

チュートリアル6 :WaveFormer を使ったシステム・モデリングとVerilog シミュレーション

本チュートリアルは、WaveFormer Pro を使って素早くシステムをモデリングしシミュレーションできることを、ステート・マシンを含む中程度の規模のデジタル・システムを例に示します。

1) 例題

12 ビット A/D コンバータ(ADC)によってサンプルされた 64K(64 × 1024)サンプルのデータに対して、ヒストグラムを計算する回路をモデリングする。

この回路は以前に伝統的な EDA ツールを使ってモデリングし、シミュレーションを行うのに、数ヶ月を要した、実際のシンプルな VME ボードの例です。しかし、WaveFormer Pro を使うことで、きわめて短時間のうちにモデリングとシミュレーションを完了できます。また、VME バス・インターフェース・プロトコルを扱うフル機能の回路モデリングおよびシミュレーションもすばやく行えます。

2) 本チュートリアルで解説する内容

- 論理式インターフェースを使ったステート・マシンのモデリング
- テンポラル / ラベル式を使った入力信号の生成
- シミュレーション・ログを使ってデザインの入力ミスを発見
- インクリメンタル・シミュレーション
- Verilog-HDL コードの直接入力によるモデリング
- 外部 Verilog-HDL ソース・コード・モデルの利用
- 条件命令を使った 3 ステート・ゲートのモデリング
- リダクション・オペレータを使った n ビット・ゲートのモデリング
- トランスペアレント・ラッチのモデリング方法
- \$display を使った Verilog-HDL ソース・コードのデバッグ
- Time Maker を使ったシミュレーション時間のコントロール
- Wave Former の Report ウィンドウを使った外部 Verilog-HDL ファイルの編集

チュートリアルを始める前に、これから作成する WaveFormer Pro のダイアグラムの完成版(tutsim.tim)を先に見ておくとういでしょう。しかし、このファイルを本チュートリアルで直接に使うのではなく、このチュートリアル実行の途上で必要に応じて参照してください。

3) 回路仕様

3.1) メモリ容量

このヒストグラムは、ADC から受け取った 12 ビット長の異なる各値が何回出現したかをグラフで表現するものです。サンプルした値をヒストグラムとして出力するにあたり、12 ビットの各値(0 ~ $2^{12}-1$)をもれなくカバーするのに 4K の SRAM($2^{12}=4096$)が必要です。また、SRAM のワード幅は対象とするデータの出現頻度に依存しますが、最悪の場合、一つの値に 64K 個のデータが集中することを想定し、 $64K=2^{16}$ により、8 ビット幅の 4K SRAM を 2 個使うことで(8 ビット × 2 = 16 ビット)、どのような場合でもヒストグラムはメモリ上に実現されます。

3.2) 回路の動作

回路が起動したとき、SRAM は初期値として各アドレスにおいてゼロ・データを保持するものとします。

ADC からのサンプル・データは SRAM のアドレスとして使われ、そのアドレスに対するメモリ・セルの保持するカウント値をインクリメントすることによりヒストグラムを形成します。この動作を 64K 個のデータを受け取るまで繰り返します。

3.3) 試験システムの構成
システムを、

デジタル・システム = データ・パス + コントロール

の考えに基づいて分析し、次の仕様(図)を得ます。

- * 図の右半分は「データ・パス部」で機能ブロック間で処理されるデータの流れを示しています。
- * 左半分は「コントロール部」を表し、ステート・マシンとなっています。状態信号を「データ・パス部」の一つであるカウンタより得て、制御信号を「データ・パス部」に与えています。

3.4) コントロール部の構成

回路を制御するために、ワンホット・ステート・マシンを使います。ワンホット・ステート・マシンは、各状態に一つのフリップフロップを割り付けます。したがって現在の状態を表すフリップフロップだけが1を保持し、他のステート・フリップフロップは0になります(名前 one-hot はここに由来する)。ワンホット・ステート・マシンは FPGA のアーキテクチャによくマッチするので、FPGA ベースの設計ではとてもポピュラーなもので、伝統的なバイナリ・エンコード・ステート・マシンにくらべて比較的高速に動作し、少ないリソースで実現できます。

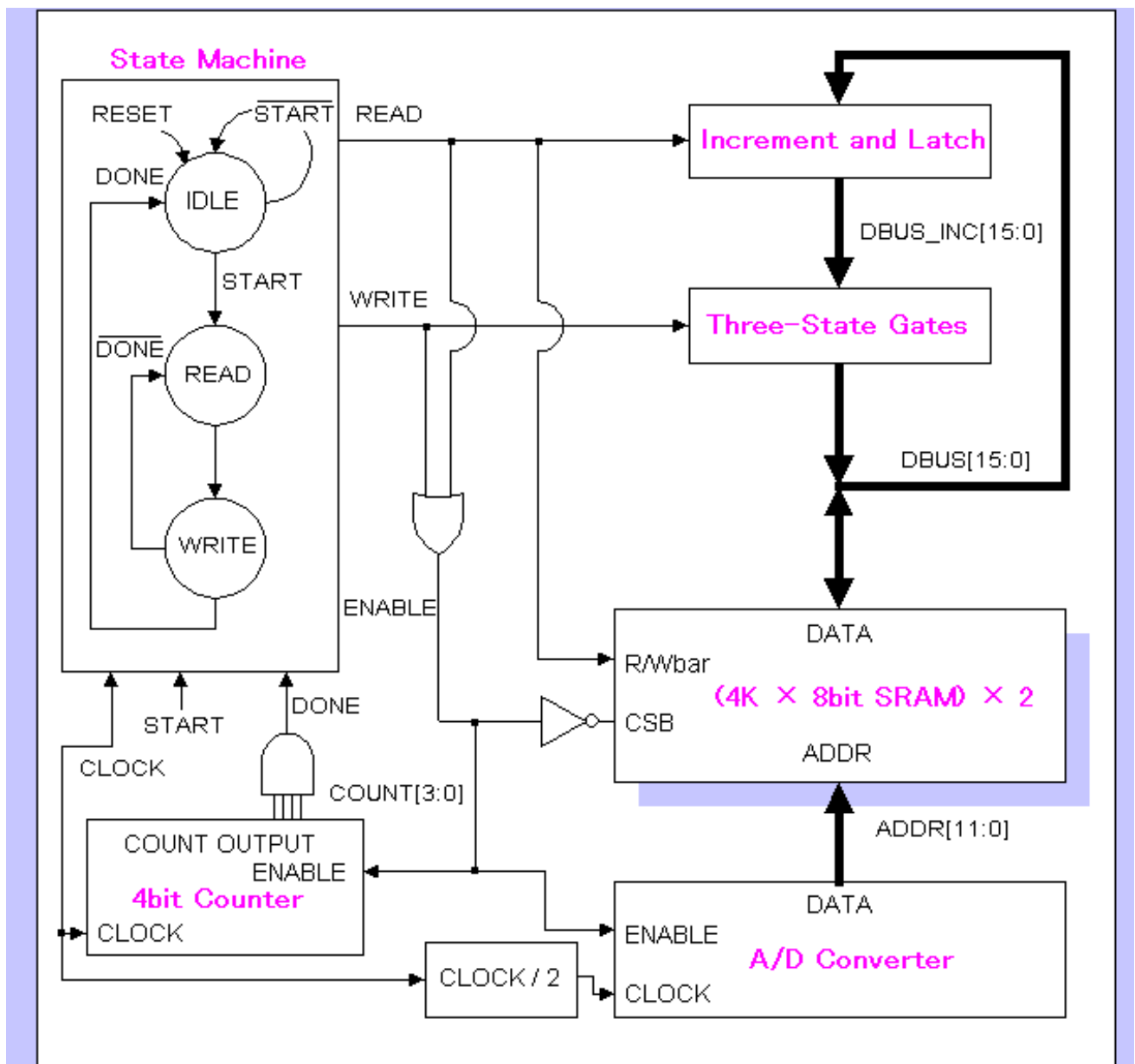


図1 システム構成図

3.5) システムの動作

ステート・マシン(SM)は最初 IDLE ステートに初期化されます。START が High になった後、クロックの立ち上がりエッジで SM は READ ステートに遷移し、ADC から出力された SRAM アドレス値に対するカウントをインクリメントしたのち、トランスペアレント・ラッチ DBUS_INC に保持します。次のクロックで、SM は WRITE ステートに遷移し、トランスペアレント・ラッチ DBUS_INC に保持された更新値を SRAM に書き戻します。

このステート・マシンは READ と WRITE ステート間を、決められた回数(バイナリ・カウンタ COUNT のサイズによって決められる)だけ、交互に繰り返します。そして既定値に到達すると、SM は IDLE ステートに戻ります。

4) コンポーネントのモデリングとシミュレーション

4.1) 論理式インターフェースを使ったステート・マシンのモデリング

ヒストグラムの回路をモデリングするための新しいダイアグラムの作成

- [File]-[New]メニューを選択し、新しいダイアグラムを作ります。
- このチュートリアルでは使わない Parameter ウィンドウを最小化しておきます。
- [Window]-[Tile Horizontally]を選択し、ダイアグラムと Report ウィンドウを見やすくします。

システム・クロックのモデリング

- [Add Clock] ボタンをクリックして、CLK0 という名前で信号を追加します。
- [OK]をクリックして、デフォルトの設定で CLK0 を登録します。

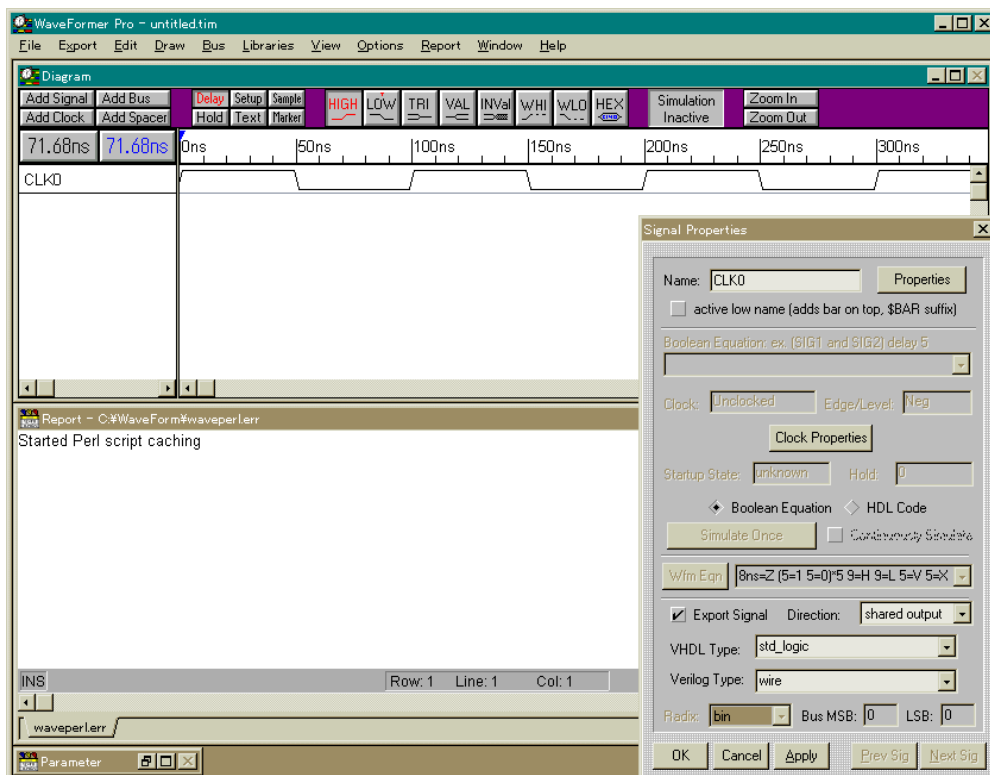


図2 システム・クロック信号の作成

START と ADDR 信号のモデリング

本システムは、以下の2つの入力信号を持っています。

- (1)処理を始める START
- (2)ADC の出力データで、SRAM のアドレス・ラインに供給される ADDR

次の2つのセクションでは、この入力信号の波形データの作成方法について説明します。START 信号の波形は比較的シンプルなのでマウス操作だけで作成できます。ADDR 信号の波形はより複雑で、人手で入力するのはやや大変です。そこで、テンポラル式やステート・ラベル式を使ってこの信号を作成します。

START 信号のモデル化 - グラフィカルに入力信号を定義する

-[Add Signal]ボタンをクリックします。

-作成された SIG0 をダブル・クリックして、Signal Properties Dialog を表示させます。このダイアログを使って、デフォルトの設定を変更します。Name の項目に START と入力します。

-[OK]ボタンをクリックして、現在の値を設定します。

Tips:

START 信号の設定を変更したときに現れた Signal Properties Dialog ウィンドウは、このチュートリアルを行っている最中、開いたままにしておくことができます ([Apply]ボタンをクリックして結果を更新できる)。他の信号の設定を変更したいときには、ダイアグラム・ウィンドウの信号名をダブルクリックします。

-START 信号として 60[ns]程度の Low 信号をはじめに描きます。Low 信号を描くには、ダイアグラム・ウィンドウ内で[Low]ボタンをクリックして、START の名前のある項目の右 60[ns]のあたりを左クリックします。

-100[ns]程度の High 信号を START に継ぎ足します。[HIGH]が押されていることを確認して、ダイアグラム・ウィンドウ内で、START の名前のある項目の右 160[ns]のあたり(はじめの Low 信号の終わりから 100[ns])を左クリックします。

-CLK0 の最初のネガティブ・エッジのときに START 信号が Low、次のネガティブ・エッジのときに High であるかを確認してください。そのようになっていけば、ステート・マシンは正常に動作します。もしそうでない場合には、入力波形をチェックし、波形を描き直します。

-最後に 800[ns]程度の Low 信号を継ぎ足します。

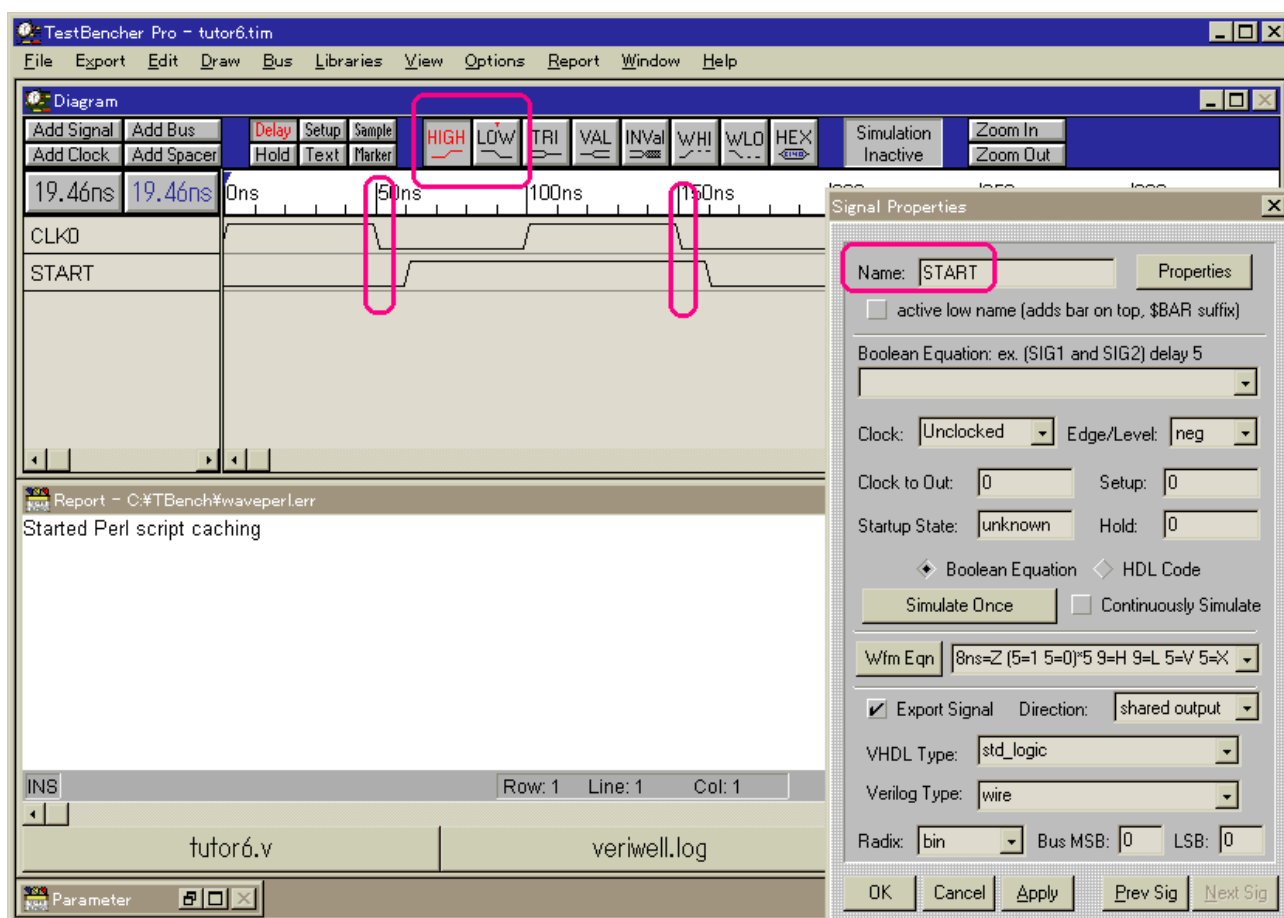


図3 START 信号の作成

波形とラベル式を使った入力信号の生成 - A/D コンバータ・データのモデリング

SRAM のアドレス・ラインを駆動する信号 ADDR (ADC からの出力) を仮想バス信号として扱えば十分なので、A/D コンバータを単なるデータ・ソースとみなしてモデリングします。

ADDR 信号の生成

- [Add Signal]ボタンをクリックし、ADDR 信号を作ります。ここで、[Add Bus]ボタンをクリックしないでください。仮想バス信号で十分です、バスの要素信号は必要ありません。
- 信号名をダブル・クリックして、Signal Properties Dialog ウィンドウを開きます。
- 信号の MSB を 11 にセットして、Radix を hex に設定します。

ADC はステート・マシンのクロック CLK0 の半分の周波数で動作します。それにより、ADDR の値は、一つおきのクロック・サイクルで変化します。以下の波形方程式を ADDR 信号の内容として使います。

$$170=X(200=V)*20$$

上の式は 170[ns]間不定、つぎに 200[ns]間有効値を 20 回繰り返すことを意味します。

- ADDR の Signal Properties Dialog ウィンドウを開きます。
- [Wfm Eqn]ボタンの右隣の項目に上記の波形方程式を入力します。
- [Wfm Eqn]をクリックすると、波形方程式に対応する波形が自動生成されます。
- [OK]をクリックして、現在の値を保存します。

次に、ラベル式を使って、ADDR バスの各有効値に値(ラベル)を割り当てます。次の式が、ADDR 信号に適用されるステート・ラベル方程式です。

$$\text{ADDR:1 Rep}((0,1,2,3,4),4)$$

上の式は 0 から 4 までの 16 進数をカウントし、それを 4 回繰り返すことを意味します。":1"は、はじめの不定ステートの後、ラベル付けを開始することを意味します。

- [Export]-[Label Waveform Equation]を選択します。
- ステート・ラベル方程式を入力します。
- [OK]をクリックして、現在の値を保存します。

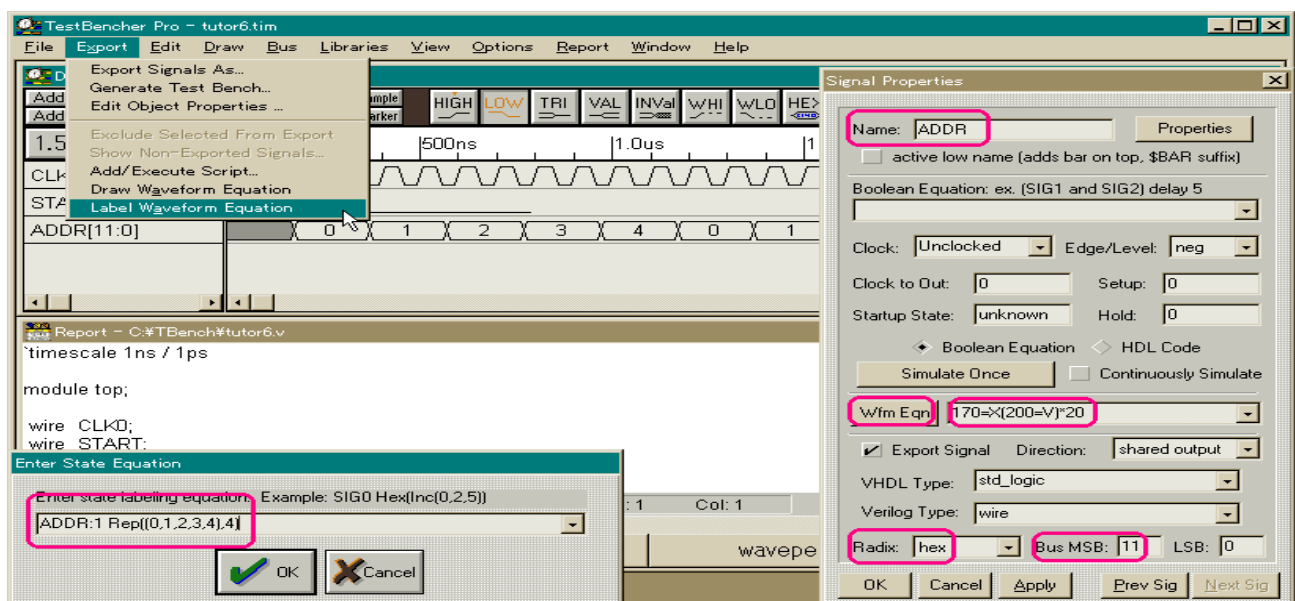


図4 ADDR 信号の作成

フリップフロップの式を使ったステート・マシンのモデリングとシミュレーション

ステート・マシンは WaveFormer Pro 上では, Signal Properties Dialog の Boolean Equation 項目使ってモデリングできます. IDLE, READ, WRITE の 3 つの信号を作成します.

作成した 3 つの信号各々について Signal Properties Dialog で以下のステップを実行します.

-ステート・マシンの式を信号に対して Boolean Equation の項目に入力します.

```

IDLE  (WRITE & DONE) | (~ START & IDLE)
READ  (IDLE & START) | (WRITE & DONE)
WRITE  READ
    
```

-3 つの信号それぞれについて, CLK0 をダイアログ内の Clock のドロップダウン・リスト・ボックスから選択します. これで, フリップフロップが CLK0 の立ち上がりエッジに同期して動作するようになります(Edge/Level のデフォルト値は neg).

-各フリップフロップのパワー・アップ時の値(デフォルト"Unknown")を編集します. まず, POWER 信号を追加し(RESET 信号に相当), 最初の 80[ns]間を Low とし, その後 2[us]までを High とします.

つぎに, 信号 IDLE の Signal Properties Dialog を開き, 以下のように設定します.

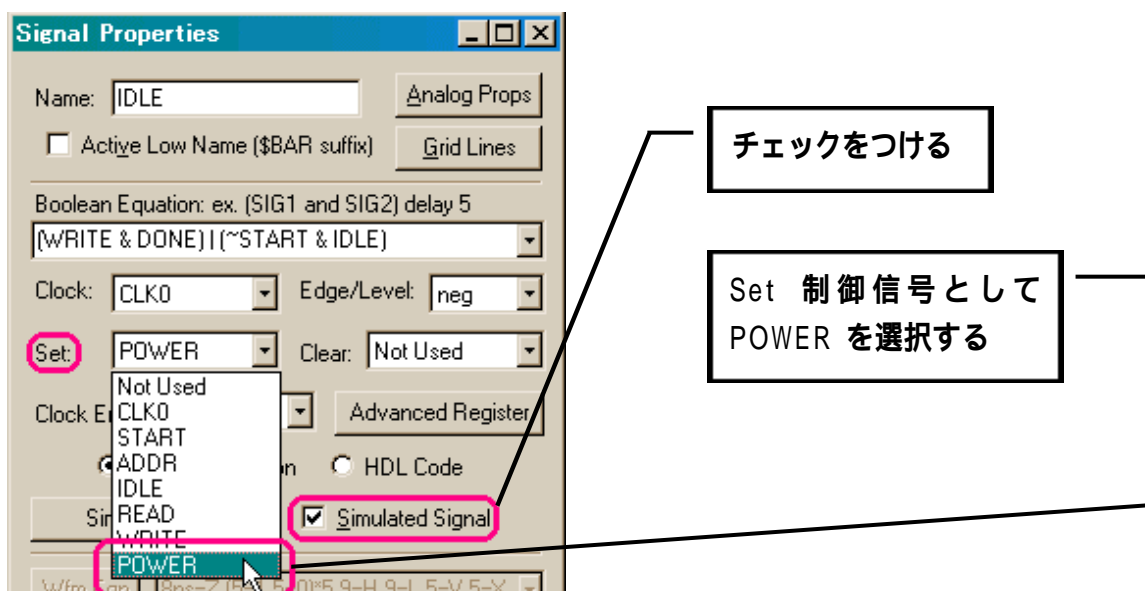


図5 IDLE 信号のパワー・アップ時の値の設定例

これで, POWER 信号が Low の間, IDLE 信号は High(1) にセットされます(IDLE = 1 * b1).

なお, WaveFormer Pro では, 指定された信号の値を元に, ロー・アクティブ/非同期でレジスタの値を High にセット, または Low にリセットするよう初期設定されています. また, この初期設定は変更が可能です.

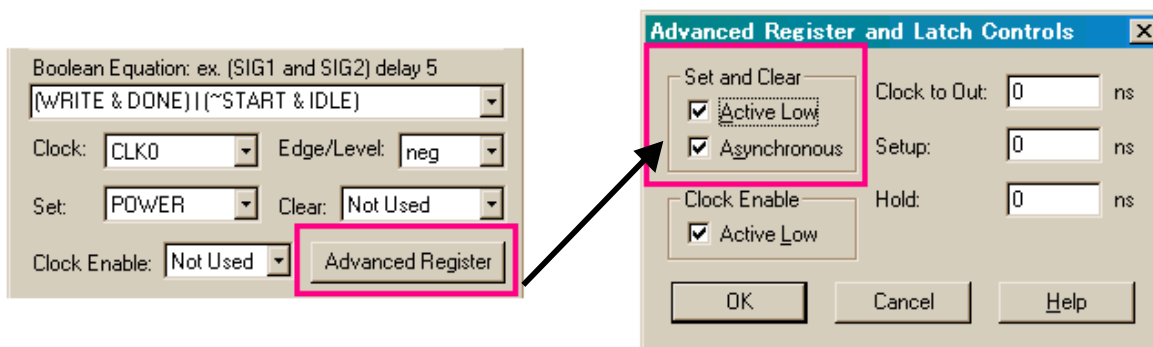


図6 レジスタ/ラッチのセット/リセット動作属性の設定

同様に，READ 1'b0,WRITE 1'b0 を設定します．これは以下に示す図にしたがって操作すれば設定できます．なお，どちらの信号についても Simulated Signal の項目にチェックを付けます．

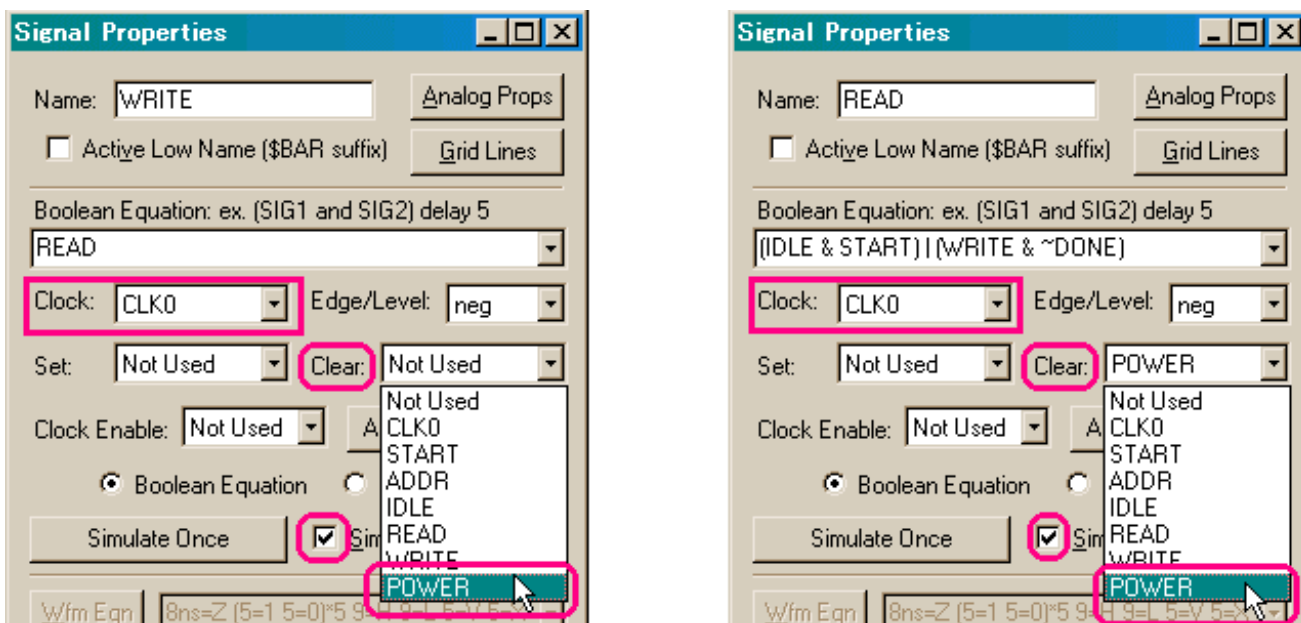


図7 WRITE 信号とREAD 信号のパワー・アップ時の値の設定例(0クリア動作の設定)

-以上の設定を行っても，シミュレーション結果が波形として現れません．これは，まだ定義していないDONE 信号を IDLE とREAD の式が参照しているためです．

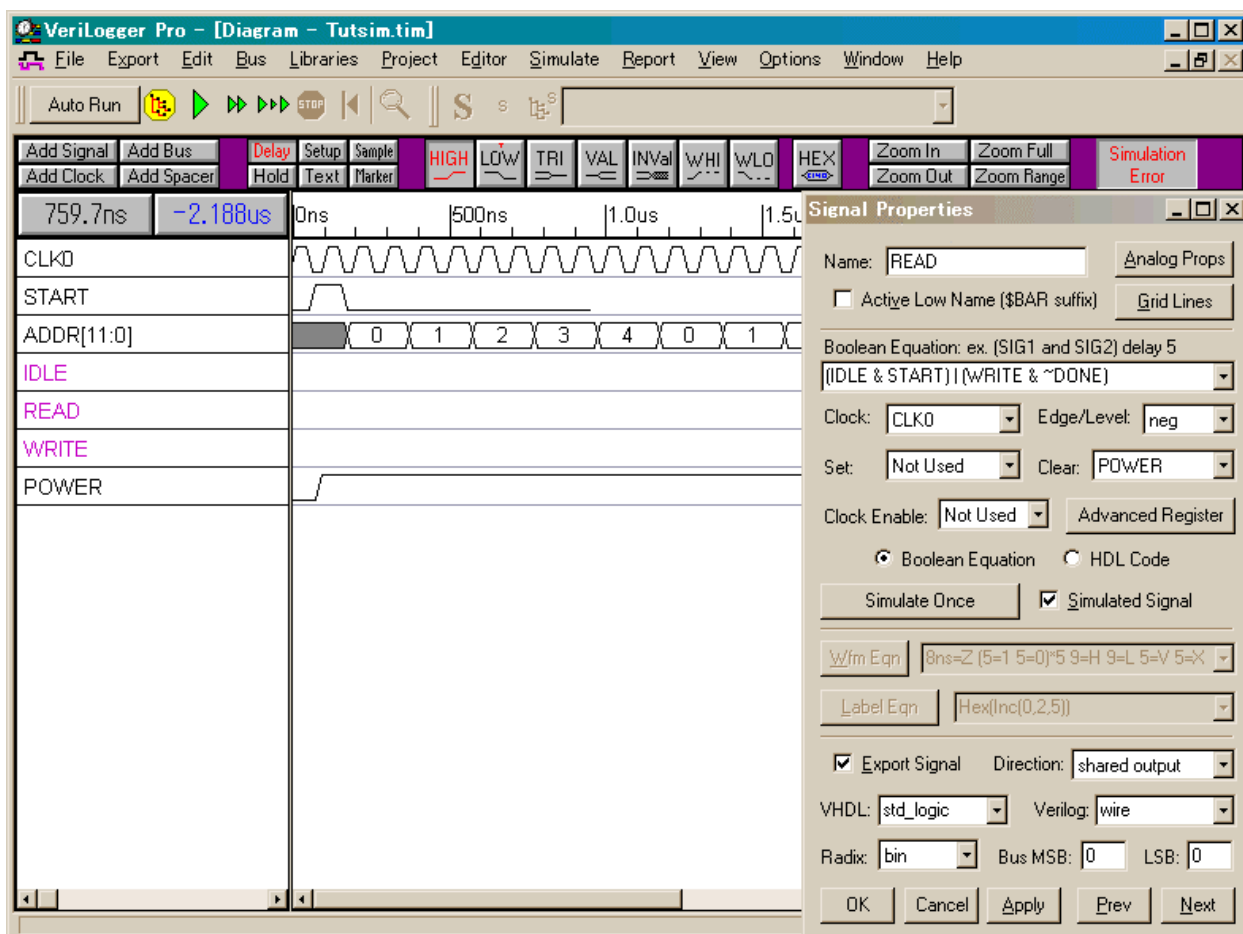


図8 IDLE ,READ ,WRITE の作成とシミュレーション結果

シミュレーション結果の検証 – Waveperl.log のレポートを見る

Waveperl.log ファイルには、作成されたフリップフロップを Verilog-HDL コードに変換する Perl スクリプトにより生成されたステータス・レポートが記述されています。このファイルには、シンタックス・エラーや、設計式中の不明信号の情報について記述されています。

-Report ウィンドウの下にある[Waveperl.log] ボタンをクリックすると、このファイルを見ることができます。IDLE または READ 信号について、Signal Properties Dialog 内の Simulate Once ボタンをクリックしてシミュレーションを行うと、このファイル中に“Unknown signal name DONE”という記述が現れます。

-IDLE, READ, WRITE の各信号については、Simulated Signal の項目がチェックされているので、ダイアグラム中の任意の信号を変更するたびにシミュレーションが行われます。これで DONE 信号を定義すると、その結果をすぐにダイアグラム・ウィンドウで見ることができます。

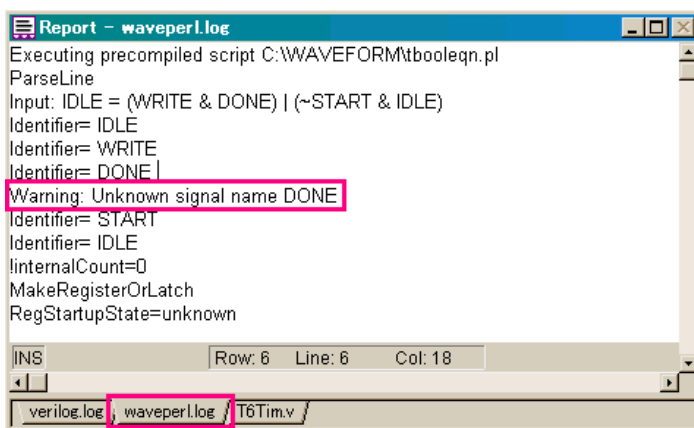


図9 Waveperl.log ファイルの内容

シミュレーション結果の検証 - シミュレーション・ログ・ファイルを見る

Report ウィンドウ内のシミュレーション・ログ・ファイル(verilog.log) 中にも、DONE 信号が定義されていないというレポート・メッセージを見ることができます。また、このログ・ファイルは WaveFormer が自動生成した Verilog-HDL ソースの対応するエラー行もレポートします。Verilog-HDL のソース・コード・ファイル名はダイアグラムと似ており、拡張子が tim から.v に変わります(ダイアグラム名が untitled.tim のとき、Verilog-HDL のソース・コード・ファイル名は untitledtim.v となる)。

これらのファイルは、WaveFormer がソース・ファイルを生成するたびに、自動的に Report ウィンドウに表示されます(デフォルトでは、設計を変更してシミュレーションを行うたびに自動生成される)。

以下の方法でエラーの起きた Verilog-HDL ソースの行を参照できます。

-verilog.log ファイルでエラーが起きた箇所を確認します。以下はその例です。

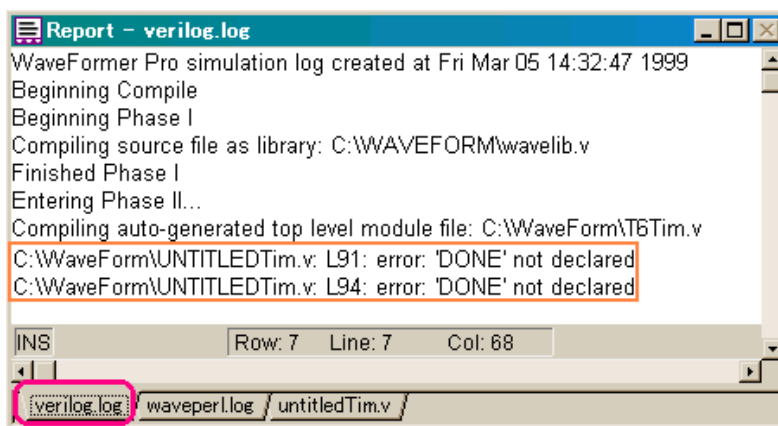


図10 verilog.log ファイルの内容

図10 に示すように、C:\untitledtim.v: L91: error: 'DONE' not declared, と表示されています。これからエラーは、91行目で起きていることがわかります(実際のエラー行番号は本チュートリアルと異なる場合があります)。

-Report ウィンドウの下にある、[* tim.v] ボタン(* は現在のタイミング・ダイアグラム名に対応)をクリックし、Verilog-HDL のソース・ファイルを見ます。

-<Ctrl>-<Shift>-L を押して、Jump to Line Number ウィンドウを表示させます。そして、91 を入力します。

-すると、IDLE と READ をシミュレーションしている Verilog-HDL コードの行が見えます。

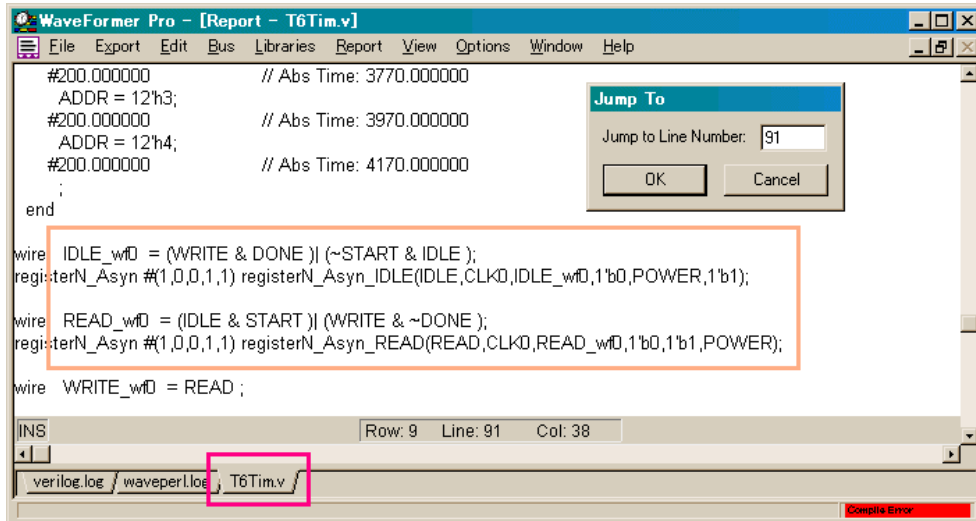


図11 Verilog-HDL ソースとエラー行

NOTE:

ソース・コードを編集しないでください。ソース・コードを編集しても、次回のシミュレーション時に、自動的にソースが上書きされてしまい、編集結果は反映されません。設計を変更する場合は、直接ソース・コードを変更するのではなく、Signal Properties Dialog ウィンドウを使ってください。WaveFormer は、正しい Verilog-HDL ファイルを生成し、シミュレーションを行います。

出力信号をモデリングしたシミュレーション(インクリメンタルな設計)

デジタル・システムのシミュレーションやデバッグを行うにあたり、一方のコンポーネントの出力信号が他方のコンポーネントの入力信号となっているので、システムの大部分を設計し終わらないとテストが始められない、という問題が起こります。

しかし、システムを細分化して、その各々の部分に対して、他の部分の出力をテスト・ベクタとして与えれば、小規模化した分だけ容易にテストを開始できます。しかし、いずれにしろテスト・ベクタの生成に時間を要しますが、WaveFormer は設計の小さい部分のテストに対して、とてもシンプルで素早い方法を提供します。

まだ設計情報のない部分に対しては、代わりにグラフィカルに信号を描いて、これを他の部分に対する入力信号としてテストを行い、後でこの信号を生成するような設計情報を付加すればよいのです。言い換えると、一時的にある部分の期待される出力を、WaveFormer 上で波形を描いてエミュレーションすることになるのです。

以下では、この方法を使って、いままでに作成したステート・マシンの動作を、DONE 信号を生成する論理式を入力する前に確認します。

-DONE 信号を追加します。

-[Low] ボタンをクリックし、1.6[us]長の Low 波形を描きます。この後、1クロック・サイクル以上続く High 信号を描きます(DONE 信号のエミュレーション)。

-生成されたダイアグラムと図1 と比較して、作成したステート・マシンが正しいことを確認してください。

-もし、相違点があれば、まず verilog.log ファイルを調べ、作成したダイアグラムのコンパイルが成功しているかどうかを確認します。エラーがあれば、ダイアグラムのエラー箇所がエラー・メッセージと一緒に表示されます。また、IDLE,READ,WRITE の各信号の Signal Properties Dialog ウィンドウで、Simulated Signal ラジオ・ボタンに

チェックがついているかを確認してください。

-シミュレーションはできるが出力が誤っている場合は、設計式、START、DONE およびPOWER 信号を確認してください。

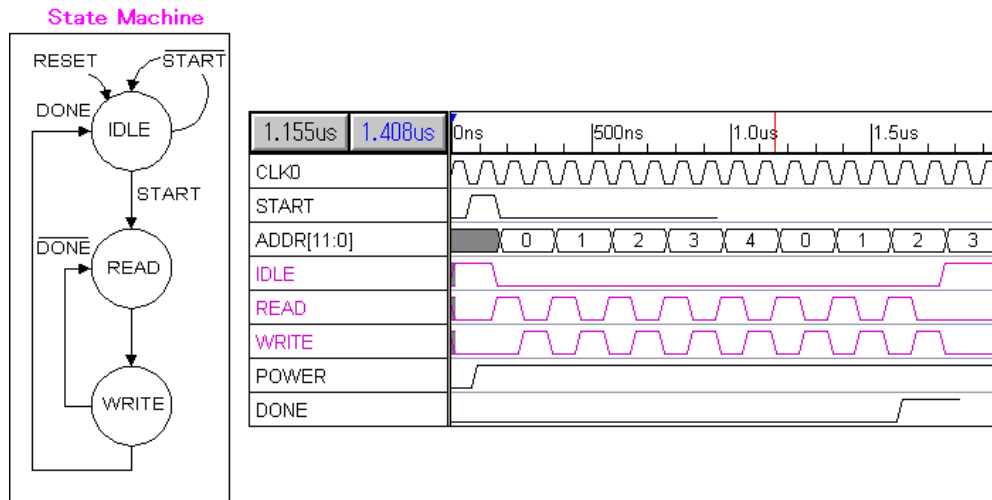


図12 DONE 信号のモデリング(エミュレーション)とステート・マシンのシミュレーション結果

なお、たとえばSTART 信号の High の期間が非常に短い場合、ステート・マシンがどのような動作をするかを確認する場合、単に START 信号のエッジ位置を変更するだけで即 WaveFormer が変化した START 信号を基に再シミュレーションを行い、結果をダイアグラム・ウィンドウに表示してくれます。

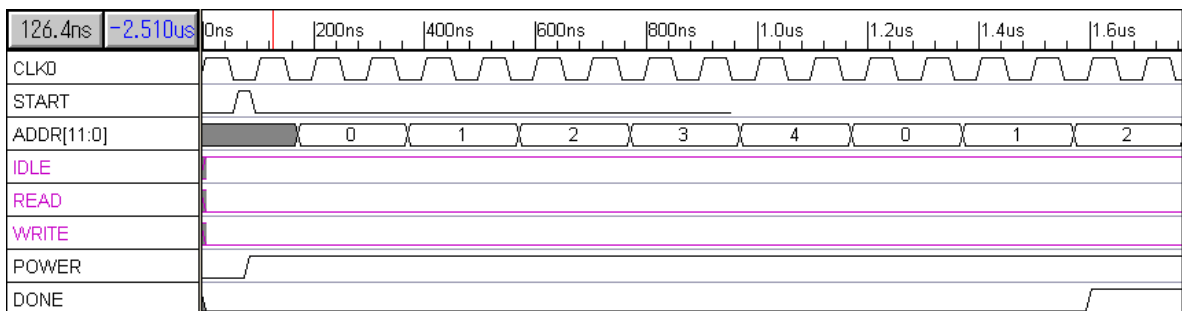


図13 START 信号の変化によるシミュレーション結果

図13 に示すように START 信号の High の期間が非常に短い場合(CLK0 の立ち下りエッジで Low)の場合、図1 に示したステート・マシンの定義どおり IDLE 状態にとどまることがすぐに確認できます。

確認が終わったら START 信号を最初の状態に戻しておいてください。

組み合わせ論理のモデリング

IDLE、READ、WRITE の状態信号に加え、ステート・マシンはENABLE 信号を持ち、SRAM、カウンタ、A/D コンバータをイネーブルにします。

以下ではこのENABLE 信号をモデリングしてみます。

-新たに ENABLE 信号を加えます。

-ENABLE 信号に対して、以下の式を Boolean Equation の項目に入力します。また、Simulated Signal の項目を忘れずにチェックしてください。

$$\text{READ} \mid \text{WRITE}$$

上の式は、READ と WRITE の論理和演算を行うことをあらわします。

なお、入力を完了すると同時に、ENABLE 信号の値がシミュレーションされてダイアグラム・ウィンドウに表示されます。

4.2) Verilog-HDL インターフェースを使ったモデリング

カウンタ回路のモデリング

カウンタの出力 COUNT は Verilog-HDL コードを使ってモデリングします。はじめに、4 ビット・カウンタを動作確認のために作成し、その後にこれを 17 ビットに拡張します。

-COUNT 信号を入力します。

- Signal Properties Dialog で、信号の MSB を 3 に、Radix を hex に設定し、Simulated Signal の項目をチェックします。

-HDL code の項目をチェックし、Equation ビューから、HDL Code ビュー/エディタに切り替え、以下に示すコードを入力します(コメント文は//ではじめ、その行の//以降は実行時にスキップされる)。

```
reg [3:0] COUNTER;
always @(negedge CLK0) //on each falling edge of CLK0
begin
  if (ENABLE) COUNTER = COUNTER + 1; // count while ENABLE is high
  else COUNTER = 0; // synchronous reset if ENABLE is low
end
assign COUNT = COUNTER; //drive wire COUNT with reg COUNTER value
```

Verilog-HDL Note:

WaveFormer 内のすべての信号は wire としてコーディングされます。COUNTER の値で COUNT (wire)を駆動させるのに、Verilog-HDL のコーディング・ブロックの最後で assign 文が必要です。

64K のデータを扱えるように、COUNT の MSB を 16 に変更して、HDL コード上のカウンタの宣言を

```
reg [16:0] COUNTER
```

とすることもできますが、本チュートリアルでは行いません。

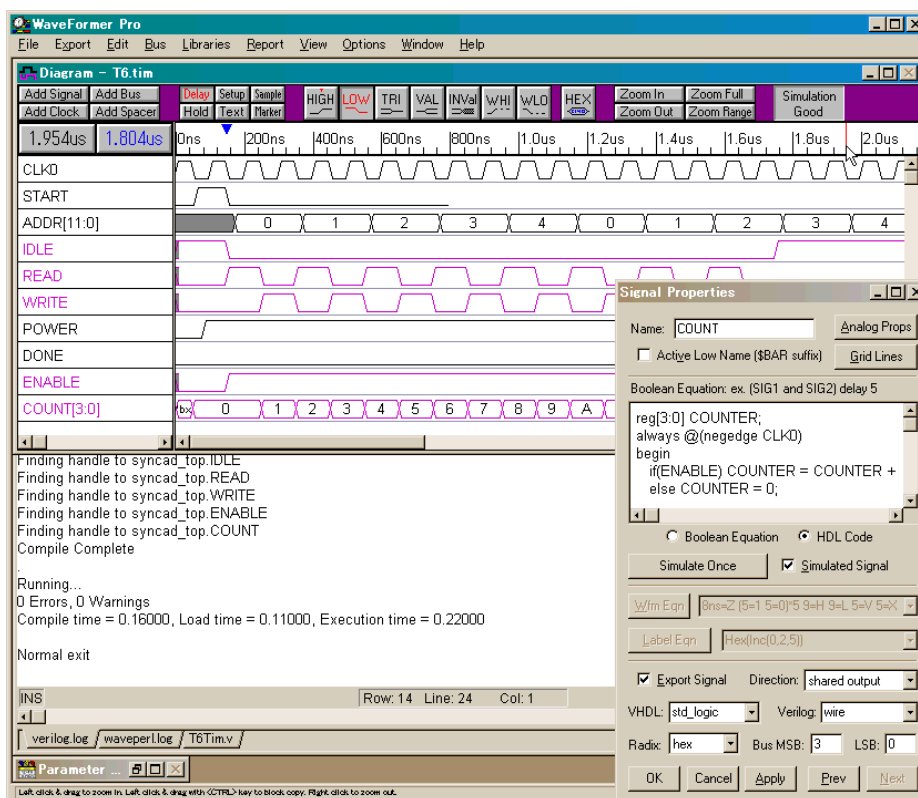
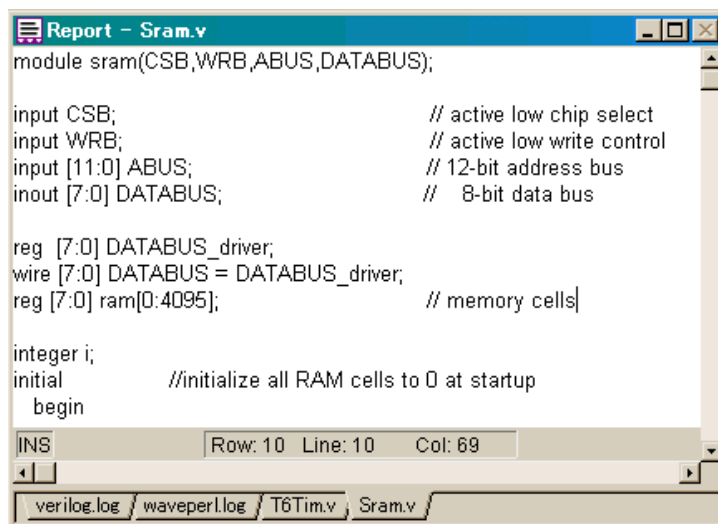


図 14 4 ビット・カウンタの実装とシミュレーション結果

既存の Verilog-HDL モデルを WaveFormer に加える(SRAM のモデリング)

ここでは、すでに Verilog-HDL で記述された SRAM モジュール(sram.v)を Wave Former に取り込んでみます。このモデルは非同期のインターフェースを正確にモデリングしているほか、SRAM 起動時のメモリ・セルのゼロ・クリアを行う記述も含まれています(実際の回路では、アドレス・バスを通して繰り返しゼロ・データを書き込む回路が別に必要になる)。

Verilog-HDL を使った完全な SRAM のモデリングはこのチュートリアル範囲外ですが、[Report]-[Open Report Tab] メニューを使って、おおまかに sram.v を見てください。



```
Report - Sram.v
module sram(CSB,WRB,ABUS,DATABUS);

input CSB; // active low chip select
input WRB; // active low write control
input [11:0] ABUS; // 12-bit address bus
input [7:0] DATABUS; // 8-bit data bus

reg [7:0] DATABUS_driver;
wire [7:0] DATABUS = DATABUS_driver;
reg [7:0] ram[0:4095]; // memory cells

integer i;
initial //initialize all RAM cells to 0 at startup
begin

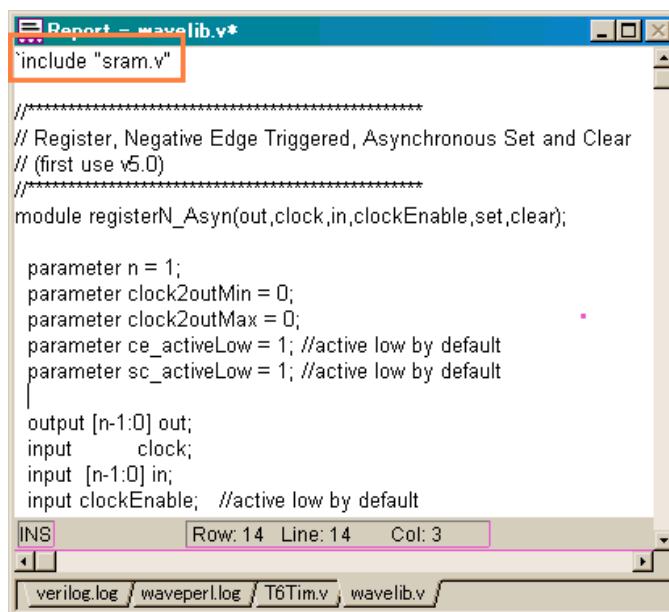
INS Row: 10 Line: 10 Col: 69
verilog.log / waveperl.log / T6Tim.v / Sram.v
```

図 15 SRAM モジュールの Verilog-HDL 記述(sram.v)

次にこの Verilog-HDL ソースを WaveFormer に取り込みます。これは、WaveFormer が使う wavelib.v に include 文を使って sram.v ファイルを取り込みます。

- [Report]-[Open Report Tab...]メニューを使って wavelib.v ファイルを開きます。
- 以下の一行を wavelib.v ファイルの先頭に追加します(すでに追加されている場合がある)。

``include "sram.v";`(include の前にダッシュをつける)



```
Report - wavelib.v*
`include "sram.v"

//*****
// Register, Negative Edge Triggered, Asynchronous Set and Clear
// (first use v5.0)
//*****
module registerN_Asyn(out,clock,in,clockEnable,set,clear);

parameter n = 1;
parameter clock2outMin = 0;
parameter clock2outMax = 0;
parameter ce_activeLow = 1; //active low by default
parameter sc_activeLow = 1; //active low by default

output [n-1:0] out;
input clock;
input [n-1:0] in;
input clockEnable; //active low by default

INS Row: 14 Line: 14 Col: 3
verilog.log / waveperl.log / T6Tim.v / wavelib.v
```

図 16 SRAM モジュール(sram.v)の追加

インクリメンタ, ラッチのモデリング

以前にネガティブ・エッジ・トリガ型のレジスタ(フリップフロップ)を生成する論理式を使ってステート・マシンをモデリングしましたが,ここではレベル・トリガ型のレジスタ(ラッチ)を同様の方法でモデリングしてみます.

SRAM の値が読み出されるとこれを DBUS に送り,インクリメンタで 1 を加算され,ラッチで保持されます.

- DBUS_INC という名前で新しい信号を作ります.
- 以下の論理式を Boolean Equation の項目に入力します.

DBUS+1

- Clock の項目から READ を選び, Edge/Level の項目から High を選びます(High レベル・ラッチの実現).
- MSB を 15 に, Radix を hex に,そして Simulated Signal をチェックします.

なお,この時点ではDBUS が定義されていないのでシミュレーション結果はあられられません.

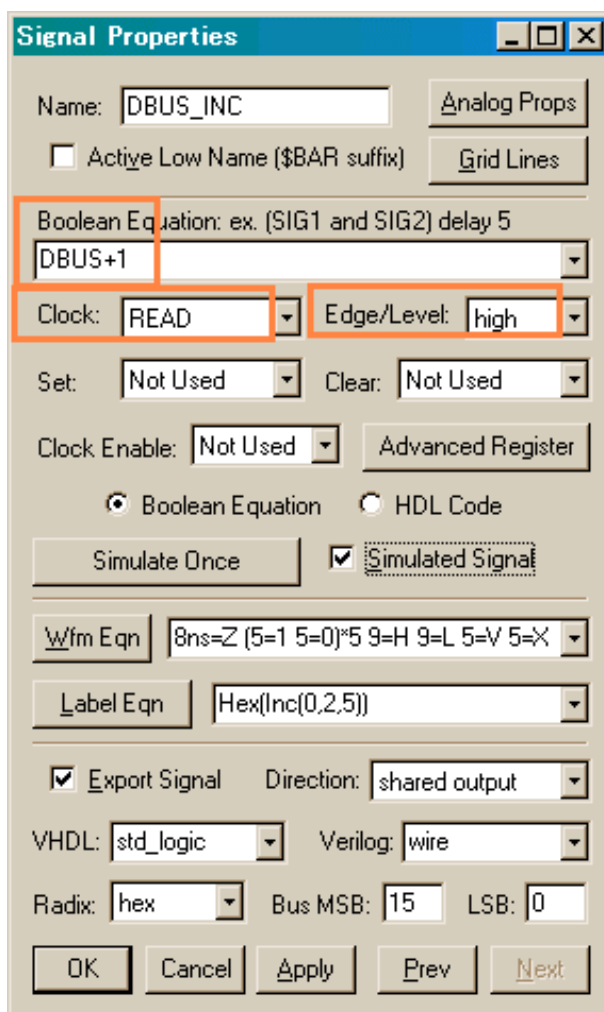


図 17 インクリメンタ, ラッチのモデリング

条件演算子を使った 3 ステート・ゲートのモデリングと,SRAM モデルの実装

- DBUS という名前で新しい信号を作ります.
- MSB を 15, Radix を hex に,そして Simulated Signal 項目をチェックします.
- 次の Verilog-HDL コードを DBUS の HDL コード・ウィンドウに入力します.

```

wire CSB = !ENABLE;
sram BinMem1(CSB,READ,ADDR,DBUS[7:0]);
sram BinMem2(CSB,READ,ADDR,DBUS[15:8]);
assign DBUS = WRITE ? DBUS_INC : 'bz;

```

最初の行ではENABLE 信号を反転した信号を生成します(SRAM はアクティブ・ロー)。
 次の2つの行は、8ビット×4KのSRAMを実装したもので、入出力信号のマッピングを行っています。
 (最初のSRAMはDBUSの下位バイト、次のSRAMは上位バイトに接続されている)
 最後の行は、DBUSを3ステート・ゲートとしてモデリングしています。ここでIf-then-else(If条件 then a else b)の働きをする条件演算子(条件 ? a : b)を使っています。WRITEがHighのときはDBUSはDBUS_INCによって駆動され、LowのときはDBUSは他と切り離されます('bzはすべてのビットがハイ・インピーダンスになることをあらわす)。
 以上の作業を行ったらシミュレーションを行ってみます。以下に示すようなダイアグラムが得られます。

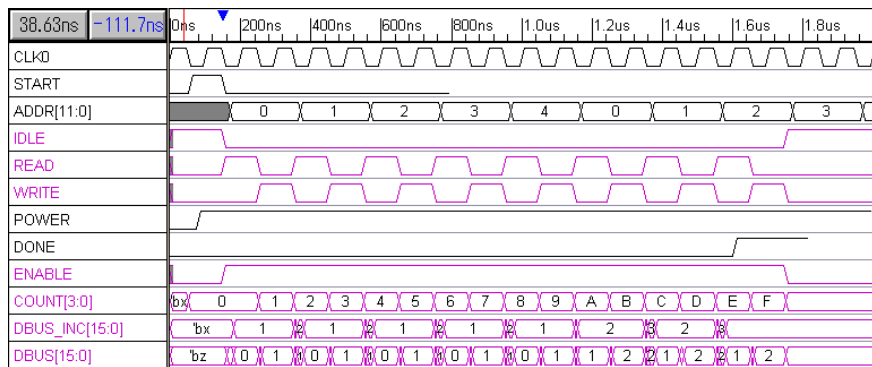


図18 DBUS およびDBUS_INC をモデリングした後のシミュレーション結果

リダクション・オペレータを使ったn ビット・ゲートのモデリングによるDONE 信号のモデリング

最後にステート・マシンに対する入力信号としてエミュレーションしたDONE 信号をモデリングします。
 DONE 信号はCOUNT 信号の各ビットの論理積を実行させることで生成されます。すでに作成済みのDONE 信号をダブル・クリックして、Simulated Signal にチェックをつけた後、次の論理式を入力します。

&COUNT

&オペレータはリダクション・オペレータと呼ばれ、入力信号の各ビット同士の論理積(AND)を演算することを意味します。これは次の式と同等です。

COUNT[0] & COUNT[1] & COUNT[2] & ...

リダクション・オペレータの利点は、ビット長を自動的に計算して上記のような論理式を自動生成するので、入力の手間が格段に省けることにあります。

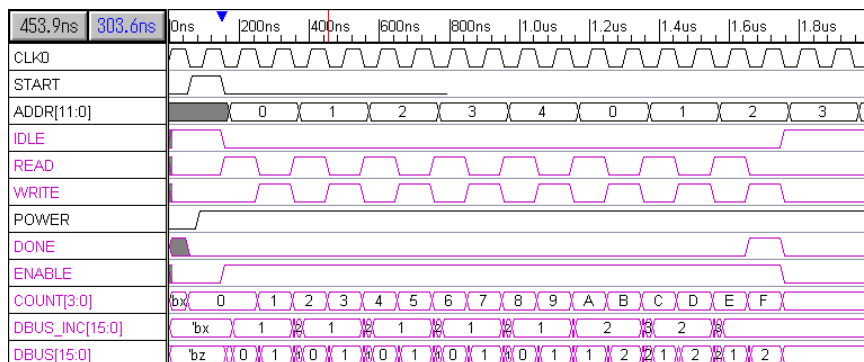


図19 DONE 信号をモデリングした後のシミュレーション結果

\$display,\$monitor コマンドを使った外部Verilog-HDL モデルのデバッグ

Verilog-HDL はシステム・タスク, \$display,\$monitor を装備しています。これらは Verilog-HDL ソースのデバッグを目的に用意されています。

\$display は C 言語の printf 文に相当し, \$display が実行されると, シミュレーション・ログ・ファイルの verilog.log に処理結果が出力されます。

\$monitor は, パラメータに指定された信号が変化したときに, ログ・ファイルにレポートを出力します。

SRAM モデル・ファイル sram.v は 2 つの \$display 文を含み, SRAM に対してデータの読み/書きが行われるときにレポートを出力します。

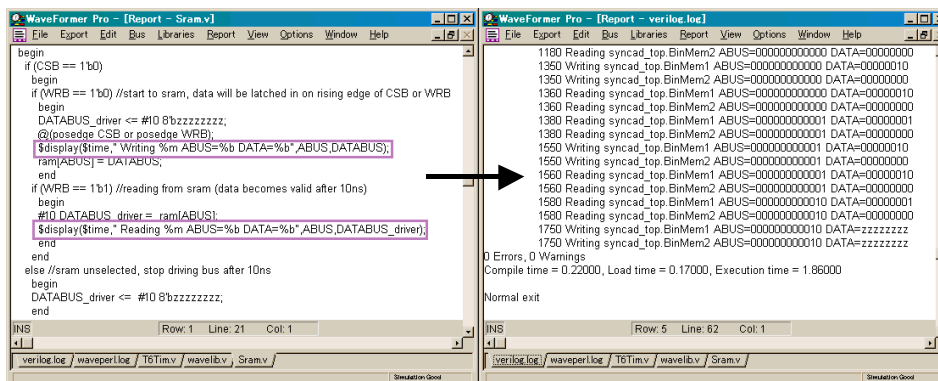


図20 sram.v に記述された \$display システム・タスクのレポート例

以上でヒストグラム回路のモデリングが完了しました。図 19 に示すような波形が描かれているでしょうか。自分の作成したダイアグラムと違う部分を見つけたら, verilog.log を使いながら間違いを訂正していきます。

SRAM から不確定なデータを読み込んだことを示す x が DBUS 上に現れているときは, \$display コマンドの出力を使ってチェックを行うのが有効です。この場合, 不確定なアドレスに対して SRAM に書き込みを行っていないかを確認してください。

End Diagram Marker を使ったシミュレーション時間の制御

デフォルトでは, WaveFormer は信号があるところまでシミュレーションを行います, タイム・マーカによってシミュレーション時間を制御することもできます。

-[Marker]ボタンをクリックし, ダイアグラム・ウィンドウ内の任意の位置でシミュレーションを終えたい時刻付近で右クリックしてマーカをつけます。

-マーカを左クリックで選択すると緑色の表示となるので, これをダブル・クリックして Edit Time Marker ダイアログ・ウィンドウを開きます。

-marker type を "End Diagram" に設定します。

-ダイアログを閉じマーカを動かすと, 自動的にマーカの置かれた時刻まで再シミュレーションを行います。

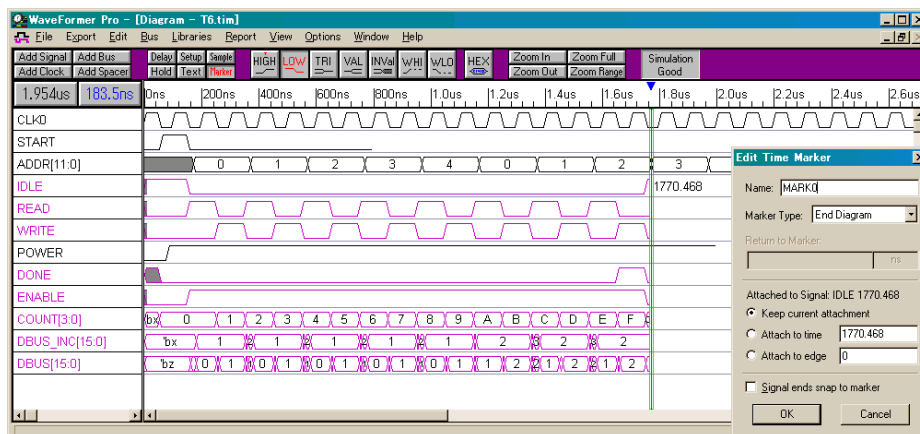


図21 タイム・マーカの設定例

WaveFormer Pro を使った Verilog-HDL ソースの編集

WaveFormer Pro に加えられた Verilog-HDL ソース・ファイルの編集は次のように行います。なお、ここでは SRAM モデル・ファイル sram.v を編集してみます。

- Report ウィンドウ内で sram.v ファイルを開きます。
- 17行目を、ram[i]=0; から ram[i]=8へ変更します。

以上の作業で、SRAMセルは、0の代わりに8で初期化されるようになります。

- Signal Properties Dialog の Simulate Once ボタンをクリックするか、入力信号のエッジを動かして再シミュレーションを行います。

NOTE:

sram.v ファイルをシミュレーションする前にセーブしましょう。sram.v を表示させた状態で、[Report]-[Save Report Tab]メニューを起動すると、Report ウィンドウ内の sram.v ファイルはセーブされます。

結果は DBUS に 8 が出力されると予想されますが、808 が表示されます。これは DBUS が2つの SRAM の初期化値 08(hex 0808= hex 808) から 16 ビットのデータを構成しているためです。

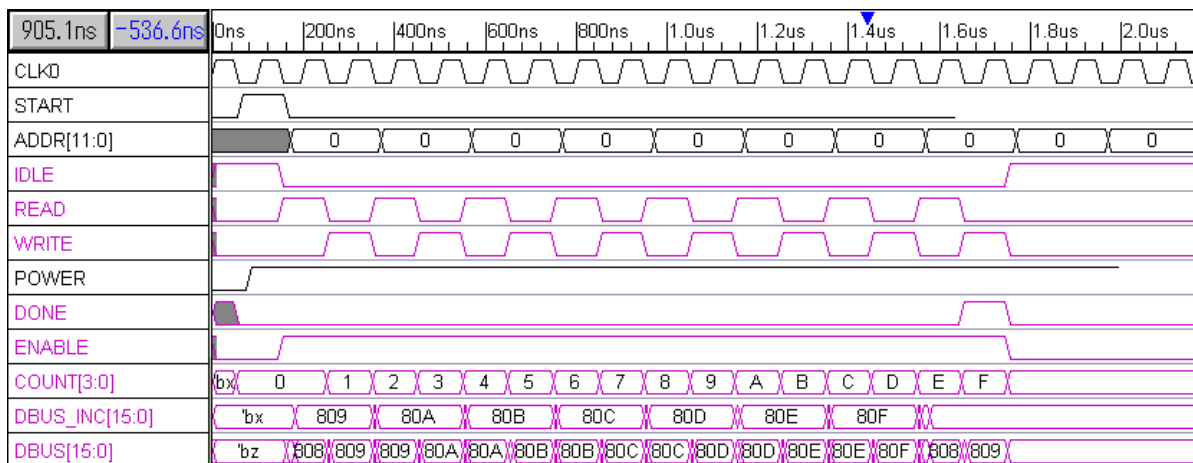


図22 SRAM の初期化値を 8 に変更したシミュレーション結果(ADDR 信号をすべて 0 とした)

5) まとめ

WaveFormer Pro は、

- 単なる波形エディタではなく、Verilog-HDL シミュレータを内蔵したモデリング・ツールとしても使える
- 細かい単位で設計 検証 設計...が容易に行えるツールである(インクリメンタルな設計作業ができる)
- WaveFormer が自動生成した Verilog-HDL ソースは他の Verilog-HDL シミュレータでも使える

という特徴を備えた強力なツールです。

以上でチュートリアルは終了です。本チュートリアルについてのご質問などは、小社デザインウェブ企画室までお願いいたします。

WaveFormer Pro, TestBench Pro そして SynaptiCAD は SynaptiCAD Inc.のトレード・マークです。

CQ 出版株式会社 デザインウェブ企画室
〒170-8461 東京都豊島区巣鴨1-14-2 CQ ビル4F
TEL :03-5395-2126 FAX :03-5395-2127

E-mail : edasupport@cqpub.co.jp web : <http://www.cqpub.co.jp/>