1996年 7月 6日
東海大学 通信工学科 清水尚彦

ハードウェア記述言語と論理シミュレーション

ハードウェア記述言語である VHDL の簡単な入門説明と VHDL のサブセットを使用する LSI CAD ALLIANCE を用いた論理シミュレーション手順について説明する.

VHDL 言語

VHDL はアメリカ政府の超高速 LSI 設計のプロジェクト VHSIC 計画の仕様記述の共通化のために設計されたハードウェア記述言語である.[1] さまざまな LSI の構造や機能をこの言語だけで記述する事を目的として言語仕様が設計された結果, 非常に多くの機能を持つ言語となった. また, 大規模なプロジェクトをサポートするための言語構造が同じくアメリカ政府の主導で設計されたソフトウェア言語 Ada から導入され階層的な設計に適した言語となっている. このため, ハードウェア設計のための言語として VHDL を眺めると不要な機能が多く, ほとんどの CAD ベンダーは VHDL をフルサポートするのではなく, その一部を使って論理設計を行う枠組を作っている.

LSI CAD ソフトウェアはその成立ちからハードウェアメーカーへの販売を仮定しているため高価なものが多いのであるが, 日本以外の主要先進国やアジアの各国では大学において LSI を設計する事が普通に行われている結果, 幾つかの LSI CAD は無償か手数料程度で入手できるようになっている.

Alliance VHDL

標準ハードウェア記述言語である VHDL をサポートする LSI CAD で無償で配布されているものの一つにフランスのマリー・ピエール・キューリー大学の開発した Alliance という CAD ソフトウェアキットがある. このソフトは VHDL の非常に制限された機能だけを使って LSI の設計を行うものであるが, 大規模な LSI を設計した実績もありハードウェア記述から LSI レイアウトまでの LSI 設計のほぼ全工程に対するツールが完備されているのでこれだけあれば LSI の設計が可能である. ALLIANCE がサポートする VHDL 言語は構造記述と動作記述である. VHDL の標準外では FSM とデータパスコンパイラもサポートしているが, 本書では触れない. 興味のある人は ALLIANCE のドキュメントを参照してもらいたい. 一般の VHDL の構文については市販の本が各種出ているのでそれらを参考にしていただきたい. ここでは, Alliance の使用方法の説明に必要な最低限の構文について説明する.

VHDL の構文は主として `ENTITY` と `ARCHITECTURE` という 2 つの部分からなり, それぞれその VHDL で宣言されているモジュールの外部仕様と内部仕様を記述する. `ENTITY` では, 外部と接続する信号線を定義し, `ARCHITECTURE` では内部信号及び動作/内部構造を記述する. 信号はプログラム言語でいうところの変数に近い概念としてとらえる事ができ, 型を持つ. 下表の信号の型が Alliance ではサポートされている.

型	説明
`BIT`	1 bit の 2 進数
`BIT_VECTOR`	`BIT` の配列
`REG_BIT`	ラッチやフリップフロップなどの状態素子. この型の宣言には別途 register という修飾子をつける必要がある.
`REG_VECTOR`	`REG` の配列
`MUX_BIT`	双方向信号. この型の宣言には別途 bus という修飾子をつける必要がある.
`MUX_VECTOR`	`MUX` の配列

[1] 現在では IEEE によって標準化がなされており, 執筆時点の最新の標準は 1993 年に制定されている.

演算子としては次のものがサポートされている.

演算子	説明
not	否定：後置された論理式の否定を値とする.
and	論理積：2 つの論理式の論理積を値とする.
or	論理和：2 つの論理式の論理和を値とする.
xor	排他的論理和：2 つの論理式の排他的論理和を値とする.
nor	論理和の否定：2 つの論理式の論理和の否定を値とする. or の論理式を not で否定したものと等価.
nand	論理積の否定：2 つの論理式の論理積の否定を値とする. and の論理式を not で否定したものと等価.
&	ビット連結：2 つのビット列を連結したビット列を値とする. 本演算子は配列宣言された信号への代入操作において, 異なる長さの信号の代入を行うために用意されている.
=	等値：2 つの論理式が等値であるときに真を値とする.
/=	非等値：2 つの論理式が非等値であるときに真を値とする.

また, 信号への代入は<=で表す. ハードウェアではプログラム言語における変数への代入と異なり, 代入の記号は配線を表しているので同一の信号に対して複数回の代入はできない事に注意が必要である. また, when による条件付の代入も許されており, 例えば

```
 a <= b or  c WHEN i=0 else
      b and c;
```

のように条件によって代入するものを切替え可能である. 図 1にこれらの演算子を使った VHDL 記述の論理回路を示す. 図中で使っている--は VHDL では同一行の本記号以降の記述がコメントである事を示している. この論理回路は 3 入力 1 出力の組み合わせ回路であり, 図に示した論理式は冗長な論理式になっている.

演習 図の論理式を簡単化せよ.

```
ENTITY test IS      -- モジュール"test"の実体は...
PORT (
  a  : IN BIT;      -- 入力信号 a
  b  : IN BIT;      -- 入力信号 b
  c  : IN BIT;      -- 入力信号 c
  f  : OUT BIT      -- 出力信号 f
  );
END test;           -- 実体の定義終了
-- モジュール"test"のデータフロー構造は
ARCHITECTURE data_flow OF test IS
BEGIN
  f <= (a and b and c) or (a and not c) or b; -- 出力 f の定義
END data_flow;      -- データフロー構造定義終了
```

図 1: Alliance による簡単な VHDL 記述の例

代入文のもう一つの型として, with, select よる条件付代入がある. これは, 前出の when のみによる条件付代入と本質的には同等であるが, with によって条件の比較対象を明示するので条件が数多くある場合などに記述量の削減と読みやすさの向上を計れる.
演習 次の代入文を with,select を使わずに記述せよ.

```
WITH current(1 DOWNTO 0) SELECT
next(1 DOWNTO 0) <=
        "0" & x_bottum WHEN "00",
        "10"           WHEN "01",
        "11"           WHEN "10",
        "00"           WHEN "11";
```

次に, VHDL によるラッチ, フリップフロップの記述について説明する. 順序回路を構成する場合, ラッチ, フリップフロップは必要欠くべからざる構成要素であり, 当然 VHDL にもこれらの回路素子を記述する方法が用意されている. 論理回路においてラッチやフリップフロップは通常回路内で接続される. そこで, 内部信号としてラッチ, フリップフロップを定義しておく必要がある. Alliance ではどちらも reg_bit register という型にて宣言する. また, これらの素子への値の設定は固定的な配線論理とは事なり条件を満たした時のみに値を取り込むようになっている. そのため, VHDL では block 文が用意されており, 条件付の代入がなされる. また, guarded はこの代入が恒久的な値の設定ではなく block 文の条件節による監視下にある事を示している. また, block 文の名前は一つのファイル内でユニークでなくてはならない.

```
...
     signal Reg : reg_bit register; --フリップフロップ Reg の宣言
     begin
...
     flip_flop : block (ck = '1' and not ck'STABLE) -- ck が 1 でかつ安定でない場合
     begin
       Reg <= guarded Din; -- 入力信号 Din の値を出力 Reg に設定する.
     end block;
```

図 2: Alliance によるフリップフロップ記述の例

```
...
     signal Lat : reg_bit register; --ラッチ Lat の宣言
     begin
...
     latch : block (ck = '1') -- ck が 1 の場合
    begin
       Lat <= guarded Din; -- 入力信号 Din の値を出力 Lat に設定する.
     end block;
```

図 3: Alliance によるラッチ記述の例

VHDL 入力

VHDL 入力には vi, emacs 等の unix のテキストエディタを用いる. UNIX のファイルの扱いやテキストエディタの使い方が分からない人は計算機センターに常備してあるドキュメント類を参照して自習する事. VHDL 動作記述の入力には拡張子として"vbe"のファイルを作成する事. シミュレーションの入力ファイルは拡張子"pat"のファイルが必要となる.

VHDL コンパイルおよびシミュレーション

VHDL コンパイルおよびシミュレーションは asimut というツールにて行う. asimut の動作の為には以下に示す環境変数の設定が必要となる. csh のユーザーを前提に記述するが他のシェルのユーザーは等価な操作を行う事.

```
setenv TOP /usr/local/lsi-cad/alliance
setenv MACHINE ews
set path = ($TOP/archi/$MACHINE/bin $path)
```

入力した VHDL 記述の構文を確認するためには
`asimut -b -c` *file_name*
とする. ここで, *file_name* には拡張子を含まない.
このファイルを論理シミュレーションにより確認するためにはまず, シミュレーションテストベクトルファイルを用意する必要がある.

```
in  a B;; -- 入力 a を 2 進数表示
in  b B;; -- 入力 b を 2 進数表示
in  c B;; -- 入力 c を 2 進数表示
out f B;; -- 出力 f を 2 進数表示

begin
test0   : 0 0 0 *;
        : 0 0 1 *;
        : 0 1 0 *;
        : 0 1 1 *;
        : 1 0 0 *;
        : 1 0 1 *;
        : 1 1 0 *;
        : 1 1 1 *;
end;
```

図 4: シミュレーションテストベクトルの例

図 1を確認するシミュレーションテストベクトルファイルは例えば次のように作成する. テストベクトルファイルは回路の信号と入出力関係, 及びファイル中の数値の基数を記述する信号定義部と, 回路に与えるもしくは回路からの応答を示すベクトルデータ部からなる. 基数としては 2 進, 8 進, 1 6 進が使え, 各々 B,O,X と表す. テストベクトル部はシミュレーションで外部から与える信号は直接数値を書き, 回路応答を観察する信号は*を書く. 尚, *は信号が配列からなる場合与えられた基数で表示できる桁数分書く必要があ

る (例えば, 8ビットの信号を１６進で記述する場合 2 桁必要なので, ベクトルファイルには**のように 2 桁記載する. このテストベクトルファイルと前出の VHDL 記述を論理シミュレーションで確認するためには

`asimut -b` *vhdl_file vector_file result_file*

とする. ここで, *file_name, vector_file, result_file* には拡張子を含まない.

演習 図 4のテストベクトルで図 1の論理回路の動作を確認せよ.

信号機の論理回路におけるシミュレーション実行例

「コンピュータ回路」の授業において扱った FSM による信号機の設計を VHDL で記述した例題を bosei に用意している.

次のように一連のコマンドを実行する事によって論理シミュレーションが実行できる.

```
cd /usr/local/lsi-cad/alliance/tutorials/signal   ;;ディレクトリの移動
make home             ;;ホームディレクトリへのファイルのコピー
cd                    ;;ディレクトリの移動
cd signal             ;;ディレクトリの移動
ls                    ;;ファイルの確認
make sim              ;;シミュレーションの実行
ls                    ;;ファイルの確認
more result.pat       ;;結果の表示
```

演習 シミュレーション結果を見て信号機が期待通り動作しているか確認せよ.

論理最適化, 論理合成

VHDL ツールの一つに論理合成/最適化ツールがある. 論理合成とは動作記述された論理ファイルをゲートの接続情報に展開するものである. 最適化は与えられた論理ファイルをゲート数や遅延時間を低減するために論理変更するものである.

先ずは, 最適化の動作を見て見よう. 図 1の回路は冗長な構成になっていると述べたがこれを最適化ツールにてゲート数の削減を計る.

`logic -o` *vbe_file result_file*

とする. ここで, *vbe_name, result_file* には拡張子を含まない.

演習 自分で最適化した結果とコンピュータによって最適化されたファイルを比較せよ.

また, 論理合成とは動作記述で記述された論理回路をゲートレベルの接続情報に変換するものである. 上記信号機のシミュレーション環境において`make vst` と入力する事によって signal のゲート接続情報ファイルが生成される. このファイルを調べるには`ls work` と入力すると展開されたファイルの一覧表が得られ, `more work/signalo.vst` と入力する事によって[2]構造記述に変換された signal の論理ファイルが参照できる.

最後に図 5に信号機の VHDL 記述例を示す.

[2]尚, `more` の終了は`q` と入力する

```
-- 押しボタン式信号器の制御回路
-- 自動車信号出力: h_red, h_yellow, h_blue
-- 歩行者信号出力: x_red, x_blue
-- 押しボタン入力: x_bottum
ENTITY sgnl IS
PORT (
  ck  : IN BIT;
  x_bottum : IN BIT;
  h_red, h_yellow, h_blue, x_red, x_blue : OUT BIT;
  );
END sgnl;
--
ARCHITECTURE data_flow OF sgnl IS
SIGNAL state : REG_VECTOR ( 2 DOWNTO 0 ) register;
BEGIN
  st: BLOCK (ck = '1' AND NOT ck'STABLE)
  BEGIN
    WITH state(2 DOWNTO 0) SELECT
    state(2 DOWNTO 0) <= GUARDED
"00" & x_bottum WHEN "000", -- h_bule,x_red
"010" WHEN "001", -- h_yellow,x_red
"011" WHEN "010", -- h_red,x_blue
"100" WHEN "011", -- h_red,x_blue
"101" WHEN "100", -- h_red,x_blue
"110" WHEN "101", -- h_red,x_red
"111" WHEN "110", -- h_bule,x_red
"000" WHEN "111"; -- h_bule,x_red
  END BLOCK;
    h_red    <= state = "010" OR state = "011" OR state = "100"
             OR state = "101";
    h_yellow <= state = "001";
    h_blue   <= state = "000" OR state = "110" OR state = "111";
    x_red    <= state = "000" OR state = "001" OR state = "101"
             OR state = "110" OR state = "111";
    x_blue   <= state = "010" OR state = "011" OR state = "100";
END data_flow;
```

図 5: 信号機の論理回路の例