“CodeTaster”解説

2009/09/27 初版作成
2010/01/21 改定 V102
シグナル・プロセス・ロジック株式会社

1. はじめに

CodeTaster は、Verilog で書かれたモジュール（以下「供試モジュール」と呼びます）を実際のハードウエア上で動作テストするためのソフトウエアです。

CodeTaster はまず、ユーザとの対話により、テストを行うための Verilog コード（以下「テストベンチ」と呼びます）を自動生成します。

生成されたテストベンチをコンパイルして FPGA にロードするまでの作業は、ユーザ側の環境で行っていただきます。

CodeTaster は次に、シリアルポートを介して FPGA と通信し、PC 側に準備したテストベクターを FPGA に送信し、これを用いて供試モジュールのテストを行い、テスト結果を PC 側に受信します。

今回ご提供するバージョンでは、転送時間の関係からテストベクタおよび結果のビット幅を、ともに 16 bit 幅×1024 点としています。これはたとえば、2 つの入力ポートをもつモジュールであれば､それぞれのポートのビット幅は 8 bit に制約されることを意味します。この制約は絶対的なものではなく、ご要望があれば異なるビット幅のバージョンをご提供することも可能です。

【V102 での変更部分】弊社は、四則演算を行う Verilog ソースコードを自動作成するソフトウエア“CodeSqueezer Basic”の供給を開始いたしました。今回の改定は、このソフトウエアが生成するモジュール検証の便宜を図るために行ったもので、以下の変更を加えました。

- CodeSqueezer が生成するモジュールソースコードの入出力ポート宣言部に、型を表すコメント（“//” に続く型指定文字列）を追加しています。
- 型指定文字列に“signed”があれば符号付整数であることを示します。
- 型指定文字列に“point:”に続く 10 新数があれば小数点位置を示します。
- CodeTaster は、これらの型指定文字列により符号および小数点位置を判定し、これらを反映した数値を出力ファイルに記録します。
- テストベクター（入力ファイル）は、従来と同様、符号なし整数で与えます。

ご注意

ソフトウエア“CodeTaster”およびこれに含まれるテンプレートファイルは、これらを使用したことに伴う結果に対して弊社が一切の責任を負わないことを条件に、無許諾無償

で使用することができます。ご使用に際して、テンプレートファイルもしくは中間出力ファイル（“CodeTaster.v”）を改変することも自由に行うことができます。

なお、これらの一部または全部を複製・配布する行為は上記の範囲外です。これらを所望される場合は弊社（cs@signal-process-logic.com）にお問い合わせください。

2. 準備

2.1. インストール

- CodeTaster_100.msi をダウンロードし、ファイル名（アイコン）をダブルクリックします。
- 表示されるウィンドウのメッセージに従って「次へ（N)」ボタンを押していくことでインストールが完了します。
- 途中、セキュリティ保護のための画面が表示されたら「許可する」をクリックしてください。
- 正常にインストールされたら、デスクトップ上に“CodeTaster”のアイコンが追加されます。また、スタートメニューにも“CodeTaster”が追加されます。これらをクリックすることで CodeTaster が起動されます。

2.2. コンパイル環境の準備

生成されたコードをコンパイルして供試 FPGA にロードするためには、ユーザの試験環境に合わせた操作があらかじめ必要になります。

以下の説明では、コンパイル環境には Altera 社の Qualtus II Ver9.0 SP2 を使用し、ハードウエアとして「Nios II エンベデッド評価キット Cyclone III エディション」（私は Cyclone III スターターキットとアップグレードパッケージで構成）を使用した場合を例として以下説明します。

2.2.1. クロックジェネレータ：ALTPLL

Nios II エンベデッド評価キットには 50 MHz の外部クロックが設けられており、これをそのままシステムクロックとして使用しています。

シリアルポート用のクロック信号は、Altera 社の提供するメガファンクション“ALTPLL”を用いて生成しています。これには、Qualtus II 付属の MegaWizard Plug-In Manager の“ALTPLL”機能を用いてモジュール“pll921_6k”を作成します。このための手順は次の通りです。

まず、プルダウンメニューから“Tools”→“MegaWizard Plug-In Manager”を選択します。つぎに、“Create a new custom megafunction variation”にチェックを入れて“Next”ボタンを押します。

左側のボックスで“I/O”の“+”マークをクリックして展開し、“ALTPLL”をクリック

して反転表示します。“device family”は“Cyclone III”、“type of output file”は“Verilog HDL”を選択し、出力ファイル名の部分に“pll921_6k”と記入して“Next”ボタンを押します。

次の画面では、入力周波数を今回使用したボードの外部クロック周波数である 50 MHz に設定し、その他はデフォルト（automatically, normal）のまま“Next” ボタンを押します。

次の画面では、argset や locked といった出力ポートをつくるかどうかを選択しますが、今回はすべて不要ですので、チェックを外して“Next”ボタンを押します。

Bandwidth 設定画面と inclk1 作成画面および Dynamic configuration 画面では、デフォルト（auto, 未使用）のまま“Next”ボタンを押します。

出力 c0 は“Use this clock”にチェックを入れ、“Enter output clock frequency”を選択した状態で“Requested settings”を“0.9216 MHz”とします。これは、ボーレート 9216000 bps に対応する周波数です。

出力 c1 も同様に“Use this clock”にチェックを入れ、“Enter output clock frequency”を選択した状態で“Requested settings”を“14.7456 MHz”とします。これはシリアル受信に使用するクロック信号で、ボーレートの 16 倍の周波数とします。実際に設定される周波数は、指定したものとは多少異なりますが、問題はありません。

その後の画面では特に設定するものはなく、すべて“Next”ボタンを押して完了します。

こうして生成された PLL モジュールは、上の手順どおりに作成した場合には、“pll921_6k.v”というファイルに生成されており、これを開くことで内容を確認できます。このポート定義の部分が、テンプレートファイル template_1.ct の PLL モジュール配置部分の記述とで、ポート名などが異なっていないことを確認してください。

2.2.2. RAM

RAM も PLL と同様の手順で MegaWizard Plug-In Manager を用いて作成します。

RAM は、“Memory Compiler”の“RAM: 2-PORT”を使用します。生成されるモジュール名は“ram16_1k.v”とします。

その後表示される画面では、read port、write port 各 1、ビット幅は 16、ワード数は 1024、memory block type は Auto とし、single clock を選択します。RAM の入出力はすべてレジスタ経由（デフォルト）とします。

メモリー初期化ファイルは指定する必要はありません(その場合は 0 で初期化されます)。

2.2.3. ピンのアサイン

テストベンチには txd, rxd, clock の 3 本の入出力があります。FPGA に論理を与える際には、これらの入出力ポートをチップ上のピンに割り当てる（アサインする）必要があります。

Qualtus II でピンをアサインする際、一旦コンパイルを行ってからアサインするのが、ポート名が自動的に設定されるために簡便です。

ピンのアサインは、プルダウンメニューの“Assignments”→“Pins”で行います。既にコンパイルが完了している場合には画面下部のポート一覧表にノード名（ポート名に同じ）と入出力の方向が表示されています。

ピン番号は“Location”の欄に設定します。この部分の表示は最終的には“PIN_B9”などとしますが、入力する際は“B9”とタイプインすることで自動的に“PIN_B9”という文字列が入力されます。

「Nios II エンベデッド評価キット Cyclone III エディション」を使用する場合のピンアサインは、clock が PIN_B9、rxd が PIN_E18、txd が PIN_H17 となります。また、これらのピンの一つは nCEO と兼用になっておりますので、“Settings”→“Device”で表示される画面の“Divice and Pin Options”ボタンを押し、“Dual-Purpose Pins”というタブを選択することで表示される nCEO を“Use as regular I/O”に切り替えなければなりません。

同じ画面で、“Unused Pins”というタブもチェックしておきましょう。古いバージョンの Qualtus II では、未使用ピンがすべて出力にアサインされており、これをそのままにしてボードの機能を部分的に使用する論理をコンパイルすると、生成したオブジェクト FOGA にロードした際に大電流が流れるという問題がありました。最近の Qualtus II では未使用ピンは weak pull-up 付きの入力となり、このような問題は生じません。

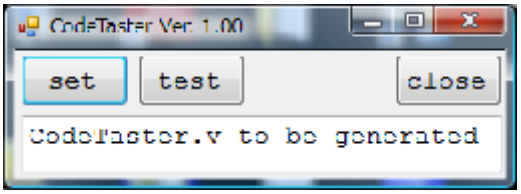
ピンアサイン等の設定を行った後、再度コンパイルを行い、設定をネットリストに反映させます。

3. CodeTaster の操作方法

CodeTaster は“set（テストベンチの生成）”および“test（テスト）”の二つの段階で行います。set 工程で自動生成されるテストベンチは、Verilog ソースコードで出力され、これをユーザがチェック・改変することができます。このテストベンチ・ソースコードを、ユーザ側の環境でコンパイルし、FPGA に書き込んでいただきます。

テストベンチを生成した際の情報はテストの段階に引き継がれます。この間に CodeTaster を終了した場合は、テストに必要な情報を再形成するため、テストに先立って同じ条件でテストベンチの生成手順を実施してください。

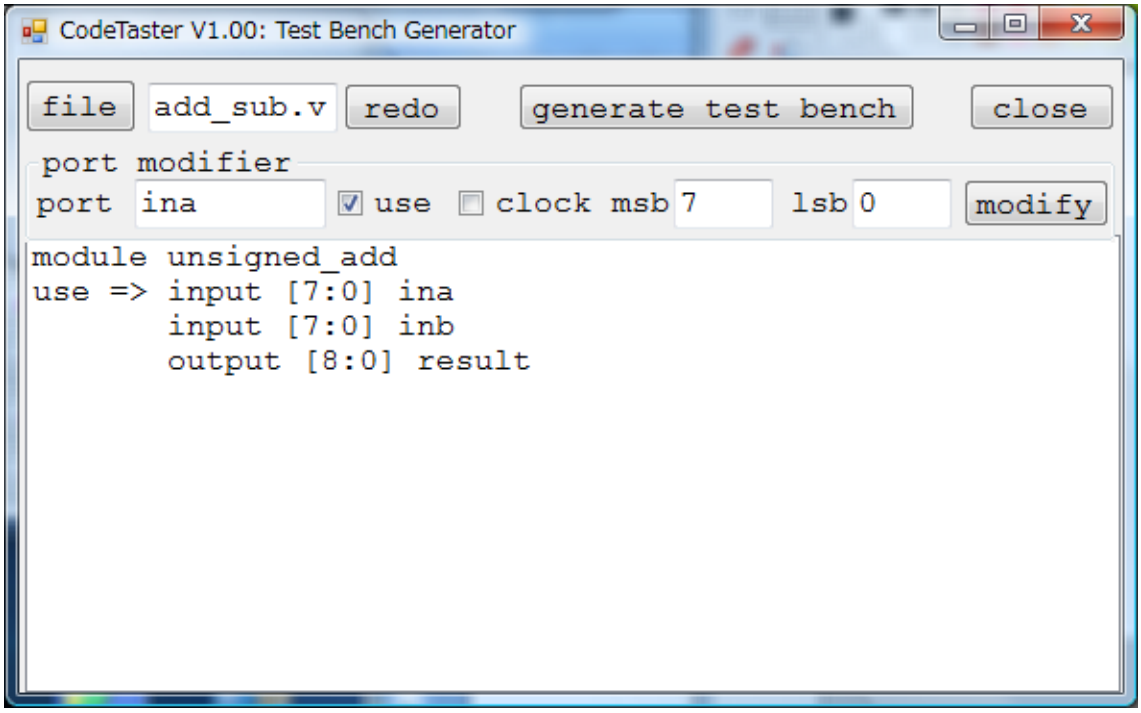
3.1. テストベンチの自動作成



テストベンチの生成手順は、次のとおりです。

- 供試モジュールを作成し、拡張子を“.v”とするファイルに格納します。ファイルは複数のモジュールを含むことができます。
- CodeTaster を起動します。右上図のウィンドウが表示されます。
- “set”ボタンを押し、次いで供試モジュールが含まれているファイルを開きます。

・ CodeTaster はファイル中のモジュール名とポート名を自動抽出して表示します。
・ ポート名が表示された部分でマウスをダブルクリックすると、該当するポートが選択されます（次ページ上図）。
  – 選択されたモジュールのみを残し、他のモジュールに関する情報は消去されます。
  – “redo”ボタンを押すことで全モジュールの情報を再表示することができます。
・ 選択されたモジュール名にはポート名の先頭に“use =>”というマークがつきます。
  – モジュール名をもう一度ダブルクリックすることで選択は解除されます。
・ モジュール名をクリックすると、“port modifier”が表示され、選択されたポートに関する情報が表示されます。
  – port modifier の各項目を書き換えたのち“modify”ボタンを押すことで、選択



されたポートに関する設定内容を変更することができます。
・ 選択が完了したら“generate test bench”ボタンを押します。
  – 生成されるテストベンチ名を指定します（デフォルトは“CodeTaster.v”）。
  – テストベンチのソースコードが自動作成され、ウィンドウが閉じます。

3.2. テストの実行と事前の準備

テストは、次の手順で実行します。

(1) FPGA 内部の RAM にテストベクターを転送します。

(2) RAM 内容を順次供試モジュール入力に送ると同時に、供試モジュール出力を RAM に格納します。

(3) RAM の内容を PC に転送してファイル出力します。

テストに先立って以下を準備する必要があります。

- テストベンチのソースコードをコンパイルし FPGA にロードしておくこと
- テストベクターの準備
- FPGA 評価ボードと PC をシリアルケーブルで接続しておくこと

これらの手順の操作方法を、以下順を追って解説します。

3.3. テストベクターの読み込みと RAM への転送

起動画面の“test”ボタンを押すと、次ページ上部の画面が現れます。

“read”ボタンを押すとテストベクターファイルの選択画面が現れます。指定されたファイルの内容は、シリアルポートを経由して、FPGA 内部の RAM に転送されます。

テストベクターは csv 形式のテキストファイルとして作成します。それぞれの行の内容がクロックごとに供試モジュールの指定した入力ポートに与えられます。

転送工程を実行している間、作業の進行状況を示すプログレスバーが表示されます。

3.4. テストの実行

“test”ボタンを押すことで RAM の内容が順次供試モジュールの入力ポートに与えられ、供試モジュールの出力が同じ RAM に書き込まれます。この動作は非常に短時間のうちに完了します（クロック 50 MHz で 1024 点を処理するに要する時間は 0.02 ms）。

3.5. 結果の受信

“receive”ボタンを押すと、受信結果を格納するファイルを指定する画面が現れます（デフォルトは“out.csv”）。ファイルを指定すると、FPGA 内の RAM 内容が順次 PC に転送され、ファイルに書き込まれます。結果の解析を容易にするため、出力ファイル各行の先頭にはテストベクターファイルの内容がコピーされ、そのあとに受信結果が記録されます。

3.6. 文字列の送信と受信

“send”ボタンを押すと、その右側のボックス内の文字列が FPGA に送信されます。FPGA 側に組み込まれた論理は、標準状態（template_2.ct の初期状態のまま）であれば、特定のコマンド文字を除いて受信文字をエコーバックします。エコーバックされた結果は画面下半分の領域に表示されます。

この機能を利用すれば、ユーザは template_2.ct のコードに手を加えることにより、CodeTaster に種々の動作をさせることが可能となります。

初期状態でコマンドに割り当てられている文字は以下の通りです。

s：RAM の内容を 1 レコード（2 Byte）送信し、読み出しアドレスを 1 増加させる

r：読み出しアドレスと書き込みアドレスをリセットする

w：続けて送られた 2 Byte を RAM に書き込み、書き込みアドレスを 1 増加させる
　　書き込みデータの受信に対して“0”または“1”をレスポンスとして返す。

t：RAM の全領域についてテストを実施する。

3.7.　シリアルポートの通信条件変更

画面上部“SIO”の文字の右側に、シリアルポート名とボーレートが表示されています。

使用可能なシリアルポートは自動検出されます。複数のシリアルポートが使用できる場合は、これらから選択することができます。

ボーレートも選択可能であり、デフォルトでは最高のボーレートに設定しています。高いボーレートの通信がうまく行えない場合は、ボーレートを下げて使用します。ボーレートを下げる場合には、FPGA 側のボーレートもこれに合わせる必要があり、PLL モジュールの出力周波数を、c0 側をボーレートに対応する周波数に、c1 側はこれを 16 倍した周波数に設定します。

4.　テンプレートファイル

CodeTaster.v の大部分はテンプレートファイルをコピーすることで作成されています。

ファイル“template_1.ct”は、CodeTaster.v の先頭部分から供試モジュール定義の前までの部分、ファイル“template_2.ct”は供試モジュール定義以降の部分に相当します。

テストベンチ“CodeTaster.v”のソースコードを以下に示します。このファイルは CodeTaster の“set”機能により自動的に生成されたもので、二つのテンプレートファイルの内容と、CodeTaster が作成するコードが含まれています。

テストベンチファイルおよびテンプレートファイルは、テストの目的やテスト環境に応じて自由に書き換えてお使いください。

コードの理解を助け、書き換えを容易にするため、以下に掲載したソースリスト中には日本語のコメントを多数追加しておきました。これらの日本語コメントは、CodeTaster の自動出力には含まれておりません。Qualtus II その他の多くの Verilog 翻訳環境は日本語を扱うことができず、以下のリストをこのままコンパイルすることはできません。

以下のリストは、あくまでソースコードを理解するためのものであり、コンパイルする際は CodeTaster の出力するソースリストを使用するか、あるいは以下のソースリストから日本語コメントの部分を削除してご利用ください。

```
// モジュール名と入出力ポートの定義。
//シリアルポートの rxd と txd および外部クロック入力線を定義している。
module CodeTaster( // CodeTaster top module
```

```
        input rxd,
        output txd,
        input clock);

        // RAM の定義
        // 16 bit 幅×1024 ワードの RAM を定義する。rd_ram は read data、ra_ram は read address、
        //      wd_ram は write data、 wa_ram  は write address、w_ram は write enable をそれぞれ示す。
        // ram difinition
        wire [15:0] rd_ram;
        reg [15:0] wd_ram;
        reg [9:0] ra_ram, wa_ram;
        reg w_ram;
        ram16_1k u1(.clock(clock), .data(wd_ram), .rdaddress(ra_ram),
                .wraddress(wa_ram), .wren(w_ram), .q(rd_ram));
        // count_out は上位バイトを出力するか、下位バイトを出力するかの選択信号
        reg count_out;
        wire [7:0] char_to_put = count_out ? rd_ram[15:8] : rd_ram[7:0];
        // 以下はシリアルポートを制御するためのモジュール定義である。
        // シリアル送受信モジュールの詳細に関しては“Verilog HDL Code Book”を参照願う。
        // modules for serial port
        wire [7:0] char_getc;
        reg [7:0] char_putc;
        wire gotten, ack_putc, ready_putc;
        reg ack_getc, req_putc;
        getc u5(.char(char_getc), .gotten(gotten), .ack(ack_getc),
                .rxd(rxd), .rclk16(rclk16), .clock(clock));
        putc u6(.char(char_putc), .req(req_putc), .txd(txd),
                .ack(ack_putc), .ready(ready_putc), .clk9600(clk9600), .clock(clock));

        // 以下は PLL モジュールとクロック信号線の定義
        // clk9600 は、当初 9600 bps を使用した名残で、実際には 921600bps としている
        wire rclk16, clk9600;
        pll921_6k u4(.inclk0(clock), .c0(clk9600), .c1(rclk16));

        // モジュールと接続するための信号線
        wire [15:0] module_out;
        reg [15:0] module_in;
// 以上がテンプレートファイル“template_1.ct”の内容
// 以下はモジュールとの接続部分で CodeTaster ソフト本体が出力した部分である
// すべての入出力ポートを選択しない場合、一部の信号線が形成されないため、
// 以下の部分をマニュアルで書き換える必要がある。
// ============================
// test module definition start
        wire [7:0] _in_0 = module_in[7:0];
        wire [7:0] _in_1 = module_in[15:8];
        wire [8:0] _out_0;
        assign module_out[8:0] = _out_0;
        unsigned_add unsigned_add (.ina(_in_0),.inb(_in_1),.result(_out_0));
// test module definition end
// ============================

// 以下がテンプレートファイル“template_2.ct”の内容
        // 以下は制御信号で、ステートマシンとして動作させるための状態 state、
        // 入力バイトの上下 count_in、テストのシーケンス番号 test_seq を定義
```

```
// control signal
reg [1:0] state;
reg count_in;
reg [10:0] test_seq;

//  以下 clock の立ち上がりに同期した処理が行われる。
// state  番号により異なる処理を実施する
always @(posedge clock) begin
	case(state)
	2'h0: begin	// idle
		// state 0 はアイドル状態。
		// ram への書き込みが行われていたら、書き込みをやめて書き込みアドレスを進める
		if(w_ram) begin
			w_ram <= 1'b0;
			wa_ram <= wa_ram + 10'h1;
		end
		else if(gotten & ~ack_getc & ready_putc & ~req_putc & ~ack_putc) begin
			//  シリアルポートから受信した場合は、まずエコーバックする
			// echo received char
			ack_getc <= 1'b1;
			char_putc <= char_getc;
			req_putc <= 1'b1;
			count_out <= 1'b0;
			count_in <= 1'b0;
			//  受信バイトがコマンドなら、これに応じた処理を実施する。
			// detect command character
			// s を受信したらステートを 1 として ram 内容を一つ送信する
			if(char_getc == "s") state <= 2'd1;	// send ram
			// r を受信したら読み出し、書き込みアドレスをクリアする
			if(char_getc == "r") begin		// reset
				ra_ram <= 10'h0;
				wa_ram <= 10'h0;
			end
			// w を受信したらステートを 2 として ram への書き込みを行う
			if(char_getc == "w") state <= 2'h2;	// write ram
			// t を受信したらテストシーケンスを初期化し、ステートを 3 としてテストを実施
			if(char_getc == "t") begin		// do test
				test_seq <= 11'h7fc;
				state <= 2'h3;
			end
		end
		else begin
			//  受信データがない場合はフラグの処理を行う
			if(~gotten) ack_getc <= 1'b0;
			if(ack_putc) req_putc <= 1'b0;
		end
	end
	2'h1: begin	// send ram
		//  ステートが 1 なら ram 内容（char_to_put に出ている）を送信する
		//  コマンド受信とエコーバックの動作完了を待つ
		if(~gotten) ack_getc <= 1'b0;
		if(ack_putc) req_putc <= 1'b0;
		else if(ready_putc & ~req_putc & ~ack_putc) begin
			//  条件がそろったら 1 文字送信する
```

```
                char_putc <= char_to_put;
                req_putc <= 1'b1;
                // 送信データの上位バイト、下位バイトの切り替え
                count_out <= ~count_out;
                // 上位バイトを送信したらアドレスを 1 つ増やしてアイドルに戻る
                if(count_out) begin
                        state <= 2'd0;
                        ra_ram <= ra_ram + 10'h1;
                end
        end
end
2'h2: begin       // write ram
        // ステートが 2 なら ram への書き込み処理を行う。
        if(gotten & ~ack_getc & ready_putc & ~req_putc & ~ack_putc) begin
                // 書き込むべきデータを受信したら上下位バイトの別を数値で返す
                ack_getc <= 1'b1;
                char_putc <= "0" + {7'h0, count_in};
                req_putc <= 1'b1;
                // 受信データを ram 書き込みデータレジスタにシフトインする
                wd_ram <= {wd_ram[7:0], char_getc};
                count_in <= ~count_in;
                if(count_in) begin
                        // 上位バイトのセットが完了したら書き込んでステートを 0 に戻す
                        w_ram <= 1'b1;
                        state <= 2'h0;
                end
        end
        else begin
                // フラグ類の後処理
                if(~gotten) ack_getc <= 1'b0;
                if(ack_putc) req_putc <= 1'b0;
        end
end
2'h3: begin       // do test
        // ステートが 3 ならテストを実行する
        if(test_seq[10] & ~|test_seq[9:0]) begin
                // test_seq が 0x400 なら終了。書き込みを中止してステート 0 に戻る。
                w_ram <= 1'b0;
                state <= 2'h0;
        end
        else begin
                // そうでない場合は、読み書きアドレスをセットし、
                ra_ram <= test_seq[9:0] + 10'h4;
                wa_ram <= test_seq[9:0];
                // モジュール出力を ram への書き込みレジスタにコピーし
                wd_ram <= module_out;
                // ram の出力をモジュール入力レジスタにコピーし
                module_in <= rd_ram;
                // モジュール出力が有効な期間であれば書き込みを実行し、
                if(~test_seq[10]) w_ram <= 1'b1;
                // テストシーケンスを一つ進める
                test_seq <= test_seq + 11'h1;
        end
end
```

```
            endcase
        end
endmodule

// シリアルポートからの 1 文字入力モジュール。Verilog HDL Code Book を参照されたい
module getc(
        output reg [7:0] char,
        output reg gotten,
        input ack,
        input rxd,
        input rclk16,
        input clock);

        reg [3:0] state;
        reg [4:0] next;
        reg [6:0] past7;
        wire [1:0] sum01 = past7[0] + past7[1];
        wire [1:0] sum23 = past7[2] + past7[3];
        wire [1:0] sum45 = past7[4] + past7[5];
        wire [2:0] sum03 = sum01 + sum23;
        wire [2:0] sum46 = sum45 + {1'b0, past7[6]};
        wire [3:0] sum06 = sum03 + sum46;
        wire mj = ~|sum06[3:2];
        reg last_clk, clk_buf;

        always @(posedge clock) begin
                clk_buf <= rclk16;
                last_clk <= clk_buf;
                if(clk_buf & ~last_clk) begin
                        past7 <= {past7[5:0], ~rxd};
                        case(state)
                        4'd0: begin
                                if(~mj) begin     // IDLE
                                        next <= 5'd24;
                                        state <= 4'd1;
                                end
                                else if(ack) gotten <= 1'b0;
                        end
                        4'd1, 4'd2, 4'd3, 4'd4, 4'd5, 4'd6, 4'd7, 4'd8: begin
                                if(|next) begin
                                        next <= next - 5'd1;
                                end
                                else begin
                                        char <= {mj, char[7:1]};
                                        state <= state + 4'd1;
                                        next <= 5'd15;
                                end
                        end
                        4'd9: begin               // stop bit
                                if(|next) next <= next - 5'd1;
                                else begin
                                        state <= 4'd0;
                                        gotten <= 1'b1;
                                end
```

```
                end
                default: state <= 4'd0;
                endcase
            end
        end

endmodule

//  シリアルポートへの 1 文字出力モジュール。Verilog HDL Code Book を参照されたい
module putc(
        input [7:0] char,
        input req,
        output txd,
        output reg ack,
        output ready,
        input clk9600, clock);

        reg [9:0] sr;
        assign txd = sr[0];
        reg [4:0] cnt;
        assign ready = ~|cnt;
        reg last_clk, clk_buf;

        always @(posedge clock) begin
                last_clk <= clk_buf;
                clk_buf <= clk9600;
                if(~last_clk & clk_buf) begin
                        if(|cnt) begin
                                sr <= {1'b1, sr[9:1]};
                                cnt <= cnt - 5'h1;
                        end
                        else if(req & ~ack) begin
                                sr <= {1'b1, char, 1'b0};
                                cnt <= 5'd10;
                                ack <= 1'b1;
                        end
                        else if(~req) ack <= 1'b0;
                end
        end
endmodule

// CodeTaster は以下に供試モジュールをコピーする。（template_2.ct には存在しない部分）
// ============================
//    test module is following
// ============================
module unsigned_add(
       input [7:0] ina, inb,
       output [8:0] result);

       assign result = ina + inb;
endmodule
```