま

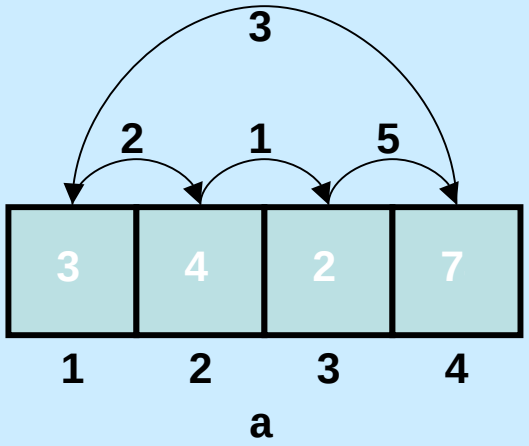
cover ステートメントによるカバレッジ

- cover は、シーケンス、プロパティに適用できる
  - assert と同じシーケンスやプロパティの発生を記録
  - 例えば、バスプロトコルの信号シーケンスをカバレッジ計測するのに適している



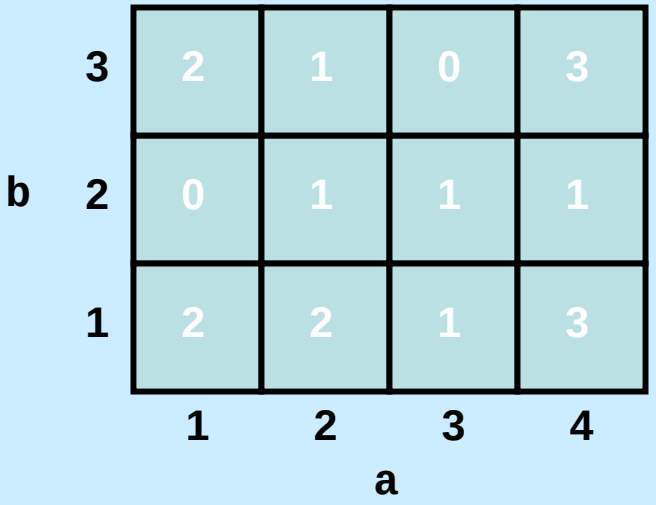
covergroup によるカバレッジ・モデル

- 変数、エクスプレッションの値に対するカバレッジ
  - テストにおいて発生するデータ値に関する情報を記録
  - 値の変化（トランジション）も定義可能
  - 例えば、テストで生成された RAM アドレスのカバレッジなどに適している



単純なカバレッジ: a==1 は何回?

トランジションカバレッジ:
a==1 の後に a==2 となったのは何回?



クロスカバレッジ:
a==1 の時に b==1 は何回?

機能カバレッジ情報の収集

covergroup ブロックにカバレッジポイントを記述する

キーワード	機能
coverpoint	収集する機能カバレッジポイントを指定する
bins	coverpointで指定した情報をどのビン(=容器)に容れるかを指定する 複数のビンが指定可能
iff	coverpointのサンプル条件を指定する

```
covergroup addr_region @(posedge clk);
  coverpoint cp_addr {
    bins    bank0 = {['h0000_0000:'h0000_03FF]} iff(!reset);
    bins    bank1 = {['h0000_0400:'h0000_07FF]} iff(!reset);
    bins    bank2 = {['h0000_0800:'h0000_0BFF]} iff(!reset);
    bins    bank3 = {['h0000_0C00:'h0000_0FFF]} iff(!reset);
  }
  coverpoint cp_burst {
    bins    BurstType0 = {2'b00} iff(!reset);
    bins    BurstType1 = {2'b01} iff(!reset);
    bins    BurstType2 = {2'b10} iff(!reset);
    bins    BurstType3 = {2'b11} iff(!reset);
  }
endgroup
```

クロスカバレッジ情報の収集

キーワードcrossによりクロスカバレッジ情報を収集する

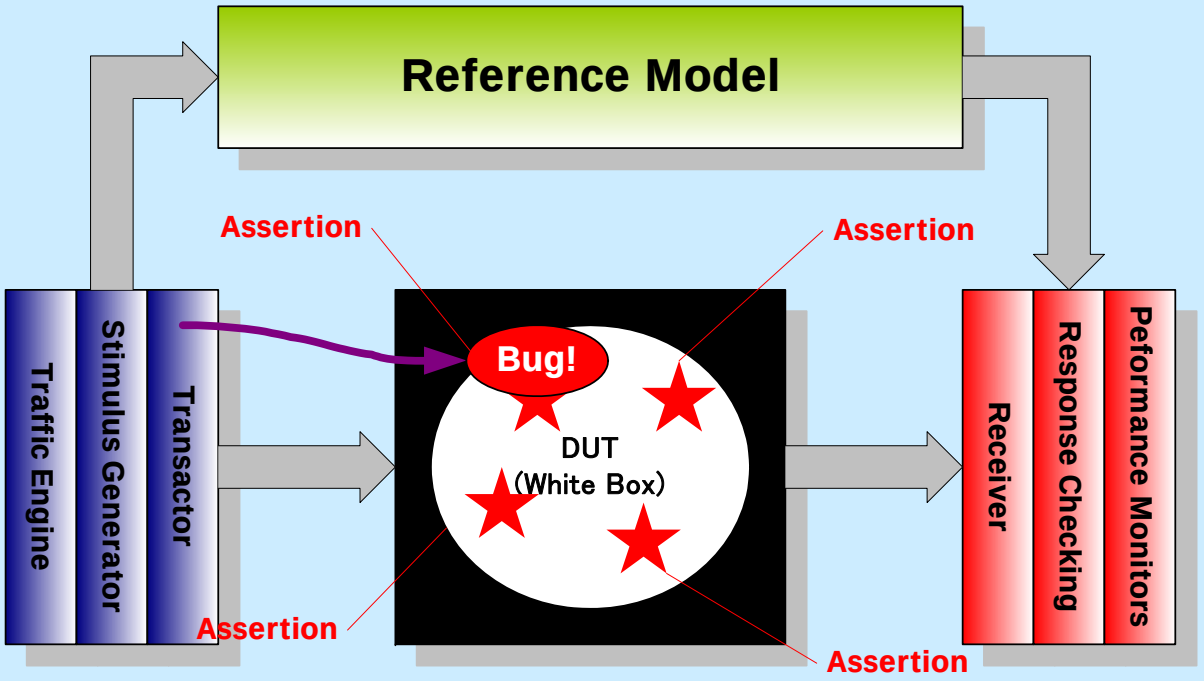
キーワード	機能
cross	クロスカバレッジ情報を収集するカバレッジポイントを指定する

```
covergroup addr_region @(posedge clk);
 coverpoint cp_addr {
    bins    bank0 = {['h0000_0000:'h0000_03FF]} iff(!reset);
    bins    bank1 = {['h0000_0400:'h0000_07FF]} iff(!reset);
    bins    bank2 = {['h0000_0800:'h0000_0BFF]} iff(!reset);
    bins    bank3 = {['h0000_0C00:'h0000_0FFF]} iff(!reset);
  }
  coverpoint cp_burst {
    bins    BurstType0 = {2'b00} iff(!reset);
    bins    BurstType1 = {2'b01} iff(!reset);
    bins    BurstType2 = {2'b10} iff(!reset);
    bins    BurstType3 = {2'b11} iff(!reset);
  }
  addr_burst : cross cp_addr, cp_burst;
endgroup
```

```
cp_addr   cp_burst      # hits
============================
bank0     BurstType0    4
bank0     BurstType1    3
bank0     BurstType2    5
bank0     BurstType3    2
 .
 .
bank3     BurstType0    2
bank3     BurstType1    3
bank3     BurstType2    3
bank3     BurstType3    3
============================
```

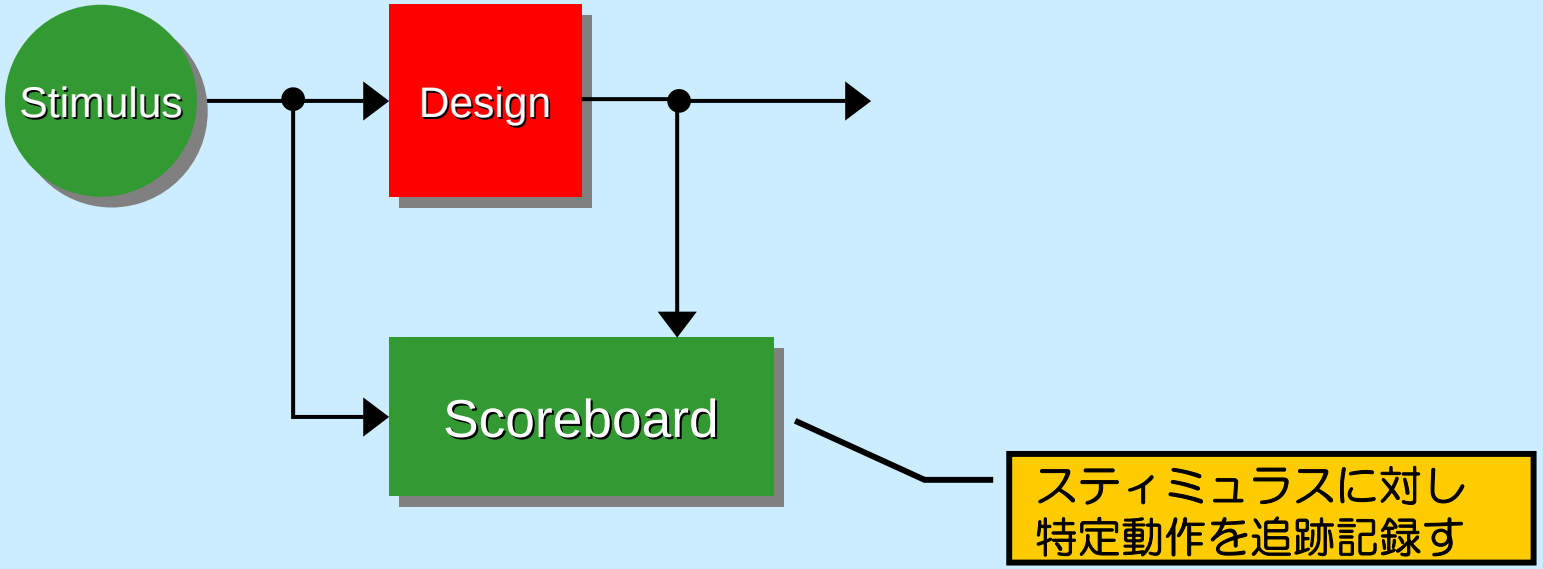
レスポンスのチェック

- レスポンスをチェックする目的
  - 不正動作の検出
  - cover ステートメントによるカバレッジ計測
- 実現方法
  - アサーション
  - スコアボード



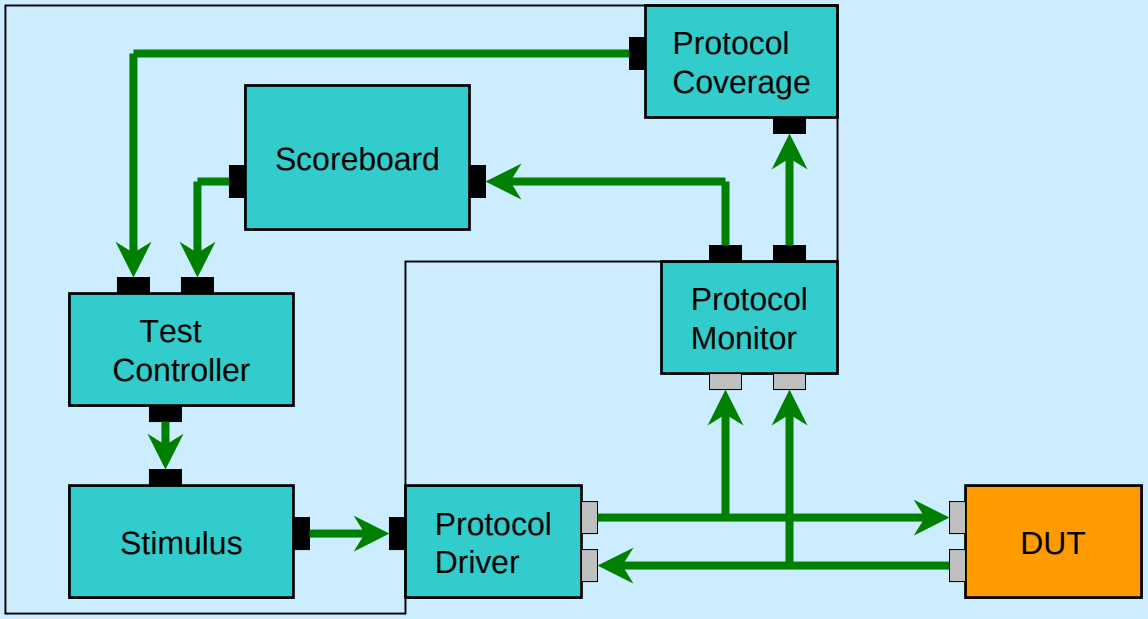
スコアボード

- スティミュラスに対する DUT の入出力を追跡記録
- 実装はアプリケーション固有だが、パターンは一般化が可能
- 記録手段：メールボックス、連想配列、キュー、など



テストベンチ、検証環境の再利用

- 部品化
  - コンポネントのカプセル化
  - プロトコルのカプセル化
- 再利用
  - 一貫したトランザクションインタフェース
  - 構築済みインフラストラクチャー
  - 高い抽象レベル



プログラム・ブロック

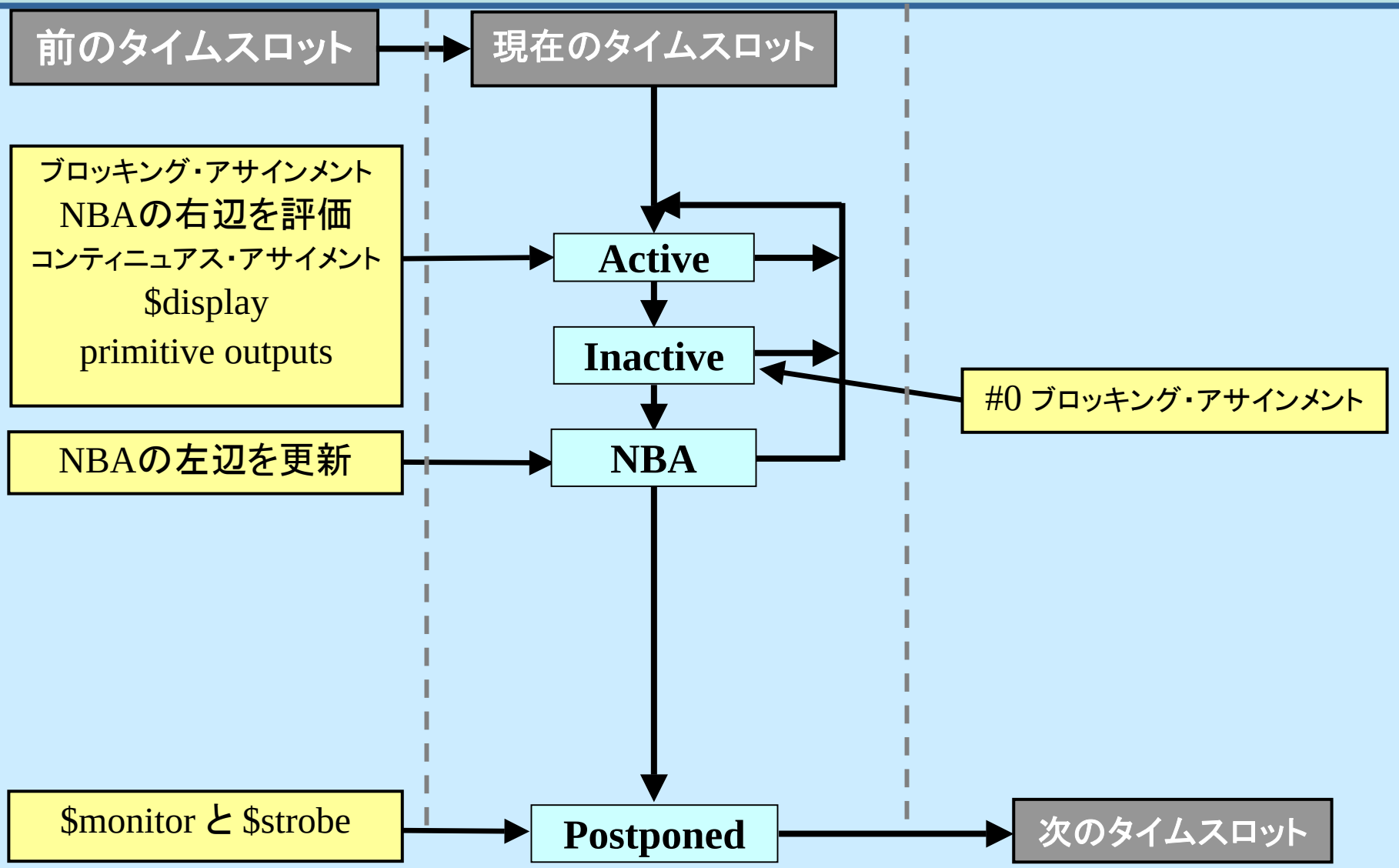
- 簡単な記述例

```
program test (input clk, input [16:1] addr, inout [7:0] data);
    initial ...
endprogram
```

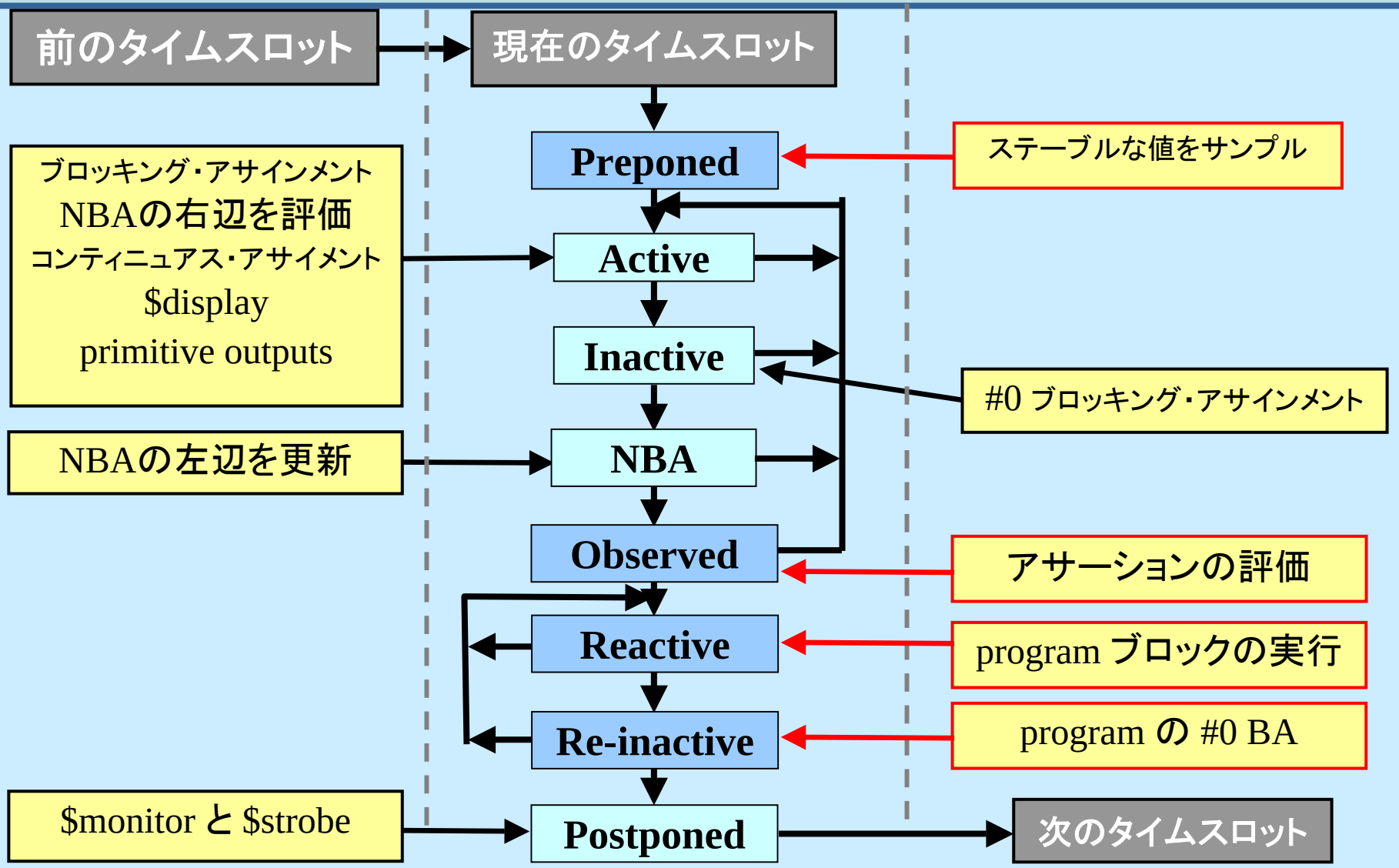
  - テストベンチのコードを明確化し、カプセル化する
  - 厳密なモデリング・スタイルを強要する
- module との主な相違点
  - Initial ブロックのみ（always, モジュールインスタンスは不可）
  - プログラム・ブロックの変数は外側から見えない
  - ブロッキングとノンブロッキングのアサインメントに関するルール
    - プログラム・ブロックの変数に対してはブロッキングのみ
    - それ以外に対してはノンブロッキングのみ

  → VHDLのprocess文と類似
- “*reactive*” 領域において実行される
  - DUTとの通信においてレーシングが起きないことを保証できる

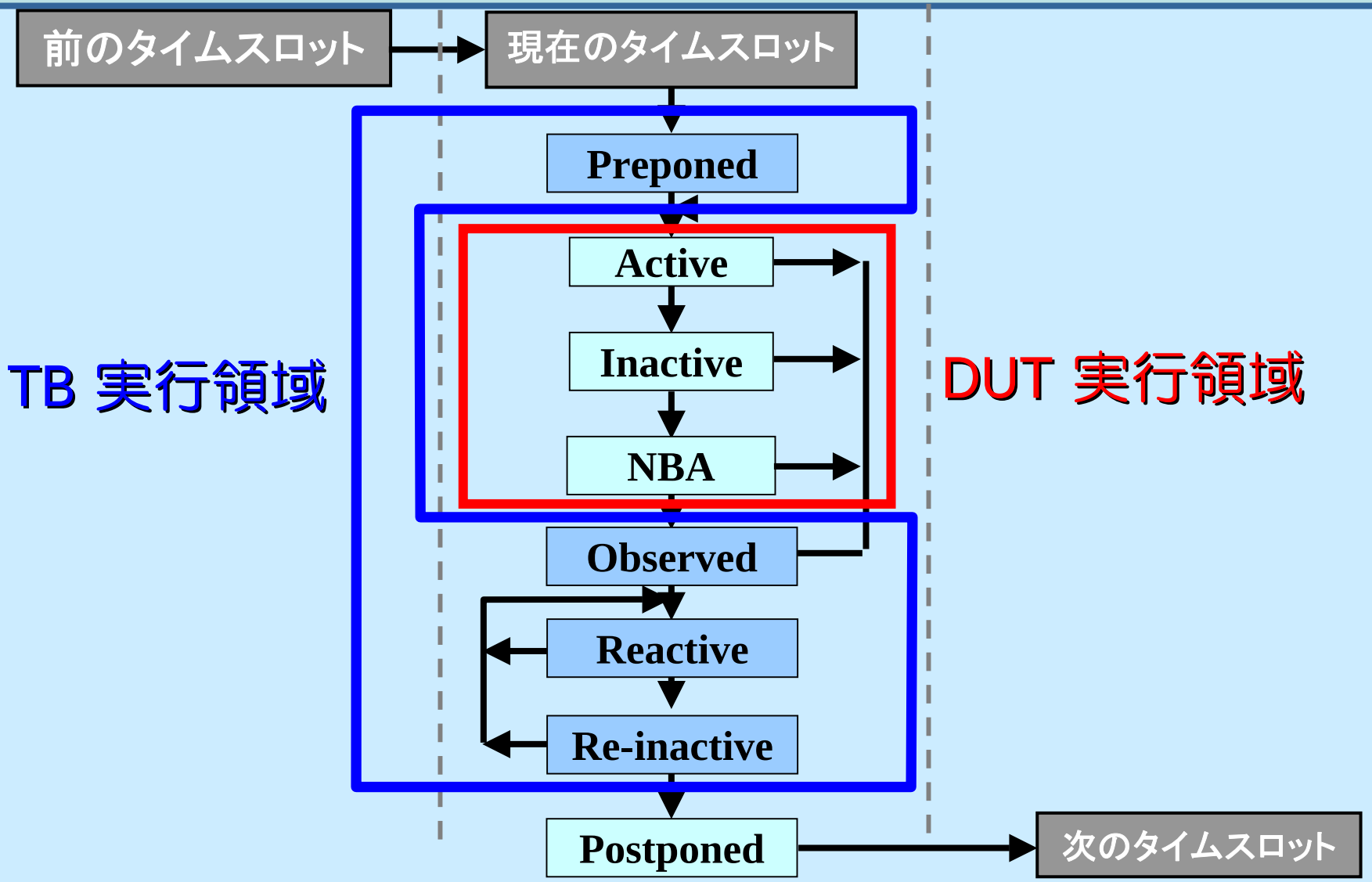
Verilog 1995/2001 のスケジューリング



SystemVerilog のスケジューリング

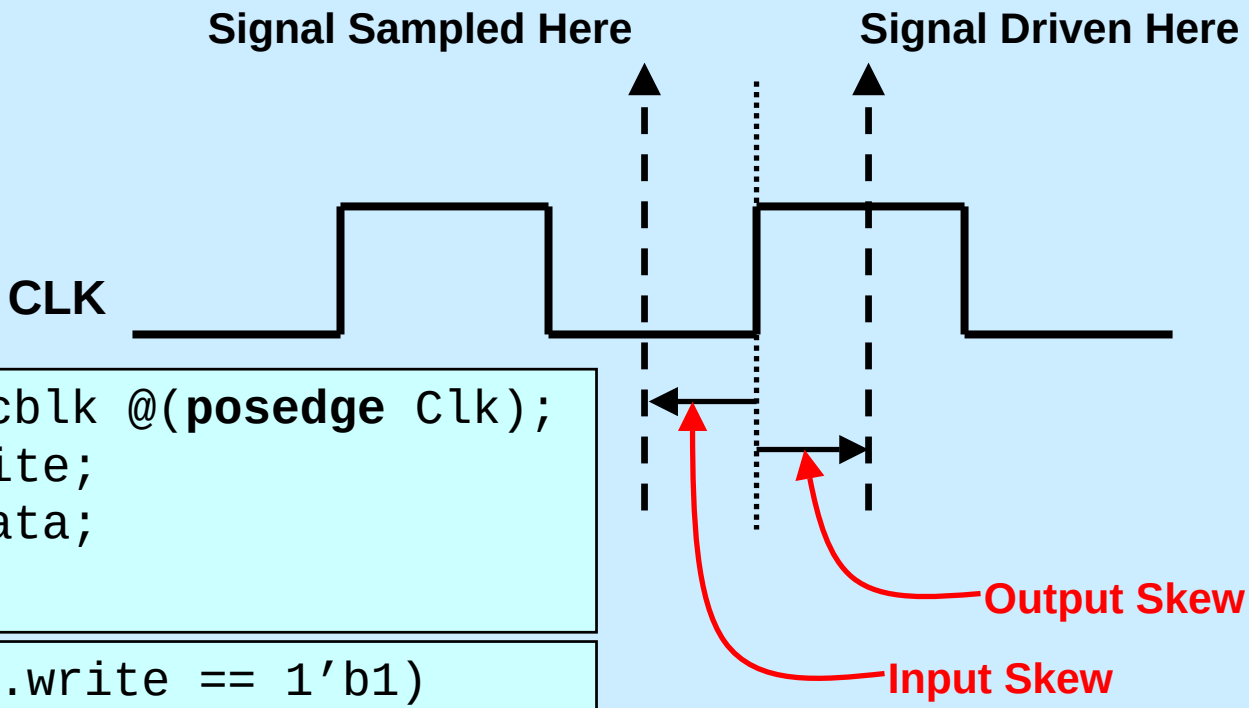


TB/DUT スケジューリング領域の分離



クロッキング・ブロック ― 入力のサンプリング

クロッキング・ブロックを使って安定した値をサンプルする




```
default clocking cblk @(posedge Clk);
  input #1step write;
  output #200ps data;
endclocking
```

```
if (cblk.write == 1'b1)
   cblk.data <= ##2 sample;
 ...
```

クロッキング・ブロック ― 同期出力

- タイミング情報を一箇所で記述
  - パターンを生成する部分と分離
  - 後からタイミングを変更することが容易

```
program tb (output stb);
  clocking slck @(posedge clk);
    output #200ps stb;
  endclocking

  initial begin
      .....
      slck.stb = 1;
      .....
endprogram
```

- プログラム・ブロックのポート stb へのドライブ
  - slck.stb = 1 の代入の後の次のクロックの立ち上がりから、200ps後

クロッキング・ブロック ― クロックへの同期

- サイクル・ディレイ記法: ##
  - default clocking
    - モジュール(あるいはprogram、interface) に1つだけ記述可能
  - default clocking で指定したクロックによるディレイ記述
    - ##5　　5サイクル（固定値）遅らせる
    - ##(j+1)　　変数で指定したサイクル遅らせる

- 同期イベント
  - @(negedge slck.bus[0]);

    bus の bit0 の立ち下りを待つ

    クロッキング・ブロック slck で指定したクロックでサンプルされる bus の値の bit0 の立ち下りを待つ

これまでのVerilog HDLテストベンチ

- デザインのようにIP化し、再利用して行くことが困難
  - 常に「似ている過去のテストベンチ」を「コピー」し、エディット
    - 生産性が低い
    - 完成済みテストベンチを壊す事となり、信頼性が低くなる
  - なぜならmodule, taskベースのテストベンチだから・・・
    - task - 入れ子が困難
    - module - input/output/inoutを通過する0/1/X/Zによる低レベルな情報伝播
  - Verilog HDLでは、デザインのように構造化しにくい

クラスとは？

- 動的オブジェクト
  - 必要な時にだけ使用する
  - moduleは静的オブジェクト – 最初から最後まで
- データ構造を容易に拡張できる
  - 制御信号の追加 – または隠蔽
- 関数(task/function)を容易に拡張できる
  - 関数の追加 – または隠蔽
  - 関数の引数追加 – または引数の隠蔽
- module, taskに比べて柔軟性が高い

SystemVerilogのクラス

基本文法:

```
class name;
    <data_declarations>;
    <task/function_declarations>;
endclass
```

Note:

クラスの宣言だけでは記憶域を取らない。
newでインスタンスを生成してはじめて記憶域を取る

```
class ab_base;
  rand reg [31:0] a, b;
  constraint c0 {
    a[3:0] == 'h0;
    b > a;
  }
  function void disp();
    $display(a,,b);
  endfunction
endclass

ab_base ab_obj = new;
```

クラスの継承

- クラスは他のクラスの特性やメソッドを継承できる
  - サブクラスは親クラスのメソッドを明示的に再定義できる
  - 良い機能はそのままに、カスタマイズが可能

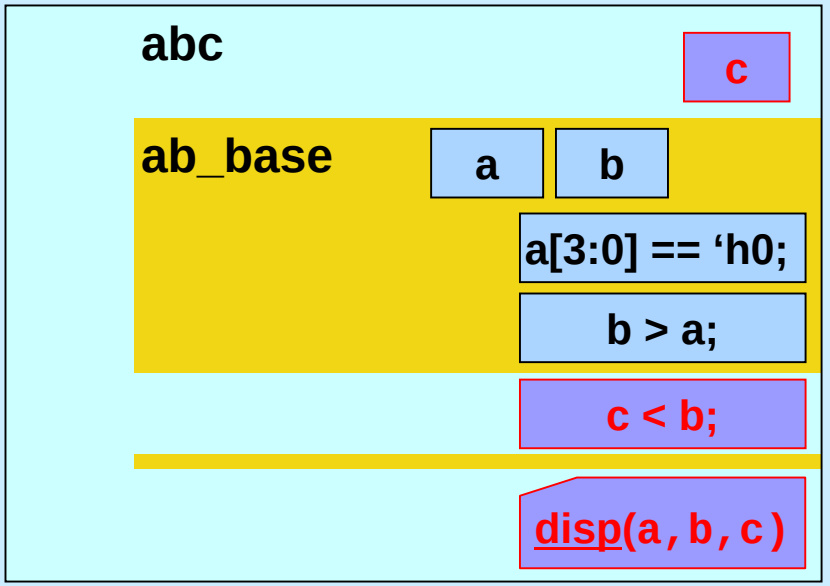
```
class abc extends ab_base;

  rand reg [31:0] c;

  constraint c1 {
    c < b;
  }

  function void disp();
    $display(a,,b,,c);
  endfunction

endclass
```



まとめ

- 最近の検証トレンド
  - カバレッジ・ドリブン検証
  - テストベンチ、検証環境の再利用
- テストベンチの説明
  - コンストレイント・ランダム・スティミュラス
  - カバレッジ
  - レスポンス
  - テストベンチ、検証環境の再利用
  - program & clocking block
  - クラス

EDAツールサポート状況

- 調査対象ツール/Version(2007/1/10時点で公開されているツール)
  - 論理シミュレータ
    - *Incisive Simulator 5.83 (Cadence)*
    - *Questa 6.2e (Mentor)*
    - *VCS 2006.06 (Synopsys)*
  - 論理合成ツール
    - *Encounter RTL Compiler 6.2 (Cadence)*
    - *Design Compiler 2006.06-SP4 (Synopsys)*
  - 等価性検証ツール
    - *Encounter Conformal 6.2 (Cadence)*
    - *Formality 2006.12 (Synopsys)*
  - プロパティチェッカー
    - *0-In V2.3q (Mentor)*
    - *Incisive Formal Verifier 5.7 (Cadence)*
    - *Magellan 2006.06 (Synopsys)*
  - Lintチェッカー
    - *Incisive Simulator 5.83 (Cadence)*
    - *LEDA2006.06 (Synopsys)*

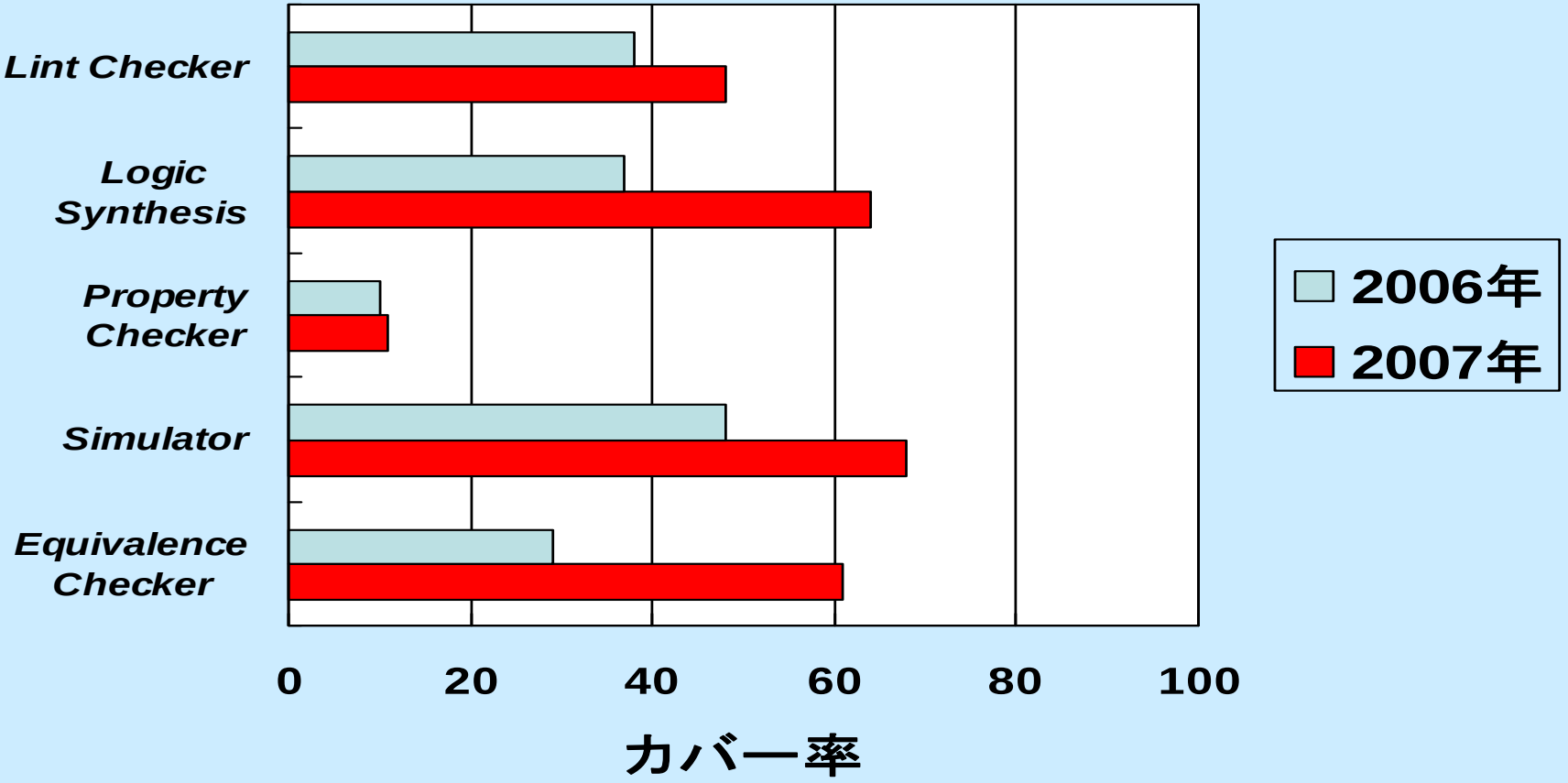
サポート状況の集計方法

- IEEE Std. 1800-2005 LRMから項目を抜粋
- 各項目についてツールごとにサポート済み/未サポートを調査
- ツールを以下の5つのカテゴリに分類、全ツール結果の*AND*をとる
  - 論理シミュレータ
  - 合成ツール
  - プロパティチェッカー
  - 等価性検証ツール
  - Lintチェッカー
- 標準言語として*ベンダーを気にせず*使用することができる
  構文カバー率を調査

カバー率はベンダーによりばらつきがあります。詳細は各ベンダーにお問い合わせください。

ツールカテゴリごとのSystemVerilog構文カバー率

- ツールカテゴリごとのLRM全項目に対するカバー率
  - Lintチェッカー、合成ツール及び等価性チェッカーにおいて合成非対象構文(アサーション,TB等)は対象外。



今回紹介した構文のサポート状況

- 論理シミュレータ上ではほぼサポートされている。
  - Randomization / Constraints
  - randsequence
  - cover / covergroup
  - cross
  - Program Block
  - Clocking Block
  - Class
- 使用にあたっては、機能が制限されているものもあります。
  EDAベンダから情報収集してください。

サポート状況まとめ

- 昨年に比べEDAツールの対応が進んでいることが確認できる。
- 合成/等価検証/シミュレータに関してはベンダーを気にせず使用することが可能な環境が、実用レベルに達しつつある。
- プロパティチェッカーについては、未だサポートが充分とは言えず、早期のサポートが期待される。