# ゼロから始めるプロセッサ入門 CPUと遊ぼう!

## 第7回●電子式スロットマシン

清水尚彦
nshimizu@keyaki.cc.u-tokai.ac.jp

5月号ではありますが、工学系大学の新入生である読者も方々も多いので、3月発売の各誌はフレッシュマン特集のような企画があちこちで真っ盛りですね。新たに工学分野を目指す人達に本連載が多少なりとも好奇心を呼び起すことができれば幸いです。

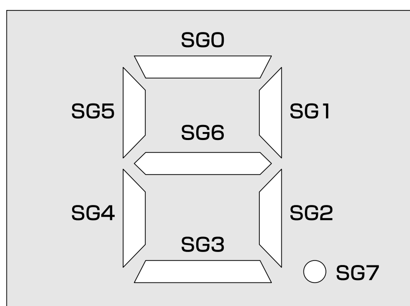
さて、せっかくハードウェア記述言語の基礎を覚えたので、今回は再び寄り道をして小さな「おもちゃ」を作ってみましょう。コンピュータのコンソールの上だけで動くものではなく、実際に手に取って遊ぶことのできるおもちゃの制作をします。早くCPUを作ってみたいと思う方も、ハードウェア記述言語の練習のつもりでお付き合いください。

おもちゃのターゲットデバイスは、連載初回に紹介したヒューマンデータ社の「FLEX10K10ブレッドボードキット」という製品です（記事末のResource【1】を参照）。

このボードの入出力インターフェイスとして搭載されている、発光ダイオードとプッシュスイッチを使って、簡単におもちゃができないものかと考えていましたが、今回は電子式スロットマシンを作ることにします。利用するボード上の入出力インターフェイスは次のようになります（図1）。

- 7セグメント発光ダイオード
  3個（a（L1）、b（L2）、c（L3））
- プッシュスイッチ
  4個（スタート（SW4）、STOPa（SW5）、STOPb（SW6）、STOPc（SW7））

【図1】




7セグメント発光ダイオードというのは図1のように発光ダイオードが並んでいるものです。7つのダイオードを組み合わせて数字を作るこの方法は、電卓やデジタル時計などで馴染みがあると思います。表示が数字に見えるように発光させるダイオードを選択する必要があります。この7セグメント発光ダイオードはスロットマシンの窓を表し、ここでは1から8までの数字が表れるものとします。プッシュスイッチは、1個のスタートボタンとこの7セグメント発光ダイオードにそれぞれ対応するストップボタンになります。

次に動作仕様を考えましょう。

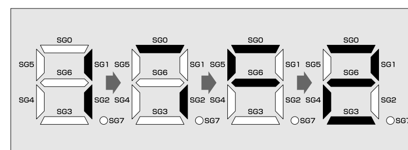
1. スタートボタンを押すと3つの7セグメント発光ダイオードの表示が動き出す。
2. ストップボタンが押されると対応する7セグメント発光ダイオードの表示を止める。
3. ストップボタンはどれから押しても構わないこととする。

スロットマシンのイメージはみなさんお持ちでしょうから、この程度のラフな動作仕様でも設計上困りません。しかし、ゲームとして成り立たせるためには、ランダムな値で停止するというだけでは面白くありません。そこには人間が介入する要素が必要です。

そこで、プレーヤーが「7セグメント発光ダイオードの変化を観察して予測し、ボタンを押すタイミングを図ることで、ゲームの勝率が上がる」といったシステムを用意します。

ただし、簡単に予測できて誰でも当るようになってはゲームにならないので、加減が必要になります。このために、変化をゆっくりと行うと共に、図2のように左から右にスクロールするように7セグメント発光ダイオードの表示を変化させ、人間の目にはスロットが回転して見えるようにします。中途半端な位置で止まると文字が読めないので、停止は必ず文字が完全に表示されているところで止まるようにします。

【図2】



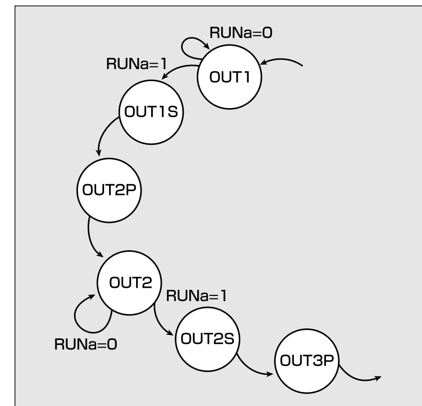

そこで、各スロットは図3のような状態遷移を行うことにします。この図では先に挙げた1と2の表示の前後の状態遷移図を示していますが、他の数値でも同じように、動作中信号RUNaが1のときのみ正常な表示から次の表示状態へ遷移するようにします。RUNaという信号は、3つの7セグメント発光ダイオードのうちのaが動作中という意味の信号を表すと思ってください。表示する数値をSEG0からSEG7までのビットマップで表してみましょう（表1）。

このビットマップに合せて7セグメント発光ダイオードを点灯させていくと流れるように（大袈裟）スロットが回転して見えます。状態の割り当てなど効率を考えれば最適化の余地は十分ありますが、必要ならコンパイラが最適化することを期待して、ここではデバッグが容易で分りやすいことを優先して設計します。

この回転するスロットを1つのモジュールとして、合計で3つ使えばスロットマシンができます。とはいえ、この他にもまだ必要な機能があります。

まずは、スロットの実行、停止の制御ユニットが必要です。スロットモジュールが回転して

【図3】



いる間はRUN信号を送り続けて、プッシュボタンによって停止が指示されたところでRUN信号を0にするように制御する必要があります。

次に表示ユニットが必要になります。表示ユニットでは3つのモジュールの出力を7セグメント発光ダイオードに送ります。ここで、LSIの入出力ピン数を低減するために3つの7セグメント発光ダイオードはカソード側を共通信号に接続しています。アノード側は3つ別々に出ています。簡単な図を図4に示します。この形は信号線を減らしコストを下げるために良く用いられますが、使うときには案外やっかいです。アノードに信号を与えた発光ダイオードだけが点灯するので、3つに別々の表示を同時にさせることはできません。そこで、人間の網膜の残像を利用して、3つを時分割に表示させ擬似的に同時に表示しているように見せます。そのため、表示ユニットでは3つのモジュールの出力を順次カソード側のピンに出力し、それに同期してアノード側の信号を出力するようにします。

以上の検討結果をまとめて図5のようなブロック図を書いてみます。このブロック図に従ってスロットマシンの設計をしていきましょう。

## ■スロットモジュールの作成

スロットモジュールは、今回のスロットマシンのキーとなる部品です。動作仕様は前述の通りですが、入出力のインターフェイスを整理しておきます。

**入力信号**

- モジュールスタート信号（start）
- スロット停止信号（stop）

**出力信号**

- 7segビットマップ（seg<8>）
- 重み（out<3>）

スタート信号と停止信号は、制御入力として受け取ることにします。また、実行中の状態(run)はモジュール内部でレジスタに記憶します。重みの信号は今回は利用しませんが、拡張とデバッグ用に出しておきます。このモジュールの宣言文はリスト1のようになります。

出力信号segは8ビットの信号になります。7セグなのに8ビットなのは、小数点の位置にも発光ダイオードがあって、この分のビットが最上位の1ビットに定義されているからです。

モジュール本体を作成しましょう。

```
module slot {
```

モジュール名をslotとしておきます。入出力信号はプロトタイプと合わせます(リスト2)。また、ステージの起動するにあたってはタスクを指定する必要がありますが、ここではタスク名としてrunを用います。

```
stage_name rotate{task run(); }
```

ステージ名もタスク名も任意なので、好きな名前を付けて構わないのですが、ステージ名はスロットの回転をイメージしてrotateと名前を付けました（リスト3）。制御入力のstart信号が入ったときに、実行状態にすると共にステージの起動をします。また、stop信号が入ったら実行状態を解除しておきます。次にステージの宣言を書きます。

```
stage rotate {
```

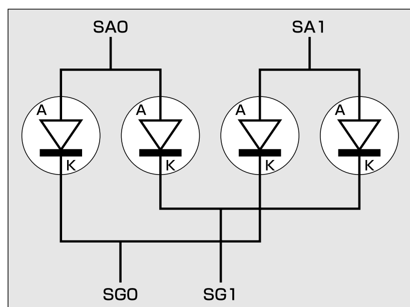
ステージ名はすでに指定済みのrotateという名前を用います。ステージは状態遷移を行う状態マシンとして記述しますが、各状態はあらかじめ宣言しておく必要があります。状態変数の最適化なんて細かなことを言わずに各状態を宣言してしまいます（リスト4）。

電源投入後の最初のステージはout1というステージにしておきます。実際には、状態はぐ

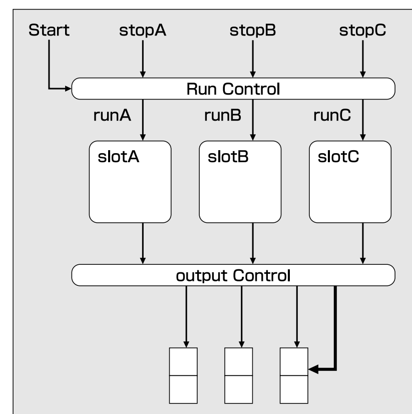
**【表1】表示する数値**

| 状態 | 表示 | 表示ビットマップ（SEG7～SEG0） |
|---|---|---|
| OUT1 | 1 | 00000110 |
| OUT1S | 1→2 | 00000101 |
| OUT2P | 1→2 | 01100001 |
| OUT2 | 2 | 01011011 |
| OUT2S | 2→3 | 01001101 |
| OUT3P | 2→3 | 01001011 |
| OUT3 | 3 | 01001111 |
| OUT3S | 3→4 | 01001100 |
| OUT4P | 3→4 | 00001011 |
| OUT4 | 4 | 01100110 |
| OUT4S | 4→5 | 00011101 |
| OUT5P | 4→5 | 01000011 |
| OUT5 | 5 | 01101101 |
| OUT5S | 5→6 | 01010001 |
| OUT6P | 5→6 | 01101011 |
| OUT6 | 6 | 01111101 |
| OUT6S | 6→7 | 01010000 |
| OUT7P | 6→7 | 00001010 |
| OUT7 | 7 | 00000111 |
| OUT7S | 7→8 | 01000101 |
| OUT8P | 7→8 | 01101011 |
| OUT8 | 8 | 01111111 |
| OUT8S | 8→1 | 01011100 |
| OUT1P | 8→1 | 00001010 |

**【図4】**




**【図5】**




**【リスト1】**

```
declare slot {
 instrin start;
 instrin stop;
 output seg<8>;
 output out<3>;
}
```

**【リスト2】**

```
instrin start;
instrin stop;
output seg<8>;
output out<3>;
reg run;
```

**【リスト3】**

```
instruct start par { run := 0b1; generate rotate.run();}
instruct stop run := 0b0;
```

**【リスト4】**

```
state_name out1, out1s, out1p, out2, out2s, out2p;
state_name out3, out3s, out3p, out4, out4s, out4p;
state_name out5, out5s, out5p, out6, out6s, out6p;
state_name out7, out7s, out7p, out8, out8s, out8p;
```

るぐる回るので、人間の目にはどこから始まろうと大した違いはありません。どこからでも構わないので適当に決めています。

```
first_state out1;
```

ここから実際に各状態の動作記述を行います。といっても、このモジュールでは各状態で行うことは出力端子への値の設定と次の状態への遷移しかありません（**リスト5**）。

正常に数字が表示されている状態では入力信号runのチェックをしています。途中の状態では止めないようにすることで、止まったときにきちんと数字として認識できるようにしておきます。

## ■モジュール完成後のシミュレーション

モジュールができたら、シミュレーションで動作確認をしましょう。モジュールの記述をslot.sflという名前のファイルに書き込みます。シミュレーションスクリプトは**リスト6**のようなものを使います。

まず、論理記述を読み込んだ後、slotモジュールをトップモジュールとしてシミュレーションイメージを作成します。その後、スロットモジュールをスタートさせます。レポートには、シミュレーションタイムとステージ中の実行ステート名、出力信号の値をそれぞれ出力するようにします。このスクリプトを動作させると**リスト7**の出力が得られます。その後、シミュレーションを2クロック進めます（**リスト8**）。さらにここで、モジュールに停止信号を入れてみましょう。またその後でシミュレーションを5クロックほど進めてみます（**リスト9**）。

すると、2を出力する状態でモジュールの実行が停止しているのが分かると思います。SFL

**【リスト5】**

```
    state out1 par {
      if(run) goto out1s;
      seg = 0b00000110; out=0b000;
      }
    state out1s par {goto out2p; seg = 0b00000101;}
    state out2p par {goto out2; seg = 0b01100001;}
    state out2 par {
      if(run) goto out2s;
      seg = 0b01011011; out=0b001;
      }
    state out2s par {goto out3p; seg = 0b01001101;}
    state out3p par {goto out3; seg = 0b01001011;}
    state out3 par {
      if(run) goto out3s;
      seg = 0b01001111; out=0b010;
      }
    state out3s par {goto out4p; seg = 0b01001100;}
    state out4p par {goto out4; seg = 0b00001011;}
    state out4 par {
      if(run) goto out4s;
      seg = 0b01100110; out=0b011;
      }
    state out4s par {goto out5p; seg = 0b00011101;}
    state out5p par {goto out5; seg = 0b01000011;}
    state out5 par {
      if(run) goto out5s;
      seg = 0b01101101; out=0b100;
      }
    state out5s par {goto out6p; seg = 0b01010001;}
    state out6p par {goto out6; seg = 0b01101011;}
    state out6 par {
      if(run) goto out6s;
      seg = 0b01111101; out=0b101;
      }
    state out6s par {goto out7p; seg = 0b01010000;}
    state out7p par {goto out7; seg = 0b00001010;}
    state out7 par {
      if(run) goto out7s;
      seg = 0b00000111; out=0b110;
      }
    state out7s par {goto out8p; seg = 0b01000101;}
    state out8p par {goto out8; seg = 0b01101011;}
    state out8 par {
      if(run) goto out8s;
      seg = 0b01111111; out=0b111;
      }
    state out8s par {goto out1p; seg = 0b01011100;}
    state out1p par {goto out1; seg = 0b00001010;}
  }
}
```

**【リスト6】**

```
sflread slot.sfl
autoinstall slot
rpt_add sim "%t\t%B\t%B\t%B\n" rotate seg out
set start 1
forward +24
```

**【リスト7】**

```
1       out1    00000110        000
2       out1s   00000101        zzz
3       out2p   01100001        zzz
4       out2    01011011        001
5       out2s   01001101        zzz
6       out3p   01001011        zzz
7       out3    01001111        010
8       out3s   01001100        zzz
9       out4p   00001011        zzz
10      out4    01100110        011
11      out4s   00011101        zzz
12      out5p   01000011        zzz
13      out5    01101101        100
14      out5s   01010001        zzz
15      out6p   01101011        zzz
16      out6    01111101        101
17      out6s   01010000        zzz
18      out7p   00001010        zzz
19      out7    00000111        110
20      out7s   01000101        zzz
21      out8p   01101011        zzz
22      out8    01111111        111
23      out8s   01011100        zzz
24      out1p   00001010        zzz
SECONDS>
```

**【リスト8】**

```
SECONDS> forward +2
25      out1    00000110        000
26      out1s   00000101        zzz
SECONDS>
```

**【リスト9】**

```
SECONDS> set stop 1
SECONDS> forward +5
27      out2p   01100001        zzz
28      out2    01011011        001
29      out2    01011011        001
30      out2    01011011        001
31      out2    01011011        001
SECONDS>
```

では明示的に遷移しない限り同じ状態に留まります。このスロット制御モジュールを3つ用意して、スタートボタンが押されたときには3つのモジュールのstart信号を同時に動作させ、停止ボタンが押された時にはそれぞれ対応するモジュールのstop信号を送ることでスロットマシンの基本構造はできます。

次に、表示制御ユニットを作成しましょう。それぞれのモジュールの7seg用の出力信号は時分割で出力ピンに送り、それに同期してアノード側の選択信号を送出します。この操作のために、3つの7セグメント発光ダイオードのどれを選択するか示す信号を出すモジュールを作成します。実は、このユニットは独立したモジュールにするほどのものでもなく、統合モジュールの中の1つのステージとして作成したほうが簡単なのです。しかし、練習の意味もありなるべく独立できるモジュールは独立させて作成します。合成後の論理規模はどちらにしても大した違いはありません。

まずは、宣言文から。

```
declare segctl {
 instrin start;
 instrout sela, selb, selc;
}
```

モジュールのスタート信号と3つの出力信号をそれぞれ制御信号としておきます。出力を制御信号とするのは、上位のモジュールでこの信号をトリガとする動作記述を書きたいからです。

モジュール本体は今回は簡単なので、リスト10に1度に示しておきます。これは3つの状態a、b、cをぐるぐる回って、それぞれの状態に対応する制御出力を起動しているだけです。このモジュールはsegctl.sflという名前で格納しておきます。

次に、このモジュールの論理シミュレーションを行いましょう。シミュレーションのスクリプトファイルをリスト11のように書きます。実行結果はリスト12のようになります。3つのステージを回りながら対応する信号線に1を出力している様子が分かります。

さて、それではこれらのモジュールを統合して動作させましょう。統合モジュールをmachine.sflという名前にします。サブモジュールとして利用するslot.sflとsegctl.sflはプリプロセッサでインクルードして利用するように設定します（リスト13）。

3つのスロットモジュールと、制御モジュールをサブモジュールとし、また入力スイッチの値を制御信号としてその入力に従った動作を記述します。このモジュールを例によってシミュレーションで動作確認します（リスト14）。

作成した3つのファイルをシミュレータに読み込ませ、7セグメント発光ダイオードの選択信号と点灯信号をシミュレーションレポートでモニタします。その後、スタートボタンを押し、4クロックごとにストップボタンを順次押していきます。スクリプトの実行結果をリスト15に示します。ストップボタンが押された後は、発光ダイオードのドライブ信号が固定されていることが分かりますか？ シミュレーションで正常な動作を確認したら、論理合成をしてみましょう。

```
auto machine nld4 ALTERA altera
```

として論理合成を行うと、

【リスト10】

```
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();}
  }
}
```

【リスト11】

```
sflread segctl.sfl
autoinstall segctl
rpt_add sim "%t\t%B\t%B\t%B\t%B\n" sel sela selb selc
set start 1
forward +10
```

【リスト12】

```
1       a       1       0       0
2       b       0       1       0
3       c       0       0       1
4       a       1       0       0
5       b       0       1       0
6       c       0       0       1
7       a       1       0       0
8       b       0       1       0
9       c       0       0       1
10      a       1       0       0
```

【リスト13】

```
declare slot {
 instrin start;
 instrin stop;
 output seg<8>;
 output out<3>;
}
declare segctl {
 instrin start;
 instrout sela, selb, selc;
}
%i "slot.sfl"
%i "segctl.sfl"
module machine {
 output seg<8>;
 instrout sa0, sa1, sa2;
 instrin sw4, sw5, sw6, sw7;
 slot slota, slotb, slotc;
 segctl segctl;
 instruct sw4 par {segctl.start();slota.start();
                  slotb.start();slotc.start();}
 instruct sw5 slota.stop();
 instruct sw6 slotb.stop();
 instruct sw7 slotc.stop();
 instruct segctl.sela par {seg = slota.seg; sa0();}
 instruct segctl.selb par {seg = slotb.seg; sa1();}
 instruct segctl.selc par {seg = slotc.seg; sa2();}
}
```

【リスト14】

```
sflread machine.sfl
autoinstall machine
rpt_add sim "%t\t%B\t%B\t%B\t%B\n" sa0 sa1 sa2 seg
set sw4 1
forward +4
set sw5 1
forward +4
set sw6 1
forward +4
set sw7 1
forward +8
```

```
machine.edf
```

というネットリストが出来上がります。

## ■ MAX＋plusII

ここまで来ればもう少しでボード上の動作までこぎつけます。ここで作成したネットリストは、アルテラ社の「MAX＋plusII」([2])をインストールしてあるPCに転送しておきます。この後からはMAX＋plusIIでの作業になります。ネットリストのままだと扱いにくいので、まずはシンボルファイルを作成してしまいます。MAX＋plusIIにおいて、ファイルメニューから［File］→［Open］でmachine.edfをオープンします。LISPのようなS式で表されるネットリストが出てくると思います。ここでさらに、［File］→［CreateDefaultSymbol］とするとシンボルファイルが生成できます。

ここで、いったんプロジェクトを新しくします。というのは、出来上がったシンボルファイルを下位モジュールとした新しいプロジェクトを作るためです。

［File］→［New］でグラフィックエディタを選択して新しいファイル生成します。何もない白いウィンドウが現れます。ここで、マウスの左ボタンをダブルクリックすると、インポート可能なシンボルが出てきます。今作ったばかりのmachineがありますので、これを選択します（画面1）。

同じく入力ピンや出力ピンはシンボルライブラリのprimディレクトリから

```
input
output
```

をインポートしてきます。作成したスロットマシンの入出力インターフェイスは正論理なのですが、ヒューマンデータのCSP-004KITでは、いくつかの論理線を負論理にしなくてはなりません。そこで、machineのシンボルの上にマウスカーソルを移動し、右ボタンでメニューを出して［Edit Ports/Parameters］を選択します。使用しないb_clockやs_clock、p_resetの信号はunusedに設定し、発光ダイオードSA0～SA2と、スイッチSW4～SW7のインターフェイス線はinversionでallを設定しておきます。

さらに、表2のように入出力ピンを基板上のピン番号と合わせて設定します。このためには、ピン番号の決定前にデバイスをまず決めておかなくてはなりません。［Assign］→［Device］でFLEX10KシリーズからEPF10K10LC84-3を選択しておいてください。

後は、画面の入出力ピンの上にマウスカーソルを持っていって、右クリックメニューから［Assign］→［Pin/Location］を選択し、表2に示したピン番号に合わせて入力していきます。ピン設定と配線の終わったブロック図を画面2に示します。配線は自由にして構いませんので、この図と必ずしも同一にはならないと思います。

さて、コンパイラで合成して、プログラマでダウンロードしてみましょう。発光ダイオードにはすべて1が表示される状態がスタートの状態です。ここでSW4を押してみてください。この表示のどこが流れるようなんだって思われるかもしれません。しかし、入力したクロック信号が早すぎて人間には全然分からないほどになっているのです。まあ、それは読者の方々のお楽しみにとっておいて、早速動かしてみましょう。SW5～SW7を適当に押してみてください。表示が停止して数字が現れると思います。いかがですか？

こんなに単純なゲームでも、凝り始めればいろいろと改造したいところも出てくるかと思います。まずは、流れるような表示をぜひ実現したいのですが、今回はここまでということで、各自工夫してみてください。

今回作成したSFLのソースコードとスロットマシンのMAX＋PlusIIのグラフィックデザインファイル（GDF）並びに合成制御ファイル（ACF）はLinux JapanのWebページからダウンロードできるようにいたしますので、ご利用ください。

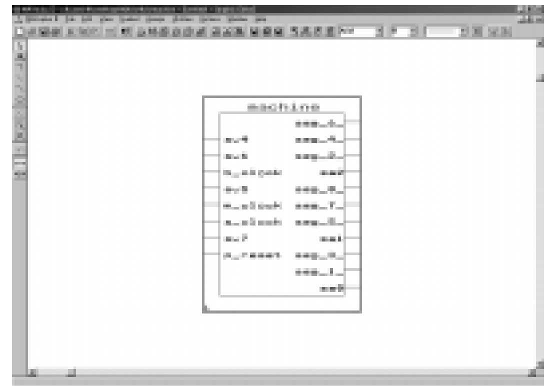
【リスト15】

```
1   1  0  0   00000110
2   0  1  0   00000101
3   0  0  1   01100001
4   1  0  0   01011011
5   0  1  0   01001101
6   0  0  1   01001011
7   1  0  0   01001111
8   0  1  0   01001100
9   0  0  1   00001011
10  1  0  0   01001111
11  0  1  0   01100110
12  0  0  1   01000011
13  1  0  0   01001111
14  0  1  0   01100110
15  0  0  1   01101101
16  1  0  0   01001111
17  0  1  0   01100110
18  0  0  1   01101101
19  1  0  0   01001111
20  0  1  0   01100110
```

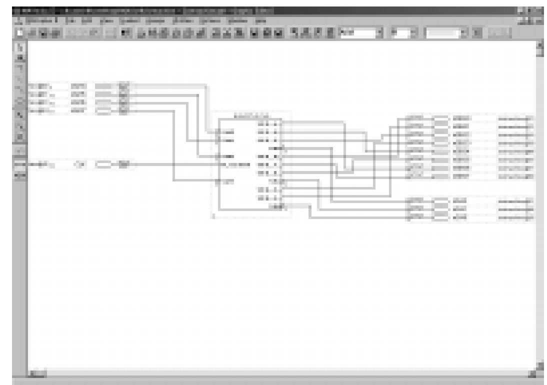
【表2】

| 入出力ピン | 基板上のピン番号 |
|---|---|
| SA0 | 25 |
| SA1 | 66 |
| SA2 | 21 |
| SEG0 | 73 |
| SEG1 | 72 |
| SEG2 | 19 |
| SEG3 | 70 |
| SEG4 | 69 |
| SEG5 | 18 |
| SEG6 | 71 |
| SEG7 | 58 |
| SW4 | 80 |
| SW5 | 81 |
| SW6 | 83 |
| SW7 | 84 |
| CLK | 43 |

【画面1】



【画面2】



### Resource

[1] 有限会社ヒューマンデータのFLEX10K10ブレッドボードキット
http://www.hdl.co.jp/CSP-004KIT.html

[2] 日本アルテラ株式会社の「MAX+PLUS II ソフトウェア・サポート」コーナー
http://www.altera.com/japan/support/software/sof-download_center_j.html