Unconventional HDL Programming

中里直人 (20090425 version)

1 Introduction

HDL で浮動小数点演算回路を設計するためには、それの基盤となるべきなんらかの参照実装が必要になる。ここでは、それがなぜかを説明したい。

HDL は Hadware Description Language という名はついているが、いわゆるプログラミング言語とは異なる性質を持つ。通常のプログラミングに慣れ親しんだ人にとって、一番の困難点はデバッグが非常に面倒ということである。そもそもデバッグとは、通常のプログラミングにおいても簡単なことではない。最も安易な方法としてよく使われるのは、printf デバッグという手法である。説明するまでもないが、これは問題の発生箇所を発見するために、適当な場所に変数の値を出力するような printf 関数 (C 言語の場合。他の言語はそれ相当のもの) を挿入し、変数の変化を「観測」することである。HDL ではこの方法が使えない[1]。

なにも出力ができない「プログラム」に意味があるのかと問われると、意味はありません、というのが通常のプログラミングをする人の答えだろうと思う。しかし、HDL から生成されるのはプログラムではなく「回路」である。プログラムのように回路を設計できる、というのが HDL のふれこみであり、そのおかげで、さらに困難な図による回路設計を回避できるようになったのは大きな進歩であった。では、HDL から生成された回路は、どうやってデバッグしたらよいのだろうか？そもそも、HDL から生成された回路を動作させるとはどういうことなのだろう？

回路には入り口と出口がある。最も単純な電気回路の例を考えよう。豆電球は片方の足を電池の＋側に、もう片方を－側に接続することで光る。豆電球の入り口は＋側の足であり、出口は－側の足である。HDL から生成された回路も、同様に入り口と出口にに何かを接続し、入り口から何かを注ぎ込んでやれば「光る」ことができる。結果として、出口からは何かが出てくる。HDL から生成された回路を動作させるとは、文字通りにこれをおこなうことである。信じがたいかもしれないけれどこれは本当である。以下、例をしめす。

[1]実際には回路設計とは無関係な拡張を使うことでできなくもないがそれはさておき

2 論理シフト

2.1 VHDL の肝を概説

Listing 1: 右論理シフトの VHDL ソース

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity rshift_5 is
6    port (
7      i : in  std_logic_vector(9 downto 0);
8      o : out std_logic_vector(9 downto 0)
9    );
10 end rshift_5;
11
12 architecture rtl of rshift_5 is
13 begin
14   o <= "00000"&i(9 downto 5);
15 end rtl;
```

最も単純な HDL による回路の例として、右方向への論理シフトを考える。その VHDL によるソースコードは図 1 のようになる。右論理シフトを例として掲げたのは、浮動小数点加算の実装で、小さい方の仮数部のを指数部の差の分だけ右論理シフトする、という操作が必要だからである。これは、簡単のために入力を 5 bit 右にシフトする回路をしめしている。

細かい文法については説明しないが、最低限以下の点を理解しないと意味が不明かもしれない。

- この回路の名前は `rshift_5` である
- 入力は `i` という「変数」
- 出力は `o` という「変数」
- 実際のシフト処理は 14 行目でおこなわれれいる

ここでわざわざ「変数」としたのは、通常のプログラムにおける変数とは異なるからである。VHDL にはいわゆる変数がないと思って欲しい。なので、本当は `i` も `o` も変数ではない。VHDL で変数のようなものが出てきたときには、それは線の束だと思って欲しい。

よって 7 行目は、通常のプログラミング言語からの推論では、入力引数を宣言しているようだと考えてしまうかもしれないが、VHDL ではこれは「`i` という名の 10 本束の入力線」という意味となる。同様に 8 行目は、「`o` という名の 10 本束の出力線」という意味となる。なぜ 10 本なのかは、`(9 downto 0)` で表現されている。「9 から 0 までの線を宣言する」というような意味あいである (C 言語の配列と同様 0 から数える)。`std_logic_vector` は、さしあ

たっては線の種類の指定と思ってもらいたい。その他の部分は、とりあえずは理解する必要はなく、決まり文句だと考える。

問題はなぜ14行のように書くと右シフトが達成されるのかということである。これは`i`や`o`が線の束であることを考慮すると見えてくるかもしれない。実は`<=`はVHDLにおける代入演算子である。ややこしいことに、言語定義上は代入演算子であるが、実際には`i`や`o`は変数ではないので、代入がおこなわれるわけではない。実はこの`<=`によって、線の束同士の接続がおこなわれるのである。`o`は10本からなる線の束であるから、そこに接続するものも10本からなる線の束でなくてはならない。そして`i(9 downto 5)`は、10本からなる線の束である`i`のうち、9番目から5番目の線の束を表す。

仕様から「`o`は`i`を右に5 bit論理シフトした線の束」になるべきなので、`o`の4から0番目の線の束に`i`の9番目から5番目の線の束を接続すればよい。そして、余った`o`には0がでてくるようになっていればよい。14行はこの接続関係を記述している。実は`"00000"`は定数のようなもので、実際にはつねに0がでてくる線の束をあらわす。0が5個あるからこれは5本の束をあらわす。どこから0がやってくるのかは、眠れなくなるので考えないで欲しい。

VHDLにおける変数のようなものが線の束であるということを明示的に表すこともできる。10本からなる線の束である`o`のn番目の線は`o(n)`と表現される。よって、14行を明示的に一本一本接続した場合以下のようになる。

Listing 2: 右論理シフトの接続

```
1   o(9) <= '0';
2   o(8) <= '0';
3   o(7) <= '0';
4   o(6) <= '0';
5   o(5) <= '0';
6   o(4) <= i(9);
7   o(3) <= i(8);
8   o(2) <= i(7);
9   o(1) <= i(6);
10  o(0) <= i(5);
```

このような記述も全く正しいVHDL記述である。図1の14行を、この記述とそっくり置き換えても動作としては同じになる。が、このように全ての接続をいちいち明示的に書く必要があるのなら、回路図で設計するのと手間が変わらないような気分となる。なので、これを図1の14行のように省略して書くことができる。また、このように明示的に表現することで、`o`には`i`を5 bit右論理シフトした何かが出てきそうであるとわかってもらえると思う。

2.2 回路を光らせる

回路に明かりを灯すためには、入力と出力に適切ななにかを接続してやり、入力から何かを流し込んでやる必要がある。こうすることを、HDLの言葉では「テストベンチ」を実行すると呼ぶ。どう呼ぶのかにかかわらず、テストベンチの実行は、普通のプログラミングに

おけるプログラムの実行と等価である。そのためには、HDL で記述された回路を実行するための環境が必要であり、これを HDL のシミュレータと呼ぶ。

2.2.1 GHDL のインストール

HDL のシミュレータには様々あるが、ここでは簡単に使うことのできる GHDL と呼ばれるプログラムを使う。GHDL は、VHDL で記述された回路を、CPU で実行可能なファイルに変換してくれる。結果として、回路を「実行する」ことが可能になる。実際には、この実行により回路の論理シミュレーションをおこなうこととなる。

以下、実行環境として Linux を想定する。CPU は x86_64 とし、Linux distribution としては、Debian Lenny 以降または Ubuntu 8.04 LTS 以降を想定する。これらの distribution では、GHDL が簡単にインストールできるため。具体的には以下のコマンドにより GHDL と関連ファイルをインストールしてもらいたい。

Listing 3: GHDL の Ubuntu 8.04 LTS でのインストール

```
1 # aptitude install ghdl gtkwave
```

gtkwave については後ほど説明する

プレコンパイルされたものやソースは http://ghdl.free.fr/ よりダウンロードできる。ソースから build するには、Ada のコンパイラが必要であり、非常に困難であるので推奨しない。Windows/Mac OS X 用のコンパイル済み GHDL も存在するので、必要であれば検索してほしい。

2.2.2 GHDL の利用方法

GHDL ではソースの VHDL ファイルを analyze して elaborate するという、二段階で実行ファイルを生成する。analyze のためのオプションは “-a” elaborate するためのオプションは “-e”’ となる。よって、仮に “top.vhd” という VHDL のソースファイルがあったとすると、以下のコマンドにより analyze と elaborate ができる。

Listing 4: analyze と elaborate

```
1 % ghdl -a --ieee=synopsys top.vhd
2 % ghdl -e --ieee=synopsys top
3 % ./top
4 % ghdl --gen-makefile --ieee=synopsys top > Makefile
```

analyze は普通のコンパイラにおけるオブジェクトファイルの生成、elaborate は普通のコンパイラにおけるリンク処理に相当する。これにより生成される実行ファイルの名前は “top” であるから、3 行にあるように実行することができる。また、4 行には “Makefile” の生成方法をしめした。これにより “Makefile” を保存すると、make コマンドで analyze と elaborate の一連の処理がおこなうことができる。

2.2.3 テストベンチの記述

図 1 のファイルを適当な名前 (以下 “rshift.vhd” とする) で保存した上で、図 4 をまねて以下のようにしたとする。

Listing 5: rshift.vhd

```
% ghdl -a --ieee=synopsys rshift.vhd
% ghdl -e --ieee=synopsys rhisft_5
% ./rshift_5
```

この一連のコマンドは何のエラーも発生させないかもしれないが、何の結果ももたらさない。図 1 の “rshift.vhd” には、普通のプログラミング言語における main 関数がないためである。テストベンチとは、概念的には VHDL における main 関数だと思って欲しい。論理シミュレーションは計算機上でおこなわれるため、その実行には main 関数が必要である。

さて、図 5 の一連のコマンドでなにも結果を得ることができないのは、“rshift_5” という回路に何も接続していないためである。普通のプログラミングにおけるユーザーの作った関数も、一度も利用しないなら (引数を渡して結果を得ることをしないなら)、その関数は存在しないことと同等である。そのため、このテストベンチという main 関数では、“rshift_5” の入り口と出口に何かを接続する。最も単純なテストベンチを以下に示す。

Listing 6: テストベンチ top

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity top is
end top;

architecture rtl of top is

component rshift_5
  port (
    i : in  std_logic_vector(9 downto 0);
    o : out std_logic_vector(9 downto 0)
  );
end component;

signal input  : std_logic_vector(9 downto 0);
signal output : std_logic_vector(9 downto 0);

begin
  DUT : rshift_5 port map (i => input, o => output);
  input <= "0000100000";
end rtl;
```

色々と記述量が増えてきているが、重要な点のみ説明する。

- このテストベンチの名前は top である。5-6 行の entity というキーワードの付近で定義されている
- このテストベンチでは rshift_5 という component(回路) を使う
- 17-18 行では新たな線の束 input と output を宣言している
- 21 行で rshift_5 を実体化している

一番重要な行は rshift_5 の実体化をしている 21 行なので、ここのみ詳しく説明する。他の部分は、毎度ながら決まり文句と思って欲しい。

実はここで「入り口と出口に何かを接続」している。行頭の DUT は、この実体化処理につけるラベルである。普通のオブジェクト指向プログラミング言語では、オブジェクトの実体化とそれへのメッセージパッシングにより、プログラムを記述するが、VHDL では回路を実体化しそれへの入力と出力を規定することで回路を記述する。オブジェクトの実体化では、オブジェクトを保持する変数が利用されるのに対して、VHDL には変数がないので、その代わりにここでラベルをつける必要がある。ラベルがないと、複数の回路の実体を言語上で区別することができなくなるためである。

:の区切りの次が、実体化する回路 rshift_5 の名前を指定している。port map はキーワードで、この後で入出力を規定しますよ、という宣言である。(i => input, o => output) の部分で、rshift_5 の入力 i と出力 o に、17-18 行で宣言されたテストベンチ側の線の束を接続している。それぞれ i => input と o => output が対応する。この記述は線の束を接続する演算子である<=と逆向きな、=>という演算子を使っている。直感的には何かおかしい気もするが、意味するところは明確だと思う。

注意してほしいのは、この main 関数は論理シミュレーションの時にのみ必要となることである。現実に VHDL で記述された回路を FPGA なり ASIC 上に実現する場合、main 関数の記述は必要ない。これは、FPGA なり ASIC 上に実現される回路では、回路への入出力は FPGA なり ASIC の現実のピンに接続されているはずであり、そのピンへには現実のデジタル回路からの入出力 (基板上の信号や PCI-Express からの信号など) ピンが接続される。そのため、現実の環境が main 関数になる、と言うこともできる。

2.2.4 テストベンチの実行

図 6 を “top.vhd” として保存し、“rshift.vhd” と同じディレクトリにおいて、以下の一連のコマンドを実行することで、このテストベンチが実行できる。

Listing 7: テストベンチの実行

```
% ghdl -a --ieee=synopsys rshift.vhd
% ghdl -a --ieee=synopsys rshift.vhd
% ghdl -e --ieee=synopsys top
% ./top
```

それぞれのファイルを analyze したうえで、テストベンチの entity 名 (図 6 の 5-6 行で指定) を elaborate することで、“top” という実行可能ファイルが生成される。しかし、このままでは、やはりなんの結果もたらさない。テストベンチは実際に動作しているが、出力の手段を指定していないため、このように実行しても実質的な意味はない。VHDL ファイルに文法的な間違いがないことが確認できるのみである。実際、よくある誤りは、<=の左右で線の束の本数があってない場合、実行時 assertion によりエラーとなり途中実行が終了する場合がある。

では、テストベンチを意味のあるように実行すればどうすればよいのだろうか？GHDL で生成した実行ファイルには、いくつかの実行時オプションが自動的に付与される。そのひとつが以下のようなオプションである。

Listing 8: 信号を保存するテストベンチの実行

```
1 % ./top --vcd=sim.vcd
```

--vcd=オプションは、指定されたファイルにテストベンチの信号データを保存する。ここでは sim.vcd というファイルが指定されている。また、信号データとは、“top.vhd” とそれが実体化している “rshift.vhd” に含まれる各 “変数” の値のことである。VHDL には普通のプログラミングのような変数はなく、それらは「線の束」であると説明したが、その線に流れているのが信号である。VHDL が表現するものはプログラムではなく文字通り回路であるから、正確にはデジタル回路であるから、そこに流れるもののは信号なのである。それを踏まえて図 6 のテストベンチを見返してみると、17-18 行では signal というキーワードが使われている。ここで、まさに input や output やというシグナルの流れる線の束を利用しますと宣言しているのである。

さて、sim.vcd とは Value Change Dump(VCD) という規格のファイルであり、要は信号のデータが変化するたびにその値が保存される。このようなファイルは、波形ファイルとも呼ばれる。信号データの変化を横軸を時間としてプロットすると、波形データのようになるからである。ただし、今実行したような単純なテストベンチでは、時間的な変化がないため、波形データとはなっていない。

VCD ファイルは、GHDL と一緒にインストールした gtkwaeve をつかって、波形を表示することができる。さしあたっては、詳しい gtkwave の使い方は説明をしない。ここでは、sim.vcd を直接覗いてみよう。というのも、VCD ファイルはテキスト形式のファイルであるから、今のような単純なテストベンチから生成される VCD ファイルであれば、直接中身を見て動作を確認できる。

その前に、rshift_5 が正しく動作した場合、どのようになるべきなのかを考える。これを考えることは、普通のプログラミングにおけるデバッグでも必須であり、正しく動作する場合がわからないと、そもそもデバッグができない。rshift_5 の仕様は「入力を 5 bit 右にシフトする」ということである。図 6 のテストベンチでは、入力として 0000100000 を与えている。これを右に 5 bit 論理シフトすると 0000000001 となるはずである。つまり今の場合、output に 0000000001 という信号が流れてくれば正しいということになる。

今の場合の sim.vcd の中身を以下に示す。timestamp 以外は同じになるはずなので、実行して同じになるか確かめてほしい。

Listing 9: sim.vcd

```
$date
  Sun Apr 26 00:00:00 2009
$end
$version
  GHDL v0
$end
$timescale
  1 fs
$end
$var reg 10 ! input[9:0] $end
$var reg 10 `` output[9:0] $end
$scope module dut $end
$var reg 10 # i[9:0] $end
$var reg 10 $ o[9:0] $end
$upscope $end
$enddefinitions $end
#0
b0000100000 !
b0000000001 ``
b0000100000 #
b0000000001 $
```

VCD ファイルの正確な規格については web 上に説明があるので検索してほしい。ここでは、以下の点のみを理解すればよい。

- $var ではじまる行で信号が定義されている
- $var ではじまる行の四つ目のコラムの記号 (10 行では "!") がこの信号を表す記号である
- #0 の次の行から各信号の値が保持されている
- 各信号の値は「値 記号」の組となっている。記号は信号の定義における四つ目のコラム

以上を踏まえると、input は 10 本の線の束であり、その記号は "!" であるから、その値は 18 行にあるとおり b0000100000 であることがわかる。先頭の b はこの値が二進数であることを示す。確かに図 6 のテストベンチの 22 行で指定したとおりになっている。同様に、output の値は b0000000001 であることがわかると思う。これは想定した動作 (5 bit 右にシフト) と同じである。つまり、このテストベンチの実行により、rshift_5 は正しく動作しているようだと予測できる。

これが「ようだと予測できる」と曖昧な記述なのは、まだ入力が b0000100000 の場合しかテストしていないからである。全てのパターンをテストしない限り rshift_5 が正しく動作

するということを確信できない。今の場合、入力が 10 bit の信号であるから入力値には 1024 パターンの可能性がある。この程度のパターン数であれば、全ての場合をテストすることはそれほど難しくはないかもしれない。コンピュータを使っているのだから、シェルスクリプトや適当なスクリプト言語により、テストベンチを全パターン生成し、その全てから得られる VCD ファイルを、これも適当なスクリプトでチェックすればよい。原理的には単純な仕事になる。

VHDL にはテストベンチを記述するために、回路記述には使えない様々な拡張がある。それらの拡張には普通のプログラムと同じような変数があり、繰り返し処理なども記述できる。また、ファイルへの入出力も可能であり、いちいち VCD ファイルを生成しなくとも、回路の結果の出力を得ることもできる。ここでは、そのような手法については説明しない。通常のいわゆる「テストベンチ」では、そのような拡張を駆使してパターンのテストを巨大なテストベンチファイルでおこなう。

2.3 VHDL による回路設計と動作検証の困難

原理的にはどのような方法を使おうとも、全パターンの確認ができれば、ある VHDL で記述された回路が正しく動作すると確信をすることができる。しかし、今のような単純回路でかつ入力のパターンが 1024 しかないのであれば、どんな方法を使ってもさして手間はかからないのは確かであるが、現実的には、全てのパターンを確認することは不可能な場合が多い。また、今のような「入力を右に 5 bit シフトする」という単純な処理では、入力の全てのパターンについて想定される出力を生成することも容易である。

では、「単精度浮動小数点加算器」の回路を VHDL で記述したとして、その動作を検証するのも同様に簡単であろうか？これはそう簡単ではない。まず、単精度浮動小数点加算器が二つの単精度浮動小数点数を足して、一つの単精度浮動小数点数を得るものだとする。入力のパターン数は、例外などをとりあえず無視して数え上げると 2^{64} パターンである。これは決して少なくない数字 ($\sim 4 \times 10^{19}$) であり、現実的には短時間 (たとえば数分) で全パターンをテストすることはできなさそうである。また、単精度演算器であれば、普通の CPU には IEEE 754 形式の浮動小数点演算器が載っているので、与えられた入力の組に対して想定される出力を得ることは比較的単純である (ただし、これは IEEE 754 に完全に準拠した浮動小数点加算器を実装する場合。現実的にはこれはかなり面倒な仕事である)。

しかし、もし「できるだけ高速な計算を実現したいので、仮数部は 16 bit くらいに削った加算器を実装したい」としたらどうだろうか？このような回路を HDL で記述してその動作をテストベンチで検証するためには、テストベンチから得られる出力に対応した「想定する出力」が事前にわかっていなければならない。そのために必要になるのが、HDL の記述に対応した「シミュレータ」である。よって、実際の開発の過程は以下のようになるはずである。

1. 実現したい回路の機能をシミュレートするプログラムをなんらかの方法で実装する
2. そのシミュレータと同じ動作をする HDL 回路を実装する
3. HDL 回路の動作をテストベンチによりシミュレータの動作とつき合わせて検証する

つまり、HDL で回路を実装するためには、まずその参照実装となるべき「シミュレータ」が必要であり、それを実装することが HDL による回路の設計の第一歩となる。これは、どのように複雑な回路であっても同様である。そのような「シミュレータ」なしに、いきなり HDL で回路を実装することは、非常に単純な場合を除いて、全く意味のないことであると言ってもよい。そのためには、まずは何をどのように HDL で実装するかを考えた上で、その仕様自体を検証するためのシミュレータを実装する。ここのシミュレータは、どのような手法で実現しても構わないが、HDL で記述可能な基本的な演算や操作は「ビット操作と整数演算」のみであると考えると[2]、そのような演算を最も効率的に実装できるようなプログラミング言語を利用するのがよい。一番単純には C 言語を利用するのが簡単である。ただし、C 言語では 64 bit 以上の整数を簡単に取り扱うことができないので、bit 長の長い回路の設計には適していないかもしれない。

3 浮動小数点演算器の実装

to be continued...

4 更新履歴

version 2009/04/25 first draft (C) 中里直人

[2]HDL で記述されたライブラリがある場合はより高級な機能も利用できる