Verilog テストベンチでプラットフォーム・デザイナー（旧 Qsys）をシミュレーションする手順の解説

Ver.17.0

2019 年 11 月 Rev.1

目次

本書をお読みになる前に

この資料の内容は 2019 年 11 月現在のものです。

この資料で紹介しているソフトウェアやハードウェア、操作手順などは、指定バージョンやデバイス等以外でも共通のものもありますが、一部については共通にならないものもありますので、ご注意ください。

文書中の記号

ⓘ **Note**	補足情報などを記載しています。
Ⓟ **Point**	重要なポイントを記載しています。
参考	理解を深めるため、参考となる資料やサイトを紹介しています。
⚠ **注記**	この資料の中では具体的には触れませんが、必要となる知識や情報を記載しています。
⃠ **禁止**	注意点や、してはいけないことを記載しています。

文中の表記

下線	クリックする事で、資料中の別の章や、外部のサイトにジャンプします。
太字斜体	画面の操作をする際の、メニューやウィンドウなどに表示されている文字を示しています。
xxxxxxx	入力するコマンド文字列を示しています。
網掛け	使用するツールを示しています。

1. はじめに

この資料は、添付のテンプレート・デザインの構成と概要を解説後、シミュレーション手順の紹介と、シミュレーション結果を掲載しています。

テンプレート・デザインは、インテル® FPGA のプラットフォーム・デザイナー（旧 Qsys）で構築したシステムを、Verilog HDL で記述したテストベンチを使用して、RTL シミュレーションを行う際のスタートポイントとしてご使用いただけるサンプルです。

2. 使用環境

この説明では、以下の開発ツールを使用しています。

【表 2-1】この説明で使用している開発ツール

項番	項目	内容
1	インテル® Quartus® Prime 開発ソフトウェア・スタンダード・エディション（以降、Quartus® Prime）	FPGA のハードウェアを開発するためのツールです。 この資料では、インテル® Quartus® Prime 開発ソフトウェア・スタンダード・エディション v17.0 を使用しています。
2	プラットフォーム・デザイナー（旧 Qsys）	FPGA のハードウェアにおいて、主に内部バス Avalon-MM インターフェイスと接続可能で、アドレスマップで定義された各コンポーネント（Nios® II Processor, DMA Controller, Timer, PIO, On-Chip Memory 等）を組み込んで、ユーザーが独自のメモリーマップド・システムを構築するためのツールで、Quartus® Prime に標準装備されています。 この資料では、v17.0 の Quartus® Prime を使用しているので、連動してプラットフォーム・デザイナー（旧 Qsys）も v17.0 を使用しています。
3	ModelSim® - Intel® FPGA Starter Edition（以降、ModelSim® - IE）	FPGA に実装する論理回路の動作を確認するために、波形を表示させたシミュレーションで、各信号の振舞いを検証するシミュレーション・ツールです。 この資料では、v17.0 の Quartus® Prime を使用しているので、それに対応した ModelSim® - IE 10.5b を使用しています。

3. このサンプルを利用する利点

テンプレート・デザイン backup_folder.zip をサンプルとして利用したときの利点は以下の通りです。

- **敷居の低い（Verilog の知識だけで記述可能な）テストベンチを使用**

  プラットフォーム・デザイナー（旧 Qsys）ではバス・ファンクション・モデル（以下、BFM）を使用したシミュレーションも利用可能ですが、このテンプレート・デザインでは BFM に敷居の高さを感じているユーザー向けに、Verilog の知識だけでシミュレーションできる仕組みとなっています。

- **テストベンチのカスタマイズが可能**

  プラットフォーム・デザイナー（旧 Qsys）は、ユーザーが構築したシステム向けに Verilog テストベンチも生成します。

  この資料では、テンプレートと・デザインに含まれるテストベンチをカスタマイズする例も紹介します。このテンプレートを利用することにより、ユーザーはテストベンチをスクラッチから作成する必要が無くなり、開発工数も削減できます。

- **コンポーネントのカスタマイズが可能**

  このテンプレート・デザインは、いくつかの代表的なコンポーネントで構成されています。ユーザーは、所望のコンポーネントを用途に応じて、プラットフォーム・デザイナー（旧 Qsys）のシステムに追加することができます。

4. テンプレート・デザインの構成

テンプレート・デザインの構成について説明します。このテンプレート・デザインのテストベンチ・アーキテクチャー並びに、プラットフォーム・デザイナー（旧 Qsys）で構築されるシステムのブロック図は、図 4-1 のように構成されます。

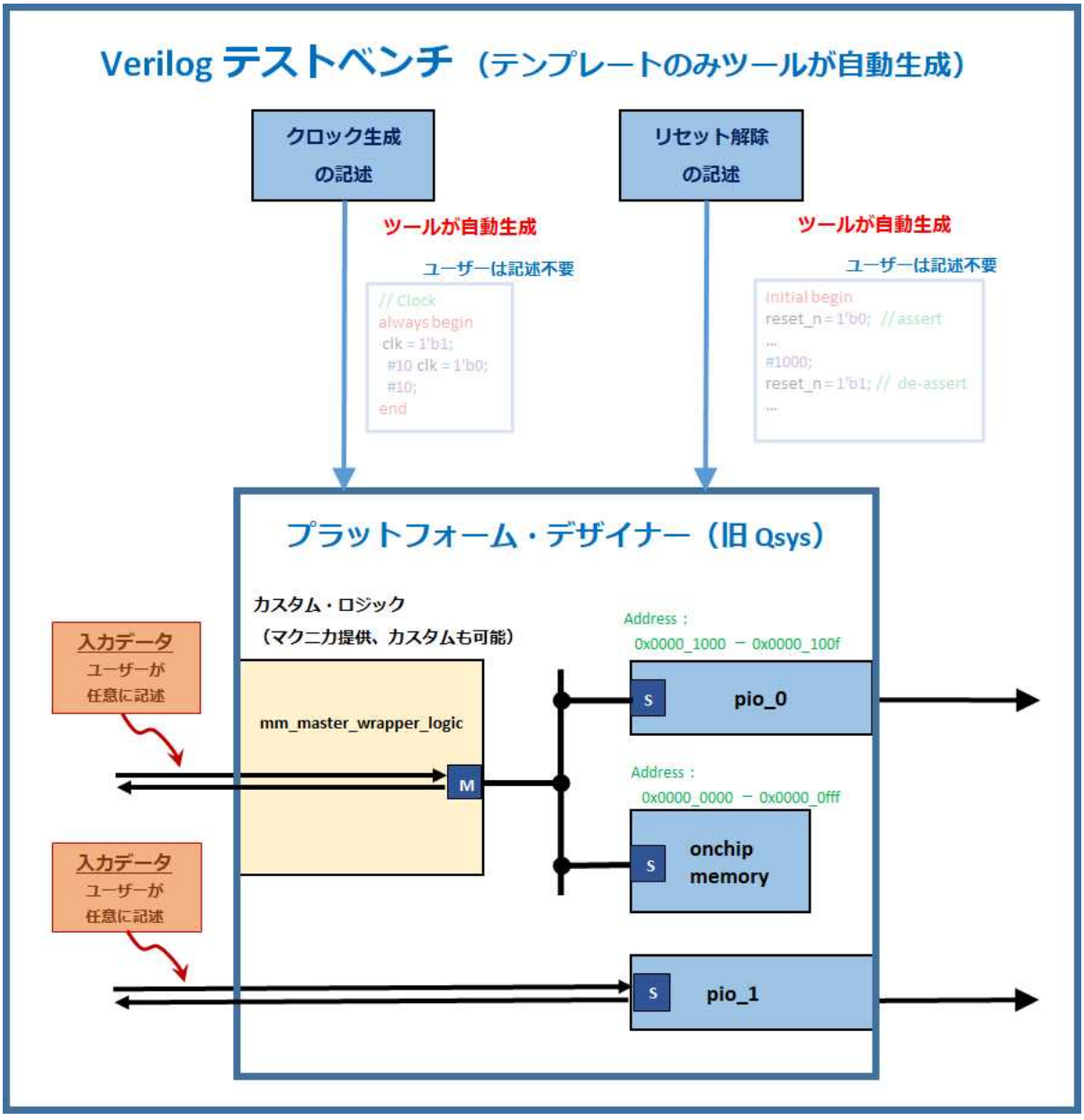
図 4-1 の中央から下側には、プラットフォーム・デザイナー（旧 Qsys）で構築されるシステムのブロック図が例示されています。このシステムは 4 つのコンポーネントで構成されています。ブロック図の上側では 3 つのコンポーネントが互いに内部バスで接続されています。この内部バスは、Avalon Memory Mapped（略して Avalon-MM）インターフェイスで構成されています。このブロック図左側のカスタムロジックは、内部バスを丸ごと外部に引き出す役割を担っています。

引き出された信号に対して入力データを与えたり、出力データを取り込んだりする動作は、ブロック図外側の Verilog テストベンチが担います。ユーザーは所望の入力データを用意して、このテストベンチに追記することで、カスタムロジックおよび内部バスを介して、その先の PIO スレーブ・コンポーネントにデータを出力させたり、オンチップメモリー（内蔵メモリー）にデータをライトもしくはリードすることが可能となります。

カスタム・コンポーネントと PIO スレーブ・コンポーネントは、Avalon-MM インターフェイスの仕様が異なっており、アドレス線やデータ線のビット幅も一致しておらず、且つ、使用している制御信号も異なります。ブロック図では、内部バスは簡略して描かれていますが、実際はインターコネクト・ファブリックと呼ばれるインテリジェンスなブロックで構成されており、上記の仕様の違いを吸収して整合を取るブリッジ機能も持ち合わせています。

ブロック図の下側には、PIO スレーブ・コンポーネントを、もう一つ追加していますが、前述のスレーブ・コンポーネントとは異なり、カスタムマスタを経由せずに、内部バスを直接外部に引き出しています。カスタム・コンポーネントの仕様に依存せずに、PIO スレーブ・コンポーネントの Avalon-MM インターフェイスを外部から直接テストベンチで制御したい場合のサンプルとして、このテンプレート・デザインに組み入れました。

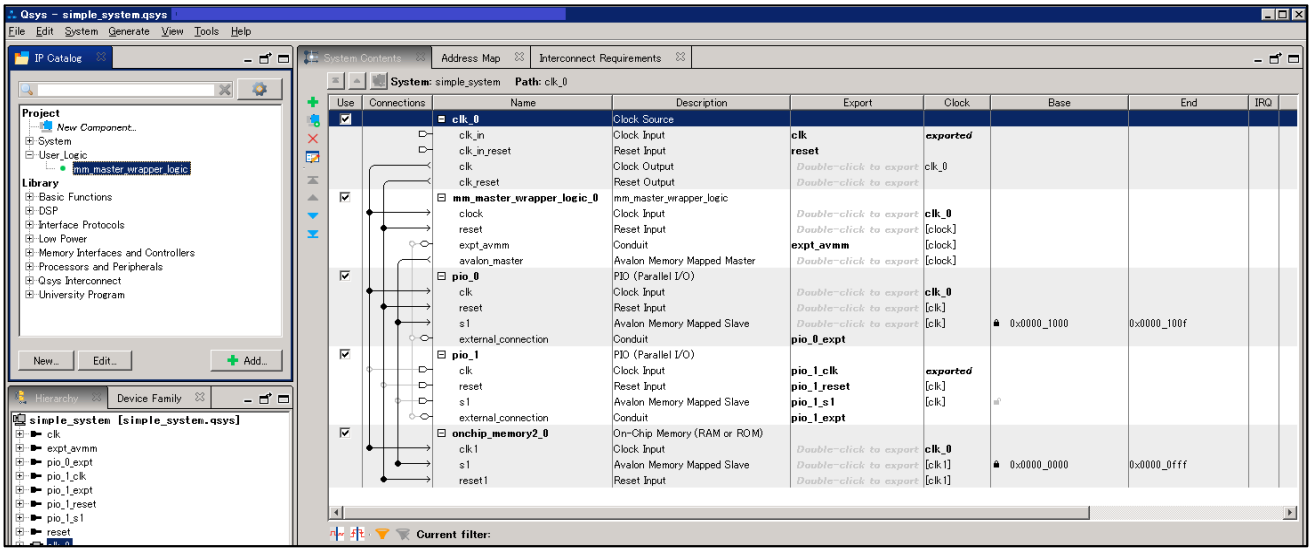
プラットフォーム・デザイナー（旧 Qsys）は、テストベンチのテンプレートを生成しますが、クロックの生成とリセット信号の解除に該当する機能もテストベンチ内に内包されています。その為、ユーザーは、これらの機能をテストベンチ内に記述する必要はありません。



【図 4-1】 Verilog テンプレート・デザインのテストベンチ・アーキテクチャー

5. テンプレート・デザインの概要

テンプレート・デザインをプラットフォーム・デザイナー（旧 Qsys）で開くと、図 5-1 のように表示され、登録されているコンポーネントを把握することができます。



【図 5-1】テンプレート・デザインで登録されているコンポーネント

このテンプレートでは、アドレスマップは、次の表 5-1 のようにマッピングされています。

マッピングされていない未定義アドレス空間にアクセスしたときの挙動は不確定であり、バスのロックアップ等が発生する可能性も考えられる為、基本的にこの空間へのアクセスは避ける必要があります。尚、このテンプレート・デザインでは、この不確定な挙動を抑止する設定を行っています。詳細は後述します。

【表 5-1】テンプレート・デザインのアドレスマップ

	Start Byte Address	End Byte Address
mm_master_wrapper_logic	Avalon-MM Master	
pio_0	0x0000_1000	0x0000_100f
pio_1	非マッピング	
onchip_memory2_0	0x0000_0000	0x0000_0000

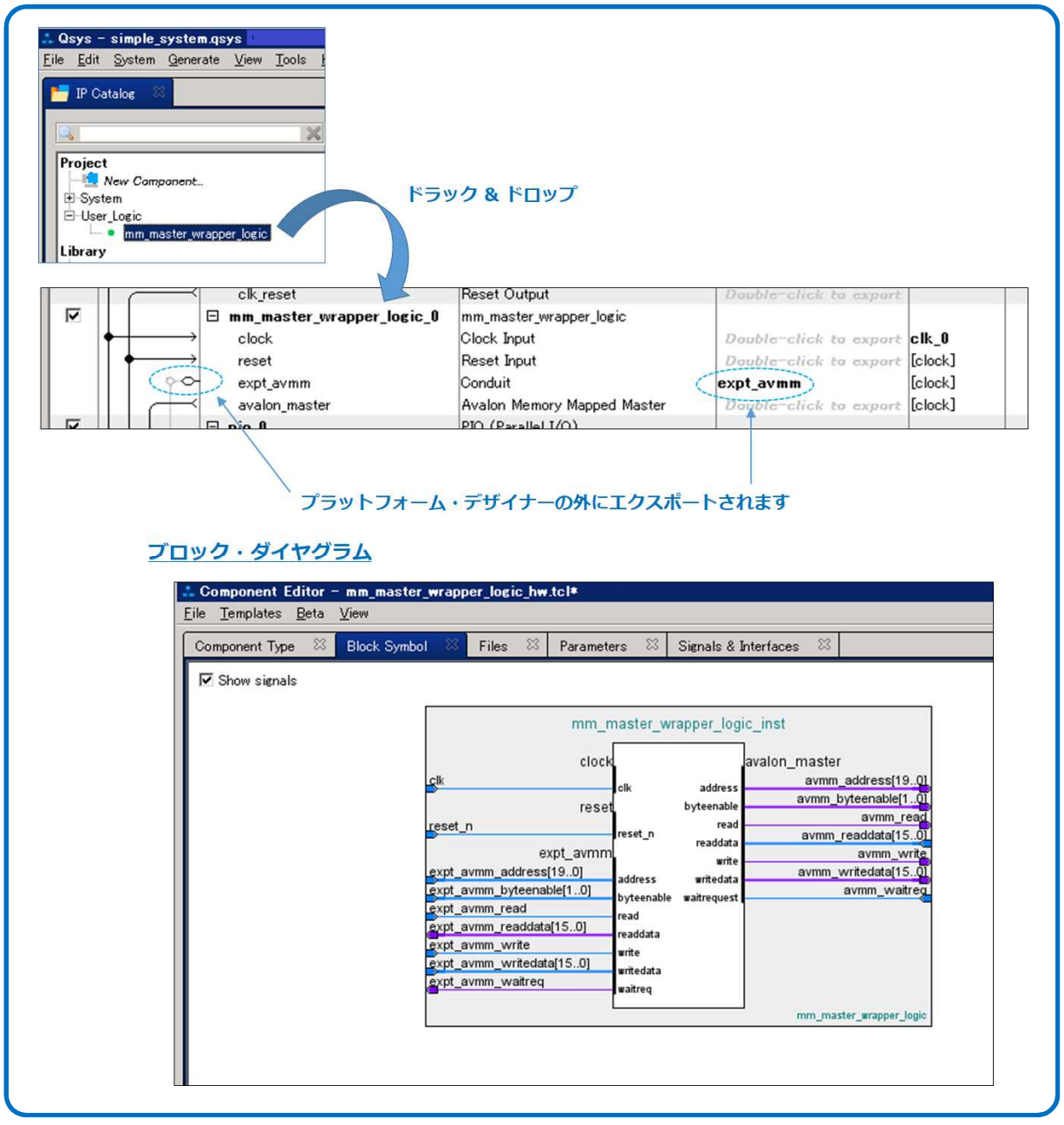
以降は、各コンポーネントの概要として、プラットフォーム・デザイナー（旧 Qsys）に登録する際の設定項目などを個別に説明します。

5-1. コンポーネント：カスタムロジック - mm_master_wrapper_logic

カスタムロジックは、ユーザーが記述した Verilog HDL を取り込んだユーザーロジックであり、プラットフォーム・デザイナー（旧 Qsys）のコンポーネントとして追加できます。

前述の図 4-1 のように、システムの内部バスを、丸ごと外部に引き出して、ユーザーが外部からバスを制御する役割を担います。

図 5-2 では、カスタムロジックをマウスでドラッグ & ドロップさせて、プラットフォーム・デザイナー（旧 Qsys）のシステムに追加して、外部に引き出す信号（Conduit インターフェイス）を外部にエクスポートさせる設定を例示しています。図 5-2 のように、図内の右下に、事前に定義したインターフェイス名 expt_avmm が表示されていれば、Conduit インターフェイスでカテゴライズされた信号が、プラットフォーム・デザイナー（旧 Qsys）のシステムの外部にエクスポートされます。



【図 5-2】カスタムロジック追加時の設定およびブロック・ダイヤグラム

カスタムロジックは Verilog で作成しています。内部バスとしてサフィックス avmm_ で定義した 6 本の信号を、assign 文を使用してサフィックス expt_ で定義した信号と接続させることで、内部バスを丸ごと外部に引き出しています。

sample_master_wrapper.v

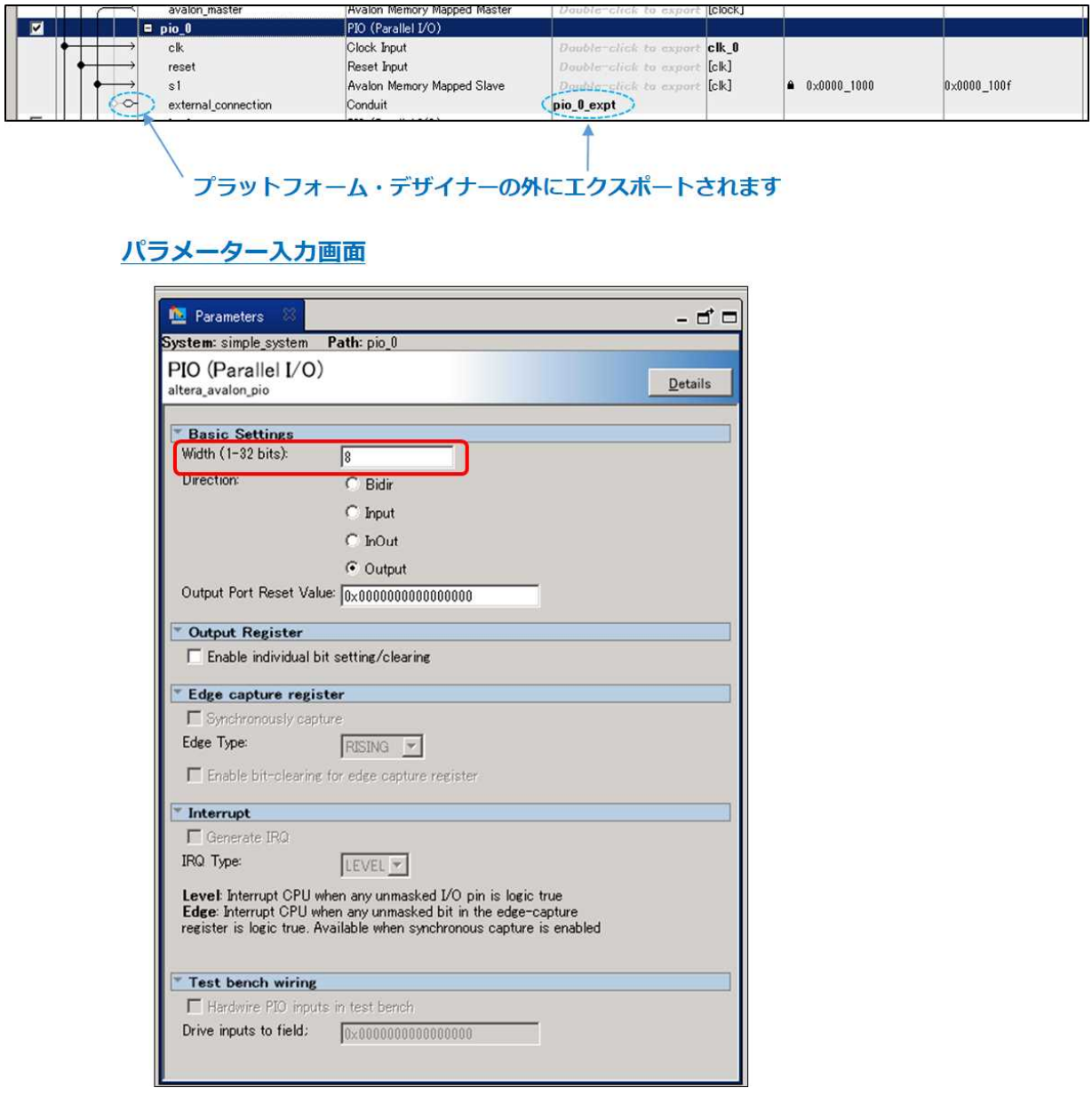
```
1  //*************************************************************************************
2  // file  sample_master_wrapper.v
3  //
4  // attention
5  // Copyright (C) 2018 MACNICA,Inc. All Rights Reserved.\n
6  //   This software is licensed "AS IS".
7  //   Please perform use of this software by a user's own responsibility and expense.
8  //   It cannot guarantee in the maker side about the damage which occurred by the ab-
9  //   ility not to use or use this software, and all damage that occurred secondarily.
10 //*************************************************************************************
11 module sample_master_wrapper(
12    // General Interface
13         input  wire           clk,              //   clk
14         input  wire           reset_n,          //   reset_n
15
16    // Avalon-MM Slave Interface
17         output  wire [19:0]   avmm_address,     //   address
18         output  wire [1:0]    avmm_byteenable,  //   byteenable
19         output  wire          avmm_read,        //   read
20         input  wire [15:0]    avmm_readdata,    //   readdata
21         output  wire          avmm_write,       //   write
22         output  wire [15:0]   avmm_writedata,   //   writedata
23         input  wire           avmm_waitreq,
24
25    // Conduit Interface
26         input  wire [19:0]    expt_avmm_address,      //   address for export
27         input wire [1:0]   expt_avmm_byteenable,   //   byteenable for export
28         input  wire           expt_avmm_read,         //   read
29         output wire [15:0]    expt_avmm_readdata,     //   readdata for export
30         input  wire           expt_avmm_write,        //   write for export
31         input  wire [15:0]    expt_avmm_writedata,    //   writedata for export
32         output wire           expt_avmm_waitreq
33
34     );
35
36
37 assign   avmm_address = expt_avmm_address;
38 assign   avmm_byteenable = expt_avmm_byteenable;
39 assign   avmm_read = expt_avmm_read;
40 assign   expt_avmm_readdata = avmm_readdata;
41 assign   avmm_write = expt_avmm_write;
42 assign   avmm_writedata = expt_avmm_writedata;
43 assign   expt_avmm_waitreq = avmm_waitreq;
44
45
46 endmodule
```

【図 5-3】カスタムロジックの記述内容

5-2. コンポーネント：PIO（8 ビット出力） - pio_0

PIO は、プラットフォーム・デザイナー（旧 Qsys）で用意されているコンポーネントであり、入力するパラメーターを変更することで、ビット幅や入力・出力属性等の仕様をユーザーがカスタマイズすることができます。

図 5-4 では、8 ビット幅の PIO 出力となるようにパラメーターを入力して、プラットフォーム・デザイナー（旧 Qsys）のシステムに追加後、Conduit インターフェイスを外部にエクスポートさせる設定を例示しています。



【図 5-4】PIO（8 ビット出力）追加時の設定およびパラメーター入力画面

5-3. コンポーネント：PIO（16 ビット出力） - pio_1

図 5-5 では、16 ビット幅の PIO 出力となるようにパラメーターを入力して、プラットフォーム・デザイナー（旧 Qsys）のシステムに追加後、Conduit インターフェイスを外部にエクスポートさせる設定を例示しています。

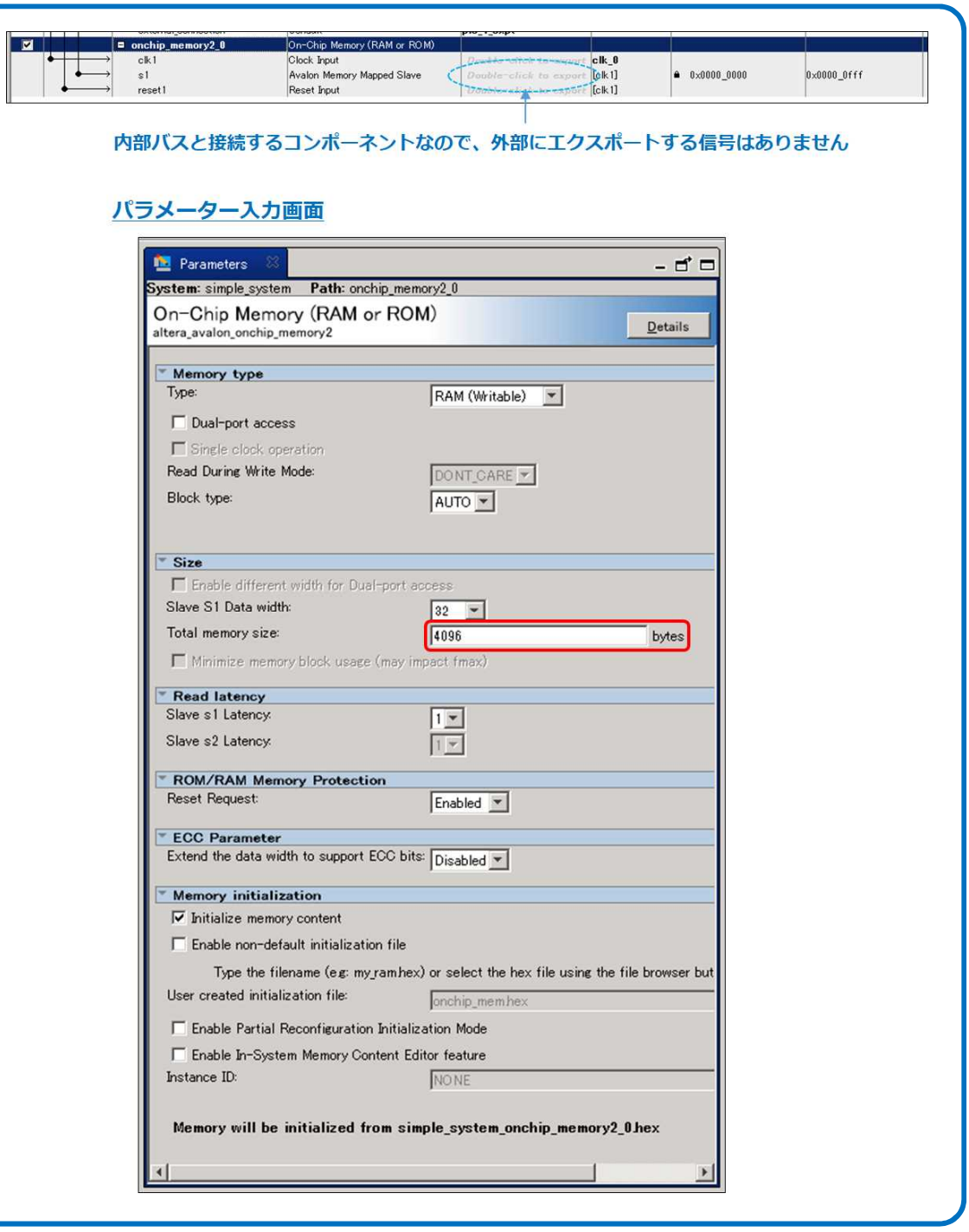
【図 5-5】PIO（16 ビット出力）追加時の設定およびパラメーター入力画面

5-4. コンポーネント：On-Chip Memory - onchip_memory2_0

On-Chip Memory は、プラットフォーム・デザイナー（旧 Qsys）で用意されているコンポーネントであり、入力するパラメーターを変更することで、ビット幅やメモリー容量等の仕様をユーザーがカスタマイズすることができます。

前述の図 4-1 で例示しているように、On-Chip Memory はプラットフォーム・デザイナー（旧 Qsys）のシステムの中にある内部バスと接続するコンポーネントなので、外部にエクスポートする信号はありません。

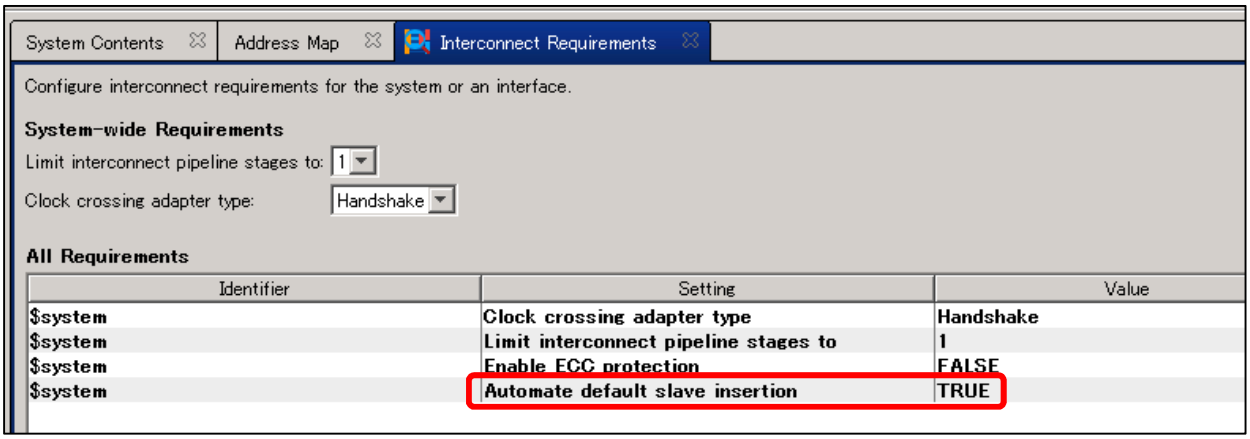
図 5-6 では、メモリー容量 4,096 バイトで構成されるようにパラメーターを入力して、システムに追加したときの設定を例示しています。



【図 5-6】On-Chip Memory 追加時の設定およびパラメーター入力画面

このテンプレート・デザインでは、前述の表 1 では未定義のアドレス空間にアクセスが発生した際の不確定な挙動（バスのロックアップ等）を抑止するように設定されています。これは図 5-7 のように、プラットフォーム・デザイナー（旧 Qsys）の Interconnect Requirement タブより、以下の項目を追加することで設定できます。

Automate default slave insertion： TRUE



【図 5-7】未定義アドレス空間アクセス時の不確定な挙動を防ぐ設定

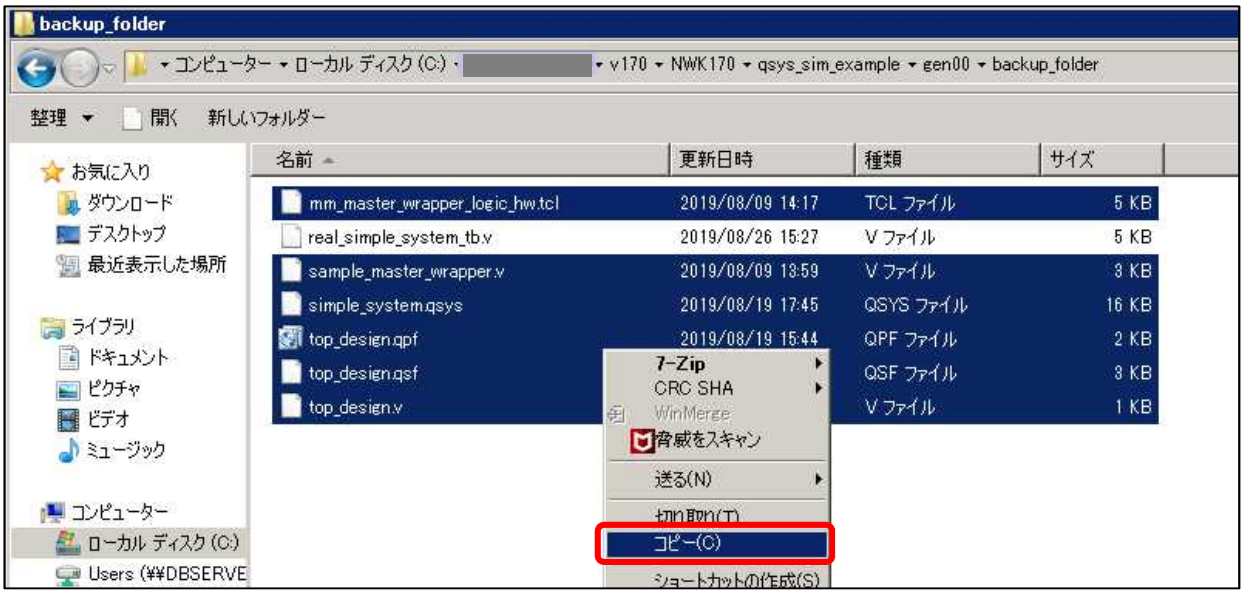
6. シミュレーション手順の解説

以下のテンプレート・デザインならびに手順書をダウンロードしてください。

- テンプレート・デザイン:
  backup_folder.zip
- シミュレーション手順書:
  Verilog テストベンチでプラットフォーム・デザイナー（旧 Qsys）をシミュレーションする手順の解説_v170_r1.pdf
  （本書）

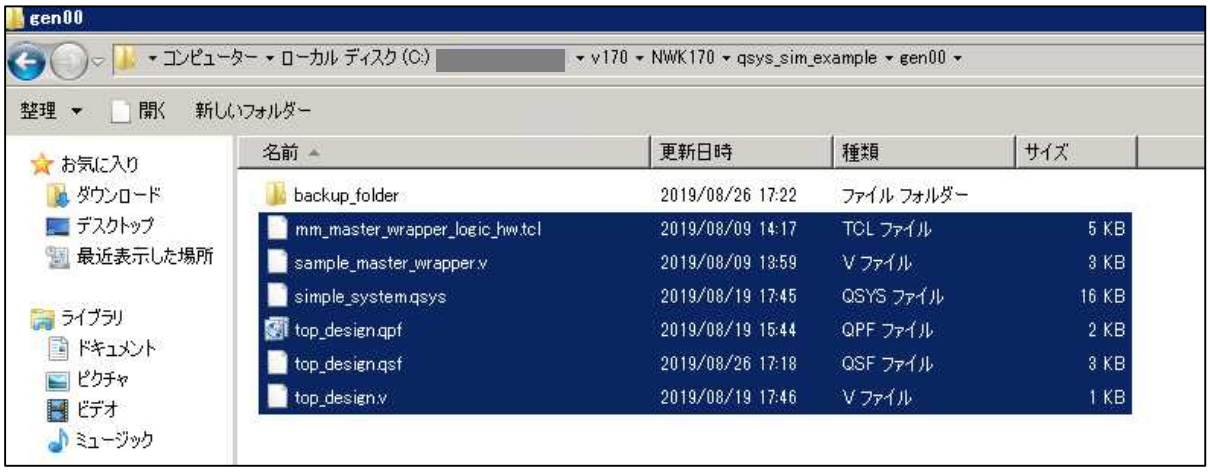
6-1. Quartus® Prime プロジェクトの起動からテストベンチの生成まで

(1) テンプレート・デザイン backup_folder に格納されているファイルの中から、real_simple_system_tb.v を除く全てのファイルを選択してコピーします。



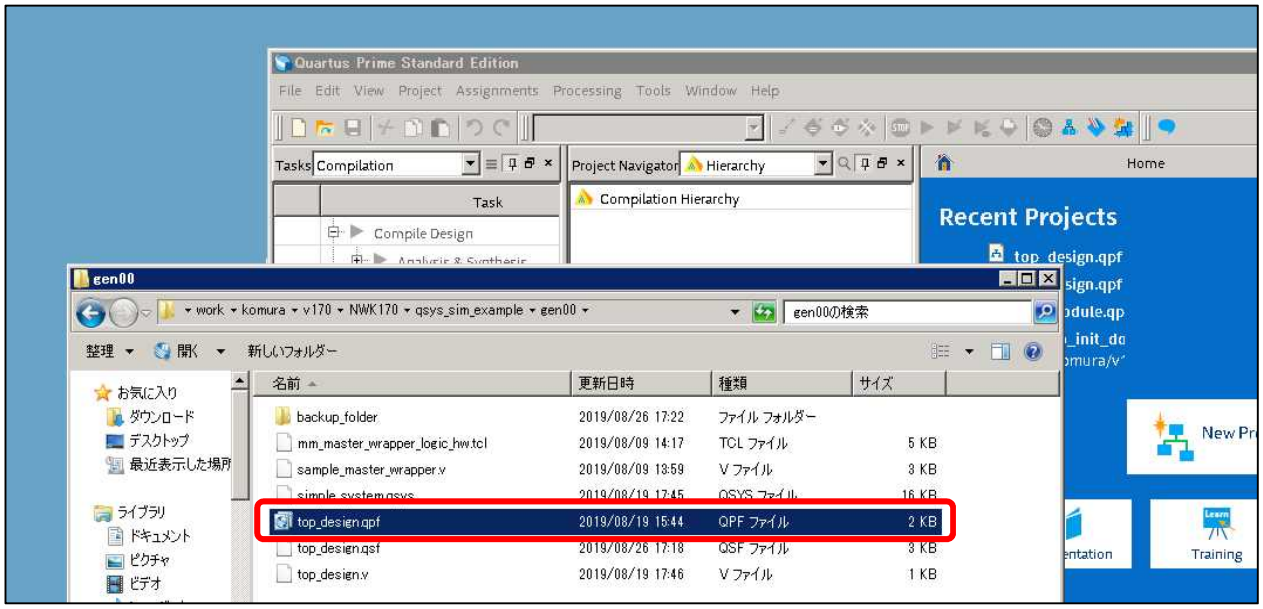
【図 6-1】テンプレート・デザインで用意されているファイル類のコピー

(2) これらのファイルを、所望のプロジェクト・フォルダーにペーストします。



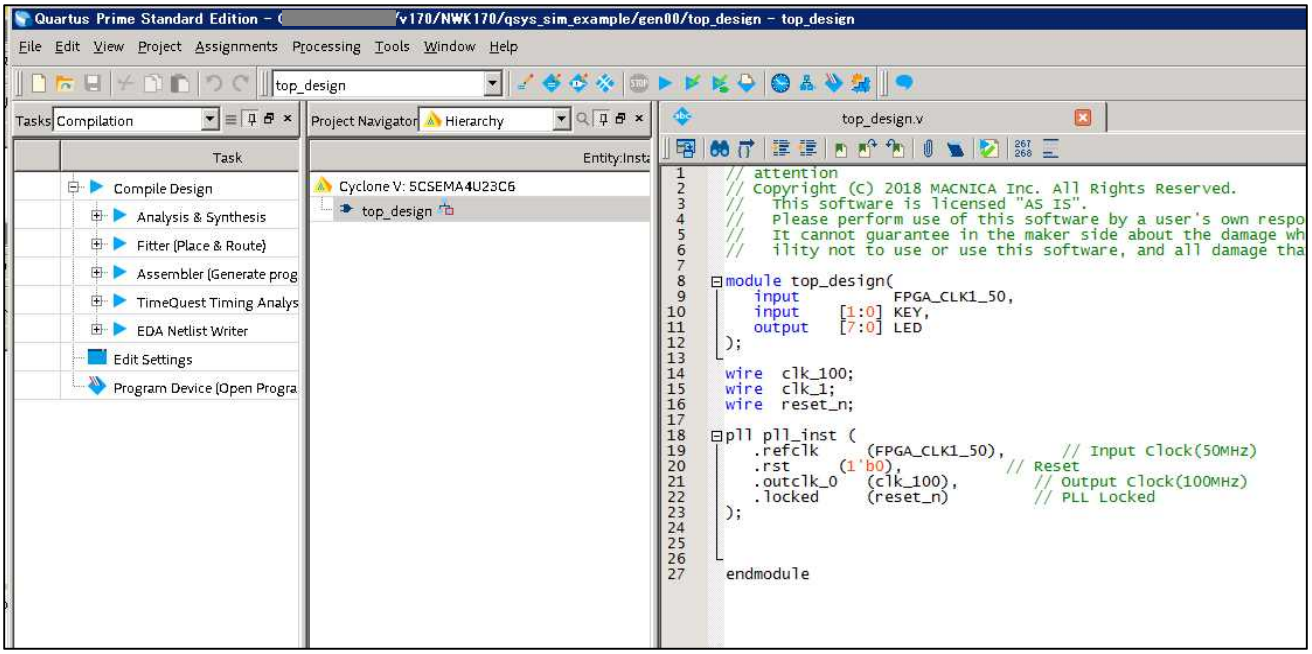
【図 6-2】テンプレート・デザインで用意されているファイル類のペースト

(3) Quartus® Prime を起動後、ペーストしたファイルの中にある top_design.qpf を読み込ませて、Quartus® Prime のプロジェクトを開きます。



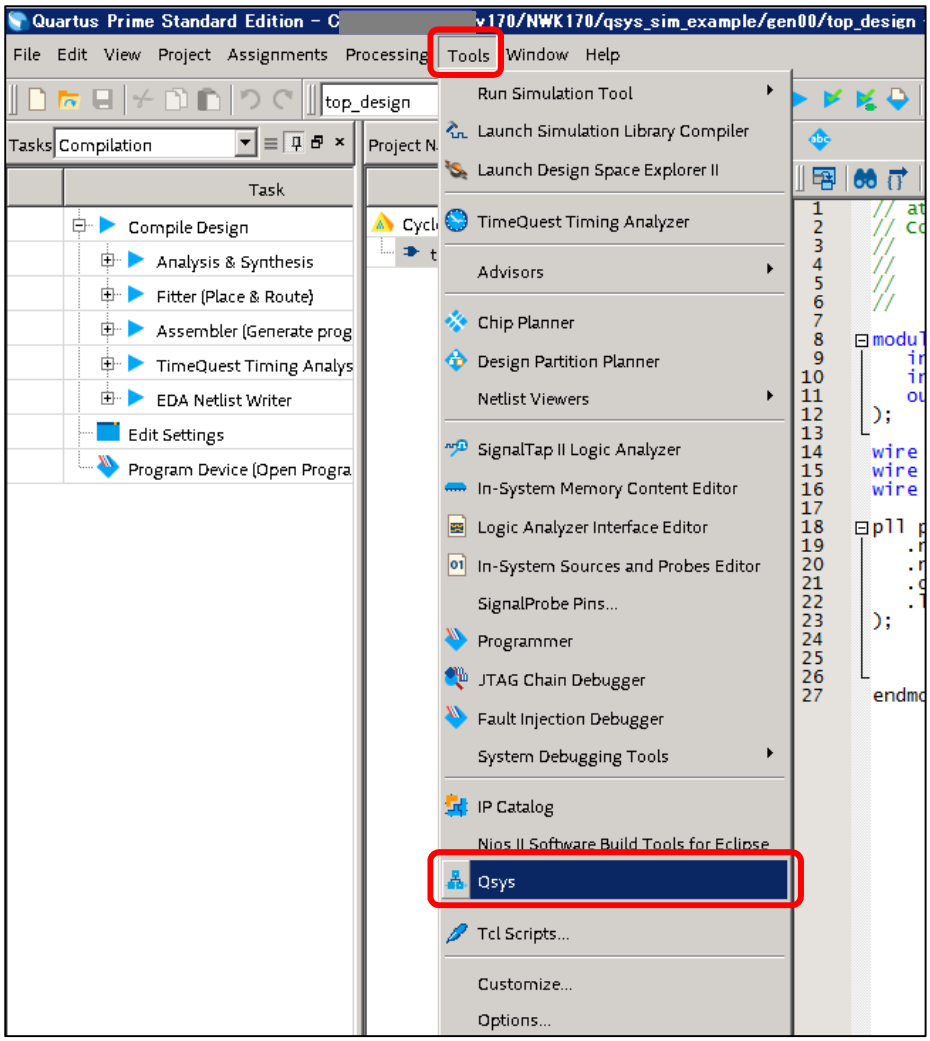
【図 6-3】Quartus® Prime プロジェクトの起動

(4) プロジェクトの最上位階層は、図 6-4 のように比較的最小限で構成されています。



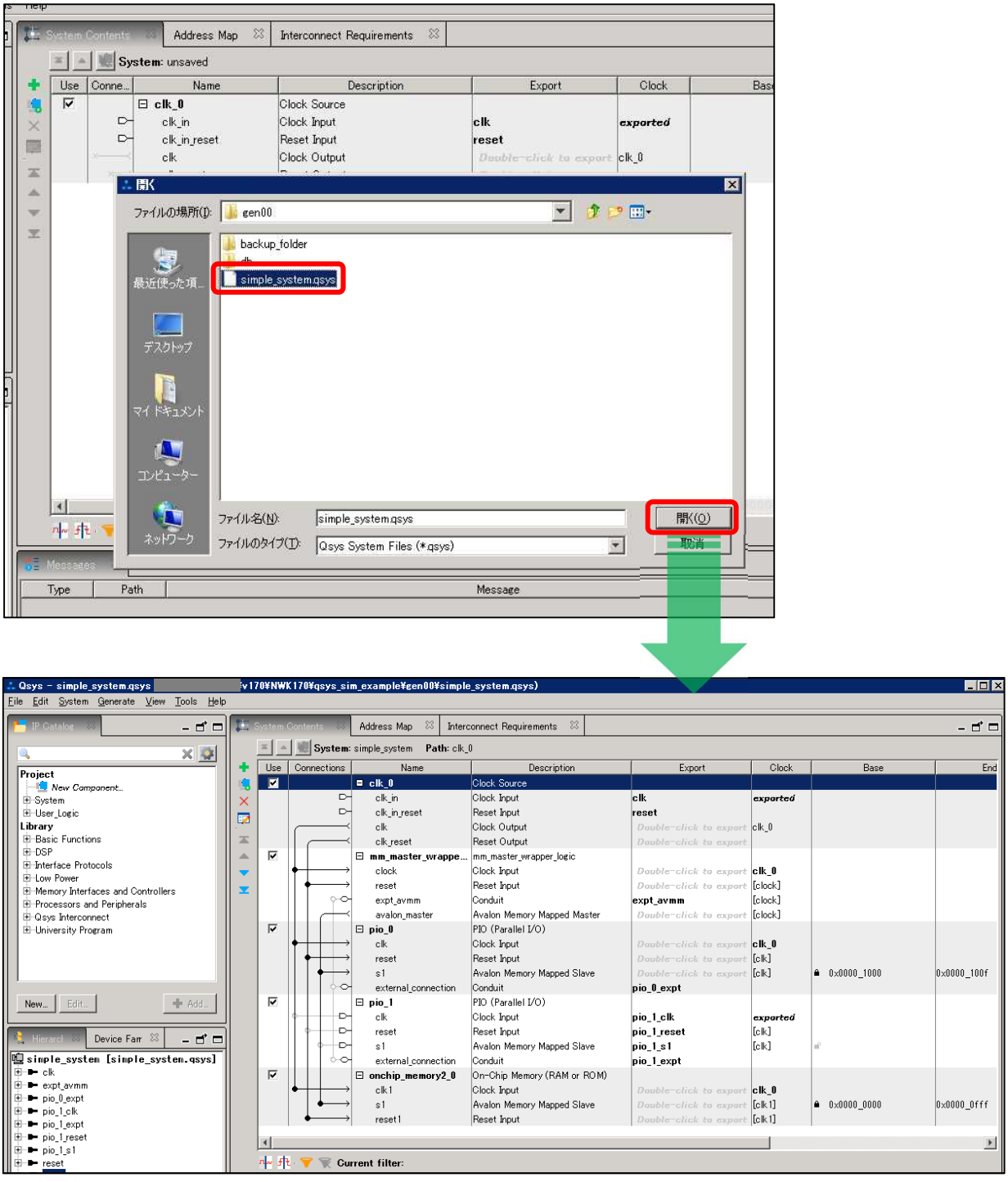
【図 6-4】Quartus® Prime プロジェクトのテンプレート・デザインの最上位階層

(5) Tools メニューから Qsys を選択します。



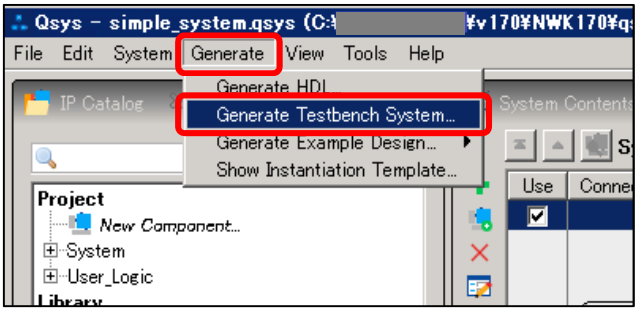
【図 6-5】プラットフォーム・デザイナー（旧 Qsys）の起動 【Tools メニュー】

(6) simple_system.qsys を選択して、作成済の プラットフォーム・デザイナー （旧 Qsys） テンプレートを起動します。



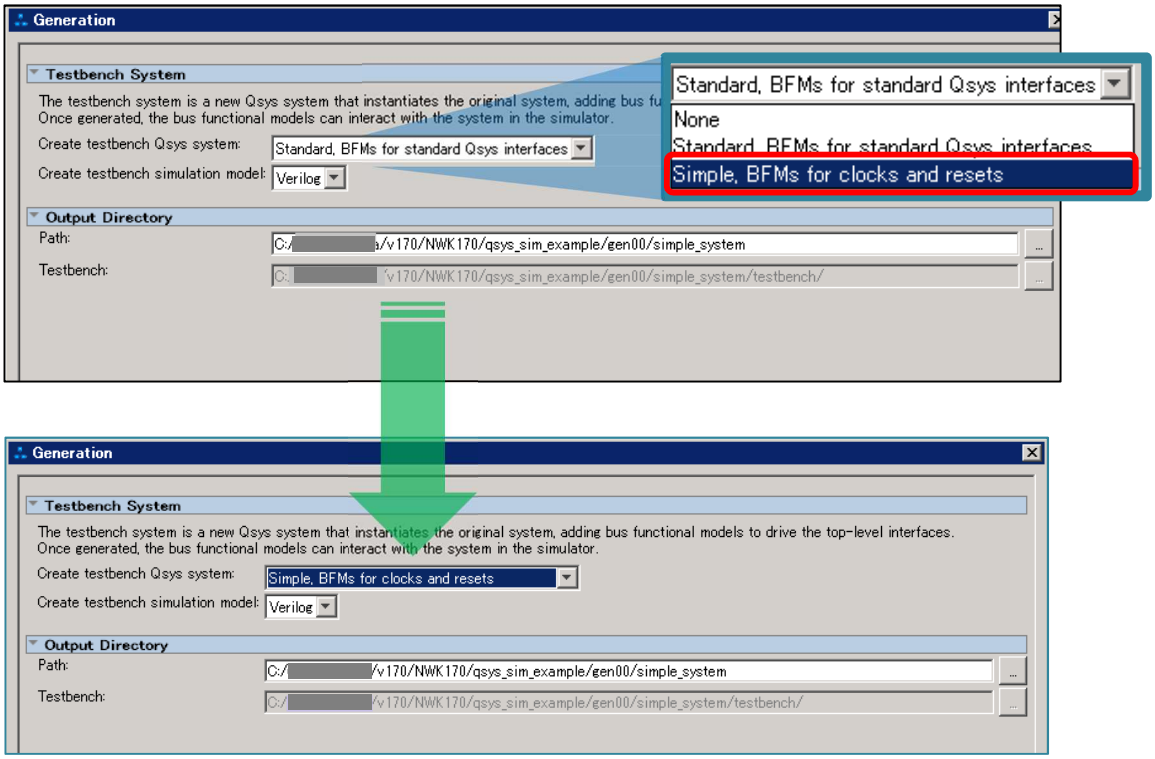
【図 6-6】プラットフォーム・デザイナー（旧 Qsys）の起動 【.qsys ファイルの選択】

(7) Generate メニューから Generate Template System を選択します。



【図 6-7】Generate メニューから Generate Template System を選択

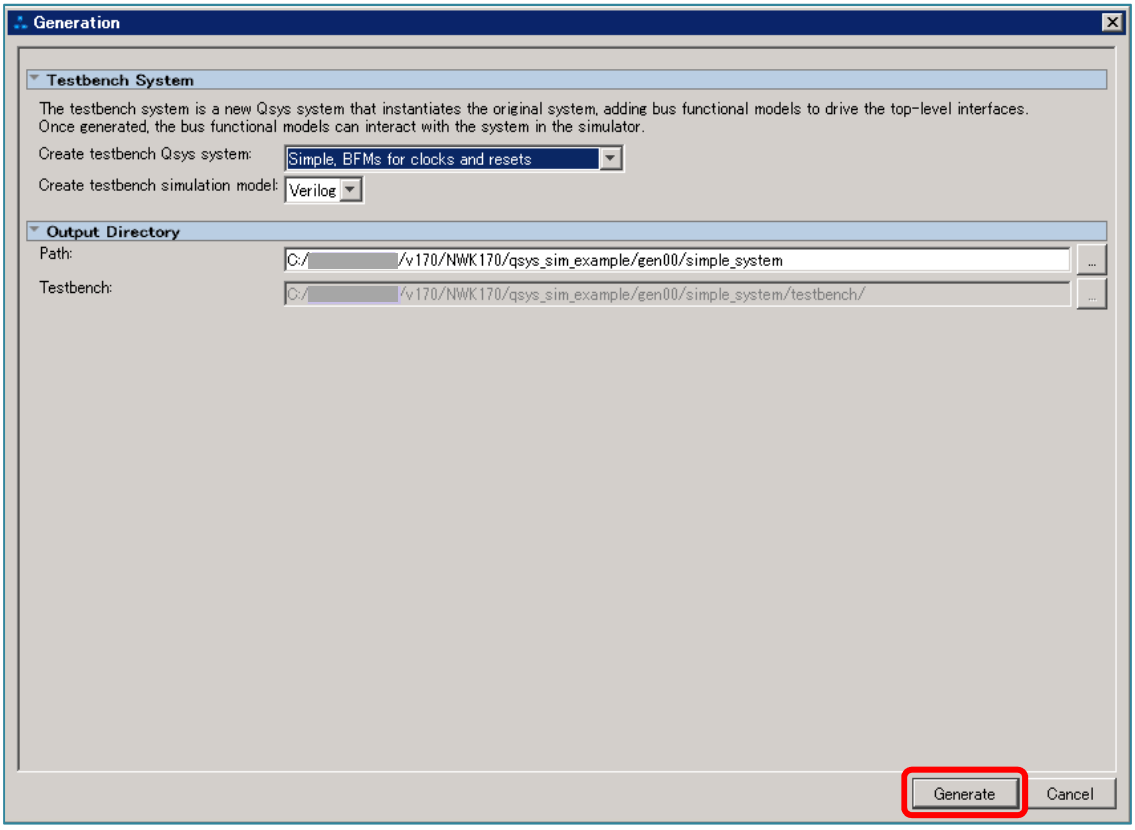
(8) Generation 画面が起動した後、Testbench System 欄の Create testbench Qsys system よりプルダウン・メニューから Simple, BFMs for clocks and resets を選択します。



【図 6-8】プラットフォーム・デザイナー（旧 Qsys）のメニューから使用するテストベンチを選択

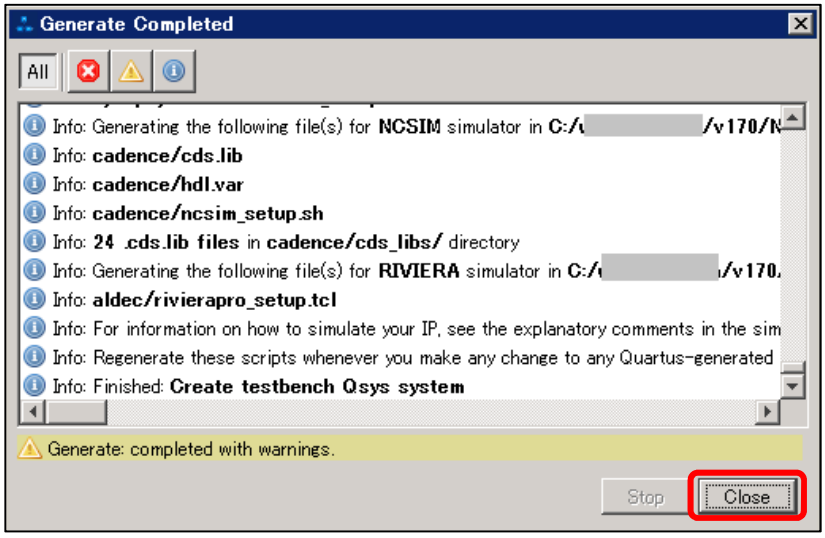
(9) 画面右下の［Generate］ボタンをクリックして、シミュレーション用の Verilog ファイルを生成します。

- この操作で、テストベンチのテンプレート、クロック生成の記述、リセット解除の記述を生成します。
- クロック生成記述およびリセット解除記述は、ツールがデフォルトで用意しているシミュレーション・モデル内に記述されています。
- このシミュレーション・モデルは、BFM になりますが、このテンプレート・デザインでは、ユーザーは BFM に関する特別な知識は必要ありません。



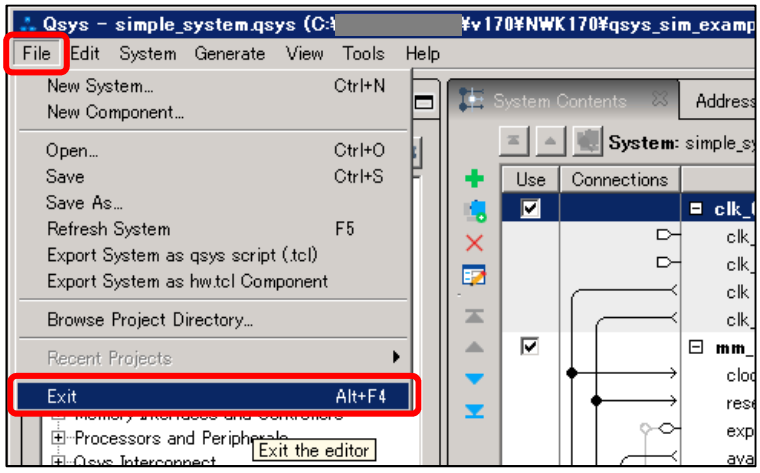
【図 6-9】プラットフォーム・デザイナー（旧 Qsys）のメニューから使用するテストベンチの生成

(10) シミュレーション用のファイルの生成が正常に行われたら［Close］ボタンをクリックします。



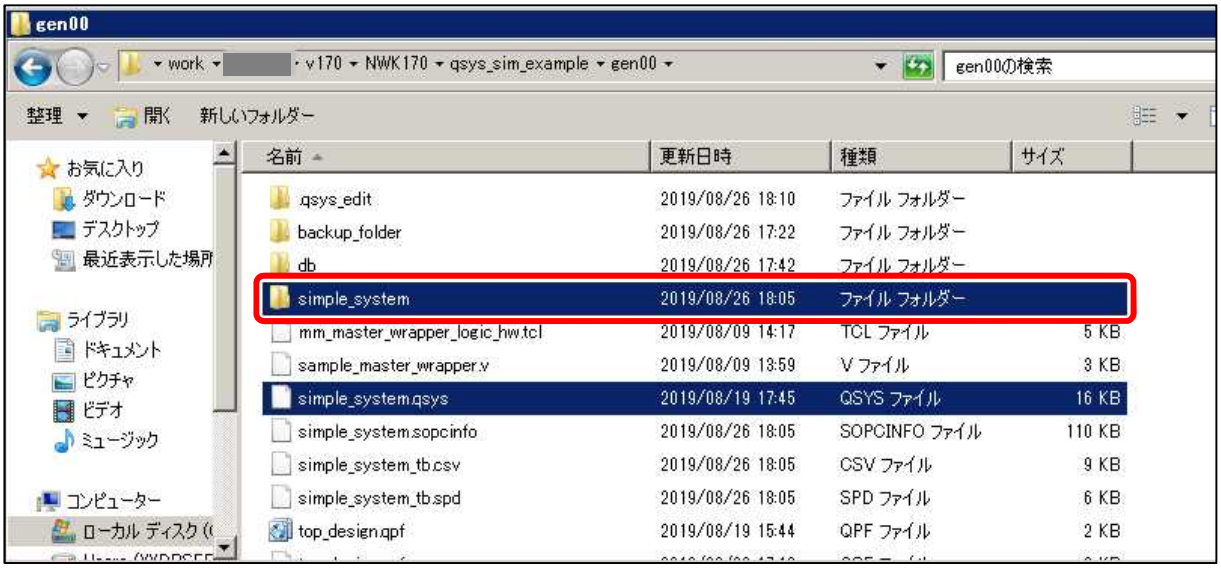
【図 6-10】テストベンチの生成終了

(11) プラットフォーム・デザイナー（旧 Qsys）のメイン画面に戻ったら、図 6-11 のように、File メニューから Exit を選択して、プラットフォーム・デザイナー（旧 Qsys）を閉じます。



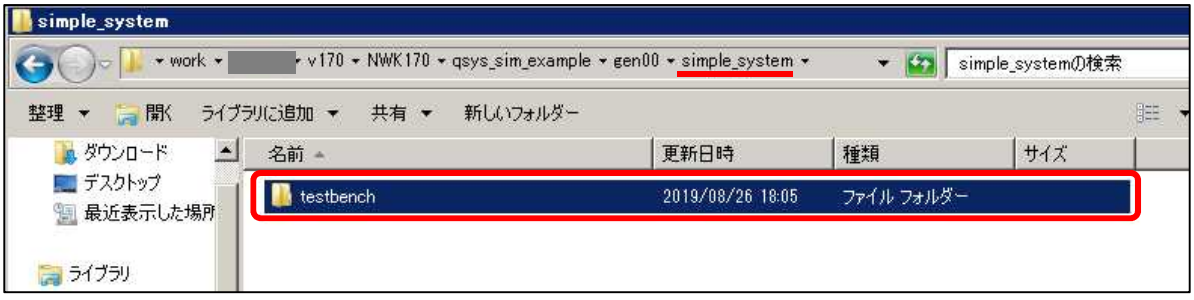
【図 6-11】プラットフォーム・デザイナー（旧 Qsys）の終了

(12) ここで、Quartus® Prime のプロジェクト・フォルダーに着目します。
拡張子 .qsys ファイルのファイル名と同じ名称のフォルダーが生成されていることが確認できます。
本例では、図 6-12 のように simple_system.qsys のファイル名に該当する simple_system フォルダーが生成されています。



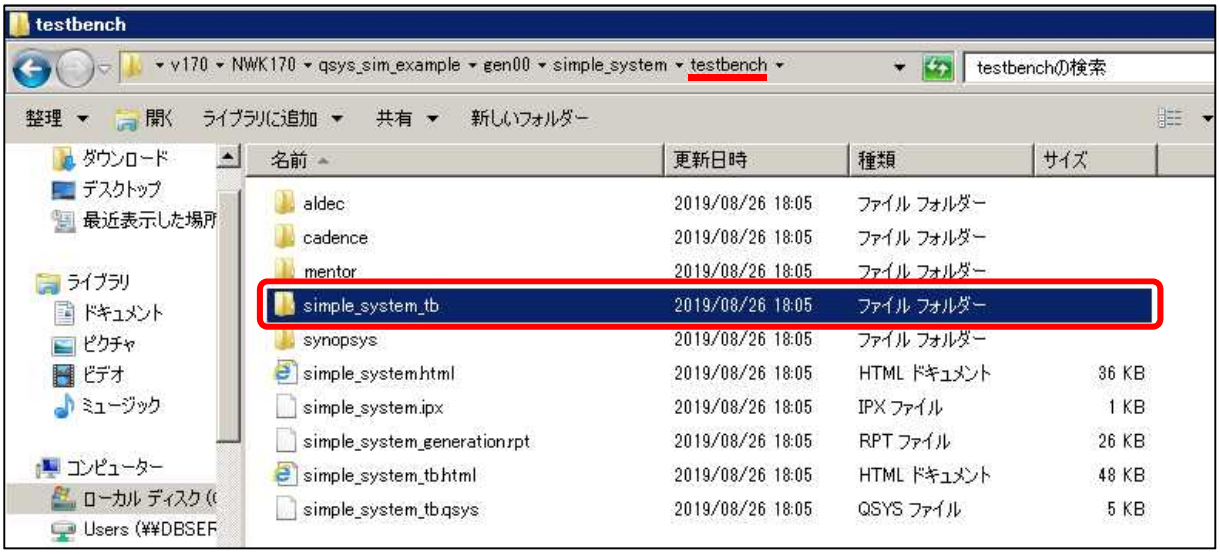
【図 6-12】拡張子 .qsys ファイルおよび、生成したフォルダー

(13) simple_system フォルダー内には testbench フォルダーが生成されています。



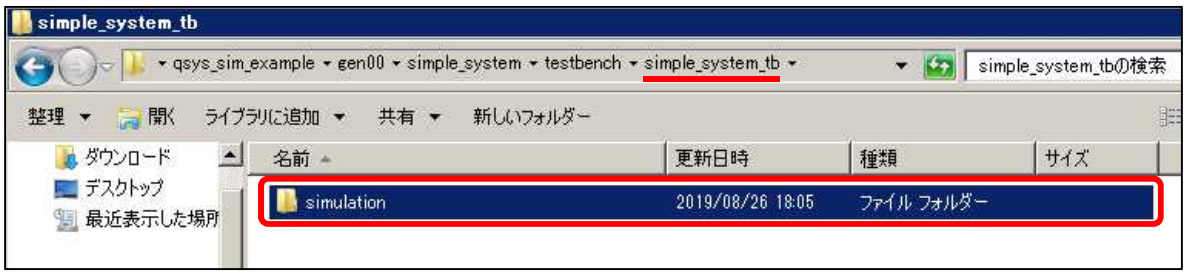
【図 6-13】simple_system フォルダー内の testbench フォルダー

(14) testbench フォルダー内に、simple_system_tb フォルダーが生成されています。



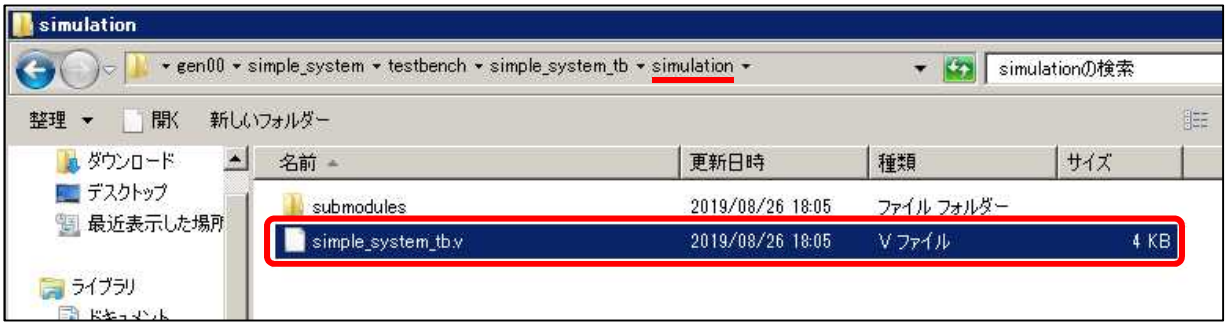
【図 6-14】testbench フォルダー内の simple_system_tb フォルダー

(15) simple_system_tb フォルダー内には、simulation フォルダーが用意されています。



【図 6-15】simple_system_tb フォルダーの simulation フォルダー

(16) simulation フォルダー内には、simple_system_tb.v が用意されています。
この <Qsys ファイル名>_tb.v が、本資料で紹介するテストベンチのテンプレートに該当します。

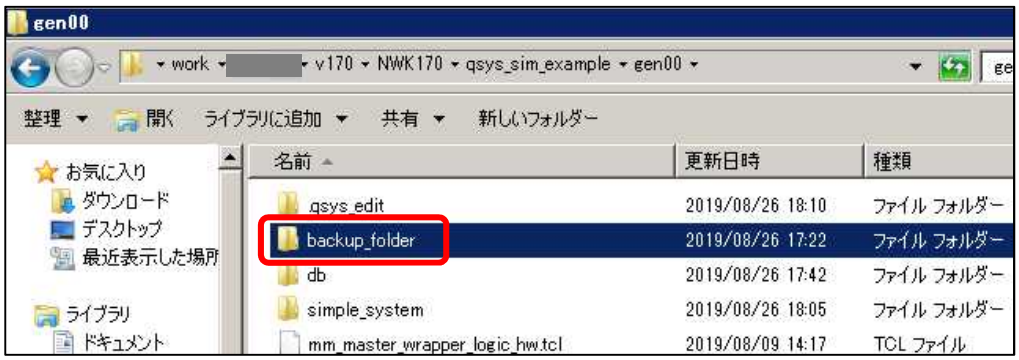


【図 6-16】テンプレート・テストベンチ（ツールが自動生成）

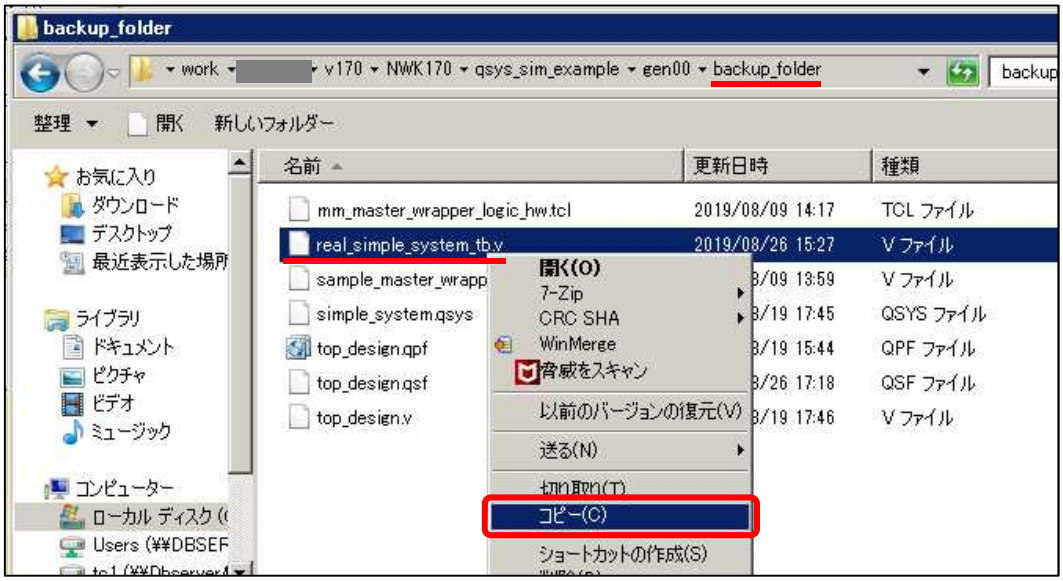
(17) simple_system_tb.v をハイライトして、ファイル名を org_simple_system_tb.v に変更します。

【図 6-17】テンプレート・テストベンチの無効化（ファイル名変更）

(18) 次に、テンプレート・デザイン backup_folder に格納されているファイルの中から、
real_simple_system_tb.v を選択してコピーします。

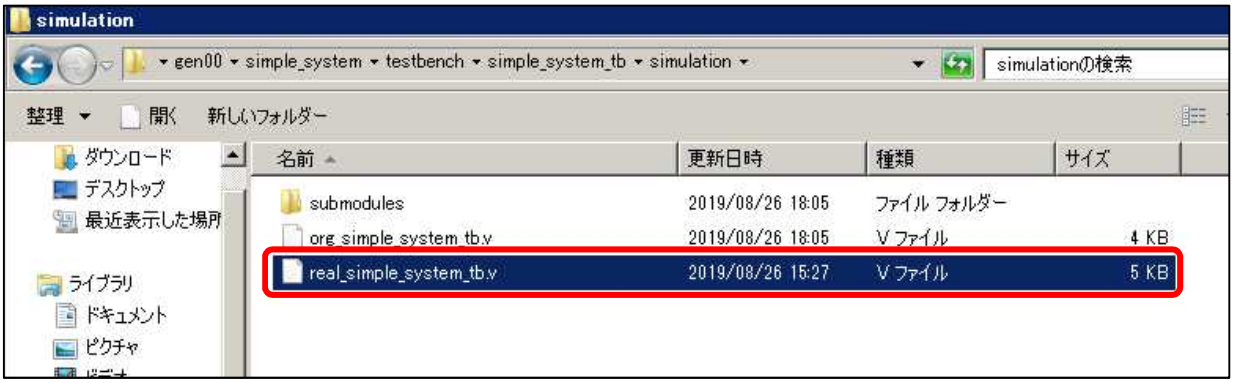


【図 6-18】カスタム済テンプレート・テストベンチを格納しているフォルダー



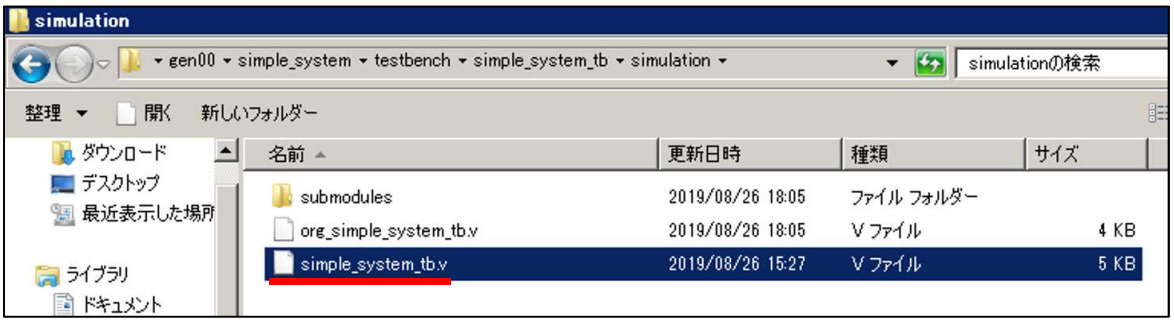
【図 6-19】カスタム済テンプレート・テストベンチ（コピー元）

(19) この real_simple_system_tb.v を、前述の org_simple_system_tb.v が格納されているフォルダー内にペーストします。



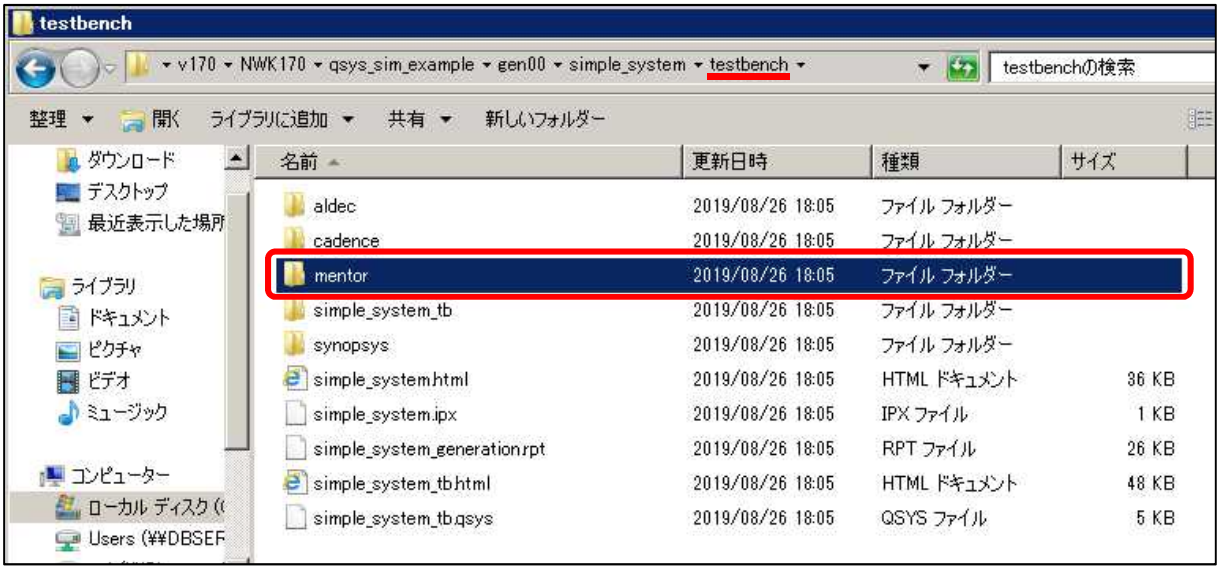
【図 6-20】カスタマイズ済テンプレート・テストベンチ（ペースト先）【ペースト後】

(20) ペーストした real_simple_system_tb.v のファイル名を、simple_system_tb.v に変更します。



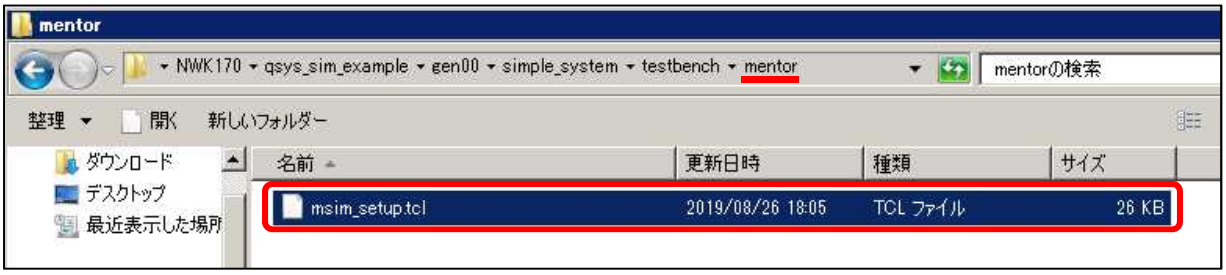
【図 6-21】テンプレート・テストベンチの有効化（ファイル名の修正）【修正後】

(21) 2 つ上の階層の testbench ディレクトリーに移動して、図 6-22 のように mentor ディレクトリーが生成されていることを確認します。



【図 6-22】testbench ディレクトリー内の mentor ディレクトリー

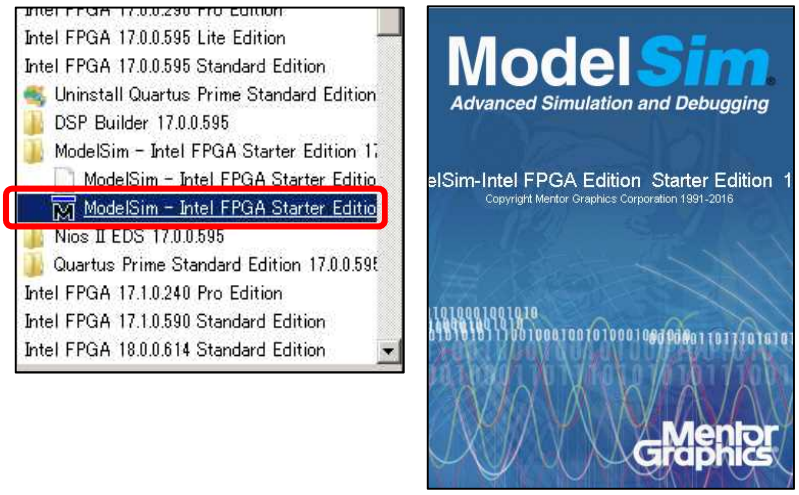
(22) この mentor ディレクトリー内に、msim_setup.tcl が用意されていることを確認します。



【図 6-23】mentor ディレクトリー内の msim_setup.tcl

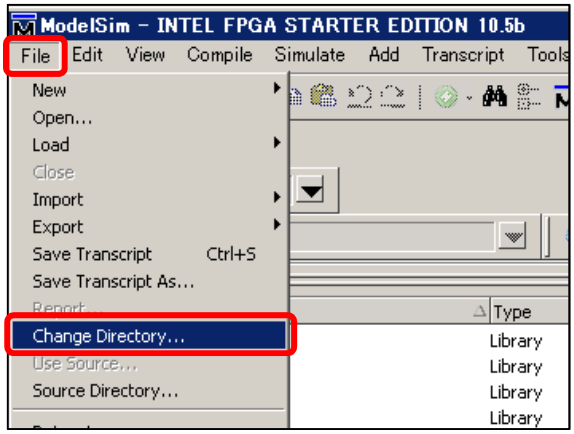
6-2. シミュレーション・ツール ModelSim® - IE の起動から波形表示まで

(1) PC のメニューから、ModelSim® - IE を起動します。



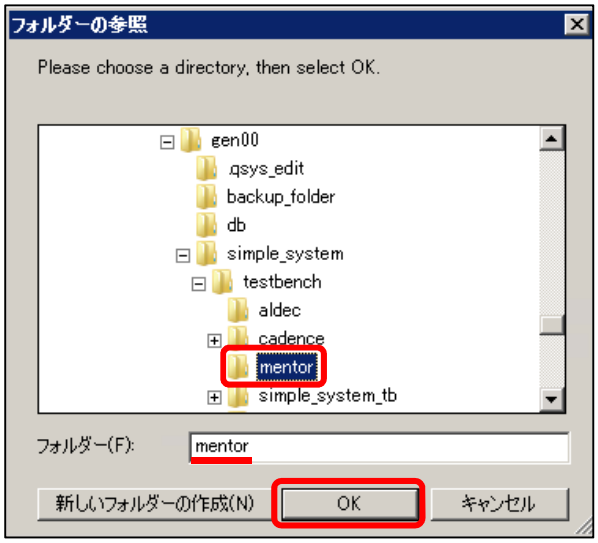
【図 6-24】ModelSim® - IE の起動画面

(2) ModelSim® が起動した後、File メニューから Change Directory を選択します。



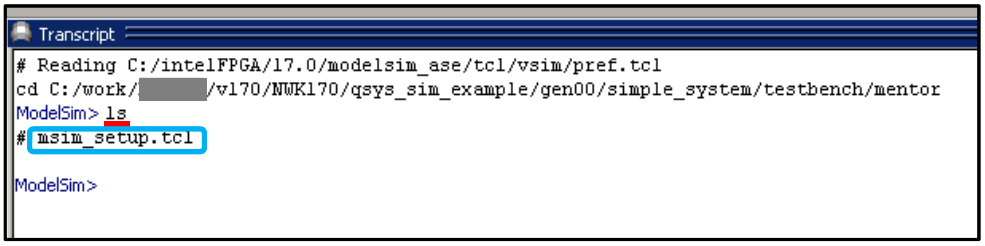
【図 6-25】Change Directory の選択

(3) ディレクトリーを移動して、図 6-26 のように前述の mentor ディレクトリーを選択して、[OK] ボタンをクリックします。



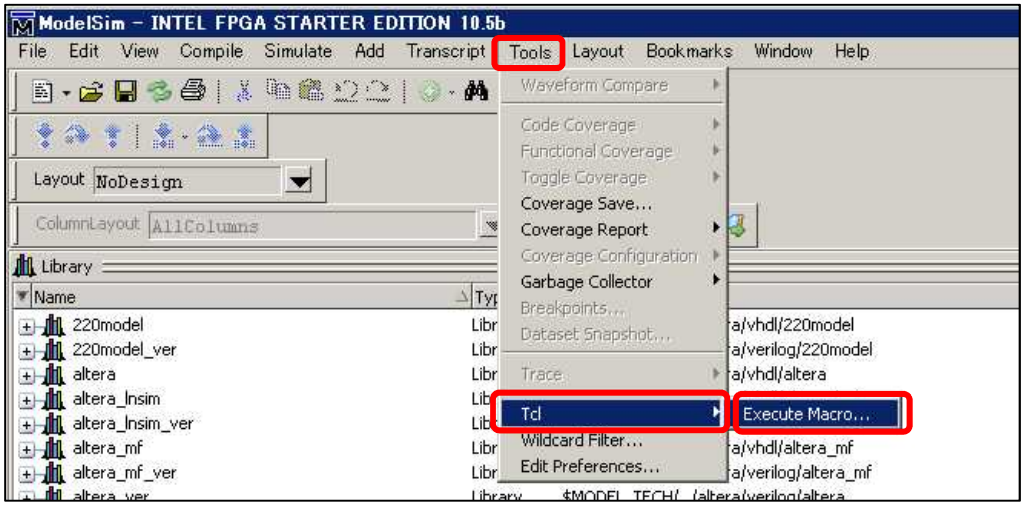
【図 6-26】mentor ディレクトリーの選択

(4) ModelSim® の下の方にある Transcript 画面でも、カレント・ディレクトリーが mentor ディレクトリーに移動していることが確認できます。この時点で ls コマンドを実行すると、前述の msim_setup.tcl があることを確認できます。



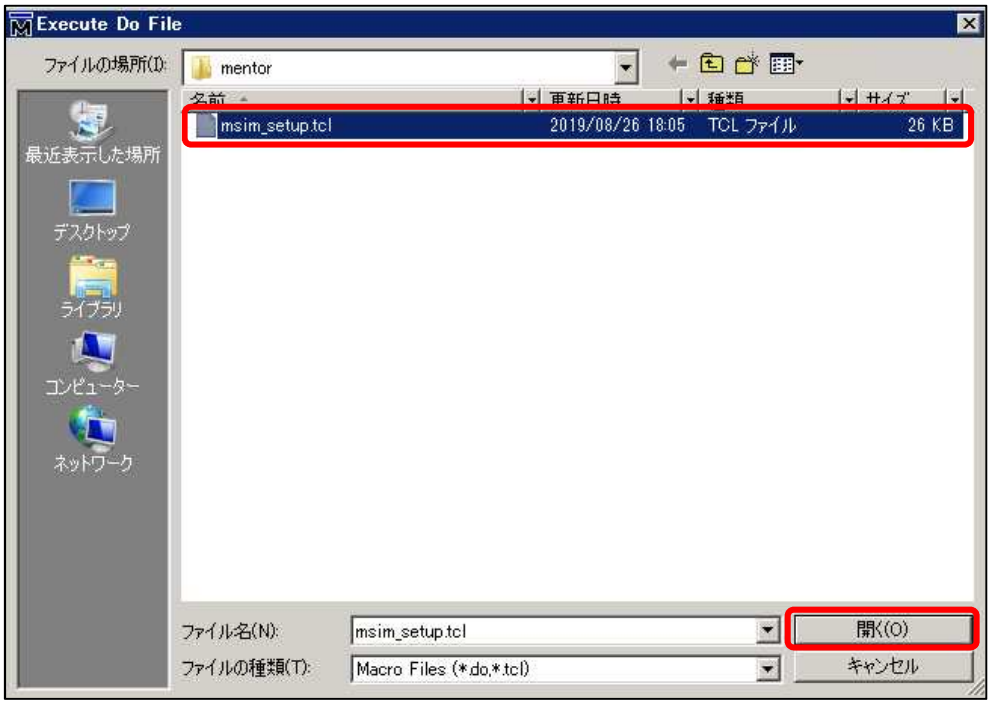
【図 6-27】カレント・ディレクトリー内にあるファイルの確認

(5) 次に ModelSim® の Tools メニューより tcl を選択後、Execute Macro を選択します。



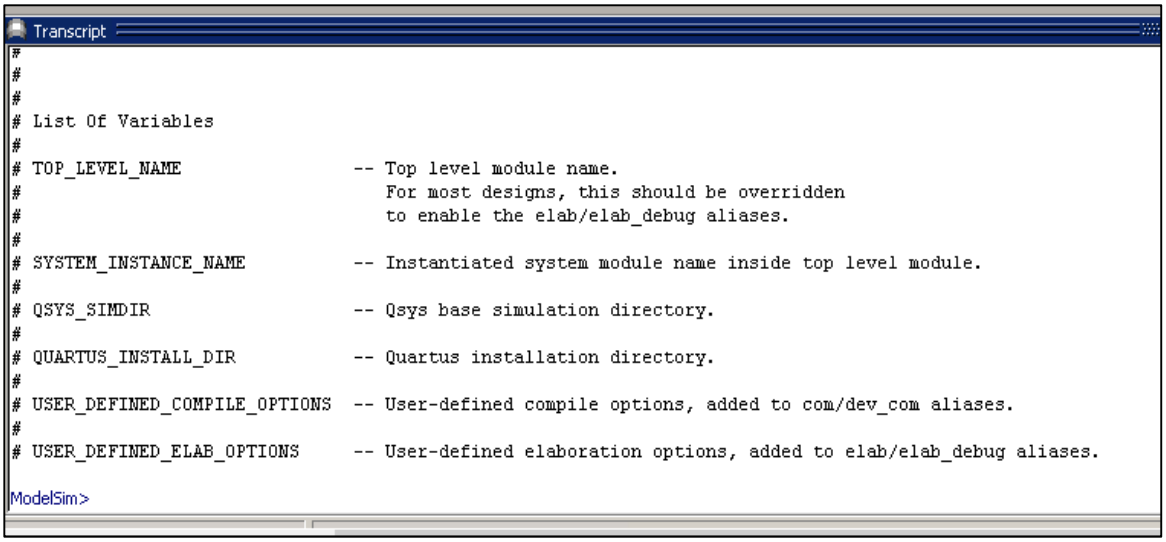
【図 6-28】Tcl スクリプトの選択

(6) msim_setup.tcl を選択して［開く(O)］ボタンをクリックします。



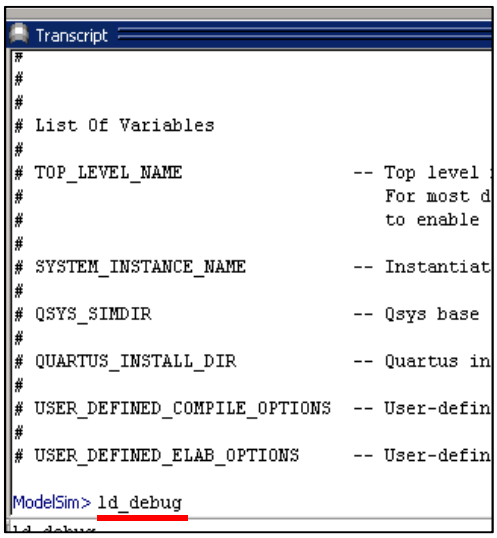
【図 6-29】Tcl スクリプトの選択【Execute Do File 画面】

(7) msim_setup.tcl が実行され、最初に下図のようなメッセージが表示されます。



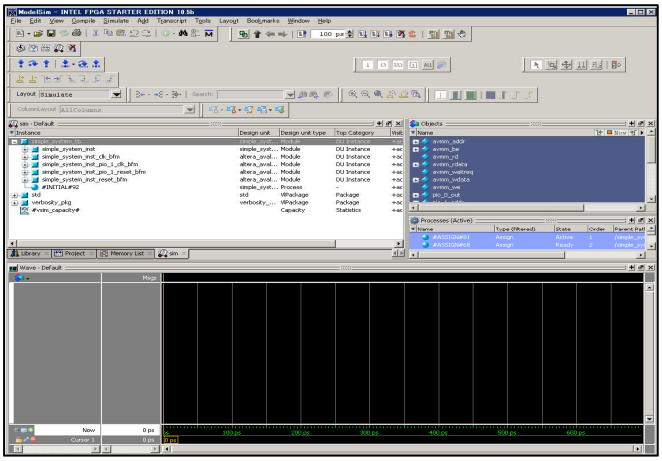
【図 6-30】ModelSim® - IE 向け設定スクリプト（本デザイン向けに生成）の実行

(8) 図 6-31 のように、ld_debug とタイプします。



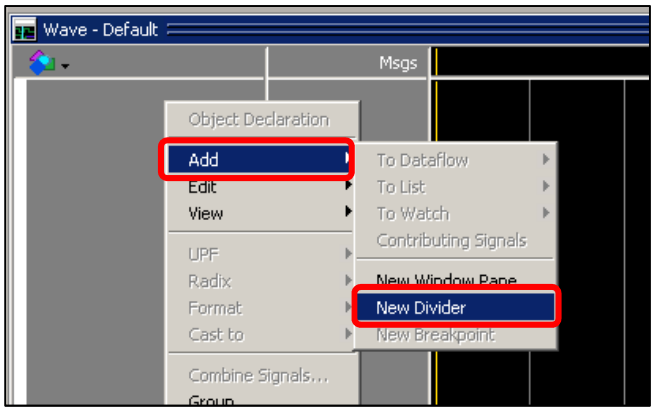
【図 6-31】ModelSim® - IE 向けの設定スクリプト提供の ld_debug コマンド

(9) 自動的に図 6-32 のような画面が表示されます。



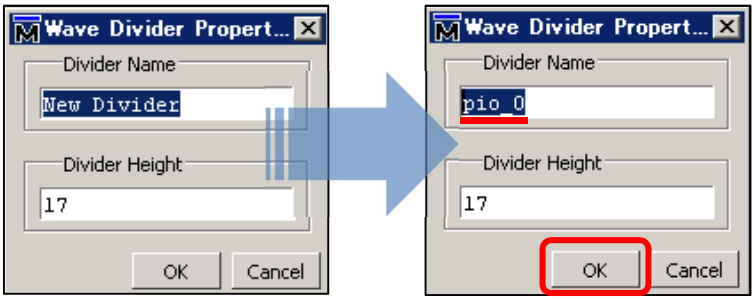
【図 6-32】Wave 画面の起動

(10) マウスカーソルを Wave 画面内に上に置いた後、右クリックから Add メニューより New Divider を選択します。



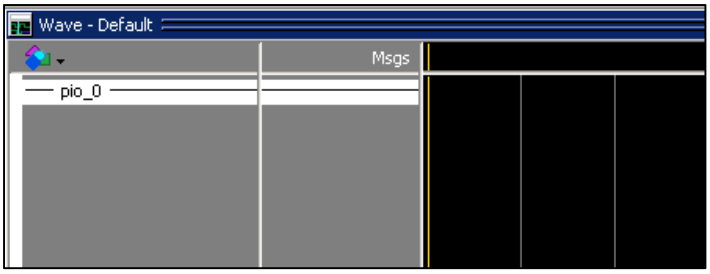
【図 6-33】New Divider の挿入

(11) Divider Name 欄を pio_0 に変更して［OK］ボタンをクリックします。



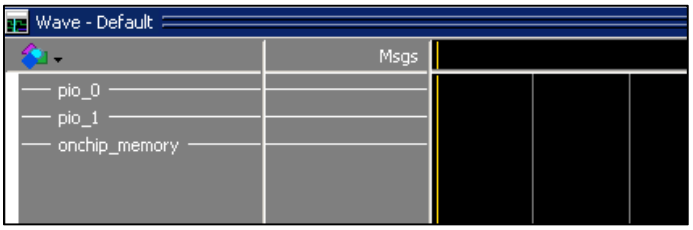
【図 6-34】Divider Name の変更

(12) Wave 画面内に区分けライン pio_0 が追加されます。



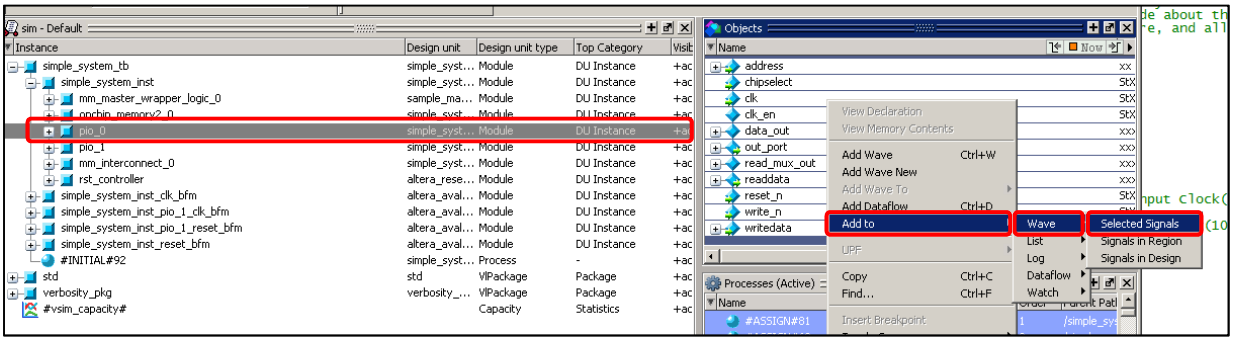
【図 6-35】New Divider の挿入

(13) 同様の操作により 図 6-36 のように、区切りライン pio_1 および onchip_memory を追加します。



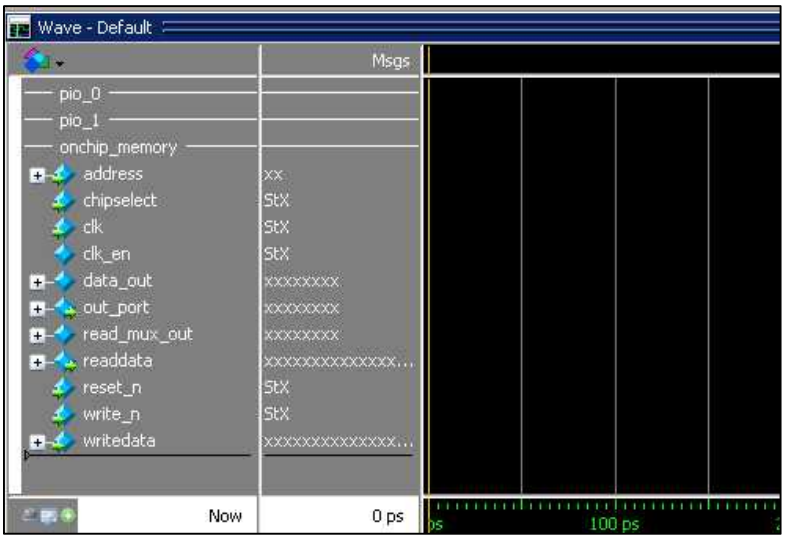
【図 6-36】Divider の追加

(14) ModelSim 画面左上の Instance 画面より、simple_system_tb/simple_system_inst ディレクトリー内にある pio_0 フォルダーをハイライト後、Objects 画面内の信号を全てハイライトして、右クリックより 図 6-37 のように Add to → Wave → Selected Signals を選択します。



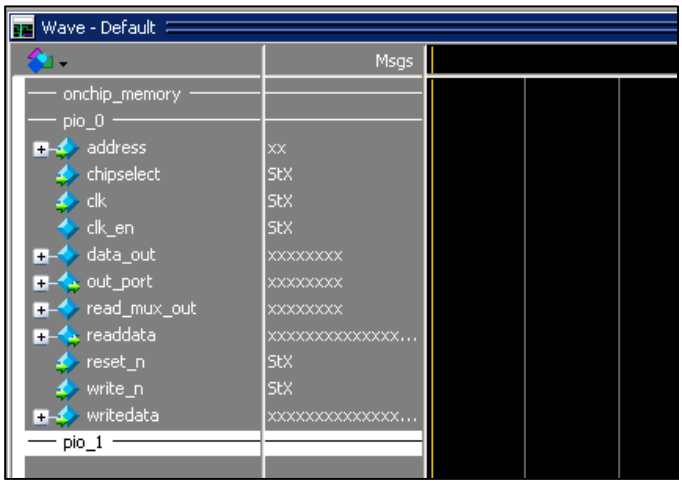
【図 6-37】Wave 画面に表示させる信号の選択（pio_0 コンポーネント）

(15) 図 6-38 のように、インスタンス pio_0 に関連した信号線が Wave 画面に追加されます。これらの信号線は図 6-39 のように区切りライン onchip_memory の下に追加されてしまう為、別途手動で修正する必要があります。



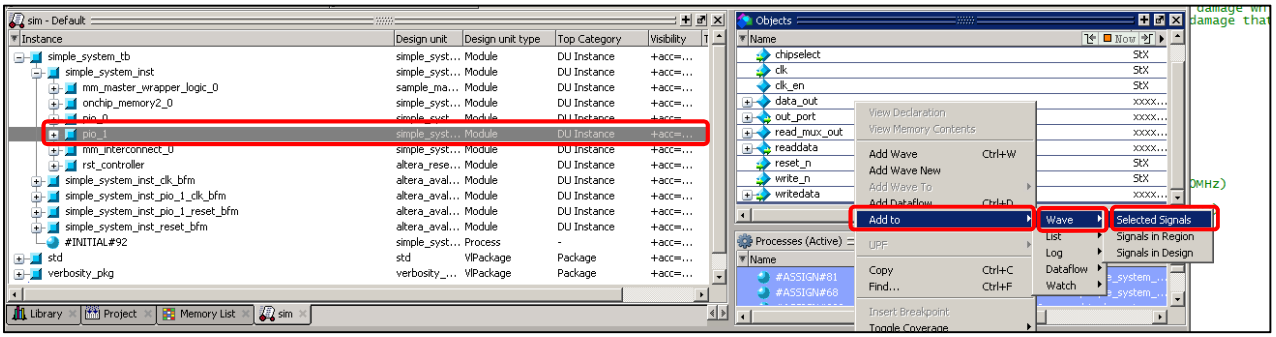
【図 6-38】選択した信号を Wave 画面に追加（pio_0 コンポーネント）

(16) 区分けライン pio_0 をハイライトして、一番上に表示されている信号線の上にマウスで移動さて修正します。



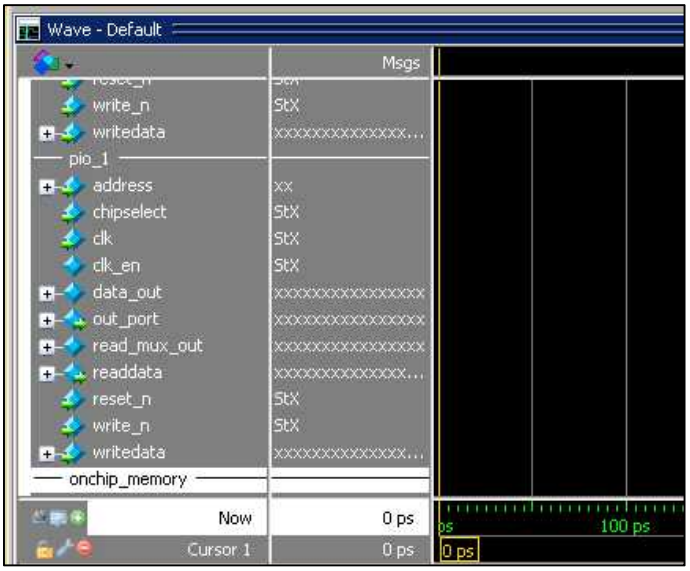
【図 6-39】追加した信号を移動（pio_0 コンポーネント）

(17) 前述と同様に、ModelSim 画面左上の Instance 画面より、simple_system_tb/simple_system_inst ディレクトリー内にある pio_1 フォルダーをハイライト後、Objects 画面内の信号を全てハイライトして、右クリックより図 6-40 のように Add to → Wave → Selected Signals を選択します。



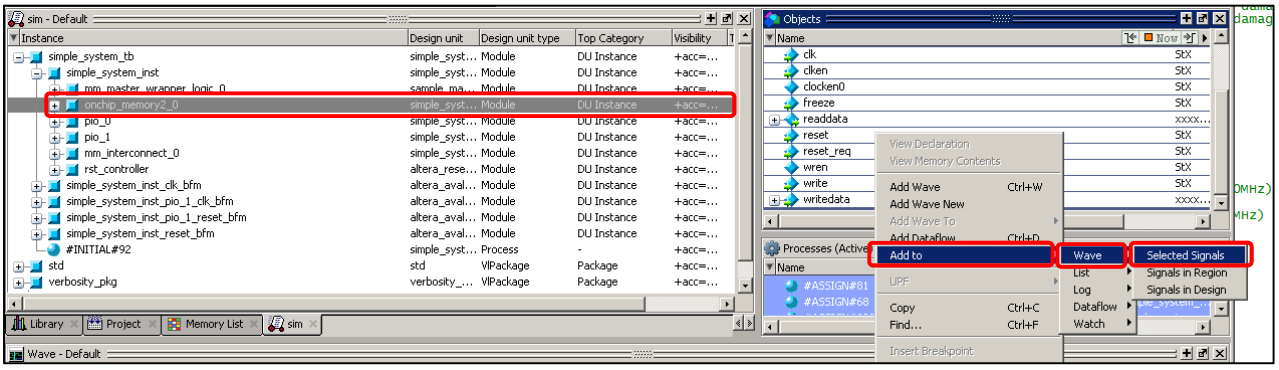
【図 6-40】Wave 画面に表示させる信号の選択（pio_1 コンポーネント）

(18) インスタンス pio_1 に関連した信号線が Wave 画面に追加された後、区切りライン pio_1 を適切な位置に移動させます。



【図 6-41】追加した信号を移動（pio_1 コンポーネント）

(19) 前述と同様に、ModelSim 画面左上の Instance 画面より、simple_system_tb/simple_system_inst ディレクトリー内にある onchip_memory2_0 フォルダーをハイライト後、Objects 画面内の信号を全てハイライトして、右クリックより 図 6-42 のように Add to → Wave → Selected Signals を選択します。

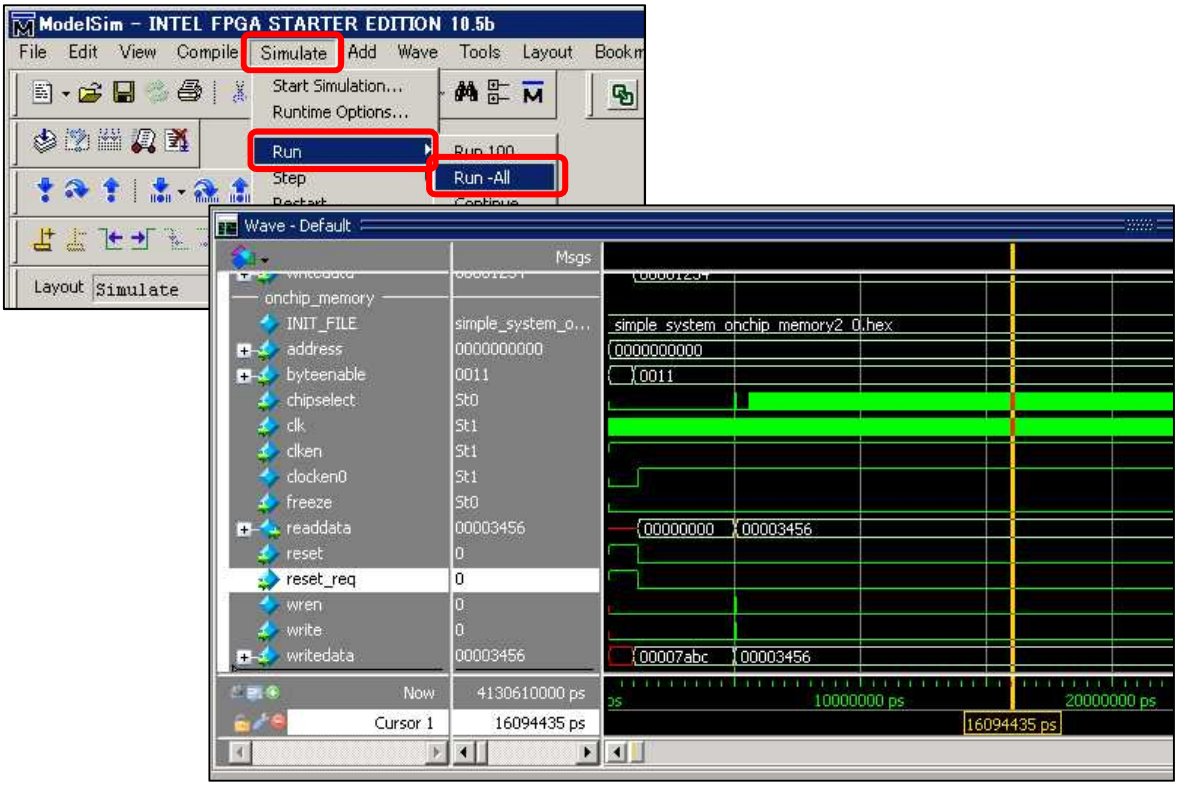


【図 6-42】Wave 画面に表示させる信号の選択（onchip_memory2_0 コンポーネント）

(20) インスタンス onchip_memory2_0 に関連した信号線が Wave 画面に追加された後、区切りライン onchip_memory を適切な位置に移動させます。

【図 6-43】追加した信号を移動（onchip_memory2_0 コンポーネント）

(21) ModelSim® - IE の Simulate メニューより、Run を選択後、Run-All を選択して RTL シミュレーションを実行します。



【図 6-44】シミュレーション波形 - 起動時を含めた On-Chip Memory のライト・オペレーション

図 6-44 では、リセット要求（画面左下の reset_req 信号）が High から Low にデアサートされた後、On-Chip Memory へのライト・オペレーションを例示しています。

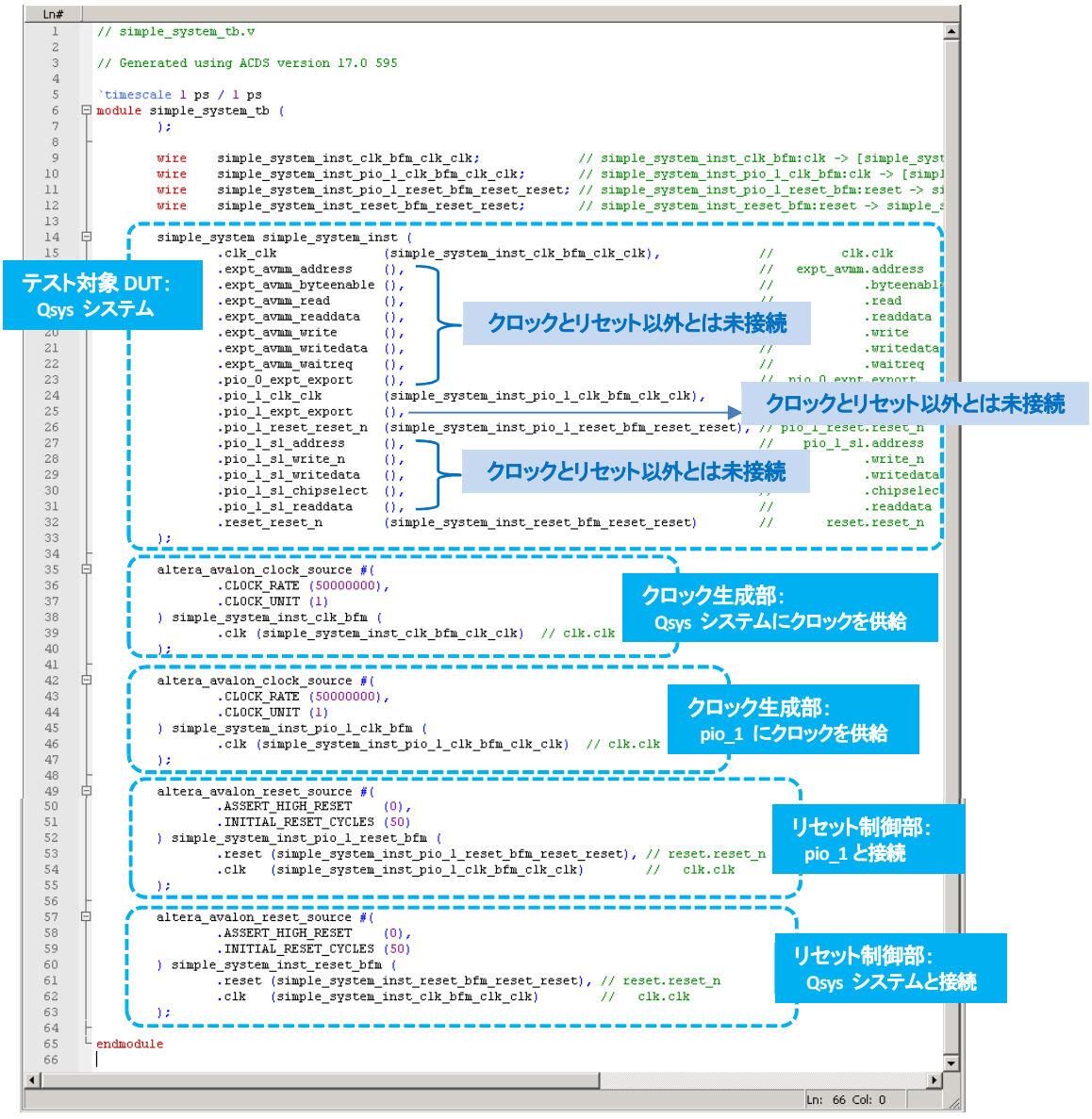
7. テストベンチ・テンプレートのカスタマイズ

付属のテンプレート・デザイン backup_folder には、カスタマイズ済のテストベンチ real_simple_system_tb.v が用意されています。以降ではカスタマイズした箇所について説明します。
最初に、ツールが生成したオリジナルのテストベンチについて説明します。

7-1. カスタマイズ前

シミュレーションにおけるテスト対象（DUT: Device Under Test）は、拡張子.qsys のファイル名を、Verilog HDL のコンポーネント名にしたインスタンスが該当します。 本資料では、simple_system.qsys のファイル名に該当する simple_system がコンポーネント名となり、simple_system_inst がインスタンス名になります。

本資料で用意したテンプレートでは、PIO_1 に対して全ての信号を Qsys の外にエクスポートしているので、クロック生成部およびリセット制御部については、Qsys システムおよび PIO_1 向けにそれぞれ用意されています。また、ツールが生成したテンプレートでは、クロックおよびリセット信号だけが DUT と接続されます。それ以外の信号については、DUT とは接続させずに未接続となっています。



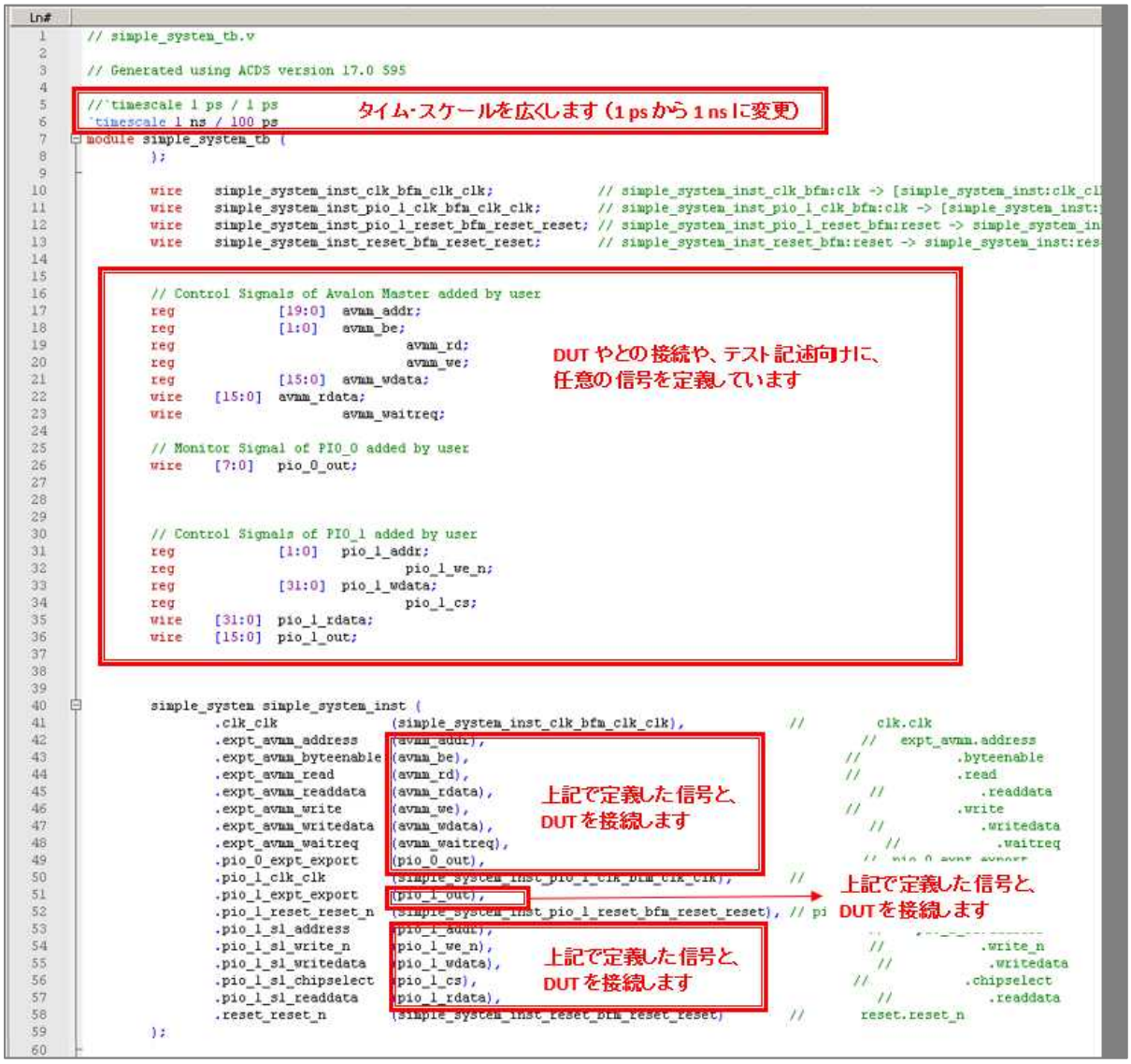
【図 7-1】オリジナルのテストベンチ（ツールが生成）

7-2. カスタマイズ後

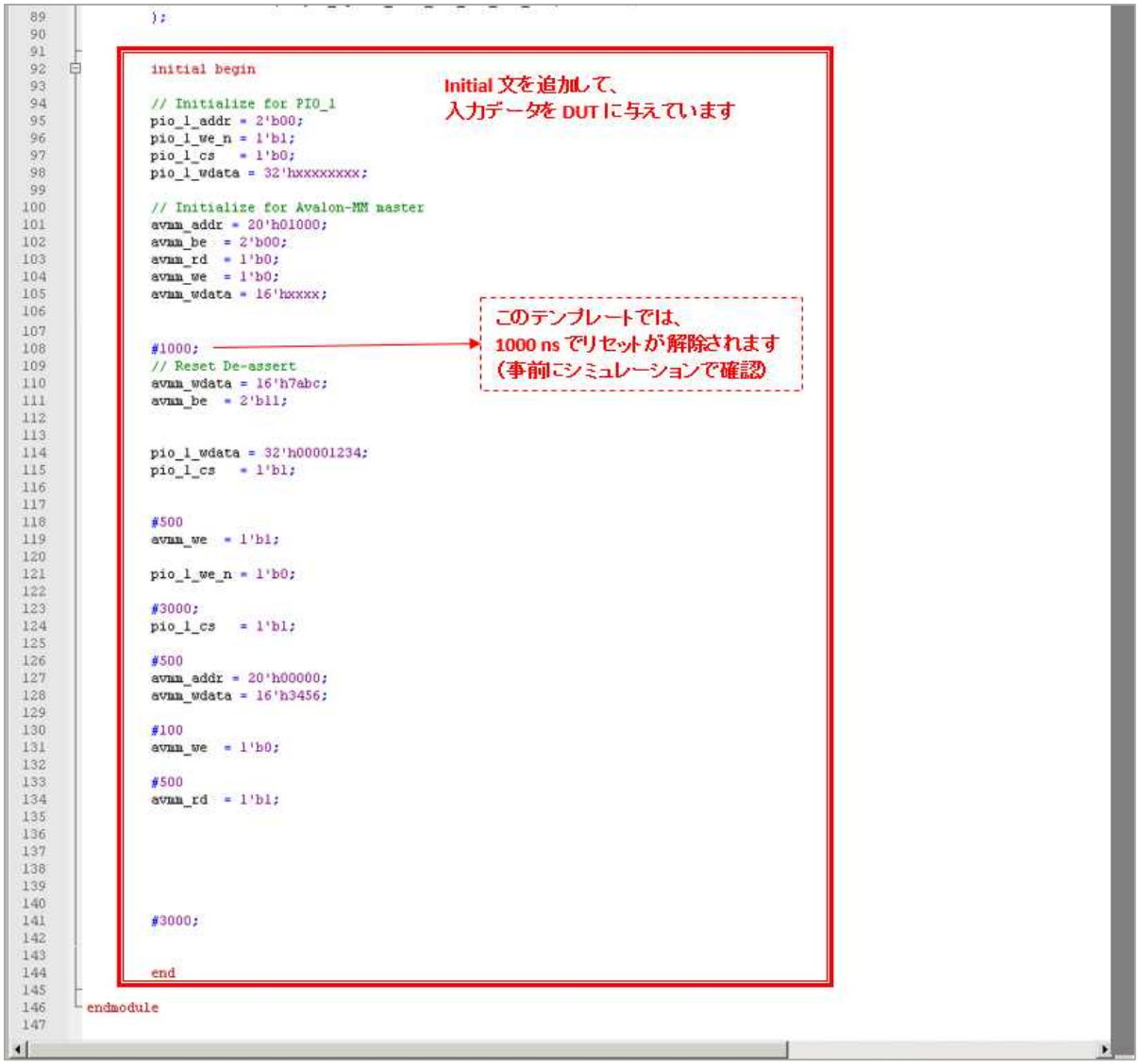
次に、カスタマイズした箇所について説明します。

図 7-2 および 図 7-3 の赤枠の箇所が追加した部分です。

タイムスケールを 1 ns 単位に変更し、テストする信号を定義して DUT と接続を行い、 Initial 文を追加して、入力データを与えています。また 1000 ns でリセットが解除される動きは、事前に把握した上で記述しています。



【図 7-2】カスタマイズ済のテストベンチ（前半部分）



【図 7-3】カスタマイズ済のテストベンチ（後半部分）

8. シミュレーション結果

8-1. コンポーネント：PIO（8 ビット出力） - pio_0

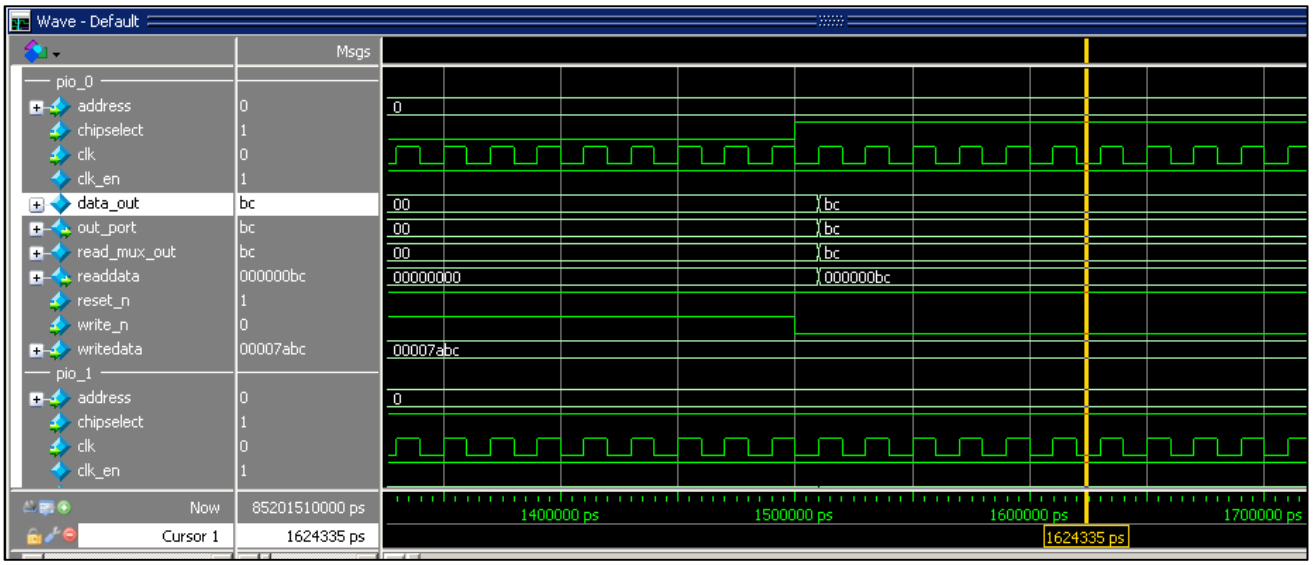
```
Ln#
 91
 92    initial begin
 93
 94    // Initialize for PIO_1
 95    pio_1_addr = 2'b00;
 96    pio_1_we_n = 1'b1;
 97    pio_1_cs   = 1'b0;
 98    pio_1_wdata = 32'hxxxxxxxx;
 99
100    // Initialize for Avalon-MM master
101    avmm_addr = 20'h01000;
102    avmm_be   = 2'b00;
103    avmm_rd   = 1'b0;
104    avmm_we   = 1'b0;
105    avmm_wdata = 16'hxxxx;
106
107
108    #1000;
109    // Reset De-assert
110    avmm_wdata = 16'h7abc;
111    avmm_be   = 2'b11;
112
113
114    pio_1_wdata = 32'h00001234;
115    pio_1_cs   = 1'b1;
116
117
118    #500
119    avmm_we   = 1'b1;
120
121    pio_1_we_n = 1'b0;
122
123    #3000;
124    pio_1_cs   = 1'b1;
125
126    #500
127    avmm_addr = 20'h00000;
128    avmm_wdata = 16'h3456;
129
130    #100
131    avmm_we   = 1'b0;
132
133    #500
134    avmm_rd   = 1'b1;
135
                                Ln: 147 Col: 0
```

【図 8-1】テストベンチ記述 - pio_0 を制御する部分をハイライト（白色）

図 8-1 のテストベンチ記述 101 行目では 20 ビットのアドレス線で構成されているカスタム・コンポーネントに対して 0x01000 番地のアドレスを与えています。このアドレスは、表 5-1 で定義されている pio_0 のベースアドレスに該当します。

また、110 行目では 16 ビットのデータ幅を持つカスタム・コンポーネントに対して 16 ビット・データ 0x7abc を与えています。

シミュレーション波形



【図 8-2】シミュレーション波形 - pio_0 のデータ出力

図 8-2 では、pio_0 のデータレジスタ（オフセット 0x0）に、データ 0xbc をライトしている動作を例示しています。

図 8-1 のテストベンチでは、16 ビットのデータ幅を持つカスタム・コンポーネントに 16 ビット・データ 0x7abc を与えていますが、8 ビットのデータ幅を持つ pio_0 では、下位 8 ビットの 0xbc だけが出力している動きが、図 8-2 左側でハイライトした data_out の値が中央付近で 0xbc に変化していることから把握できます。

8-2. コンポーネント：PIO（16 ビット出力） - pio_1

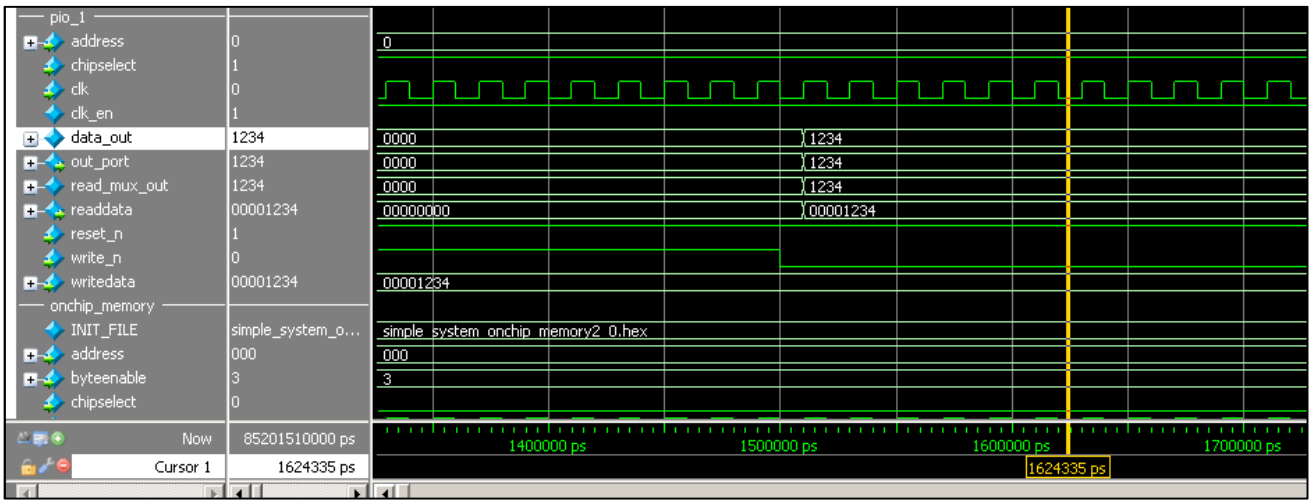
```
Ln#
 91
 92    initial begin
 93
 94    // Initialize for PIO_1
 95    pio_1_addr = 2'b00;
 96    pio_1_we_n = 1'b1;
 97    pio_1_cs   = 1'b0;
 98    pio_1_wdata = 32'hxxxxxxxx;
 99
100    // Initialize for Avalon-MM master
101    avmm_addr = 20'h01000;
102    avmm_be   = 2'b00;
103    avmm_rd   = 1'b0;
104    avmm_we   = 1'b0;
105    avmm_wdata = 16'hxxxx;
106
107
108    #1000;
109    // Reset De-assert
110    avmm_wdata = 16'h7abc;
111    avmm_be   = 2'b11;
112
113
114    pio_1_wdata = 32'h00001234;
115    pio_1_cs   = 1'b1;
116
117
118    #500
119    avmm_we   = 1'b1;
120
121    pio_1_we_n = 1'b0;
122
123    #3000;
124    pio_1_cs   = 1'b1;
125
126    #500
127    avmm_addr = 20'h00000;
128    avmm_wdata = 16'h3456;
129
130    #100
131    avmm_we   = 1'b0;
132
133    #500
134    avmm_rd   = 1'b1;
135
                                Ln: 147 Col: 0
```

【図 8-3】テストベンチ記述 - pio_1 を制御する部分をハイライト（白色）

図 8-3 のテストベンチ記述 95 行目では、カスタム・コンポーネントを介さずに、直接 pio_1 のアドレス線に 0x0 番地を指定しています。Avalon-MM インターフェイスを直接テストベンチまで引き出しているので、表 5-1 のアドレスマップにはマッピングされません。

114 行目で、pio_1 の Avalon-MM インターフェイスで定義されている 32 ビット幅のデータ線に、値 0x00001234 を書き込んでいます。

シミュレーション波形



【図 8-4】シミュレーション波形 - pio_1 のデータ出力

図 8-4 では、pio_1 のデータレジスタ（オフセット 0x0）に、データ 0x1234 をライトしている動作を例示しています。

図 8-3 のテストベンチでは、pio_1 の Avalon-MM インターフェイスに 32 ビットのデータ 0x00001234 をライトしていますが、16 ビットのデータ幅を持つ pio_1 では、下位 16 ビットの 0x1234 だけが出力している動きが、図 8-4 左側でハイライトした data_out の値が中央付近で 1234 に変化していることから把握できます。

8-3. コンポーネント：On-Chip Memory - onchip_memory2_0

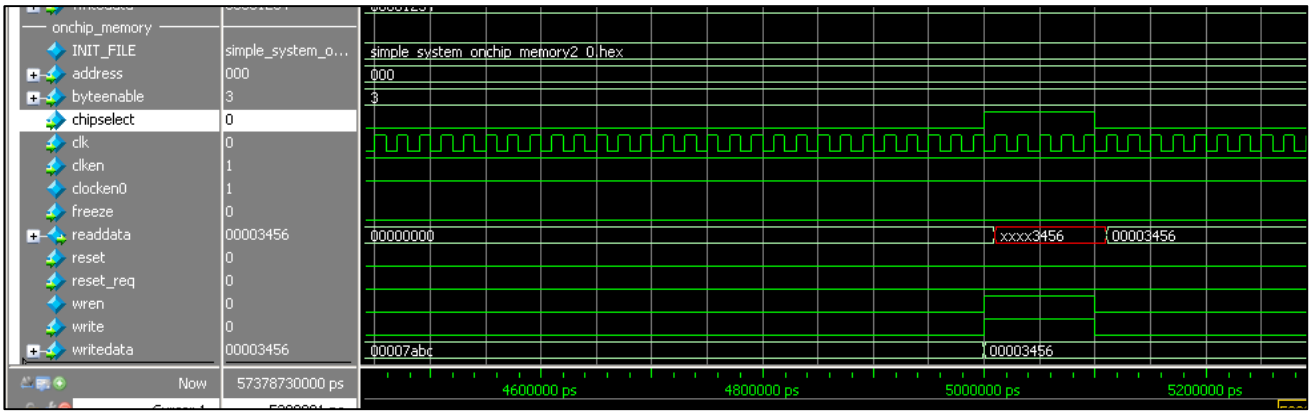
```
Ln#
 91
 92    initial begin
 93
 94    // Initialize for PIO_1
 95    pio_1_addr = 2'b00;
 96    pio_1_we_n = 1'b1;
 97    pio_1_cs   = 1'b0;
 98    pio_1_wdata = 32'hxxxxxxxx;
 99
100    // Initialize for Avalon-MM master
101    avmm_addr = 20'h01000;
102    avmm_be   = 2'b00;
103    avmm_rd   = 1'b0;
104    avmm_we   = 1'b0;
105    avmm_wdata = 16'hxxxx;
106
107
108    #1000;
109    // Reset De-assert
110    avmm_wdata = 16'h7abc;
111    avmm_be   = 2'b11;
112
113
114    pio_1_wdata = 32'h00001234;
115    pio_1_cs   = 1'b1;
116
117
118    #500
119    avmm_we   = 1'b1;
120
121    pio_1_we_n = 1'b0;
122
123    #3000;
124    pio_1_cs   = 1'b1;
125
126    #500
127    avmm_addr = 20'h00000;
128    avmm_wdata = 16'h3456;
129
130    #100
131    avmm_we   = 1'b0;
132
133    #500
134    avmm_rd   = 1'b1;
135
                                Ln: 147 Col: 0
```

【図 8-5】テストベンチ記述 - On-Chip_memory を制御する部分をハイライト（白色）

図 8-5 のテストベンチ記述 127 行目では 20 ビットのアドレス線で構成されているカスタム・コンポーネントに対して 0x00000 番地のアドレスを与えています。
このアドレスは、表 5-1 で定義されている onchip_memory2_0 のベースアドレスに該当します。

また、128 行目では 16 ビットのデータ幅を持つカスタム・コンポーネントに対して 16 ビット・データ 0x3456 を与えています。

シミュレーション波形



【図 8-6】シミュレーション波形 - On-Chip Memory のライト・オペレーション

図 8-6 では onchip_memory2_0 の 0x000 番地に、データ 0x00003456 をライトしている動作を例示しています。メモリー容量 が 4,096 バイトで設定（図 5-6 参照）されている為、onchip_memory2_0 のアドレスは 12 ビットで表現されます。

図 8-5 のテストベンチでは、16 ビットのデータ幅を持つカスタム・コンポーネントに 16 ビット・データ 0x3456 をライトしていますが、32 ビットのデータ幅を持つ onchip_memory2_0 では、16 ビットから 32 ビットに拡張された値 0x00001234 がライトしている動きが、図 8-6 左側下でハイライトした writedata の値が中央付近で 00003456 に変化していることから把握できます。

改版履歴

Revision	年月	概要
1	2019 年 11 月	初版