



# SystemVerilog設計検証ツール

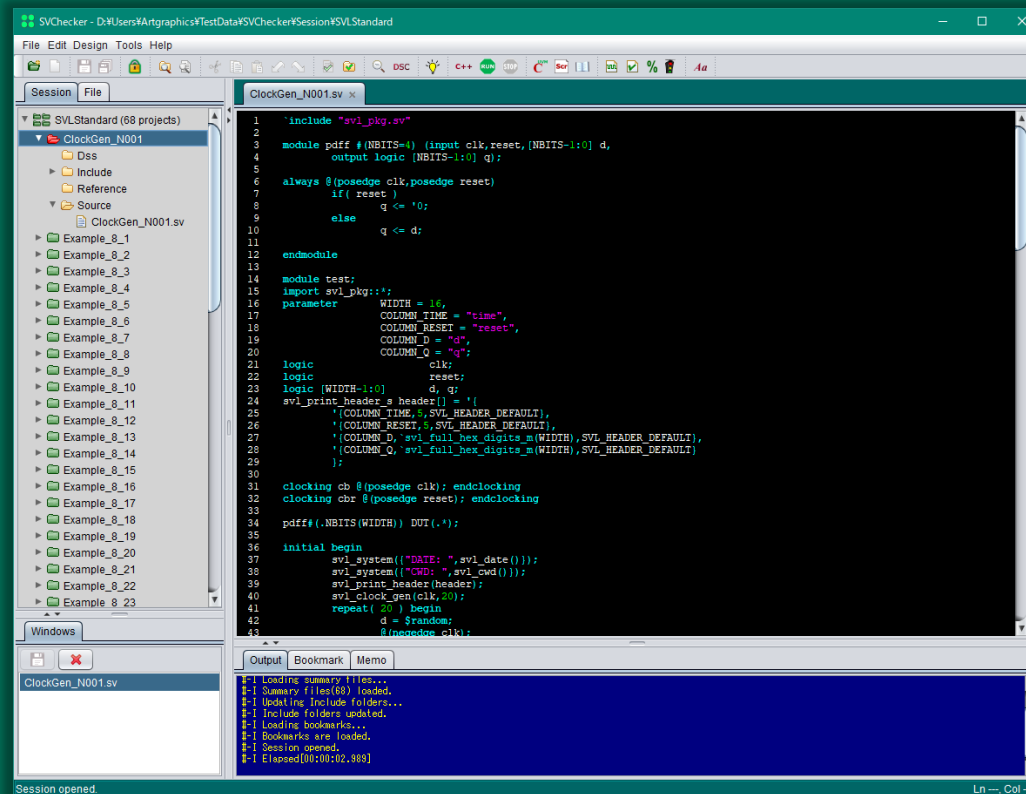
最先端の検証技術を効果的に適用する為には、最適なGUIの使用が不可欠です。  
SystemVerilog IDE は時代に即した機能を提供します。

# SystemVerilog IDE

---

- プロジェクトマネージャー
- クイック参照機能
- SystemVerilog コンパイラー
- 検証機能
- UVM サポート
- 論理合成システム
- 自動バックアップ機能
- コード開発と支援機能
- デザインスタイルチェック
- SystemVerilog シミュレータ
- 検証ビューワー
- HTML 文書生成
- ソフトウェア更新機能

# SystemVerilog IDEの外観



# IEEE Std 1800-2023

- SystemVerilog IDE は IEEE Std 1800-2023 に準拠しています。例えば、以下のような指定をできます。

```
1  class base_t;
2  rand bit [2:0] port;
3  rand bit [3:0] data;
4  constraint C1 { port inside {[0:5]}; }
5  covergroup cg;
6      coverpoint port { bins port_bins[2] = {[0:5]}; }
7      coverpoint data { bins data_bins[3] = {[0:15]}; }
8  endgroup
9  function new();
10     cg = new;
11 endfunction
12 endclass
13
14 class sub_t extends base_t;
15     rand logic [3:0] value;
16     covergroup extends cg;
17         coverpoint value { bins value_bins[4] = {[0:15]}; }
18     endgroup
19     function new(default);
20         super.new(default);
21     endfunction
22 endclass
23
24 ..
```

カバーグループを拡張できます

キーワードdefaultを指定できます

# トークンナビゲータ

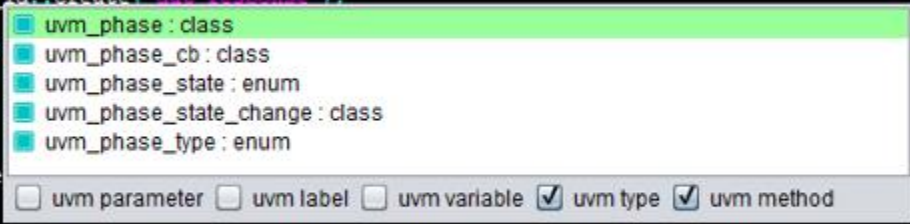
- ナビゲータが入力文字の近くに自動的に現れ、次に入力すべき文字候補のリストを表示しタイプ入力の負荷を軽減します。
- ↑ と ↓キーにより選択を変更できます。
- Enterキーを押すと選択されているワードで入力が完了します。

```
1  module ud_counter(input clk,reset,[1:0] up_down,output logic [3:0] count);
2
3  | always|
4
5   always
6   always_comb
7   always_ff
8   always_latch
9
10
11   uvm parameter  uvm label  uvm variable  uvm type  uvm method
12
13
```

# UVMナビゲータ

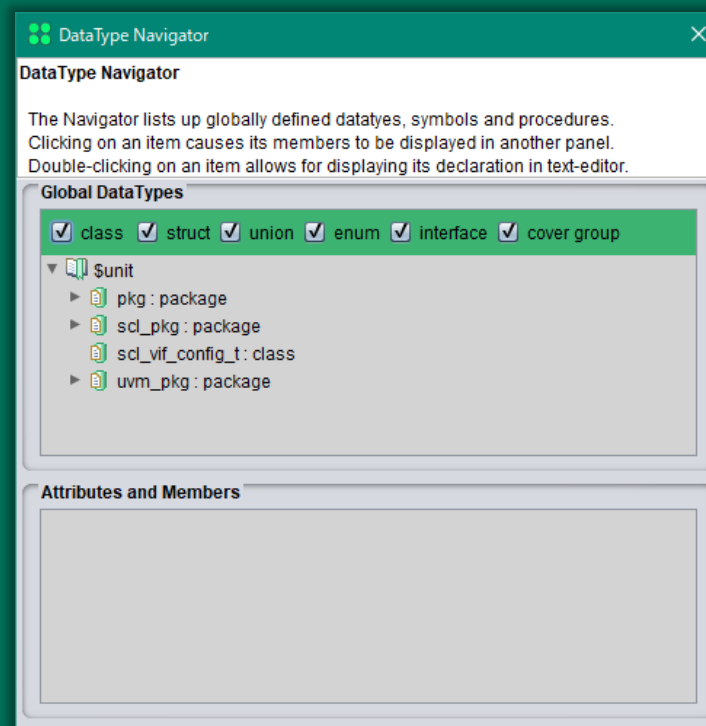
- ナビゲータは、UVMキーワード、データタイプ、クラス、メソッド等も認識します。

```
26 endfunction
27
28 task simple_collector_t::run_phase(uvm_pha);
29     item = simple_item_t::type_id::create("dut_response");
30     fork
31         collect_response();
32         collect_reset();
33     join
34 endtask
35
36 task simple_collector_t::collect_re
37     forever begin
38         @vif.cb;
39         send_item();
40     end
41 endtask
```



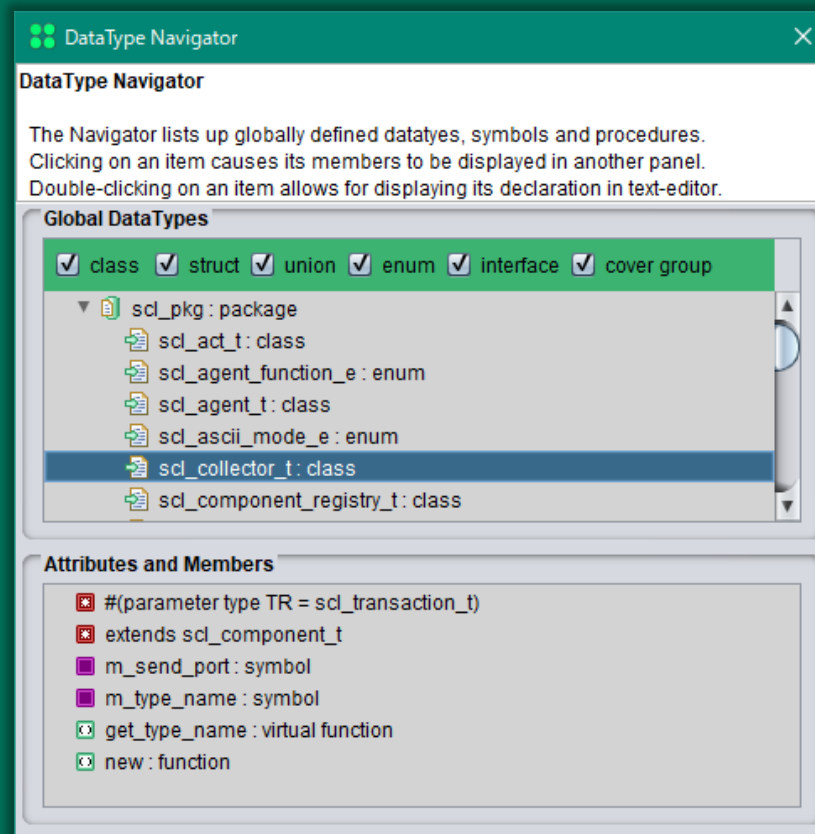
# データタイプナビゲータ

- ユーザ定義のデータタイプやUVMのデータタイプを簡単にアクセスできます。
- ナビゲータ内のデータタイプ名をダブルクリックすると、そのデータタイプが定義されているファイルが開きます。
- データタイプ定義が変更されると、ナビゲータの情報も同期して自動的に更新されます。



# データタイプナビゲータの使用例

- データタイプ名をクリックすると定義情報の概要が下のサブペインに表示されます。
- 右の例では、`scl_collector_t` がクリックされた状態を示しています。





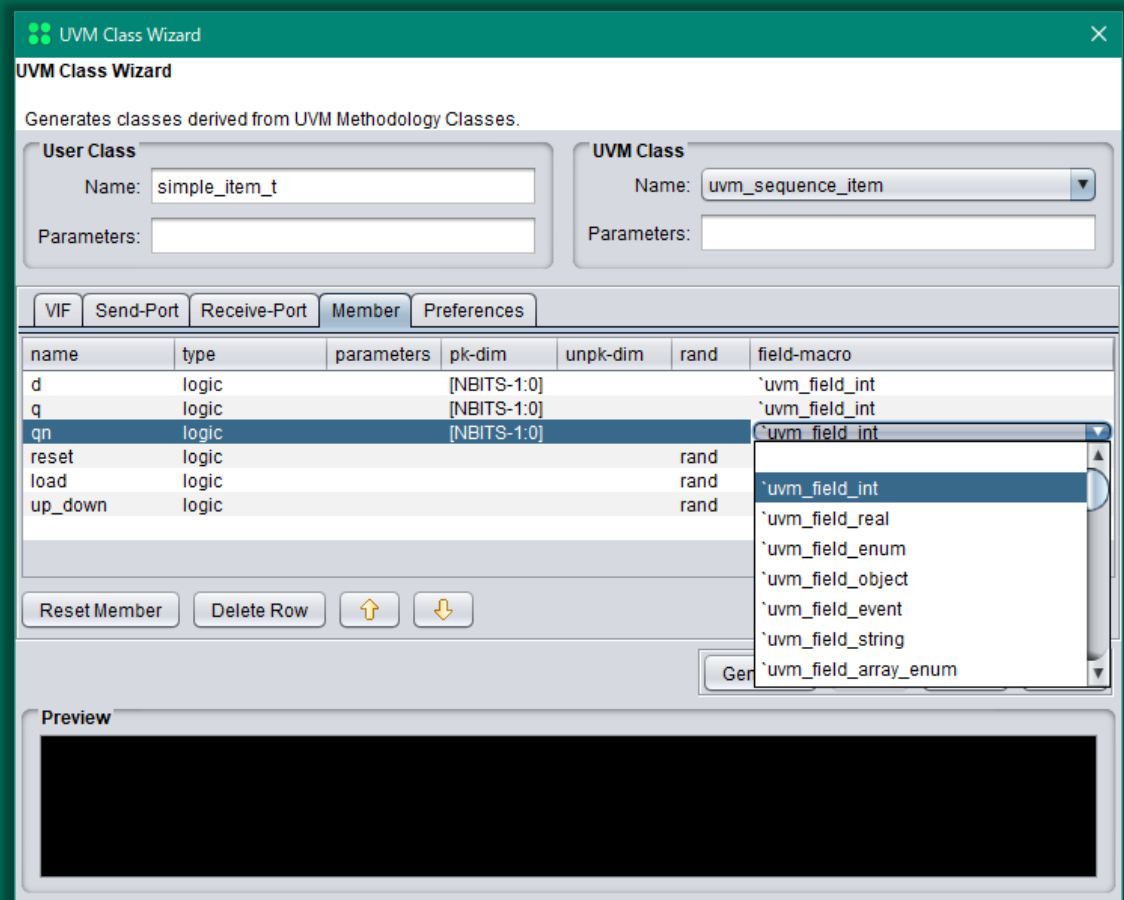
# UVMクラスウィザード

---

- UVMを使用する際、必須項目が多くタイプ入力の負荷が大きい問題と間違い易い記述が多い問題があります。UVMクラスウィザードは、これらの負荷を軽減します。
- UVMクラスウィザードは、必須のUVMマクロおよびフィールドマクロを生成します。
- UVMクラスウィザードは、\*\_phaseメソッドのスケルトンも生成します。

# UVMクラスウィザードの使用例

- 類似の定義が続く場合には、!!を入力すると内容がコピーされます。その後、必要に応じて編集すれば完了します。
- フィールドマクロはドロップダウンリストから選択するだけで完了します。



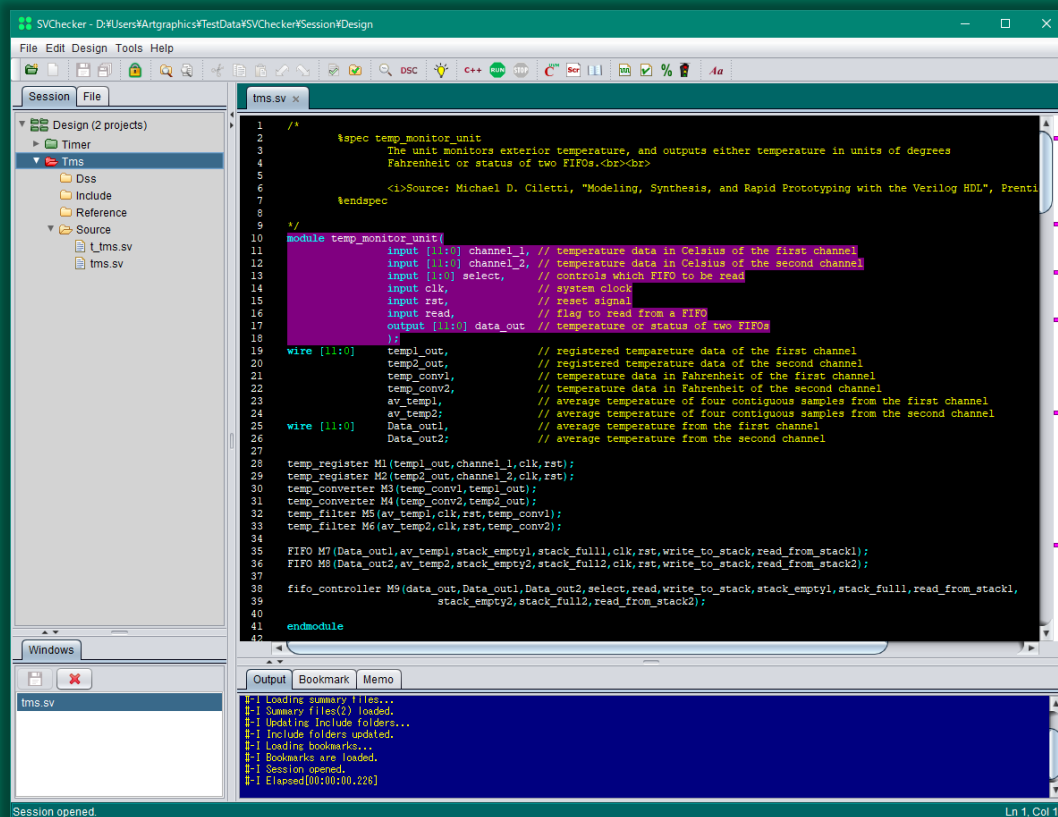
# UVMクラスウィザード生成例

- クラス生成ボタンをクリックすると右のようにクラスが生成されます。
- 必要なマクロが生成されている様子を確認できます。

```
1 // -----
2 //
3 //     simple_item_t
4 //
5 // -----
6 class simple_item_t extends uvm_sequence_item;
7 logic [NBITS-1:0]    d;
8 logic [NBITS-1:0]    q;
9 logic [NBITS-1:0]    qn;
10 rand logic           reset;
11 rand logic           load;
12 rand logic           up_down;
13 `uvm_object_utils_begin(simple_item_t)
14     `uvm_field_int(d,UVM_DEFAULT)
15     `uvm_field_int(q,UVM_DEFAULT)
16     `uvm_field_int(qn,UVM_DEFAULT)
17     `uvm_field_int(reset,UVM_DEFAULT)
18     `uvm_field_int(load,UVM_DEFAULT)
19     `uvm_field_int(up_down,UVM_DEFAULT)
20 `uvm_object_utils_end
21 //
22 //     new
23 //
24 function new(string name="simple_item_t");
25     super.new(name);
26 endfunction
27 endclass
28
```

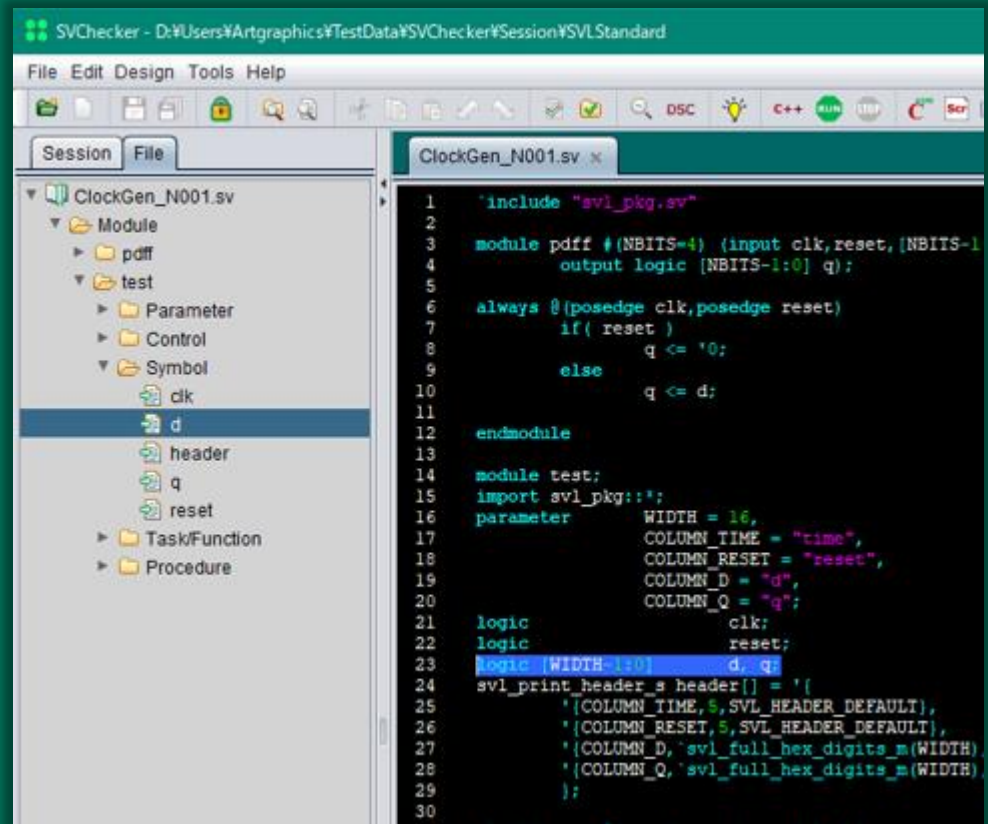
# ハイライト機能

- module、initial、always、task、function、class 等をハイライトできます。
- 右端のロケータをクリックするとハイライトされた箇所へ飛びます。



# ファイルビュー

- コンパイルするとファイル構造が作成され、ファイルの構成要素をクリックするだけでアクセスできます。
- 右図では、dノードをクリックしているので、変数dが定義されている行がハイライトされています。
- 大きなファイル内で変数の定義場所を位置付ける場合に便利な機能です。



# プロジェクトマネージャー

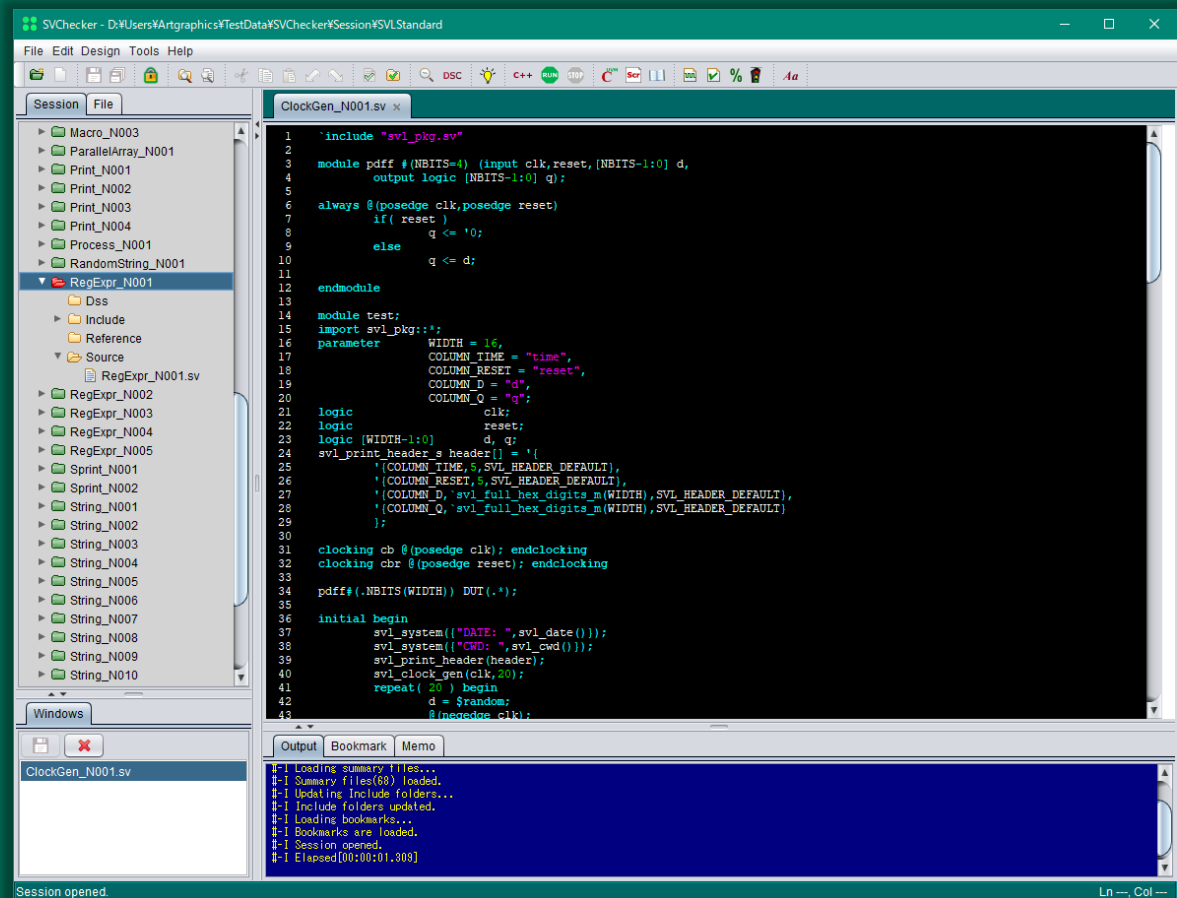
---

- SystemVerilogでは、ソースコードはソースファイルとインクルードされたファイルから構成されます。
- SystemVerilog IDEでは、ソースファイルとインクルードされたファイルはプロジェクトフォルダのもとで管理されます。
- ソースファイルはSourceフォルダ、インクルードファイルはIncludeフォルダで管理されます。
- フォルダで管理するファイル数に制限はありません。
- マウスクリックだけで、フォルダへのファイルの追加、フォルダからファイルの削除をできます。
- プロジェクトマネージャーは、フォルダの情報を基にしてコンパイルする情報を構築します。
- Includeフォルダーの内容はコンパイラーにより自動的に更新されます。

# プロジェクトマネージャーの例

プロジェクト

- 赤く表示されているフォルダーはアクティブなプロジェクトを意味します。
- コンパイルする際には、アクティブなプロジェクトが対象になります。



# MakeCpp

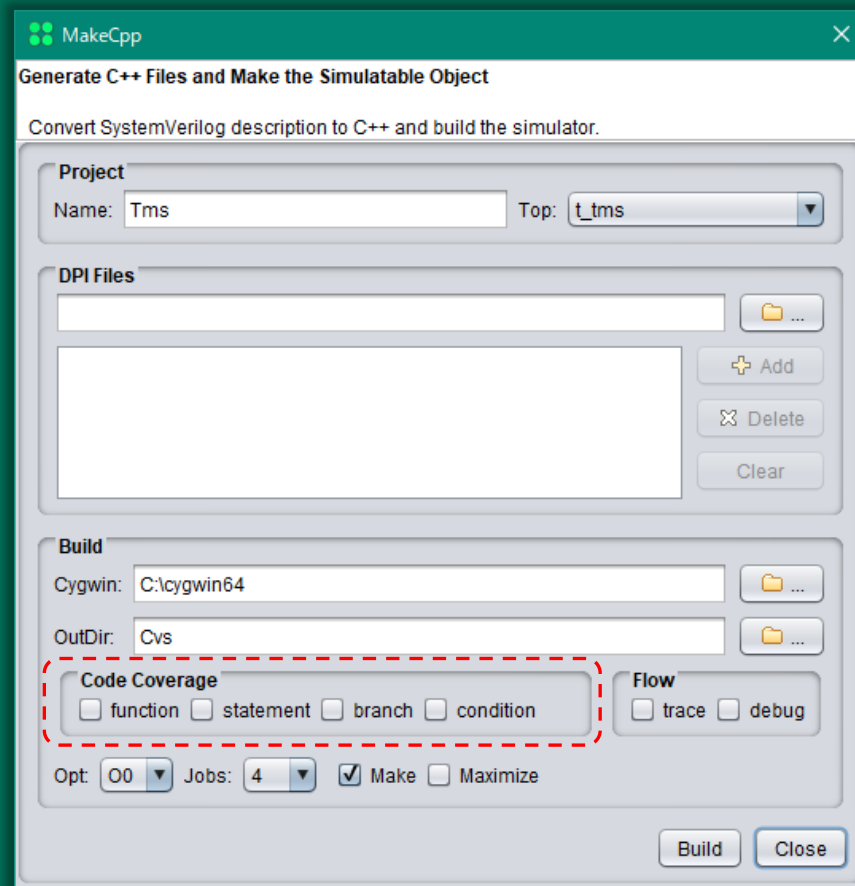
---

- シミュレーションするための実行モジュールを作成する手順は、MakeCppにより行われます。
- MakeCppはSystemVerilog記述をC++に変換し、C++コンパイラを使用して実行モジュールを作ります。
- 実行モジュールが作成されると、何度でも実行を繰り返す事ができます。一度のコンパイルで多くのテストケースを実行する事ができます。



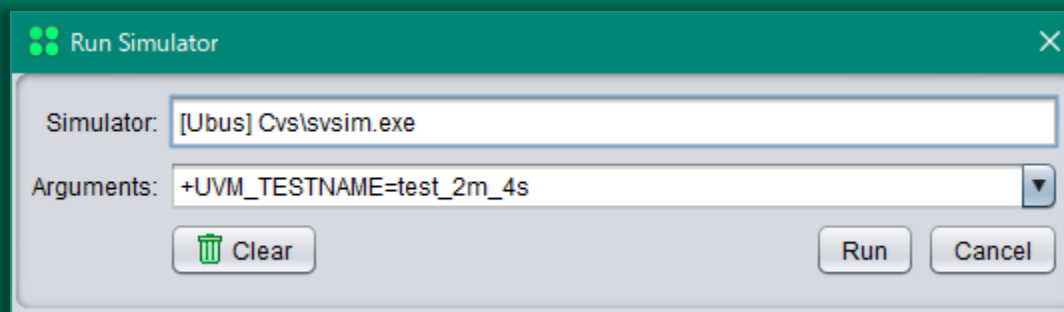
# MakeCppの実行例

- コードカバレッジはSystemVerilog機能の一部ではありませんが、SystemVerilog IDEによりサポートされています。
- コードカバレッジのオプションを設定すれば、シミュレーション時にコードカバレッジの処理が行われます。



# シミュレーションの実行

- シミュレーションの実行には、下図に示すRun Simulatorダイアログが使用されます。
- ダイアログ内で実行に必要なパラメータを指定できます。
- 一度指定されたパラメータは記録されるので、次回以降はドロップダウンリストからパラメータリストを選択するだけで済みます。



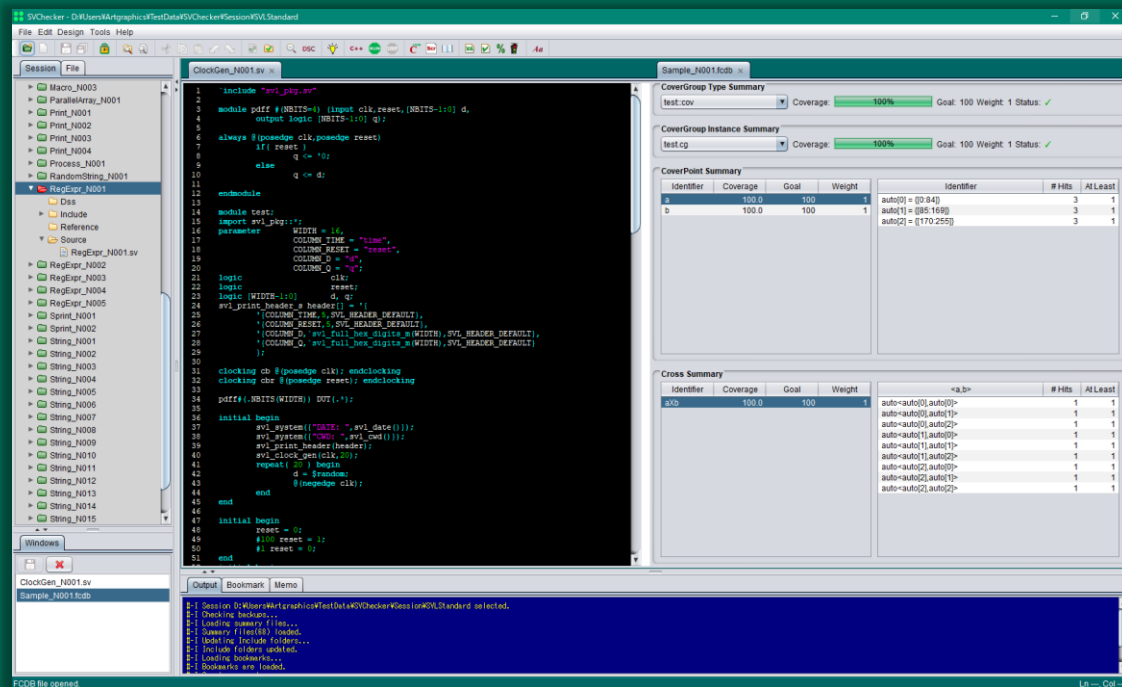
# 検証機能

---

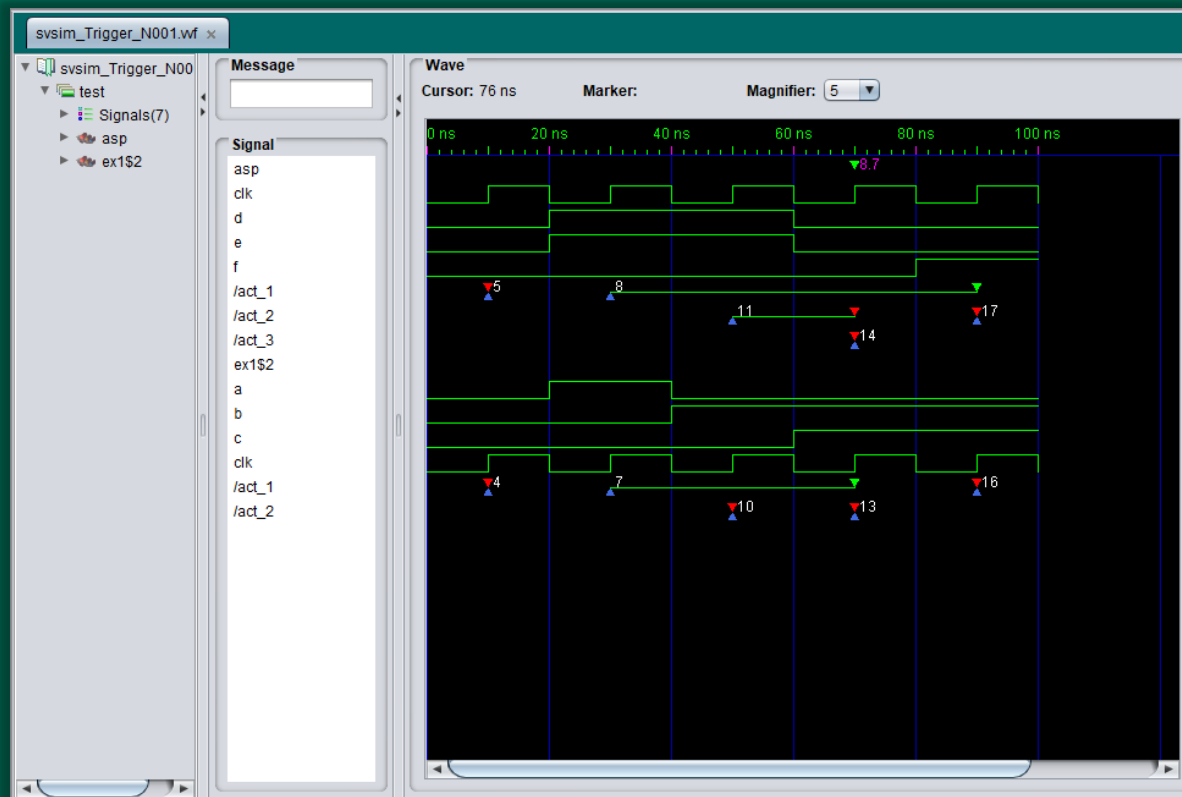
- ファンクショナルカバレッジとアサーションは、標準機能としてサポートされています。従って、SystemVerilogソースコード中にそれらの機能の記述が指定されていれば、シミュレータにより実行されます。
- シミュレーションの実行が終了すると検証機能の結果がファイルに生成されているので、何時でもビューワーで検証結果を確認できます。
- SystemVerilog IDEは、検証結果を通常のファイルと同じように扱って表示します。

# テキストウィンドウと検証結果の例

- SystemVerilog IDEはテキストウィンドウと検証結果を同じように扱います。下図は、テキストエディタとファンクショナルカバレッジの結果を同時に表示しています。



# アサーション実行例



# ファンクショナルカバレッジ実行例

The screenshot displays a functional coverage tool interface for a file named 'Sample\_N001.fcdb'. It is divided into four main sections: CoverGroup Type Summary, CoverGroup Instance Summary, CoverPoint Summary, and Cross Summary. Each summary section shows a dropdown menu, a coverage progress bar at 100%, and a goal of 100 with a weight of 1 and a status of 'checked'.

**CoverGroup Type Summary**  
test:cov Coverage: 100% Goal: 100 Weight: 1 Status: ✓

**CoverGroup Instance Summary**  
test.cg Coverage: 100% Goal: 100 Weight: 1 Status: ✓

**CoverPoint Summary**

Identifier	Coverage	Goal	Weight
a	100.0	100	1
b	100.0	100	1

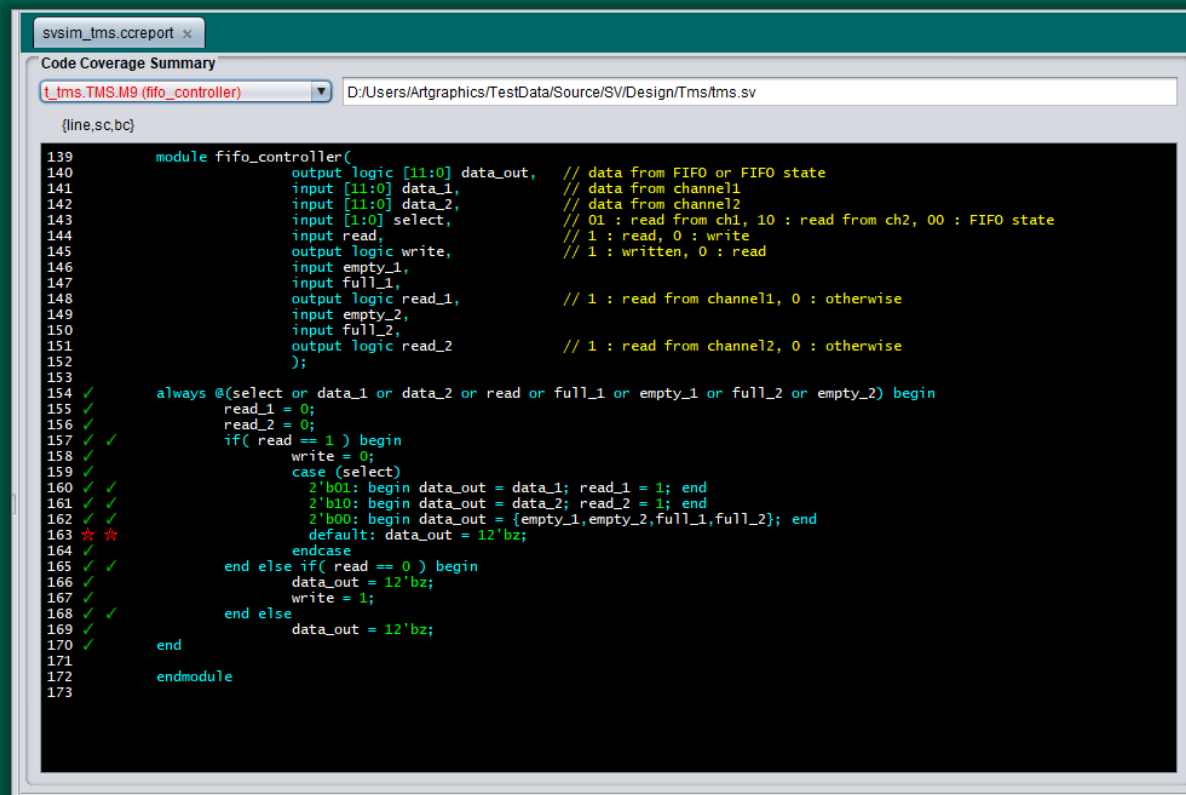
Identifier	# Hits	At Least
auto[0] = {[0:84]}	3	1
auto[1] = {[85:169]}	3	1
auto[2] = {[170:255]}	3	1

**Cross Summary**

Identifier	Coverage	Goal	Weight
aXb	100.0	100	1

<a,b>	# Hits	At Least
auto<auto[0],auto[0]>	1	1
auto<auto[0],auto[1]>	1	1
auto<auto[0],auto[2]>	1	1
auto<auto[1],auto[0]>	1	1
auto<auto[1],auto[1]>	1	1
auto<auto[1],auto[2]>	1	1
auto<auto[2],auto[0]>	1	1
auto<auto[2],auto[1]>	1	1
auto<auto[2],auto[2]>	1	1

# コードカバレッジ実行例



The screenshot shows a code coverage report for a Verilog module named 'fifo\_controller'. The report is displayed in a window titled 'svsim\_tms.ccreport x'. The code is shown with line numbers from 139 to 173. The report indicates that the code is 100% covered, with green checkmarks next to all lines. The code defines a module 'fifo\_controller' with inputs 'data\_1', 'data\_2', 'select', 'read', 'empty\_1', 'full\_1', 'empty\_2', and 'full\_2', and outputs 'data\_out', 'write', 'read\_1', and 'read\_2'. The code implements a FIFO controller with a select signal that can be used for reading from either channel or for writing to the FIFO. The code is as follows:

```
139     module fifo_controller(  
140         output logic [11:0] data_out, // data from FIFO or FIFO state  
141         input [11:0] data_1, // data from channel1  
142         input [11:0] data_2, // data from channel2  
143         input [1:0] select, // 01 : read from ch1, 10 : read from ch2, 00 : FIFO state  
144         input read, // 1 : read, 0 : write  
145         output logic write, // 1 : written, 0 : read  
146         input empty_1,  
147         input full_1,  
148         output logic read_1, // 1 : read from channel1, 0 : otherwise  
149         input empty_2,  
150         input full_2,  
151         output logic read_2 // 1 : read from channel2, 0 : otherwise  
152     );  
153  
154     always @(select or data_1 or data_2 or read or full_1 or empty_1 or full_2 or empty_2) begin  
155         read_1 = 0;  
156         read_2 = 0;  
157         if( read == 1 ) begin  
158             write = 0;  
159             case (select)  
160                 2'b01: begin data_out = data_1; read_1 = 1; end  
161                 2'b10: begin data_out = data_2; read_2 = 1; end  
162                 2'b00: begin data_out = {empty_1, empty_2, full_1, full_2}; end  
163                 default: data_out = 12'bz;  
164             endcase  
165         end else if( read == 0 ) begin  
166             data_out = 12'bz;  
167             write = 1;  
168         end else  
169             data_out = 12'bz;  
170     end  
171  
172     endmodule  
173
```

# 論理合成システム

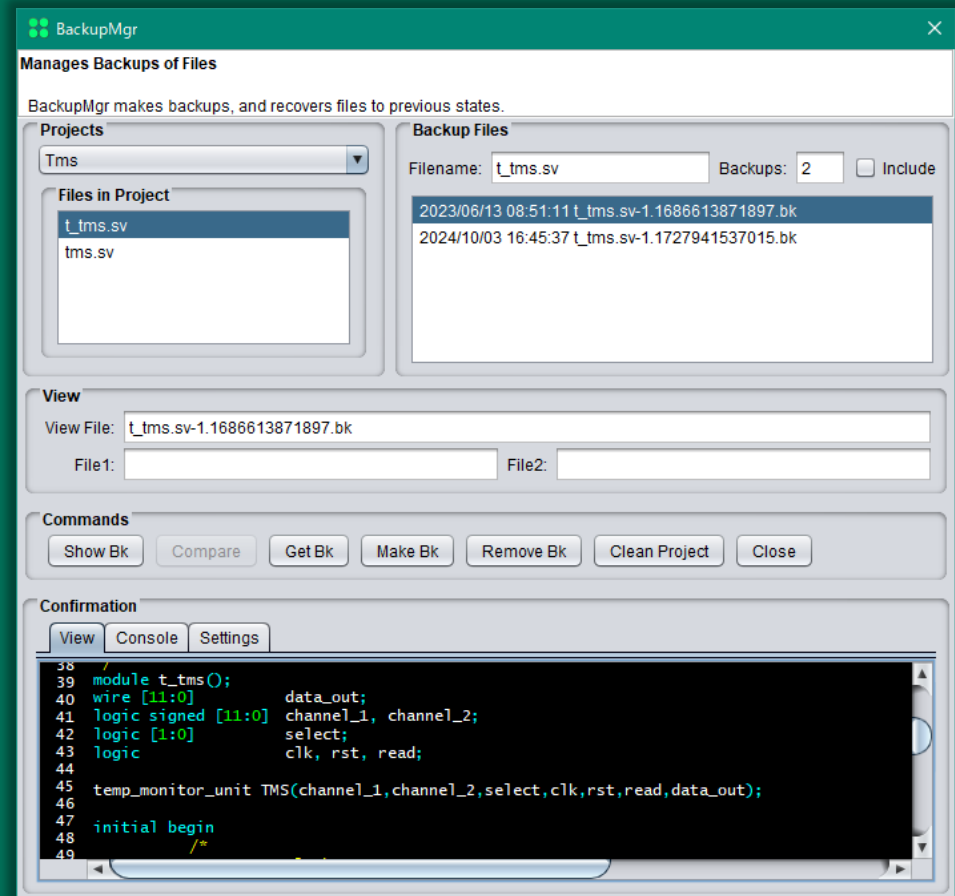
---

- 論理合成システムはRTL記述によるデザインの性能をいち早く予測するための機能です。
- 論理合成システムは、ユーザ指定のライブラリーを使用してRTL記述からネットリストを生成します。ユーザは、生成されたネットリストを基にして様々な解析ツールによりデザインの性能を測定する事ができます。
- 詳細は論理合成システムの紹介資料をご覧ください。



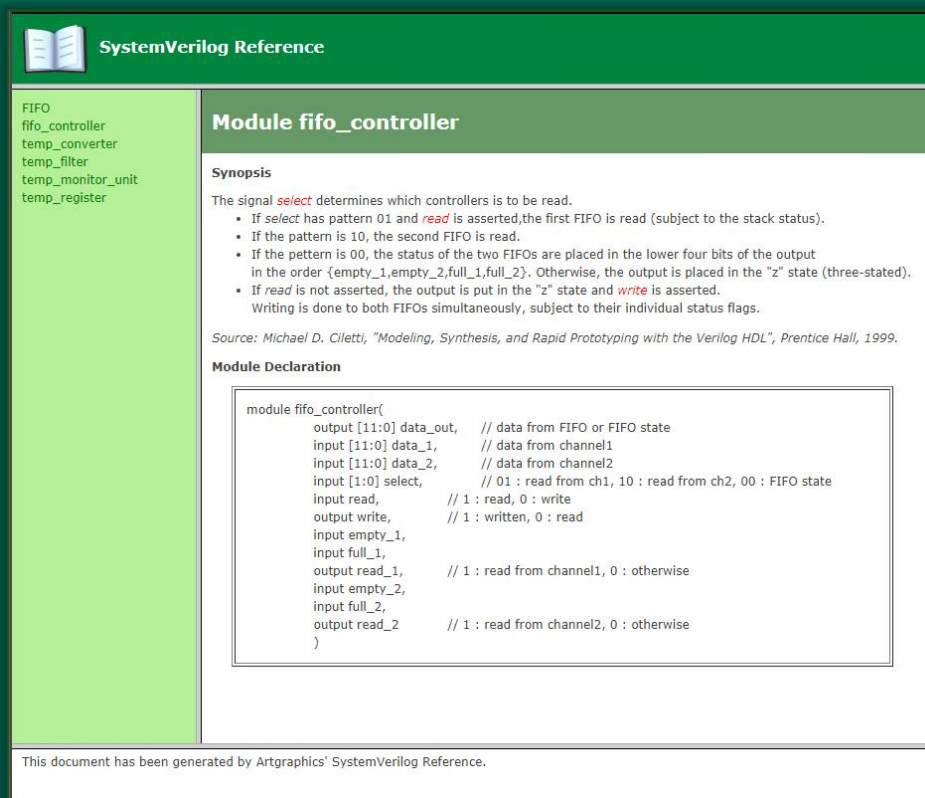
# 自動バックアップ

- 編集用にソースファイルを開くと、自動的にファイルのバックアップが作られます。但し、前回のバックアップと内容が同じであれば、新たなバックアップは作られません。
- バックアップは圧縮されて管理されるのでディスクスペースを無駄にしません。
- 編集中にファイルが壊滅状態になっても編集開始時点の状態に戻れます。
- バックアップマネージャーを使用すると、以前のファイルの状態に戻る事ができます。
- 保有するバックアップのバージョン数を指定できます。



# HTML文書生成

- HTML文書生成機能は、SystemVerilog記述から機能概要を抽出してHTMLファイルを生成する機能です。
- 生成されたHTMLファイルをブラウザで閲覧できます。
- 仕様確認に便利なツールです。



The screenshot displays the SystemVerilog Reference website. The page title is "SystemVerilog Reference". On the left, a sidebar lists navigation items: "FIFO", "fifo\_controller", "temp\_converter", "temp\_filter", "temp\_monitor\_unit", and "temp\_register". The main content area is titled "Module fifo\_controller".

**Synopsis**

The signal *select* determines which controllers is to be read.

- If *select* has pattern 01 and *read* is asserted, the first FIFO is read (subject to the stack status).
- If the pattern is 10, the second FIFO is read.
- If the pattern is 00, the status of the two FIFOs are placed in the lower four bits of the output in the order {empty\_1, empty\_2, full\_1, full\_2}. Otherwise, the output is placed in the "z" state (three-stated).
- If *read* is not asserted, the output is put in the "z" state and *write* is asserted. Writing is done to both FIFOs simultaneously, subject to their individual status flags.

Source: Michael D. Ciletti, "Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL", Prentice Hall, 1999.

**Module Declaration**

```
module fifo_controller(
    output [11:0] data_out, // data from FIFO or FIFO state
    input [11:0] data_1, // data from channel1
    input [11:0] data_2, // data from channel2
    input [1:0] select, // 01 : read from ch1, 10 : read from ch2, 00 : FIFO state
    input read, // 1 : read, 0 : write
    output write, // 1 : written, 0 : read
    input empty_1,
    input full_1,
    output read_1, // 1 : read from channel1, 0 : otherwise
    input empty_2,
    input full_2,
    output read_2 // 1 : read from channel2, 0 : otherwise
)
```

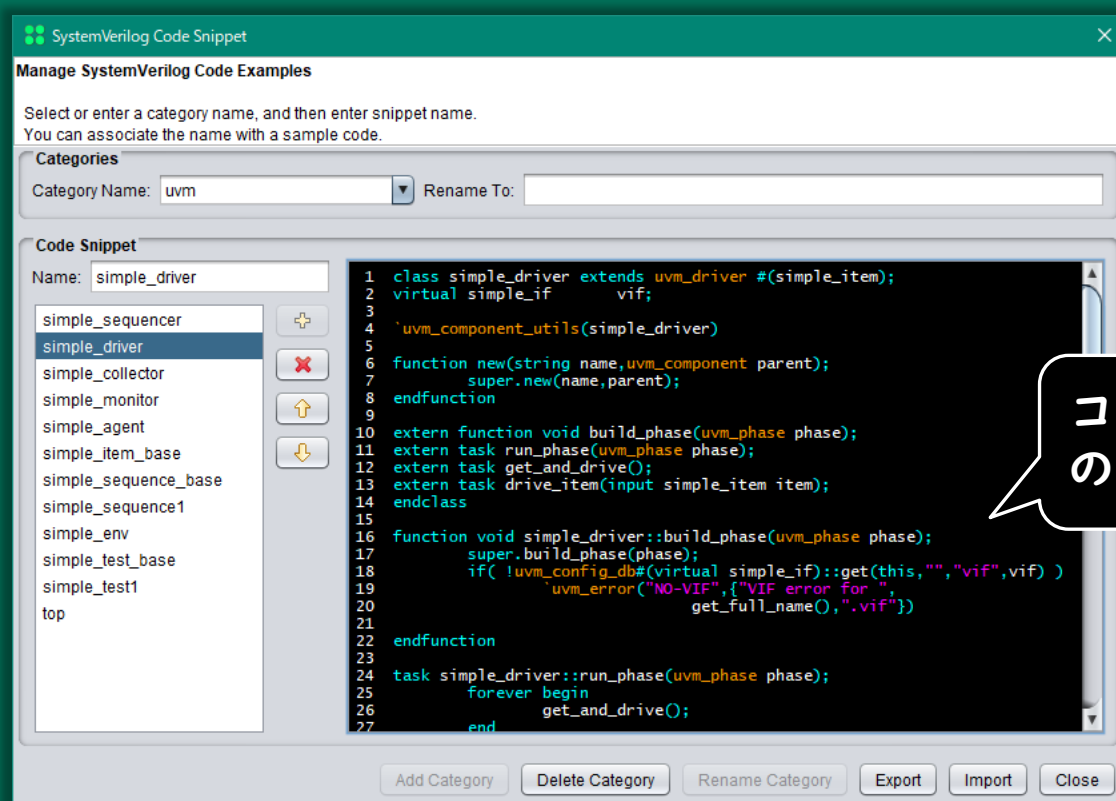
This document has been generated by Artgraphics' SystemVerilog Reference.

# コードスニペット

---

- SystemVerilog IDEのコードスニペット機能は、記述の断片を記録する機能です。
- カテゴリーで分類してコードの断片を管理できるので便利です。
- 複雑なシンタックスを持つ機能の実装例、模範記述例、頻繁に使用する実装例等をコードスニペットとして登録しておくことで生産性が向上します。
- コードスニペットを他の技術者と共有する事ができます。

# コードスニペット例



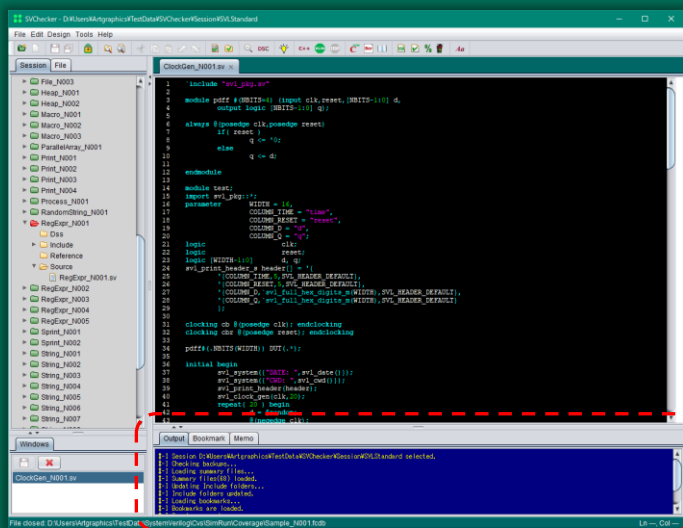
コードスニペットからの部分も抽出できます

# ブックマーク

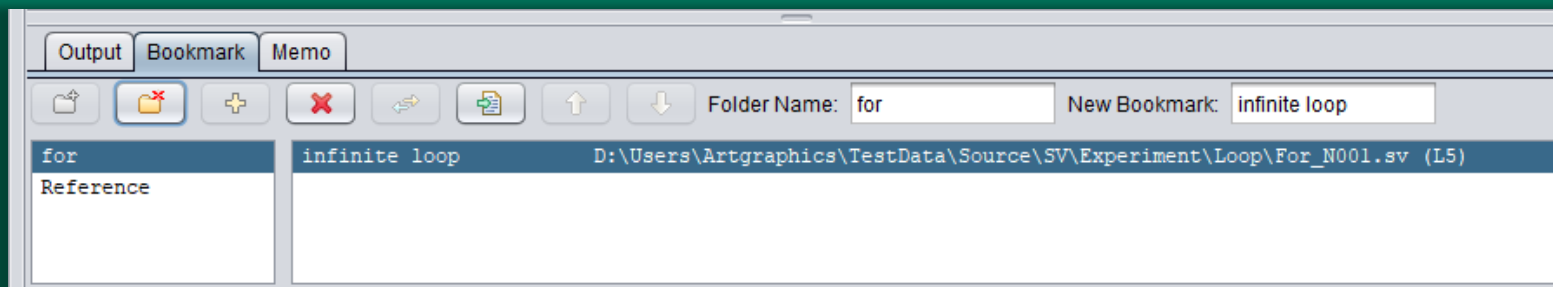
---

- 頻繁にアクセスするファイルをブックマークとして登録しておく便利です。
- テキストファイルの任意の行をブックマークとして登録できます。
- ブックマークをクリックすると、ブックマークされた行を含むファイルが開き、ブックマークされた行がハイライトされます。
- ブックマークをフォルダで分類できるので、ブックマークの管理が容易です。

# ブックマーク例



ブックマーク



# SystemVerilog IDE 実行環境

---

Windows	Cygwin
	WSL-Ubuntu
Linux	Ubuntu desktop

# まとめ

---

- SystemVerilog IDE は現代的なGUIを採用した先進的なIDEです。
- ファンクショナルカバレッジやアサーション等の検証機能は標準的にサポートされています。また、コードカバレッジ機能も備えられています。
- SystemVerilogの最新仕様 IEEE Std 1800-2023がサポートされています。
- SystemVerilog IDE はWindowsおよびLinuxで動作します。
- 低価格で、しかも使い易いSystemVerilog IDE を是非お試しください。