半導体産業にもようやく明るい兆しが見えてきました。
IT バブル崩壊だなんだと言われてきましたが、不況の構造は毎度
おなじみのシリコンサイクルと
大差ないように感じたのは私だけでしょうか。長期的な利益を度外視
して各社が安売りに走るこの業態はなんとかならないものかと常々
思っています。結局は多くの会社が同じものを作るのが問題で
シェアを取って数量を出すことで原価を低減しようと安売りに
走る構造問題をかかえています。

一方、ニッチな市場で利益を出してがんばっている会社も
多いので、それぞれに強みを生かすことの大切さを感じます。
世界中で多くの会社や地域がより付加価値の高いマーケットへの
参入を目指しています。皆が一斉に参入するマーケットは
その時点で付加価値は大きく目減りしていきます。そこで、
いかに知的所有権(IP)を守りながらハイエンドのマーケットでの
有力プレーヤとして地位を築いていけるかが勝負になります。

IP には様々なレベルのものがありますが、最近のシステムオンチップ
(SoC)の世界では回路の機能ブロックを差して IP と呼ぶことが多く
なっています。IP で有名な会社として ARM 社があります。彼らは
プロセッサの IP を中心としてライセンスとロイヤリティを収入の
柱とするビジネス戦略を持っています。

http://www.arm.com/

CPU だけが IP ではなく、現在では様々な IP が流通しています。
その昔、プリント基板の上に LSI を並べて製品を組み立てた
時代がありましたが、今では 1 つの LSI にほとんどの機能を集積
可能なほど LSI の規模が大きくなっています。また、
システムオンチップの時代は IP をいかに活用するかが
製品開発の成否を決めるといってもよいほど
開発期間が短くなっています。

CPU を設計し遊び倒そうということがこの連載の主旨ですが
その技術は実は IP 設計に通じるものがあります。GNU の精神で
IP を公開するのもよし、IP を通じて世界を相手にベンチャーを
起すのもよし、色々な技術への接しかたがありますが、IP 設計の
世界に興味を持つ読者が一人でも増えてくれればと思います。

この連載ではハードウェアを記述するために SFL 言語を利用し
そのシミュレーションと論理合成に PARTHENON を使っていますが、
Verilog/VHDL に対する産業界のニーズが非常に高いのも理解して
います。最終的に作成するものは論理回路であり、それは
ネットリストに変換されたのちに LSI のレイアウトになるので
当初の記述言語が何であれ実はそれほど大きな違いはないので
記述しやすい言語を取り上げているのがその理由です。また、
ハードウェアの動作についての理解がしっかりできていれば
記述言語は容易に修得できるため、論理合成の可否があいまいな
Verilog/VHDL などよりは切り分けがしっかりしている SFL で
HDL の学習をスタートしたほうが早く理解が深まります。
大学において VHDL と SFL の講義を両方試した結果、VHDL で受講した
学生は CPU 設計に到達する以前にギブアップすることが多く、
一方、SFL で受講した学生は CPU 設計まで比較的スムースに
到達したことから、初学者のためには SFL を使うことにしています。

今回は SFL と Verilog の違いを理解するために、また、SFL で
作成した論理回路を Verilog に手動で変換する方法を示すために
スロットマシンを Verilog 記述に変換してみます。コンパイラとして
2001 年 6 月号で紹介した Verilog コンパイラ Icarus Verilog を
使います。

http://www.icarus.com/eda/verilog/index.html

執筆時に Icarus Verilog は V0.6 になっています。
以前紹介した時には Xilinx 社の XNF ネットリストしか
合成ターゲットとしてサポートされていませんでしたが、
今回調べてみると EDIF フォーマットもサポートされています。
そこで、マッピングファイルを書けば ALTERA 社の Max+Plus2 にも

合成結果をもっていけそうです。しかしながら、Icarus Verilog
が合成できる動作記述は非常に限定されているため、
ほとんど 構造記述並みの記述を強いられることになるので
ここでは合成は行なわず、シミュレーションだけとしておきます。
Icarus Verilogでシミュレーションが終り動作が確実だと
思えばMax+Plus2のVerilogリーダーを用いて論理合成を行ないます。

スロットマシンのSFLファイルを対応するVerilog記述に
書換えていきましょう。

まずは、一番小さなLED制御回路であるsegctl.sflからです。
SFLの記述は次のようになっていました。

```
module segctl {
 instrin start;
 instrout sela, selb, selc;

 stage_name selseg{task go(); }
 instruct start generate selseg.go();

 stage selseg {
   state_name a,b,c;
   first_state a;

   state a par {goto b; sela();}
   state b par {goto c; selb();}
   state c par {goto a; selc();}
  }
}
```

制御入力信号startが入ると回路がスタートし、
3つの制御出力信号sela,selb,selcを順番に出力する回路に
なっています。SFLではクロック信号やリセット信号は
記述の中には直接は出てきません。状態の遷移を動作で
記述する形になります。これに対してVerilogではクロック信号、
リセット信号等、回路の中で利用する信号は全て明示的に扱います。
また、モジュールの状態という概念はないため、これらは
状態レジスタに明示的に記述することになります。

segctlでは3つの状態を用います。そこで、状態を表わすためには
2ビットのレジスタが必要になります。
ステージが動作しているかどうかを表わすためのレジスタも
作りましょう。本来のスロットマシンの動作では、これらの変換には
かなり省略可能な部分がありますが、定型的な変換を行なうことで
将来の自動変換に含みを持たせておきます。
まずは、変数やパラメータの宣言部分を考えてみましょう。

明示的に追加する必要があるクロック、リセット信号は
それぞれclock, resetとすると、モジュールのインターフェイス部分は

```
module segctl (clock, reset, start, sela, selb, selc) ;
```

のようになります。信号としてstart,sela,selb,selcはSFLと
同様にインターフェイス信号になります。
インターフェイス部分に定義される信号は個別に型を指定する
必要があります。

```
 input clock, reset, start;
 output sela, selb, selc;
```

SFLでは制御入力、制御出力という信号に意味を持たせた
型を利用しましたが、Verilogでは単に入力、出力の型を
用います。次にSFLでは状態変数を名前で指定して
いましたが、Verilogには状態変数を特別に扱う仕組はない
ので明示的に状態をレジスタに作成しますが、レジスタの

値と状態名を分り易くするためにパラメータで状態を宣言
しておきましょう。

```
 parameter sta=1, stb=2, stc=3;
```

SFL の制御出力は出力が明示的に指定されていない場合には
0 を出力する決まりになっていますが、Verilog の出力信号には
意味付けはないので、出力時には 0 も明示的に指定します。
記述を簡単にするために、ここにもパラメータを宣言して
おきます。

```
 parameter outa=3'b100, outb=3'b010, outc=3'b001;
```

ここでは 3 つの出力を 3 ビットの数値としてまとめて
宣言しています。出力信号は一度 result という変数に
設定した後、sela,selb,selc の各信号に出力させます。
この一時変数は中間の端子として定義すれば良いだけ
なのですが、Verilog では値が保持される信号線には
レジスタ属性を付ける必要があるので、reg タイプと
して宣言します。

```
 reg  [2:0] result;
```

ここで注意することは reg としてレジスタ
宣言したとしても Verilog では必ずしも記憶素子が
合成される必要はないということです。

実際、上の result のレジスタは属性こそレジスタと
していますが、記憶素子は割当てられることはありません。
しかし、レジスタ属性にしておかなければ動作記述でエラーと
なってしまいます。そこで、セレクタ回路を自動生成させたい
信号はレジスタ属性を用います。
完全に構造記述を行なえば曖昧さは無くなりますが、
その場合には HDL を使うメリットも大幅に減ってしまいます。

次に、回路が動作している状態を表わす信号として
stageon、状態変数として segstate をそれぞれ宣言します。

```
 reg stageon;
 reg [1:0] segstate;
```

これらの信号には実際にレジスタが割当てられることになります。
宣言の部分では何の違いもないのですが、動作記述の書き方で
ただの信号線として扱われるかレジスタが生成されるかが変わります。
この曖昧な解釈において設計者とコンパイラの意見が食い違うと
思いもしない回路になってしまうので注意が必要です。

さて、いよいよ動作記述の作成です。同期回路を作成するので
クロックの立ち上がりで信号が遷移するので、always @文を
用いて動作を記述します。リセット信号は通常非同期に入力する
ので、記述は次のようにクロックかリセットが入力した時に
回路が動作するように記述します。実はリセット信号は信号
レベルで動作するレベルセンシティブな信号とする方が
普通なのですが、ALTERA 社の Max+Plus2 では always 文に
レベルセンシティブとエッジトリガが混在していると合成できない
ためリセットもエッジトリガにしています。他の合成エンジンを
使って混在でも合成できる方はリセットの前の posedge を
消してください。

```
 always @ (posedge clock or posedge reset)
```

動作記述ではまずリセット時の動作を記述しておきます。
リセットとリセット以外の動作は通常大きく異るため
リセット記述を先にまとめておくと分りやすいと思います。
今回はリセットにおいて行なう必要があるのは
回路が動作していることを表わす stageon 信号をクリアする
ことと、状態遷移を初期化することです。

```
  if (reset)
    begin
     stageon <= 0;
     segstate <= sta;
    end
  else begin
```

このように if 文を用いてリセット時の状態を記述しておきます。
リセット以外の動作を else の後にまとめることで、リセット動作と
それ以外の動作で信号の二重アサインなどのトラブルを防ぐことが
できます。
前出のように状態変数 segstate にはパラメータで定義した状態を
表わすシンボル sta を入力し A 状態に初期化しておきます。

このリセット時の動作は SFL の記述では first_state として
初期状態を宣言している部分と暗黙のうちにステージは
リセット後非動作状態になっているという仕様に相当します。

さて、外部から start 信号が来たら回路を動作状態に変更するので
次のように信号 stageon に 1 を代入します。

```
   if (start) stageon <= 1 ;
```

ここでノンブロッキング代入文を用いているのは、この効果は
次のクロックサイクルで発生するべきものだからです。
イコール記号 1 つだけのブロッキング代入文を用いると代入が
直ちに行なわれると解釈されますので、レジスタを想定している
この信号の代入には不適当です。
このように代入文一つとっても合成後の回路を想定して選択する
必要があります。

さて、回路が動作状態にある場合には状態遷移を行ないます。

```
   if (stageon)
    case (segstate)
     sta:
      begin
       segstate <= stb;
      end
     stb:
      begin
       segstate <= stc;
      end
     stc:
      begin
       segstate <= sta;
      end
    endcase
```

状態変数である segstate レジスタの値を見て現在の状態に
したがって処理を切り替えます。まず、状態 A では次状態を
状態 B に切り替えます。
この処理は状態 B や状態 C でも遷移先と出力データが異る
だけで同様の処理となります。次状態のレジスタへの
代入は次のクロックで遷移が発生するのでノンブロッキングで
行ないます。

ここで、リセットを判定する if 文において else 側の begin に対応する
end を記述しておきます。

```
end
```

次に出力信号への代入は同じ always 文の中で行なうと
条件付の代入としてラッチが生成されてしまうので
もう一つ always 文を作成し、そこで
ブロッキング代入文で行ないます。
sela,selb,selc にそれぞれ 1 つだけ 1 を出力するためパラメータで

定義した信号を result 信号に出力します。
ここで、result 信号は出力結果を直ちに反映させるため
ブロッキング代入文を用いて値を設定します。
always 文の条件として信号名を記述した場合には
その信号が変化した場合に動作が発生することに
なります。そこで、segstate という状態変数に
従って状態の動作を記述することができます。
また、出力信号のための中間変数である result をモジュールの
出力信号に代入します。この代入は固定的なので assign 文で
行ないます。

```
 always @(segstate)
    case (segstate)
     sta:
      begin
       result = outa;
      end
     stb:
      begin
       result = outb;
      end
     stc:
      begin
       result = outc;
      end
    endcase

 assign {sela, selb, selc} = result;


endmodule
```

お疲れさまでした。ここまでで、モジュールの記述が終了です。
SFL で記述されたモジュールを Verilog で記述し直してみたのですが、
行数、ワード数とも 2 倍以上になっています。

```
$ wc segctl.v segctl.sfl
     51     115     900 segctl.v
     16      44     309 segctl.sfl
     67     159    1209 合計
```

まとめると次のリストのようになります。

```
-------------------------------------------------------
module segctl (clock, reset, start, sela, selb, selc) ;
 parameter sta=1, stb=2, stc=3;
 parameter outa=3'b100, outb=3'b010, outc=3'b001;
 input clock, reset, start;
 output sela , selb , selc ;

 reg  [2:0] result;
 reg stageon;
 reg [1:0] segstate;

 always @(posedge clock or posedge reset )
  if (reset)
    begin
     stageon <= 0;
     segstate <= sta;
    end
  else begin
   if (start) stageon <= 1 ;
   if (stageon)
    case (segstate)
     sta:
      begin
       segstate <= stb;
      end
```

```
     stb:
      begin
       segstate <= stc;
      end
     stc:
      begin
       segstate <= sta;
      end
    endcase
  end

 always @(segstate)
    case (segstate)
     sta:
      begin
       result = outa;
      end
     stb:
      begin
       result = outb;
      end
     stc:
      begin
       result = outc;
      end
    endcase
 assign {sela, selb, selc} = result;
endmodule
```

さて、モデルが出来たら論理シミュレーションで動作の
確認を行ないます。Verilogは論理シミュレーションのために
設計された言語ですから、論理シミュレーションは得意です。
論理合成は苦手な Icarus Verilog もシミュレーションは問題なく
実行できます。シミュレーションコードは比較的短いので
リストを示した後、まとめて解説します。

```
module main;
parameter STEP=10;
reg clk, reset, start;
wire sela, selb, selc;
segctl sgc(.clock(clk), .reset(reset), .start(start),
           .sela(sela), .selb(selb), .selc(selc));
always #(STEP/2) clk = ~clk;
always @(negedge clk) begin
 $display("a(%b) b(%b) c(%b)", sela, selb, selc);
end
initial begin
 clk = 0;
 reset = 1;
 #(STEP) reset=0;
 #(2*STEP) start=1;
 #(3*STEP) start=0;
 #(6*STEP) $finish;
end
endmodule
```

このモジュールを segctl.tst とすると論理シミュレーションの
ためには次のようにコンパイルします。

```
$ iverilog segctl.v segctl.tst -o segctl.exe
```

すると segctl.exe という実行ファイルができるので、

```
$ ./segctl.exe
```

と生成したファイルを実行します。この実行形式は
実はスクリプトになっています。

```
$ file segctl.exe
segctl.exe: a /usr/local/lib/ivl/../../bin/vv script text
```

と表示されます。テキストファイルですから、more コマンドで
中身の確認ができます。

スクリプトの実行結果は次のようになります。

```
------------------------------------------------------------
$ ./segctl.exe
a(x) b(x) c(x)
a(1) b(0) c(0)
a(1) b(0) c(0)
a(1) b(0) c(0)
a(1) b(0) c(0)
a(0) b(1) c(0)
a(0) b(0) c(1)
a(1) b(0) c(0)
a(0) b(1) c(0)
a(0) b(0) c(1)
a(1) b(0) c(0)
a(0) b(1) c(0)
------------------------------------------------------------
```

回路がリセットされると状態 A に初期化され、その後 start 信号が
発行されるとクロックごとに A->B->C->A と状態が遷移して
行きます。

さて、Icarus ではこの回路の合成はできないので、
ALTERA 社の Max+Plus2 で合成をかけてみました。
小さな回路なのターゲットはなんでもいいのですが、
とりあえず FLEX 10K シリーズをターゲットにしてみます。
内部論理の利用状況の概要は次の表のようになります。

```
** BURIED LOGIC **
                                          Fan-In    Fan-Out
 IOC    LC   EC  Row Col Primitive    Code  INP  FBK  OUT  FBK  Name
  -     2   -   A   01      DFFE   +        1    0    0    2  stageon (:32)
  -     4   -   A   01      DFFE   +        0    2    0    4  segstate1 (:35)
  -     6   -   A   01      DFFE   +    !   0    2    0    4  segstate0 (:36)
  -     1   -   A   01       OR2           0    2    1    0  :65
  -     3   -   A   01       OR2           0    2    1    0  :66
  -     5   -   A   01       OR2           0    2    1    0  :67
```

全部で 6 個のロジックセルを使用して合成されたことが分ります。

スロットマシンの SFL 記述は 3 つのファイルに分れています。
一つは今解説した通りです。部品として利用するもう一つの
回路である slot モジュールは segctl とほとんど同様の状態遷移を
記述するだけなので、これまでの解説で簡単に変換できると
思います。他の一つである machine というファイルは segctl や
slot をインスタンスとして利用するので、インスタンスの
利用方法について解説しましょう。

まずは、次のリストを見てください。

```
------------------------------------------------------------
module machine (clock, reset, sw4, sw5, sw6, sw7, sa0, sa1, sa2, seg) ;
 input clock, reset, sw4, sw5, sw6, sw7;
 output sa0, sa1, sa2;
 output [7:0] seg;
```

```
 reg [7:0] seg;

 wire [7:0] sega, segb,segc;

 slot slota (.clock(clock), .reset(reset), .start(sw4), .stop(sw5), .seg(sega));
 slot slotb (.clock(clock), .reset(reset), .start(sw4), .stop(sw6), .seg(segb));
 slot slotc (.clock(clock), .reset(reset), .start(sw4), .stop(sw7), .seg(segc));

 segctl segctl(.clock(clock), .reset(reset), .start(sw4),
               .sela(sa0), .selb(sa1), .selc(sa2));

 always @({sa0, sa1, sa2})
  case ({sa0, sa1, sa2})
    3'b100: seg=sega;
    3'b010: seg=segb;
    3'b001: seg=segc;
  endcase
endmodule
-------------------------------------------------------
```

これがmachine.vの本体になります。
ここでは、segctlとslotからインスタンスをそれぞれ1個、3個ずつ
生成します。
Verilogではインスタンスの生成の時に利用モジュール中での名前と
インスタンスの元ファイル中の名前のマッピングを記述します。
例えば、
.start(sw4)
という記述がslotのインスタンスの宣言にありますが、これは
インスタンス化する元モジュールのstartという信号をsw4と
いう名前で利用することを表わしています。
machine.vのファイルの中にはclockもresetも必要ないため
これらの信号はそのままインスタンスに渡します。
segctlから出てくるsa0からsa2の信号に従って7セグメントLEDの
出力データを切り替えるのですが、ここではalways文を用いて
います。3つの信号のどれか一つでも変化したらalways文は
有効になり、それぞれ対応する信号に応じた動作を行ないます。

次にmachineのシミュレーションファイルを作成しましょう。
実はmachineのシミュレーションを行なうにはslotモジュールも
作らなくてはならないのですが、あんまり全部答を書いても
面白くないでしょうから、読者の宿題としておきましょう。

```
-------------------------------------------------------
module main;
parameter STEP=10;
reg clk, reset, sw4, sw5, sw6, sw7;
wire sa0,sa1,sa2;
wire [7:0] seg;
machine machine(.clock(clk), .reset(reset), .sw4(sw4),
 .sw5(sw5), .sw6(sw6), .sw7(sw7), .sa0(sa0), .sa1(sa1),
 .sa2(sa2), .seg(seg));

always #(STEP/2) clk=~clk;
always @(negedge clk)
 $display("sa[%b%b%b] seg[%b]",sa0,sa1,sa2,seg);

initial begin
 clk = 0;
 reset = 1;
 #(STEP) reset=0;
 #(2*STEP) sw4=1;
 #(3*STEP) sw4=0;
 #(8*STEP) $finish;
end
endmodule
-------------------------------------------------------
```

このシミュレーションテストベクターにはスロットを停止する
記述が入っていませんが、適宜追加してみてください。

以上見てきたようにSFL言語で記述された回路は形式的な変換を行なって
Verilogなどの他のハードウェア記述言語に変換することができます。
SFLでもVerilogでもVHDLでも自分で覚えやすい言語を覚えておけば
他の言語を修得するのは簡単です。最終的に出来上がる回路自体は
同じものを目指しているのですから、当然ですね。
ただし、VerilogやVHDLはかなり無茶苦茶な回路を平気で作成可能で
あることから、これらの言語で悪い習慣が付いてしまうと後で苦労する
と思うので、私はSFLを初学者には勧めています。