情報工学実験 II

第 1 回： オリエンテーションと基本的な設計の流れの確認

2018 年 11 月 29 日

柴田 裕一郎（shibata@cis.nagasaki-u.ac.jp）

1 はじめに

本実験では，ハードウェア記述言語を用いたディジタルシステムの設計法を習得することを目的とし，「論理回路」で学んだ内容について実践します．ハードウェア記述言語を用いて回路を設計し，設計ツールを用いて論理合成と最適化を行い，最終的には設計した回路を FPGA（Field Programmable Gate Array: 書き換え可能な LSI）上に実装し，実機による動作確認を行います．

2 実験の進め方・評価方法など

本実験は毎回資料を配布しながら進めます．毎回，その日の実験に必要な事柄の説明を行ってから実験へと移ります．ハンドアウトやスライドなどの講義資料は

```
http://slab.cis.nagasaki-u.ac.jp/~shibata/lab2/wiki/
```

からいつでも参照できます．本実験ではハードウェア記述言語の文法や設計用ツールの使用法など，今回始めて習得する事柄も多いので，説明中はメモを取りながら集中して聴いてください．そうすれば極端に時間をオーバーすることはないはずです．

成績評価は実験レポート 100% とし，試験は行いません．未提出の課題がひとつでもある場合は不合格です．他人のコピーとみなせるレポートは採点しません．なお，3 年生の「情報工学実験 IV」は本実験を修得しないと履修できないため，本実験が不合格になると 4 年生に進級できなくなってしまう可能性が生じます．どうか積極な気持で参加してください．

3 LSI 設計の流れ

さて，LSI（大規模集積回路：Large Scale Integrated circuit）の設計手順は概ね以下のようになっています．

（1）仕様レベルの設計
　まずはどのようなチップを作るのか，アルゴリズムなども含めて決定していきます．

（2）レジスタトランスファレベル（RTL: Register Transfer Level）の設計
　クロックサイクルごとのハードウェアの振る舞いを設計し，シミュレーションして動作を検証します．最近では C や JAVA などのプログラミング言語を用いて，もっと抽象度の高いレベルで設計する方式（高位合成による設計）も盛んです．

（3）論理レベルの設計
　仮配線遅延を用いた論理シミュレーションによる動作の検証です．テストベクタの作成，故障シミュレーションも行います．

（4）レイアウトおよび配置配線
　ゲートアレイの場合自動的に行われます．セミカスタムは一部自動．フルカスタムは人手で行います．
　終了後，実配線遅延を用いた論理シミュレーションによりテストベクタを用いて動作を検証します．
（5）マスク生成
（6）製造
（7）製品検査

本実験でも実際に LSI チップを試作してみたいところではありますが，予算も限られており現状では困難です．そこで，本実験では FPGA（書き換え可能なゲートアレイ：Field Porgrammable Gate Array）を用います．FPGA というのは何度も書き換えられるハードウェアと考えると分かりやすいでしょう．配置配線をした後の回路情報データ（コンフィギュレーションデータ）をチップにダウンロードすれば，自分で設計した回路の実際の動作を確認できます．つまり，論理レベルで設計し，論理シミュレーションを行えば，一応後のステップは自動的に CAD が全てを行ってくれます．ゲートアレイを製造するにはたいへんな費用がかかるので，製造した後に設計ミスが見つかると大きな損失を被ることになります．そこで，実際にチップをおこす前に FPGA を使って機能を検証することがしばしばあります．

FPGA は柔軟性を持っている分だけ，スピードや面積効率はどうしてもカスタム LSI に劣るのですが，高性能で大きな FPGA が比較的安価に流通するようになりました．一方，最先端プロセスでの LSI 開発は莫大な費用がかかることから，近年では機能検証だけではなく，家庭用テレビなどの一般の商用製品にも広く FPGA が使われるようになりました．さらにデータセンタなどでは，大規模データ解析処理の一部を FPGA 上にハードウェア化することで高速化する技術がすでに実用化されています．

本実験ではハードウェア記述言語の一種である SystemVerilog を設計に用います．これは，前述の設計レベルでいうとレジスタトランスファレベルの設計を行うことになります．このレベルのシミュレーションで機能を確認すれば，論理回路の生成はツールによって自動的に行われます．

まとめると，今回の実験で人手で行うのは主に 1 と 2 のステップです．つまり設計，記述，シミュレーションが主な仕事となります．きちんとシミュレーションで動作を確認すれば，実装に失敗することはまあ稀だと考えて良いでしょう[*1]．あとは，論理回路の自動生成時に性能に関する見積りデータが得られるので，これを考察してレポートをまとめるという流れになります．

4　ハードウェア記述言語とは

これまでの論理回路やハードウェアの講義では，論理回路を回路図を用いて表現し検討してきました．このように図面上でディジタル回路を設計し入力する方法は，もちろん今でも用いられていますが，現在はディジタル回路をプログラミングのように言語で記述し，その記述から回路を自動合成する方法が一般的です．このような言語のことをハードウェア記述言語（HDL: Hardware Description Language）といいます．HDL には次のような利点があります．

- 抽象度の高いレベルで記述するためバグが発見しやすく他人がみても分かりやすい
- 論理の簡単化など詳細なレベルの設計は論理の自動合成時に自動的に行われるので，設計のための時間が短縮できバグも減る

[*1] あくまで実験レベルではです．本当は世の中はそんなに甘くないです．

ソースコード 1　myand.sv

```
1  `default_nettype none
2  /*
3   * Simple AND gate
4   */
5  module myand
6    (
7     input wire  in1,
8     input wire  in2,
9     output wire out1
10    );
11
12    assign out1 = in1 & in2;
13 endmodule
14 `default_nettype wire
```

しかし，一方でこのような方法は以下の弱点を持っています．

- 合成可能な回路の形式が制限される（特に非同期回路など）
- 最大性能は人手による設計に及ばないこともある（最近はツールも賢いのでほとんど無い）

現在では HDL を用いた設計が主流で，回路図設計はあまり行われなくなりました．また，HDL よりもさらに抽象度の高い言語による設計技術（高位合成）が一般化しつつあります．HDL にもいくつかの種類がありますが，本実験では SystemVerilog を使用します．ただし，あまり文法の詳細には立ち入らず，ハードウェアが設計できるようになることに重点をおきます．

5　はじめての SystemVerilog・FPGA

それでは，ごく簡単な回路を例にしてツールの基本的な使い方を見ていきましょう．

5.1　SystemVerilog ファイルの記述

まず，練習用のディレクトリを作ります．一般にハードウェア設計用ツールは，カレントディレクトリに中間ファイルをたくさん作ります．うかつにホームディレクトリで作業をすると痛い目にあいますので，新しいモジュールをつくる度に専用のディレクトリを作ってそこに移動するようにしましょう．

練習用のディレクトリに移動したらソースコード 1 のようなファイルを myand.sv という名前で記述しましょう．これは単純な and ゲートを記述したものです．

まず，最初の行と最後の行（14 行目）はおまじないについて説明します．実は SystemVerilog は未定義の変数はすべて 1 ビットの配線とみなすという危険な文法を持っているのですが，これだとタイプミスしても文法エラーにならずデバッグ作業が大変になってしまうため，1 行目でこの機能を OFF にしています．ただし，他の人が作った別のファイルやライブラリなどは，この暗黙の変数宣言を仮定している可能性があるので，最後の行で再びこの機能を元に戻しています．

続く 3 行（2 行目〜4 行目）はコメントです．SystemVerilog では C 言語と同じように /* と */ で囲まれたところにコメントを書くことができます．また，// 以降改行までもコメントになります．

次に module 文を使って myand という名前の回路モジュールを定義しています．SystemVerilog では回路の基本単位をモジュールと呼び，モジュール同士の階層構造を用いて，より複雑な回路を表現していきます．モジュール名とファイル名は同じにすることをお勧めします（本実験では，そうしないとうまく回路ができないことがあります）．5 行目のキーワード module から始まって 13 行目のキーワード endmodule までがモジュールの範囲です．C 言語に慣れていると，つい endmodule の後にセミコロン（;）をつけたくなりますが，こうするとエラーになってしまいますので注意が必要です．

モジュール名の括弧の中（7 行目から 9 行目）では，そのモジュールの入力（input）と出力（ouput）を宣言します．ここでは，入力として in1 と in2，出力として out1 をそれぞれ配線（wire）として宣言しています．SystemVerilog では，C 言語と同じように好きな場所で改行を行ったり，1 行に複数の文を書いたりできます．また，

```
input wire in1, in2,
```

のようにカンマ（,）を用いて，同じ属性の複数の入力や出力を一緒にで宣言することもできます．この例のようにビット幅を特に指定しない場合には，入力・出力ともそれぞれ 1 ビットの端子として扱われます．

さて，12 行目の assign 文が，この回路モジュールの動作を表現しています．C 言語からの類推で，出力 out1 に入力 in1 と in2 の論理積（AND）が出力される様子が分かると思います．演算子 & は C 言語と同じようにビット演算（AND）を表しています．以下に SystemVerilog でよく用いられる演算子を示します．ほとんど C 言語と同様であることが分かると思います．

- 算術演算： +（加算），-（減算），*（乗算），/（除算），**（べき乗算）
- 比較演算： <（小なり），<=（以下），>（大なり），>=（以上）
- 等号演算： ==（等しい），!=（等しくない）
- 論理演算： &&（論理積），||（論理和），！（否定），
- ビット演算： ~（NOT），&（AND），|（OR），^（XOR），~^（XNOR）
- 条件演算： ?:（条件式 ? 真のときの式 : 偽のときの式）
- シフト演算： >>（右論理シフト），<<（左論理シフト）

また，C 言語は「=」は代入の意味で用いられますが，SystemVerilog の assign 文における「=」は，代入というよりも，むしろ「接続する」とか「出力する」いう意味でとらえるべきです．C 言語における代入文は上から順番に実行されていきますが，SystemVerilog の assign 文はハードウェアの構造を示すもので，「いつ実行される」という概念がありません．常に右辺の信号変化が左辺に伝達されることを示しており，文法的には「継続的代入」と呼ばれます．

5.2 シミュレーション

記述した SystemVerilog ファイルから実際にハードウェアを作り出す前に，論理シミュレータでその動作の確認を行うのが普通です．通常，実機での動作確認を行うには FPGA へのコンパイルなどに長い時間がかかり，効率的な不具合の修正が困難です．これに対してシミュレーションには FPGA へのコンパイル作業が必要なく，効率よく機能検証が行えます．また，実機では難しい内部信号の観測が簡単というメリットもあります．

シミュレーションによる機能検証の基本は，

ソースコード 2　sim_myand.sv

```
1  `default_nettype none
2  `timescale 1ns/1ps
3  // Simulation module for myand
4  module sim_myand();
5     logic in1, in2;
6     wire  out1;
7
8     myand myand_inst(.in1(in1), .in2(in2), .out1(out1));
9
10    initial begin
11       in1 <= 1'b0;
12       in2 <= 1'b0;
13       #10
14         print();
15       in2 <= 1'b1;
16       #10
17         print();
18       in1 <= 1'b1;
19       in2 <= 1'b0;
20       #10
21         print();
22       in2 <= 1'b1;
23       #10
24         print();
25       $finish;
26    end
27
28    task print();
29       $write("in1=%b in2=%b out1=%b\n", in1, in2, out1);
30    endtask
31 endmodule
32 `default_nettype wire
```

- 外部から入力を与えること
- 正しい出力が得られるかを観測すること

です．このシミュレーションの作業自体を，SytemVerilog の言語の体系の中で行うことができます．具体的には，シミュレーション用のモジュールを作成し上記の 2 つの処理を行います．

5.2.1　シミュレーションモジュールの記述

先の AND 回路のためのシミュレーションモジュールの記述例（sim_myand.sv）をソースコード 2 に示します．先ほどの回路記述に比べるとやや複雑ですが，順に内容を説明します．

■暗黙の配線定義の無効化・有効化（1 行目・32 行目）　最初と最後の行は，先ほどの回路記述と同様です．未定義の変数が 1 ビットの配線と解釈される機能を無効化するとともに（1 行目），他の人が作った別の設計ファイルに悪影響を与えないように最終行でこれを元に戻します（32 行目）．

■タイムスケール（2 行目）　シミュレータが時計を「1」進めたときに，それが実時間では何の単位に相当するかを設定しています．この例では，「`timescale 1ns/1ps」と設定することで，シミュレータ時間の「1」

が 1 ナノ秒（10^{-9} 秒）に対応することを設定しています．また，シミュレーション時間の解像度を 1 ピコ秒（10^{-12} 秒）に設定しています．

■コメント（3 行目） C++ や最近の C 言語と同様に，// から改行まではコメントになります．

■モジュール名と入出力（4 行目） モジュール名を sim_myand としています．また，シミュレーションモジュールはいわば仮想的なモジュールで，入力も出力もありませんので，中括弧の中には何も記述していません．

■検査対象への入力用変数（5 行目） 検査対象のモジュール（先ほどの myand モジュール）に，入力を与えるための変数を定義しています．これらの変数には，このシミュレーションモジュール内で，次々に色々な値を代入していきます．これはプログラミング言語での変数への代入と同じイメージで，SystemVerilog の文法では「手続的代入」と呼ばれています．wire 型の配線には手続的代入を行うことはできません．そこで，これらの変数の型は wire ではなく logic にしています．

■検査対象からの出力用配線（6 行目） 検査対象のモジュール（先ほどの myand モジュール）から出力される信号線を接続する配線を定義しています．こちらは，このシミュレーションモジュール内で値を直接代入することはなく，検査対象のモジュールの出力が継続的に接続されることになります．そこで，型を配線（wire）としています．

■検査対象モジュールの実体化（8 行目） 検査対象のモジュール（先ほどの myand モジュール）を実体化（インスタンシエーション）しています．つまり，モジュール定義から実体（インスタンス）を作り出して，名前（インスタンス名）を与えます．型から変数を作り出したり，クラスからオブジェクトを作り出すことをイメージするとわかりやすいと思います．ここでは，myand というモジュール定義から myand_inst という名前のインスタンスを作り出しています．

また，中括弧の中はインスタンスの入出力ポートと，シミュレーションモジュール内部の変数や配線の接続を示しています．例えば，

```
.in1(in1)
```

で in1 という名前のポートに in1 という名前の変数を接続することを示しています．この場合はどちらも同じ名前ですが，ドットがついているのがインスタンスのポート名，括弧の中にあるのがシミュレーションモジュール内の変数名です．

■シミュレーション動作の initial ブロック（10〜26 行目） 具体的なシミュレーションの動作を記述しています．initial 文はシミュレーション開始時に一度だけ，begin から end までの文を順番に実行します．この例では，10 行目から 26 行目までが 1 つの initial ブロックになっています．

■手続的代入（11 行目，12 行目など） initial ブロックの中では C などのプログラミング言語のように，「順番に実行する」という概念があるため，「手続的代入」が行われます．手続的代入は，

```
in1 <= 1'b0;
```

のように，左辺の logic 型の変数に対して「<=」を用いて記述できます．この例の右辺は「1 ビットの 0」を示しています．

■定数の表記（11 行目，12 行目など）　ハードウェア設計ではさまざまなビット幅の信号を扱います．定数の記述にもビット幅を示す文法が備わっており，「'」の左側に書くことがきます．「'」の右のアルファベットは定数を何進数で表記されているかを示します．b（2 進数），o（8 進数），d（10 進数），h（16 進数）が使えます．

SystemVerilog では代入の左辺と右辺のビット幅が異なってもエラーにはならず，文法で定められたルールに従って自動的にビット幅を変換されます．また，ビット幅を省略してもルールに従ってビット幅が推定されます．ただ，意図しない変換によるバグを防ぐためにも，ビット幅は省略しないほうが良いでしょう．ただし，例外的な記法として，すべての全ビットが 0 であることを示す「'0」と全ビットが 1 であることを示す「'1」という記述も可能になっています．

■シミュレーション時刻制御（13 行目，16 行目など）　前述のように initial ブロックの中では，文を「順番に実行する」という概念があるため，文と文の間に時計経過の概念を挿入することが可能です．この例では，「#10」のようにしてシミュレーションの時計を 10 ステップ進めます．2 行目でシミュレーションの時計の 1 ステップを 1 ナノ秒としていますので，「# 10」10 ナノ秒シミュレーションの時刻を進めることを意味します．

■タスク呼び出し（14 行目，17 行目など）　C 言語での関数呼び出しと同様で，ここでは print という名前のタスクを呼び出しています．この print タスクは 28 行目から 30 行目で定義されおり，変数や配線の値を画面に出力する処理が記述されています．詳しくは後述します．

■シミュレーションの終了（25 行目）　$ で始まる名前のタスクはシミュレータが解釈する特殊なタスク（システムタスク）です．$finish はここでシミュレーションを終了することを示します．

■表示用のタスク記述（28 行目〜30 行目）　検証対象モジュールの入力と出力を画面に表示するためのタスクです．タスクとは C 言語における返戻値のない関数のことだと考えるとイメージしやすいと思います．task の後がタスク名で，ここでは print という名前のタスクを定義しており，entask までがタスクの記述です．$write は C 言語の printf 関数とほぼ同じ機能のシステムタスクでが，printf には関数にはない「%b」という 2 進数で表示するためのフォーマット指示子が使えます．

5.2.2　シミュレーションの実行

本実験では SystemVerilog のシミュレーションに Modelsim Altera Starter Edition を用います．シミュレータを起動するには，いくつかのコマンドを実行する必要がありますが，本実験で配布している Makefile を使えば，

```
% make sim
```

とすることで簡単にシミュレーションを実行できます[*2]．うまくいけば画面に下記のようなメッセージが表示されます．

[*2] 本稿では，コマンドの実行例を示すときにはシェルのプロンプトを % で表すことにします．したがって，先頭の % を入力する必要はありません．この例では make sim の m から打ち込めば OK です．

```
# vsim -do {add wave -r /sim_myand/myand_inst/*; run -all} -l sim_myand.log -c -wlf sim_myand.wlf
sim_myand
# Loading sv_std.std
# Loading work.sim_myand
# Loading work.myand
# add wave -r /sim_myand/myand_inst/*
#  run -all
# in1=0 in2=0 out1=0
# in1=0 in2=1 out1=0
# in1=1 in2=0 out1=0
# in1=1 in2=1 out1=1
# ** Note: $finish    : sim_myand.sv(25)
#    Time: 40 ns  Iteration: 0  Instance: /sim_myand
```

細かなメッセージの違いについては気にしなくて結構ですが，4 通りの in1 と in2 の値の組合わせに対して，出力 out1 が正しく AND として機能しているかを確認してください．なお，このシミュレーションのメッセージは自動的に sim_myand.log ファイルにも保存されます．

5.2.3 波形の確認

上記のようにシミュレーションを実行すると，sim_myand.wlf というファイルも生成されます．これはシミュレーションの波形データが格納されたファイルです．この波形データを表示して確認するためには，

```
% vsim sim_myand.wlf &
```

とします．このシミュレーションはレジスタトランスファレベル（RTL: Register Transfer Level）シミュレーションと呼ばれ，入力が変化してから出力が変化するまでのゲート遅延や配線遅延が考慮されていない点に注意して下さい．

5.3 FPGA への実装

5.3.1 コンパイル

では，いよいよ SystemVerilog 記述から論理合成をして回路をつくってみましょう．

合成した回路を FPGA 上に実現するには，FPGA の内部に回路をどう配置し，どう配線するのかを決める必要があります．この作業を配置配線と呼びます．配置配線が済むと FPGA 回路情報のデータファイルが出来上がります．このデータを FPGA のコンフィギュレーションデータと呼びます．コンフィギュレーションデータを FPGA に書き込んで実際に回路を構成することを「FPGA をコンフィギュレーションする」といいます．

本実験では Altera 製[*3]の FPGA を使います．このため，論理合成・配置配線は Altera の設計ツールである Quartus II を使います．さて，以上のように SystemVerilog の記述から FPGA に回路を構成するには多くのステップが必要ですが，本実験では make コマンドを使ってこれらのことを一気に行います．まず，設計した SystemVerilog ファイルのあるディレクトリに，make コマンドを制御するための Makefile，Quartus II に FPGA の部品名やピンの配置などを指示するための tcl ファイル（ここでは myand_top.tcl），Quartus II に回路の遅延制約を与えるための SDC（Standard Delay Constraints）ファイル（ここでは myand_top.sdc）実

[*3] 現在，Altera は Intel 社のブランドです．

ソースコード 3　複数の出力があるモジュール（two.sv）

```
1  `default_nettype none
2  /*
3   * Sample with two output ports
4   */
5  module two
6    (
7     input wire  in1,
8     input wire  in2,
9     output wire out1,
10    output wire out2
11    );
12
13    assign out1 = in1 & ~in2;
14    assign out2 = in1 |  in2;
15 endmodule
16 `default_nettype wire
```

験ボードのインタフェース回路の Verilog 記述（ここでは myand_top.v），を用意します．これらのファイルの中身を理解する必要は全くありません．おまじないだと思って下さい．

これらを用意したら，make コマンドを実行します．

```
% make
```

実行にはしばらく時間がかかりますが，拡張子が.rbf のコンフィギュレーションデータ（ここでは myand_top.rbf）が生成されれば成功です．

作業ディレクトリには，途中結果のレポートファイルなどがたくさん生成されます．エラーなどが生じて合成が止まったときなど，これらの中間ファイルをすべて消去したい場合には，

```
% make distclean
```

と打ち込んでやります．

5.3.2　FPGA のコンフィギュレーション

いよいよ実験ボードの FPGA に回路を構成してみましょう．実験ボードと PC を USB ケーブルで接続し，ボードの電源を入れます．その後，

```
% make config
```

と打ち込むと，USB ケーブル経由でコンフィギュレーションデータがボードに送られ回路が構成されます．ボードのスイッチと LED（発光ダイオード）で回路の動作を確認しましょう．

6　練習： 複数の出力がある場合

それでは，新しいディレクトリに移動して別の回路で今までの流れを練習してみましょう．AND 回路の例ではモジュールの出力は 1 つだけでしたが，複数の出力を持つ回路モジュールを設計してみます．SystemVerilog

の記述はソースコード 3 の様になります．

ソースコード 1（myand.sv）と異なり，assign 文が 2 つ用いられています．C 言語などのプログラミング言語は，基本的に個々の文は書いてある順番に実行されますが，前述のように継続的代入には「順番に実行」の概念はありません．並記された文は同時（独立）に常に成立していると解釈するのが自然です．したがって，2 つの assign 文の順番を入れ替えて記述しても，シミュレーションの結果や合成するハードウェアには影響はありません．