情報Ⅰ FPGA その１

年　　組　　番氏名＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

＜FPGA を利用した論理回路＞

コンピュータは様々な論理回路で構成されています。

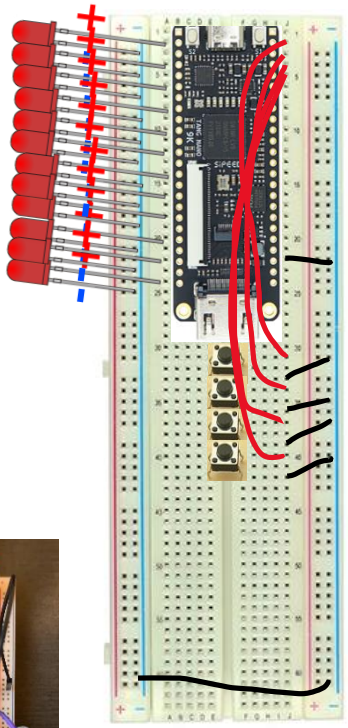
通常の論理回路は CPU や GPU や各種チップは回路構成を変更できない LSI（集積回路）ですが、FPGA は Field Programmable Gate Array の略で設計者が論理回路の構成をプログラミングできる機器です。

今回は FPGA を利用して組み合わせ回路や順序回路、CPU などの論理回路を実装していきます。

＜使用する機器＞

- Tang Nano 9k（FPGA を利用したボード）
- ブレッドボード　BB－102　（はんだ付けなしで回路を構成できる板）
- 抵抗付き LED
- タクトスイッチ
- ジャンプワイヤ

＜基板の作成＞

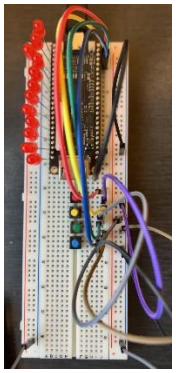


まずブレッドボード上に Tang Nano 9k の USB 端子側の上に向けて、左側のピンを B1 から B24,右側のピンを I1 から I24 に刺します。

次に 1 番目の LED の長い線を A2短い線を―、2 番目の LED の長い線を A4短い線を―、順に 12 番目の LED の長い線を A24短い線を―に刺します。

そのあとタクトスイッチを左右からピンが出る向きで、E31E33 と F31F33、E34E36 と F34F36、E37E39 と F37F39、E40E42 と F40F42 に刺します。

次にジャンプワイヤを J2と J31、J3 と J34、J4 と J37、J5 と J40、J23と―、J33と―、J36 と―、J39と―、J42 と―、左の―と右の―でつなぎます。



最後に Tang Nano 9k に USB ケーブルをつなぎ、パソコンの USB 端子につないでください。USB ハブではなくパソコンの USB 端子に直接つないでください。

＜GOWIN のインストール＞（元のプリントに追加 2025/5/28）

Tang nano 9k を使うための環境構築(GOWIN FPGA Designer)

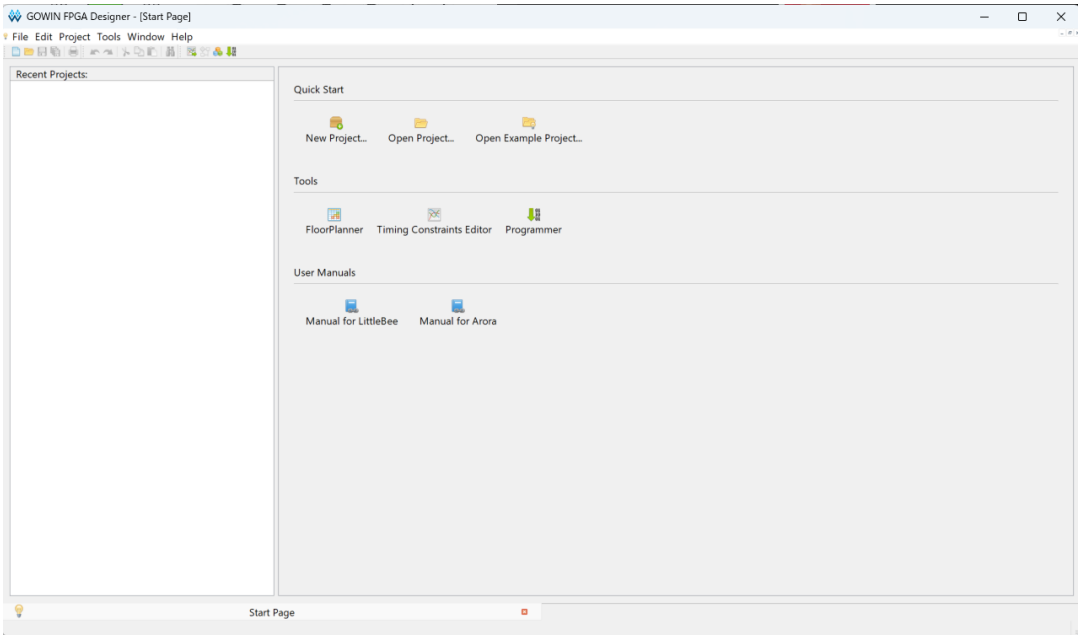
https://zenn.dev/nihinihikun/articles/2eb2844479638b

を参考に Gowin IDE をインストールしてライセンス設定してください。

＜動作確認 1＞

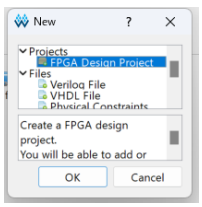
Gowin IDE を利用して FPGA に回路を書き込みます。

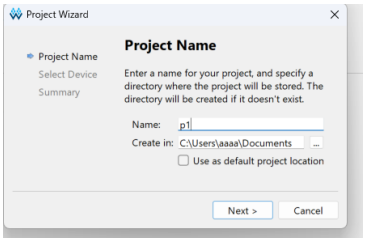
左下のメニュー→Gowin→Gowin を起動します。

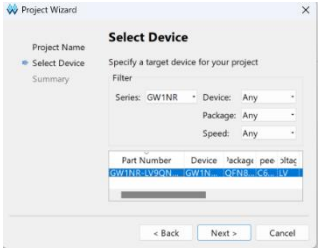


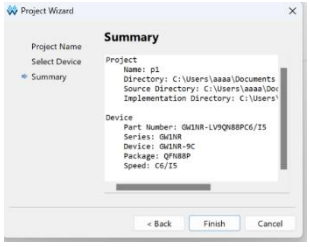
[動作確認用プロジェクトの作成]

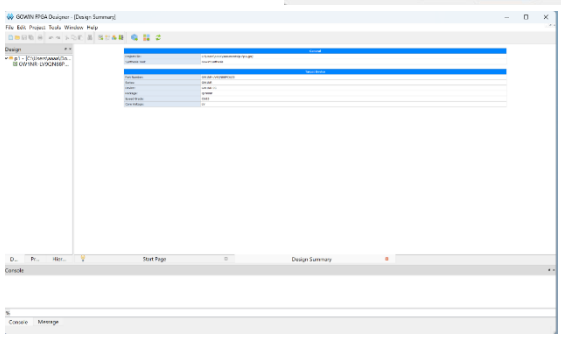
File→New を選び、Projects の FPGA Design Project を選び OK を押します。次に Project Name の Name を p1 にして Create in:はユーザーの Documents 以下にして Next を押します。Select Device の Series を GW1NR にして下の GW1NR-LV9QN…を選択し、Next を押します。最後に Summary で Finish を押します。







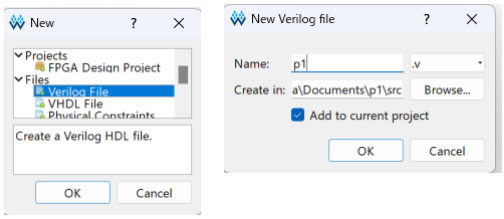


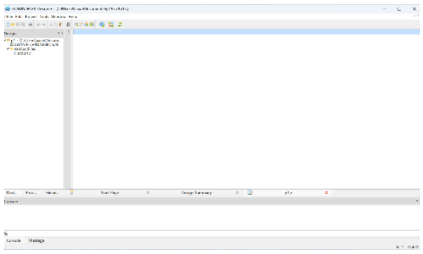


[動作確認用 Verilog コードの記述]

論理回路を Verilog というハードウェア記述言語で書きます。Verilog ではプログラミング言語でプログラムを記述するようなレジスタ転送レベル(Register Transfer Level)と呼ばれる抽象的なレベルで回路を記述できます。Verilog で記述した回路を論理合成してゲート回路（ネットリスト）を合成します。ネットリストを物理配線制約データと合わせて FPGA の物理回路表現に対応するビットストリームして FPGA に書き込みます。

File→New で Verilog File を選び OK を押します。New Verilog File で Name:に p1 を入れて OK を押します。





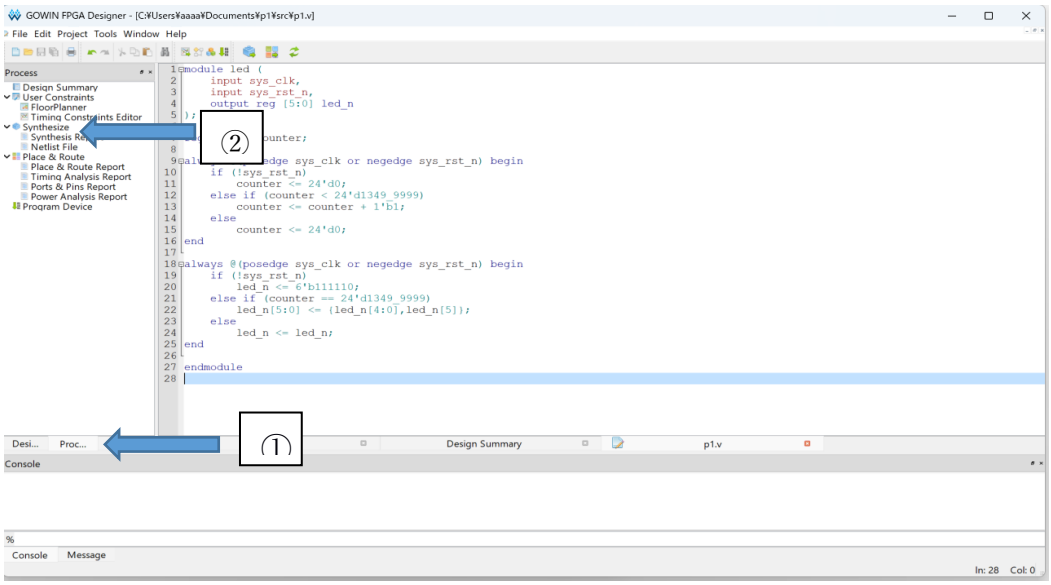
次のコードを記述します。

```
module led (
        input sys_clk,
        input sys_rst_n,
        output wire [5:0] led_n
);

reg [23:0] counter;
reg [5:0] leddata;

assign led_n = ~leddata[5:0];

always @(posedge sys_clk or negedge sys_rst_n) begin
        if (!sys_rst_n)
                counter <= 24'd0;
        else if (counter < 24'd1349_9999)
                counter <= counter + 1'b1;
        else
                counter <= 24'd0;
end

always @(posedge sys_clk or negedge sys_rst_n) begin
        if (!sys_rst_n)
```

```
            leddata <= 6'b000001;
        else if (counter == 24'd1349_9999)
            leddata[5:0] <= {leddata[4:0], leddata[5]};
end

endmodule
```

[動作確認用 Verilog コードの論理合成]

今記述した Verilog コードの論理合成を行います。



File→Save “p1.v”を押してファイルを保存し、左ペインの下の Process を押し、左ペインの Synthesize をダブルクリックして論理合成します。

問題なければ下のペインに

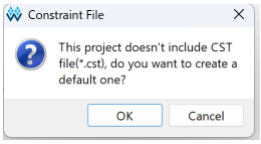
```
[100%] (略) completed
GowinSysthesis inish
```

と表示されて論理合成に成功し、Systhesis の左に緑チェックが付きます。

赤字でエラーが出た場合エラーメッセージに従い、エラーが出ないまで Verilog のコードを修正し保存して論理合成してください。Verilog コードを修正するときに右ペインにコードが表示されない場合は、左ペインの下の Design を押し、src＼p1.v をダブルクリックしてください。

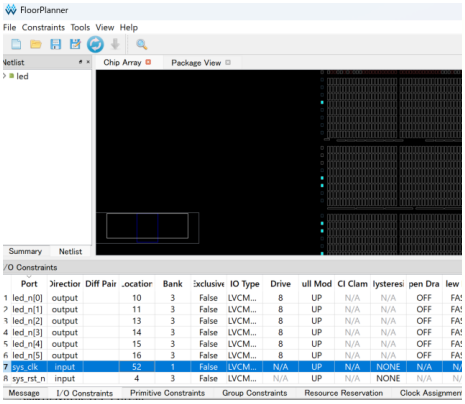
[動作確認用制約ファイルの作成(回路配線)]

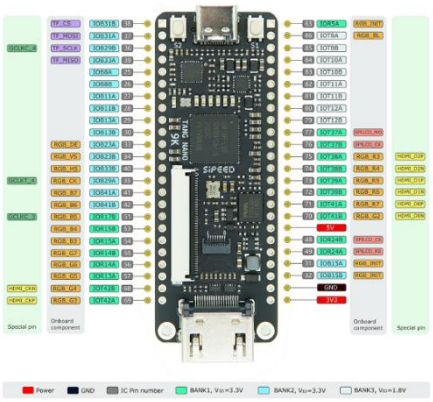
論理合成されたネットリストに Tang Nano 9k の LED やスイッチ、クロック信号線を配線します。FloorPlanner をダブルクリックし、Constraint File で OK を押して制約ファイルを作成します。



led_n[0]	led_n[1]	led_n[2]	led_n[3]	led_n[4]	led_n[5]	sys_clk	sys_rst
10	11	13	14	15	16	52	4

下ペインの I/O Constraints に led_n[0]の Location を 10、led_n[1]を 11、led_n[2]を 13、led_n[3]を 14、led_n[4]を 15、led_n[5]を 16、sys_clk を 52、sys_rst_n を4にします。





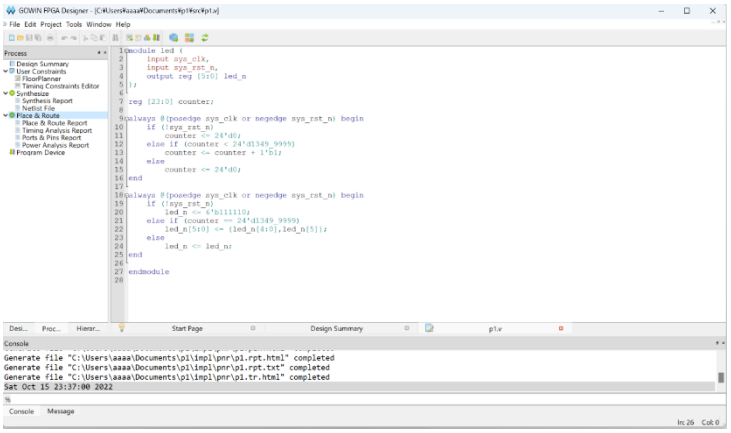
10,11,13,14,15,16 は Tang Nano 9k の USB 端子の下にある 6 個の LED につながる pin です。52 は 27 MHz のクロック信号、4 は USB 端子の右のスイッチです。

Tang Nano 9k は標準では Pull Mode が UP になるので、OFF のときはどこともつながらず ON のときに線とつながるタクトスイッチをつないだ場合、スイッチがオフの時は抵抗経由でプラス線、オンの時はマイナス線につながるようになっています。

File→Save を押して保存し×を押して閉じます。

[動作確認用ビットストリームの作成]

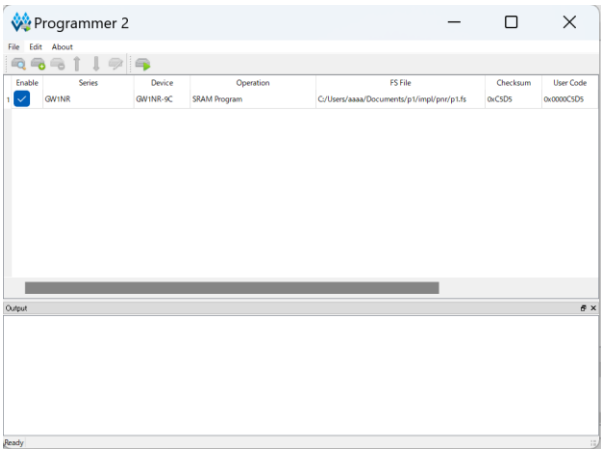
Place&Route をダブルクリックしてビットストリームを合成します。問題なければ Place&Route の左に緑チェックが付きます。問題がある場合は Floor Planner をダブルクリックして制約ファイルを確認したり、Design から Verilog コードを確認して、再度論理合成とビットストリームの作成を行ってください。

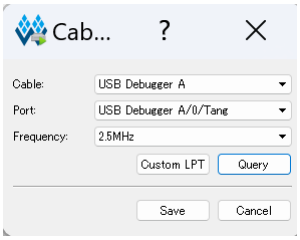


[動作確認用コードの FPGA への書き込み]

Program Device をダブルクリックして Programmer 2 を起動します。

初回起動時は Tang Nano 9k をうまく認識しないので、GW1NR 上で右クリックし Cable Setting→cable を選び、Query を押して Save を押してください。



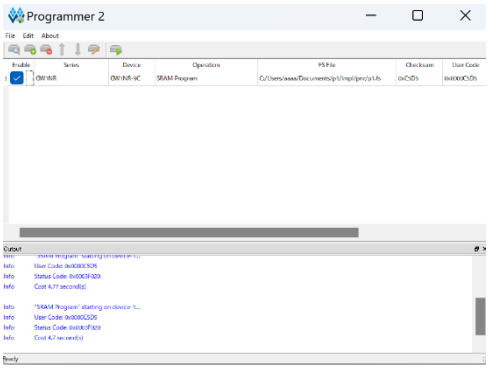


次に右クリックし Program/Configure を押してください。Progress Information が表示され、回路が FPGA に書き込まれます。

正しく書き込まれたならば下のペインに青地でメッセージが表示されます。

Tang Nano 9k の USB 端子の下の 6 つの LED が 1 秒ごとに右から順に点灯するのを繰り返すはずです。USB 端子の右のボタンを押すとリセットされ右端の LED から点灯します。

×を押して Programmer 2 を閉じてください。



＜動作確認 2＞

ブレッドボードに設置した 12 個の LED の動作確認をするコードを入力します。動作確認1で入力したファイルを修正して実装します。

もし動作確認1のプロジェクトを閉じてしまった場合は File→Open で保存した場所(ドキュメントの p1)内の p1.gprj を開き、Design タブから p1.v をダブルクリックしてください。

[動作確認用コードその 2 の実装]

次のコードを入力します。// 修正箇所 となっている行を追加・修正してください。最初のコードに led2を追加してブレッドボード上の LED を表示できるようにしています。

```
module led (
    input sys_clk,
    input sys_rst_n,
```

```
        output wire [5:0] led_n,
        output wire [11:0] led2 //  修正箇所
);

reg [23:0] counter;
reg [17:0] leddata; //  修正箇所

assign led_n = ~leddata[5:0];
assign led2 = leddata[17:6]; //  修正箇所

always @(posedge sys_clk or negedge sys_rst_n) begin
        if (!sys_rst_n)
                counter <= 24'd0;
        else if (counter < 24'd1349_9999)
                counter <= counter + 1'b1;
        else
                counter <= 24'd0;
end

always @(posedge sys_clk or negedge sys_rst_n) begin
        if (!sys_rst_n) begin
                leddata <= 18'b00_0000_0000_0001; //  修正箇所
        end else if (counter == 24'd1349_9999) begin
                leddata[17:0] <= {leddata[16:0],leddata[17]}; //  修正箇所
        end
end

endmodule
```

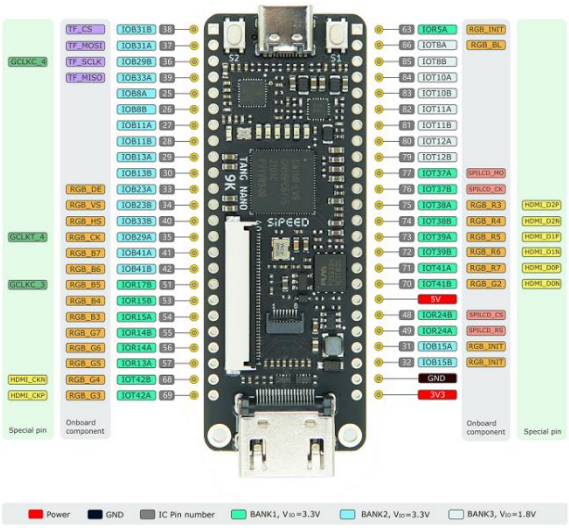
File→Save “p2.v”を押して保存し、Process から Synthesize をダブルクリックして論理合成します。

FloorPlanner をダブルクリックし、I/O Constraints に次のように入力してください。

led2[0]	led2[10]	led2[11]	led2[1]	led2[2]	led2[3]	led2[4]	led2[5]
37	57	69	39	26	28	30	34

led2[6]	led2[7]	led2[8]	led2[9]	led_n[0]	led_n[1]	led_n[2]	led_n[3]
35	42	53	55	10	11	13	14

led_n[4]	led_n[5]	sys_clk	sys_rst
15	16	52	4



変な番号付けになっているのは Tang Nano 9k の基板の都合によるものです。右のような番号付けになっています。

File→Save を押し、×を押して Floor Planner を閉じてください。

このまま Place & Route をダブルクリックするとエラーが出ます。57 番 pin が標準で別の目的に使うように設定されているからです。

Project→Configuration の Place & Route の Dual-Purpose Pin を押して、Use SSPI as regular IO にチェックを入れてOKを押してから、Place & Route をダブルクリックしてください。

最後に Program Device をダブルクリックして Programmer 2 を起動し、GW1NR で右クリックして Program/Configure を押してFPGAに書き込みしてください。

問題なければ、USB端子の下のLEDの右端から左端まで順に点灯し、そのあとブレッドボード上の左のLEDが上から順に点灯し、これを繰り返すはずです。USB端子の右のボタンを押すとリセットされ、USB端子下の右端のLEDからの点灯から開始します。

考察と感想

情報Ⅰ FPGA その 2

年　　組　　番氏名

＜コンピュータの構成要素＞

ここから本格的にコンピュータの内部構造を学習していきます。

コンピュータは物理的に処理するハードウェアとハードウェアを動かすための命令や手順を記述したソフトウェアから構成されています。

ハードウェアは次の4つの要素からできています。

- CPU（制御・演算）
- 記憶装置（主記憶装置・補助記憶装置）
- 入力装置
- 出力装置

ソフトウェアはデータとともに通常補助記憶装置に保存され、CPU が主記憶装置に送り、CPU が主記憶装置上のソフトウェアを実行します。

CPU はソフトウェアの命令に従い各装置を制御します。ソフトウェアの命令に従い、補助記憶装置からデータを読み取ったり入力装置からの入力を読み取り、主記憶装置上で処理・演算し、出力装置に送って出力したり補助記憶装置に書き込みしたりします。

このようにコンピュータの動作は制御・演算・記憶・入力・出力によって成り立っています。

＜組み合わせ回路＞

コンピュータを構成するハードウェアは、電気信号を電気が流れているか流れていないかの 2 つの状態を 0（偽）と1(真)に対応させ、その0と1に対して演算する論理回路でできています。

入力信号に対してさまざまな論理ゲートで演算して出力信号を返す回路を組み合わせ回路といいます。

今回はANDゲート、ORゲート、NOTゲート、XORゲートとそれを組み合わせて作る加算回路（半加算回路と全加算回路）を作成します。

[正論理と負論理]

通常電気が流れてない状態を0、電気が流れている状態を1に対応付けします。これを正論理といいます。逆に電気が流れていない状態を1、電気が流れている状態を0に対応付けすることを負論理といいます。

回路作成の都合上、正論理・負論理両方が組み合わさる場合があります。

今回の Tang Nano 9k の場合、内蔵 LED は負論理になっていて、LEDに 1 を入力すると消灯、0 を入力すると点灯となっています。またどこにも接続されない配線は標

準で電源に抵抗を介して接続するプルアップ回路なので、スイッチを押したときにの配線先をグラウンド（0V）に接続し、スイッチを押さない場合は電源につながるので1、スイッチを押すとグランドにつながるので0を出力します。

＜組み合わせ回路1　スイッチとLED＞

File→New から FPGA Design Project を選び、Name を comb1 にし、Series を GW1NR にして GW1NR-LV9QN…を選び、新しいプロジェクトを作成し、File→New から Verilog File を選び、Name を comb1 にして次の Verilog コードを入力してください。

```
module sw (
    input wire [3:0] sw_n,
    output wire [5:0] led_n,
    output wire [1:0] led2
);

assign led_n[3:0] = sw_n;
assign led_n[4] = 1'b0;
assign led_n[5] = 1'b1;

assign led2[0] = 1'b0;
assign led2[1] = 1'b1;

endmodule
```

次に File→Save を押して保存し、Process の Synthesis をダブルクリックして合成し、FloorPlanner の I/O Constraints に次のように入力してください。

led2[0]	led2[1]	led_n[0]	led_n[1]	led_n[2]	led_n[0]	led_n[4]	led_n[5]
37	39	10	11	13	14	15	16

sw_n[0]	sw_n[1]	sw_n[2]	sw_n[3]
86	85	84	83

File→Save を押して保存し、×を押して閉じて、Place&Route をダブルクリックして合成し、Program Device をダブルクリックして、GW1NR で右クリックして Program/Configure を実行してFPGAに書き込みしてください。

[Verilog の基本構造]

```
module  モジュール名(
入力信号や出力信号
);
回路本体

endmodule
```

今回のVerilogコードは sw というモジュール名で作成します。

[入出力信号線定義]

```
    input wire [3:0] sw_n,
    output wire [5:0] led_n,
    output wire [1:0] led2
```

input wire [3:0] sw_n が入力信号、output wire[5:0] led_n と output wire[1:0] led2 が出力信号です。wire は配線の意味で wire[3:0] sw_n とすると sw_n[0]、sw_n[1]、sw_n[2]、sw_n[3]の4本の配線を作成します。wire[5:0] led_n は同様に led_n[0]、led_n[1]、led_n[2]、led_n[3]、led_n[4]、led_n[5]の 6 本の配線、 wire [1:0] led2 は led2[0]、led2[1]の 2 本の配線です。1 本だけの配線をしたい場合は wire led3 のようにします。配線名に_n を付けているのは負論理の配線です。区別しやすくするためにそうしています。

複数定義する場合は,で区切ります。

[回路配線定義]

```
assign led_n[3:0] = sw_n;
assign led_n[4] = 1'b0;
assign led_n[5] = 1'b1;

assign led2[0] = 1'b0;
assign led2[1] = 1'b1;
```

assign 出力配線名 = 入力配線名で配線をつなぐことができます。;で区切って定義していきます。

assign led3 = sw3 とすれば sw3 に入力された信号が led3 に出力されます。複数本の配線は配線名[使いたい配線]で指定できます。assign led_n[3:0] = sw_n とすると、led_n[3]と sw_n[3]、led_n[2] = sw_n[2]、led_n[1] = sw_n[1]、led_n[0] = sw_n[0]の配線がつながります。assign led_n[4] = 1'b0 は led_n[4]に 0 を入力する意味です。1'b0 は 1bit の二進数の0の意味です。4'b1011 ならば二進数 1011、4'd9 ならば 4bit の十進数の 9(二進数の 1001)の意味になります。

led_n は負論理なので 0 を入力すると点灯、1 を入力すると消灯です。内蔵 led 右から 5 番目は常に点灯、6 番目は常に消灯になります。外部 led は正論理なので 0 を入力すると消灯、1を入力すると点灯です。外部 led 上から 1 番目が常に消灯、2 番目が常に点灯になります。

内蔵 led の右から 1 番目から 4 番目はスイッチの状態によって変わります。どちらも負論理です。スイッチを押すと 0、スイッチを押さないと 1 で、led は 0 だと点灯、1だと消灯なので、スイッチを押すと対応する led が点灯します。

実際にそのように動作しているか動作確認してください。

＜組み合わせ回路 2 AND、OR、＞

これまでと同様に comb2 で FPGA Design Project と Verilog File を作成し、次のコードを入力して、Synthesize してください。

```
module comb2 (
	input wire [1:0] sw_n,
	output wire [6:0] ledout
);

wire [1:0] sw;
wire [6:0] led;
assign sw = ~sw_n;
assign ledout = led;

assign led[0] = sw[0];
assign led[1] = sw[1];

assign led[2] = sw[0] & sw[1]; // AND
assign led[3] = sw[0] | sw[1]; // OR
```

```
assign led[4] = sw[0] ^ sw[1]; // XOR
assign led[5] = ~sw[0]; // NOT sw[0]
assign led[6] = ~sw[1]; // NOT sw[1]

endmodule
```

FloorPlannar は次のようにします。

ledout[0]	ledout[1]	ledout[2]	ledout[3]	ledout[4]	ledout[5]	ledout[6]
37	39	26	28	30	34	35

sw_n[0]	sw_n[1]
86	85

Place&Route を行い、Program Device からFPGAへ回路を書き込みしてください。

[入出力回路]

```
    input wire [1:0] sw_n,
    output wire [6:0] ledout
```

入力はスイッチ 2 個 sw_n[0]と sw_n[1]、出力は外部 led7 個 ledout[0]、ledout[1]、ledout[2]、ledout[3]、ledout[4]、ledout[5]、ledout[6]です。

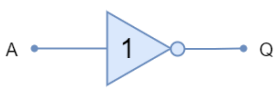
[NOT ゲート]

このままスイッチが負論理のままだと扱いにくいので、正論理の sw と led に sw_n と ledout をつなぎます。

```
wire [1:0] sw;
wire [6:0] led;
assign sw = ~sw_n;
assign ledout = led;
```

~配線名とすると信号が反転します。0 なら 1、1 なら 0 になります。反転する論理ゲートを NOT ゲート(否定)といいます。次のように 0 を偽、1を真として真偽の値を表にしたものを真偽値表といいます。

入力 A	出力 Q
0	1
1	0



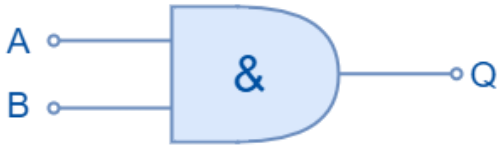
[AND ゲート・OR ゲート・XOR ゲート]

```
assign led[0] = sw[0];
assign led[1] = sw[1];

assign led[2] = sw[0] & sw[1]; // AND
assign led[3] = sw[0] | sw[1]; // OR
assign led[4] = sw[0] ^ sw[1]; // XOR
assign led[5] = ~sw[0]; // NOT sw[0]
assign led[6] = ~sw[1]; // NOT sw[1]
```

led[0]と led[1]は sw[0]と sw[1]をそのままつないでいます。スイッチを押せば対応する led が点灯します。

配線名 1 & 配線名 2 とすると両方が1だった場合は1, それ以外は 0 を出力します。これを AND ゲート(論理積)といいます。

入力 A	入力 B	出力 Q
0	0	0
0	1	0
1	0	0
1	1	1



led[2]は sw[0]と sw[1]の論理積なのでスイッチを両方押したときだけ上から 3 番目の led が点灯します。

// はコメントです。この右側に書かれている内容は何も意味しません。

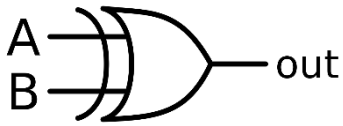
配線名 1 | 配線名 2 とするとどちらかが1ならば1、両方 0 ならば 0 を出力します。これを OR ゲート(論理和)といいます。

入力 A	入力 B	出力 Q
0	0	0
0	1	1
1	0	1
1	1	1

led[3]は sw[0]と sw[1]の論理積なのでどちらかのスイッチを押せば上から 4 番目の led が点灯します。

配線名 1 ^ 配線名 2 とすると違う信号なら 1、同じ信号なら 0 を出力します。これを XOR ゲート(排他的論理和)といいます。XOR ゲートは、OR ゲートの出力と AND ゲートの出力の否定を AND ゲートに入力した場合と同じ出力になります。

入力 A	入力 B	出力 out
0	0	0
0	1	1
1	0	1
1	1	0



led[4]は sw[0]と sw[1]の排他的論理和なので片方のみ押した場合、上から 5 番目の led が点灯します。

led[5]は sw[0]の否定なので、上から 6 番目の led は sw[0]を押すと消灯押さないと点灯、led[6]は同様に sw[1]の否定なので、上から 7 番目の led は sw[1]を押すと消灯押さないと点灯です。

このように動作しているか確認してください。

＜組み合わせ回路 3　半加算回路＞

1bit+1bit の足し算回路を作成します。

comb3 として次の Verilog コードを入力し、FloorPlannar を設定して FPGA に回路を書き込んでください。

```
module comb2 (
	input wire [1:0] sw_n,
	output wire [5:0] ledout
);

wire [1:0] sw;
wire [5:0] led;

assign sw = ~sw_n;
assign ledout = led;

assign led[0] = sw[0];
assign led[1] = sw[1];

assign led[3:2] = sw[0] + sw[1]; // ADD s0 + s1

assign led[4] = sw[0] ^ sw[1]; // XOR
```

```
assign led[5] = sw[0] & sw[1]; // AND

endmodule
```

[半加算回路]

```
    input wire [1:0] sw_n,
    output wire [5:0] ledout
```

```
wire [1:0] sw;
wire [5:0] led;

assign sw = ~sw_n;
assign ledout = led;

assign led[0] = sw[0];
assign led[1] = sw[1];
```

ここまでは comb2 と同じです。

```
assign led[3:2] = sw[0] + sw[1]; // ADD s0 + s1
```

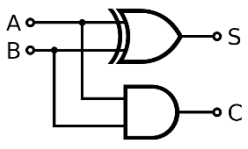
1bit+1bit の演算は桁上げがあるので 2bit の出力になります。Verilog の加算機能を利用して sw[0]+sw[1]の 1bit 目が led[2]に、2bit 目(桁上げ)が led[3]に出力されます。

```
assign led[4] = sw[0] ^ sw[1]; // XOR
assign led[5] = sw[0] & sw[1]; // AND
```

加算回路を論理ゲートの組み合わせで実装したのがこちらです。

0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 1 + 1 = 10 なので次の真偽値表の回路を作成します。これを半加算回路といいます。

入力 A	入力 B	出力 S	出力桁上げ C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



sw[0] + sw[1]の 1bit 目は sw[0]と sw[1]の排他的論理和の出力と同じです。なので sw[0] ^ sw[1]を led[4]に出力しています。sw[0]+sw[1]の 2bit 目(桁上げ)は sw[0]と sw[1]の論理積の出力と同じです。なので sw[0] & sw[1]を led[5]に出力しています。

上から 3 番目と 5 番目、4 番目と 6 番目が同じ動作しているか確認してください。

＜組み合わせ回路4　全加算回路＞

今度は 2bit+2bit の加算回路を実装します。comb4 として次の Verilog コードと FloorPlannar を入力して FPGA に書き込んでください。55 ピンが SSPI と重なるので Project→Configuration の Dual-PurposePIN の Use SSPI as regular IO にチェックを入れてから合成してください。

```
module comb4 (
	input wire [3:0] sw_n,
	output wire [9:0] ledout
);

wire [3:0] sw;
wire [9:0] led;
wire s1, c1, st, ct1, ct2, s2, c2;

assign sw = ~sw_n;
assign ledout = led;

assign led[0] = sw[0];
assign led[1] = sw[1];
assign led[2] = sw[2];
assign led[3] = sw[3];

assign led[6:4] = {sw[1], sw[0]} + {sw[3], sw[2]}; // ADD s1s0 + s3s2

assign s1 = sw[0] ^ sw[2]; // XOR
assign c1 = sw[0] & sw[2]; // AND
assign led[7] = s1;

assign st = sw[1] ^ sw[3]; // XOR
assign ct1 = sw[1] & sw[3]; // AND
assign s2 = st ^ c1; // XOR
assign ct2 = st & c1; // AND
assign c2 = ct2 | ct1; // OR
```

```
assign led[8] = s2;
assign led[9] = c2;

endmodule
```

FloorPlannar は次のようにします。

ledout[0]	ledout[1]	ledout[2]	ledout[3]	ledout[4]	ledout[5]	ledout[6]	ledout[7]
37	39	26	28	30	34	35	42

ledout[8]	ledout[9]	sw_n[0]	sw_n[1]	sw_n[2]	sw_n[3]
53	55	86	85	84	83

[入出力回路と全加算回路用信号線]

```
    input wire [3:0] sw_n,
    output wire [9:0] ledout
```

```
wire [3:0] sw;
wire [9:0] led;
wire s1, c1, st, ct1, ct2, s2, c2;

assign sw = ~sw_n;
assign ledout = led;
```

4個のスイッチ入力の sw_n と 10 個の led 出力用 ledout を正論理の sw と led に接続しています。

2bit+2bit の演算は 1bit 目は comb3 の半加算回路でいいのですが、2bit 目の演算は 2bit 目+2bit 目+桁上げの 3 入力の回路が必要になるので、そのための配線として、s1,c1,st,ct1,ct2,s2,c2 の配線を用意しています。

[2bit 同士の加算回路]

```
assign led[6:4] = {sw[1], sw[0]} + {sw[3], sw[2]}; // ADD s1s0 + s3s2
```

Verilog の機能を加算回路の実装では、sw[0]を入力 1 の 1bit 目、sw[1]を入力 1 の 2bit 目、sw[2]を入力2の 1bit 目、sw[3]を入力 2 の 2bit 目とした場合、{ sw[1], sw[0]}

+ {sw[3], sw[2]}とします。演算結果は上から 5 番目の led が 1bit 目、6 番目が 2bit 目、7 番目が桁上げになります。

[1bit 目の加算回路]

```
assign s1 = sw[0] ^ sw[2]; // XOR
assign c1 = sw[0] & sw[2]; // AND
assign led[7] = s1;
```

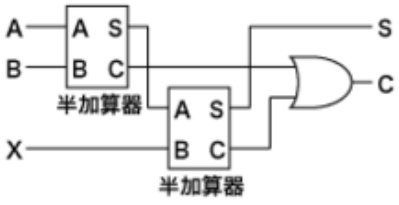
1bit 目はそのまま半加算回路で OK です。上から 8 番目の led[7]に出力しています。

[2bit 目の加算回路と全加算回路]

```
assign st = sw[1] ^ sw[3]; // XOR
assign ct1 = sw[1] & sw[3]; // AND
assign s2 = st ^ c1; // XOR
assign ct2 = st & c1; // AND
assign c2 = ct2 | ct1; // OR

assign led[8] = s2;
assign led[9] = c2;
```

2bit 目は 1bit 目の桁上げを含めて 3 つの入力の加算を行う必要があります。
桁上げを含めて加算できるようにした回路が全加算回路です。次の真偽値表になります。

入力 A	入力 B	入力 X (桁上げ)	出力 S	出力 C (桁上げ)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



複雑ですが、A と B で半加算回路を作り、出力を st 桁上げ ct とし、st と X の半加算回路を作り、その出力を s2 桁上げを ct2 とすると、s2 が S、ct と ct2 の論理和が C になります。

2bit 目の演算結果を上から 9 番目の led[8]に出力し、2bit 目の桁上げを上から 10 番目の led[9]に出力しています。

Verilog の機能を利用した結果が上から 5～7番目なので、8～10 番目と同じかどうか確認してみてください。

考察と感想

情報Ⅰ FPGA その 3

年　　組　　番氏名＿＿＿＿＿＿＿＿＿＿＿＿

＜フリップフロップ回路＞

フリップフロップ回路は 1bit の情報を記憶できる回路です。

フリップフロップ回路を複数組み合わせることで複数の bit の保存ができます。通常レジスタと呼ばれます。多数のフリップフロップ回路を組み合わせてアドレスを指定することにより読み書きできる回路を SRAM(Static RAM)といいます。

＜順序回路＞

組み合わせ回路は基本的に同じ入力に対して同じ出力を行います。

組み合わせ回路にフリップフロップ回路を組み合わせると、過去の入力等の状態を保存することで、同じ入力に対してさまざまな処理を行い多様な出力ができるようになります。入力と出力だけでなく内部状態を保存して動作する論理回路を順序回路といいます。

CPU は命令の処理結果や入力を保存して内部状態に応じて動作が変化する順序回路です。GPU や多くの周辺機器を提供するチップも順序回路になっています。

＜順序回路 1　スイッチカウンター＞

コンピュータ回路をより深く理解するために、スイッチを押すごとに数が増える簡単な順序回路を実装してみましょう。

seq1 として FPGA Design Project と Verilog File を作成し、FloorPlannar で配線して、論理合成して FPGA に回路を書き込んでください。

```
module seq1 (
   input wire button_n,
   input wire reset_n,
   output wire [3:0] led_n
);

wire [3:0] leddata;
assign led_n[3:0] = ~leddata;

reg [3:0] counter = 4'b0000;
assign leddata = counter;

always @(negedge button_n or negedge reset_n) begin
```

```
    if (!reset_n) begin
        counter <= 4'b0000;
    end else begin
        counter <= counter + 1'b1;
    end
end

endmodule
```

button_n	led_n [0]	led_n[1]	led_n[2]	led_n[3]	reset_n
3	10	11	13	14	4

[入出力回路]

```
    input wire button_n,
    input wire reset_n,
    output wire [3:0] led_n
```

```
wire [3:0] leddata;
assign led_n[3:0] = ~leddata;
```

内蔵 led は負論理なのでこれまでと同様に NOT ゲートを通して leddata に正論理で処理します。USB 端子右のボタンを reset_n、USB 端子左のボタンを button_n に割り当てます。今回ボタンは負論理のまま扱います。

[レジスタ]

```
reg [3:0] counter = 4'b0000;
assign leddata = counter;
```

reg [bit 数] レジスタ名 = 初期値とすると、初期値に指定した値の bit 数のフリップフロップ回路が作られレジスタ名で読み書きできるようになります。reg [3:0]なので counter[3],counter[2],counter[1],counter[0]の 4bit のレジスタで、二進数 0000 で初期化されます。この内容を leddata に配線するので counter の内容が内蔵 led に出力されます。

[順序回路]

```
always @(negedge button_n or negedge reset_n) begin
    if (!reset_n) begin
        counter <= 4'b0000;
```

```
    end else begin
        counter <= counter + 1'b1;
    end
end
```

begin … end で囲まれた部分が一つの処理ブロックになります。always@(...) begin …end が順序回路の処理ブロックです。if() begin … end else begin … end も同様です。

順序回路は入力の変化に応じて動作し、処理結果をレジスタに保存します。always @(negedge button_n or negedge reset_n)の@()内にどんな入力の変化に反応するかを記述します。negedge button_n は button_n が 1 から 0 に変化する場合、negedge reset_n は reset_n が 1 から 0 に変化する場合を意味し、negedge button_n or negedge reset_n は button_n または reset_n のどちらかが 1 から 0 に変化した場合に反応します。USB 端子右のボタンを押すと reset_n が 1 から 0 に変化し、USB 端子左のボタンを押すと button_n が 1 から 0 に変化するので、どちらかのボタンを押したときに、次の行以降に記述されている処理内容が実行されます。

!reset_n は reset_n を反転されたものなので、reset_n が 0 ならば!reset_n は 1 です。if(!reset_n)で reset_n が押されたかそうでないか判定し、reset_n が押された場合 counter <= 4'b0000 を行い、reset_n が押されていない場合この処理内容は button_n か reset_n かどちらかが押された場合なので、button_n が押されたことを意味し、counter <= counter + 1'b1 が行われます。counter <= 4'b0000 はレジスタ counter の保存内容を二進数 0000 にするという意味です。counter <= counter + 1'b1 は counter+1 を counter に入れる、つまり counter の内容を+1 するという意味です。

USB端子左のボタンを押すとそのたびに counter が+1 され、4 つの led が counter の内容を二進数で表示していきます。二進数 1111 の次は 0000 になります。USB 端子右のボタンを押すと counter が 0000 にリセットされて led が消灯するはずです。

10 進数	0	1	2	3	4	5	6	7
2 進数	0000	0001	0010	0011	0100	0101	0110	0111
10 進数	8	9	10	11	12	13	14	15
2 進数	1000	1001	1010	1011	1100	1101	1110	1111

なお、一回押しただけなのに複数押された状態になることがあると思います。スイッチは金属をくっつけるか離すかだけの機械なので、正確に ON/OFF できるわけではなく細かい ON/OFF を発生させてしまう場合があるからです。正確に ON/OFF できない現象をチャタリング(chattering)といいます。内蔵ボタンはまだましで、配線変更して外部

スイッチにしてみると激しくチャタリングを起こすので、時間があれば試してみてください。

＜順序回路 2　クロックカウンター＞

CPU などハードウェアのほとんどは一定時間ごとに 1 と 0 を切り替えるクロック信号に従って動作し、内部の状態を更新していきます。これを同期式順序回路といいます。

スイッチに応じて動くなどクロック信号に従わず動作する順序回路を非同期式順序回路といいます。

Tang Nano 9k には 1 秒間に 27,000,000 回(27MHz)　1 と 0 を切り替えるクロック信号が内蔵されているので、これを利用して 1 秒ごとに counter を+1 する回路を実装してみましょう。

seq2 で FPGA Design Project と Verilog File を作成し、FloorPlannar で配線し、論理合成して FPGA に回路を書き込んでください。

```
module seq2 (
        input wire clock,
        input wire reset_n,
        output wire [3:0] led_n
);

wire [3:0] leddata;
assign led_n[3:0] = ~leddata;

reg [3:0] counter = 4'b0000;
assign leddata = counter;

reg [24:0] ct = 25'd0;

always @(posedge clock or negedge reset_n) begin
    if (!reset_n) begin
        ct <= 25'd0;
        counter <= 4'b0;
    end else if (ct < 25'd2699_9999) begin
        ct <= ct + 1'b1;
    end else begin
        ct <= 25'd0;
```

```
        counter <= counter + 1'b1;
    end
end

endmodule
```

clock	led_n [0]	led_n[1]	led_n[2]	led_n[3]	reset_n
52	10	11	13	14	4

[入出力回路]

```
        input wire clock,
        input wire reset_n,
        output wire [3:0] led_n
```

```
wire [3:0] leddata;
assign led_n[3:0] = ~leddata;

reg [3:0] counter = 4'b0000;
assign leddata = counter;
```

USB 端子左ボタン button_n の代わりに 27MHz の clock 信号を入力に使います。それ以外は seq1 と同じです。

[クロックカウンター]

```
reg [24:0] ct = 25'd0;
```

1 秒ごとに動作させるために何回クロック信号が来たかをカウントする 25bit のレジスタ ct を作成します。1 秒間に 27MHz クロック信号が来るならば 27000000 まで数えられればいいので、25bit あれば 2 の 25 乗= 33554432 まで扱えるので問題ありません。

[順序回路]

```
always @(posedge clock or negedge reset_n) begin
    if (!reset_n) begin
        ct <= 25'd0;
        counter <= 4'b0;
    end else if (ct < 25'd2699_9999) begin
        ct <= ct + 1'b1;
    end else begin
```

```
        ct <= 25'd0;
        counter <= counter + 1'b1;
    end
end
```

if (!reset_n)で分岐し、USB 端子右のボタンが押された時には ct <= 25'd0 と counter <= 4'b0 で ct と counter を 0 にしてリセットします。

クロック信号に対しては、ct が 26999999 未満ならば ct を＋1し、ct が 26999999 になったら、ct を 0 にし counter を＋1します(ct は 0 から始まっているので 26999999 が 27000000 番目です)。1 秒間にクロック信号が 27000000 回来るので、こうすることで 1 秒ごとに counter を＋1できます。else if (ct < 25'd2699_9999)で十進数 26999999 未満か比較し、そうならば ct <= ct + 1'b1 で ct を+1 し、そうでないならば、ct <= 25'd0 で ct を0にし、counter <= counter + 1'b1 で counter を+1 しています。

これで 1 秒ごとに counter が+1 されて led に二進数で counter の内容が表示されるはずです。

＜CPU1 メモリの読み込み＞

CPU は、メモリから命令を読み込み、命令を解読し、命令に従い入力装置や記憶装置からデータを読み込み、演算し、結果を出力装置や記憶装置に書き込みます。

ここからは 4bit TD4 の仕様に基づくCPUを実装します。

[TD4 の構成要素]

- 8bit×16 命令メモリ rom
- 4bit プログラムカウンタ pc(命令アドレスを保持)
- 4bit データレジスタ A(reg_a)、B(reg_b)
- 4bit 出力レジスタ OUT レジスタ(reg_out)
- 4bit 入力スイッチ 入力ポート(sw_in)
- 4bit + 4bit 加算回路 ALU
- 桁上げフラグレジスタ cflag

[TD4 の命令]

1 命令 8bit 固定で上位 4bit が命令内容で下位 4bit がデータ（即値,Im）です。この命令を機械語といいます。ニモニックは人間にわかりやすい表記で命令を表したもので、アセンブラにより機械語に変換できます。今回は直接機械語で命令を入力します。

命令	ニモニック	意味
0000	ADD A, Im	A + 即値(Im)を A に代入 (A + Im => A)
0001	MOV A, B	B を A に代入(B + 0000 => A)
0010	IN A	入力ポートの値を A に代入(IN + 0000 => A)
0011	MOV A, Im	即値(Im)の値を A に代入(Im + 0000 => A)
0100	MOV B, A	A を B に代入 (A + 0000 => B)
0101	ADD B, Im	B + 即値(Im)を B に代入 (B + Im => B)
0110	IN B	入力ポートの値を B に代入 (IN + 0000 => B)
0111	MOV B, Im	即値(Im)の値を B に代入 (Im + 0000 => B)
1001	OUT B	B を OUT レジスタに代入 (B + 0000 => OUT)
1011	OUT Im	即値(im)を OUT レジスタに代入 (Im + 0000 => OUT)
1110	JNC Im	桁上げしていない(cflags が 0)なら Im にジャンプ (Im + 0000 => PC if not carry)
1111	JMP Im	Im にジャンプ (Im + 0000 => PC)

上位 4bit の最上位 2bit が 00,01,10 の場合、最上位 2bit が ALU の結果の出力先で下位 2bit がALUの入力元、最上位 2bit が 11 の場合、下位 2bit が 11 の場合(無条件分岐命令)と下位 2bit が 10 で cflag が 0 の場合(条件分岐)ALU 出力先が pc、ALU の入力元が 0 固定となっています。

cpu で FPGA Design Project と Verilog File を作成します。

```
module cpu (
    input wire clock,
    input wire reset_n,
    input wire[3:0] sw_n,
    output wire [4:0] led_n,
```

```
        output wire [11:0] led2
);

wire [16:0] leddata;
assign led_n[4:0] = ~leddata[4:0];
assign led2 = leddata[16:5];

wire [3:0] sw_in;
assign sw_in = ~sw_n;

reg [3:0] pc = 4'b0000;
assign leddata[3:0] = pc;

reg [24:0] ct = 25'd0;

reg [7:0] rom[0:15];
initial $readmemb("rom1.txt", rom);

wire [7:0] opcode;
assign opcode = rom[pc];
assign leddata[12:5] = opcode;

always @(posedge clock or negedge reset_n) begin
    if (!reset_n) begin
        ct <= 25'd0;
        pc <= 4'b0;
    end else if (ct < 25'd2699_9999) begin
        ct <= ct + 1'b1;
    end else begin
        ct <= 25'd0;
        pc <= pc + 1'b1;
    end
end

endmodule
```

次に論理合成する前に、エクスプローラーでプロジェクトのフォルダ(通常はドキュメントのプロジェクト名(cpu))の src を開き右クリックの新規作成→テキストドキュメントを作成し名前を rom1.txt(拡張子非表示の場合は rom1)にして次の 16 行分のテキストを入力してください。この 2 進数 8bit×16 行が命令メモリの内容になります。

```
10110111
00000001
11100001
00000001
11100011
10110110
00000001
11100110
00000001
11101000
10110000
10110100
00000001
11101010
10111000
11111111
```

このあと Synthesize で論理合成してください。rom.txt 作成前に論理合成するとファイルがないという警告が下ペインに表示されるはずです。ファイルを作成したら Synthesize で右クリックして強制的に論理合成させてください。

FloorPlannar で配線し、Project→Configuration で Dual-Propose Pin で Use SSPI as regular IO にチェックを入れ、論理合成して FPGA に回路を書き込んでください。ここからはプロジェクトを分けると大変なので、このプロジェクトに追記していく形で最後まで実装していきます。

clock	led2[0]	led2[10]	led2[11]	led2[1]	led2[2]	led2[3]	led2[4]
52	37	57	69	39	26	28	30

led2[5]	led2[6]	led2[7]	led2[8]	led2[9]	led_n[0]	led_n[1]	led_n[2]
34	35	42	53	55	10	11	13

led_n[3]	led_n[4]	reset_n	sw_n[0]	sw_n[1]	sw_n[2]	sw_n[3]
14	15	4	86	85	84	83

[入出力回路]

```
    input wire clock,
    input wire reset_n,
    input wire[3:0] sw_n,
    output wire [4:0] led_n,
    output wire [11:0] led2
```

```
wire [16:0] leddata;
assign led_n[4:0] = ~leddata[4:0];
assign led2 = leddata[16:5];

wire [3:0] sw_in;
assign sw_in = ~sw_n;
```

クロック信号(clock)、USB 端子右ボタン（reset_n）、スイッチ 4 個(sw_n)、内蔵 led5 個(led_n)(左端の led は未使用)、外部 led12 個(led2)を使用し、負論理の sw_n を正論理の入力ポート sw_in にし、内蔵 led を正論理にして外部 led と合わせて leddata[16:0]として扱うようにしています。

[クロックカウンターとプログラムカウンター]

```
reg [3:0] pc = 4'b0000;
assign leddata[3:0] = pc;

reg [24:0] ct = 25'd0;
```

seq2 から counter から pc に名前を変えていますが同様に 1 秒ごとに+1 されます。内蔵 led4 個に内容を表示するようにしています。クロックカウンターct は seq2 と同じです。

[命令メモリとオペコード]

```
reg [7:0] rom[0:15];
initial $readmemb("rom1.txt", rom);
```

```
wire [7:0] opcode;
assign opcode = rom[pc];
assign leddata[12:5] = opcode;
```

reg [7:0] rom[0:15]で 8bit のレジスタ rom を rom[0]から rom[15]まで 16 個作成しています。Rom は rom1.txt の内容で初期化されます。rom1.txt は 1 行ごとに 8bit の二進数が書かれていて、1 行目の 10110111 が rom[0]、2 行目の 00000001 が rom[1]の順に初期化されます。

8bit の opcode に rom[pc]の内容が配線されます。pc が 0 ならば rom[pc]は rom[0]なので opcode は 10110111 になります。

opcode を外部 led の 1 番目から 8 番目につないでいます。opcode[0]が leddata[5]なので led2[0]、opcode[1]が leddata[6]=>led2[1]のように下位 bit から順に led に対応付けされています。

[順序回路]

```
always @(posedge clock or negedge reset_n) begin
   if (!reset_n) begin
      ct <= 25'd0;
      pc <= 4'b0;
   end else if (ct < 25'd2699_9999) begin
      ct <= ct + 1'b1;
   end else begin
      ct <= 25'd0;
      pc <= pc + 1'b1;
   end
end
```

seq2 とほぼ同じです。Counter が pc になっているだけです。

1 秒ごとに pc が+1 され内蔵 led に値が二進数で表示され、rom1.txt の行のうち pc に対応する行の二進数が下位 bit から上位 bit の順に外部 led の上から 1 番目から 8 番目までに表示されるはずです。

考察と感想

情報 I FPGA その 4

年　　組　　番氏名

＜CPU2 命令デコーダの実装 1 と reg _a と ALU と cflag の実装＞

ここからは、rom から読み取った命令を解読する命令デコーダと、命令内容に従い演算を行う ALU と結果を反映するライトバックを実装します。即値 im の内容を A レジスタに書き込む『MOV A, Im』命令(0011xxxx)と、即値 im と A レジスタを加算して A レジスタにいれる『ADD A, Im』命令(0011xxxx)を実装します。

CPU1の内容を次のように修正してください。修正箇所は// 修正箇所で示しています。

```
module cpu (
        input wire clock,
        input wire reset_n,
        input wire[3:0] sw_n,
        output wire [4:0] led_n,
        output wire [11:0] led2
);

wire [16:0] leddata;
assign led_n[4:0] = ~leddata[4:0];
assign led2 = leddata[16:5];

wire [3:0] sw_in;
assign sw_in = ~sw_n;

reg [3:0] pc = 4'b0000;
assign leddata[3:0] = pc;

reg [24:0] ct = 25'd0;

reg [7:0] rom[0:15];
initial $readmemb("rom2.txt", rom); //  修正箇所

wire [7:0] opcode;
assign opcode = rom[pc];
assign leddata[12:5] = opcode;
```

```
// reg_a
reg [3:0] reg_a = 4'b0; //  修正箇所
assign leddata[16:13] = reg_a; //  修正箇所

// cflag
reg cflag = 1'b0;   //  修正箇所
assign leddata[4] = cflag;   //  修正箇所

//  デコーダー
wire [1:0] alu_sel;   //  修正箇所
assign alu_sel = opcode[5:4];   //  修正箇所
wire [1:0] load_sel;   //  修正箇所
assign load_sel = opcode[7:6];   //  修正箇所
wire [3:0] im;   //  修正箇所
assign im = opcode[3:0];   //  修正箇所

// ALU
wire [3:0] alu_in;//  修正箇所
assign alu_in = (alu_sel == 2'b00) ? reg_a : 4'b0000; //  修正箇所

wire [3:0] alu_out;//  修正箇所
wire nextcflag;//  修正箇所
assign {nextcflag, alu_out} = alu_in + im; //  修正箇所

always @(posedge clock or negedge reset_n) begin
   if (!reset_n) begin
      ct <= 25'd0;
      pc <= 4'b0;
      reg_a <= 4'b0; //  修正箇所
      cflag <= 1'b0; //  修正箇所
   end else if (ct < 25'd2699_9999)
      ct <= ct + 1'b1;
   else begin
      ct <= 25'd0;
      pc <= pc + 1'b1;
```

```
      reg_a <= (load_sel == 2'b00) ? alu_out : reg_a; //  修正箇所
      cflag <= nextcflag; //  修正箇所
   end
end

endmodule
```

メモ帳で rom1.txt と同じ場所に次の rom2.txt を作成してください。

```
00111010
00110001
10110001
10111010
00000101
00000010
00001101
01000010
00100000
00100000
00100000
00100000
00100000
00100000
00100000
00100000
```

[命令メモリ]

変更内容を見ていきます。

```
initial $readmemb("rom2.txt", rom); //  修正箇所
```

rom2.txt を読み込むように変更しています。

[レジスタ A と cflag_n]

```
// reg_a
reg [3:0] reg_a = 4'b0; //  修正箇所
assign leddata[16:13] = reg_a; //  修正箇所

// cflag_n
```

```
reg cflag = 1'b0;   // 修正箇所
assign leddata[4] = cflag;   // 修正箇所
```

4bit のフリップフロップ回路として reg_a を作成し、leddata[16:13](外部 led 下から 4 つ)に接続。1bit のフリップフロップ回路として cflag を作成し、leddata[4](内蔵 led 右から 5 番目)に接続しています。

[デコーダ]

```
// デコーダー
wire [1:0] alu_sel;   // 修正箇所
assign alu_sel = opcode[5:4];   // 修正箇所
wire [1:0] load_sel;   // 修正箇所
assign load_sel = opcode[7:6];   // 修正箇所
wire [3:0] im;   // 修正箇所
assign im = opcode[3:0];   // 修正箇所
```

この後の ALU は 4bit+4bit の演算を行い、ライトバックで結果を書き込みます。

ALU への入力の片方は即値 im です。もう片方は命令で指定したものです。結果を書き込む先も命令で指定したものです。命令の 5bit 目と 4bit 目 opcode[5:4]が ALU への入力の選択で配線 alu_sel に送ります。命令の 7bit 目と 6bit 目 opcode[7:6]が結果を書き込み先の選択で配線 load_sel に送ります。命令下位 4bit opcode[3:0]が即値 im なのでそのまま配線 im につなぎます。

[ALU]

```
// ALU
wire [3:0] alu_in;// 修正箇所
assign alu_in = (alu_sel == 2'b00) ? reg_a : (alu_sel == 2'b10) ? sw_in : 4'b0000; //
修正箇所

wire [3:0] alu_out;// 修正箇所
wire nextcflag;// 修正箇所
assign {nextcflag, alu_out} = alu_in + im; // 修正箇所
```

配線 alu_in は片方の ALU への入力です。分岐処理『条件 ? 満たした場合 : 満たさない場合』を使います。alu_sel が 00 つまり命令の 5bit4bit が 00 の場合、reg_a の内容を ALU に入力します。そうでない場合(01 と 10 と 11)、0000 を ALU に送ります。

4bit 配線 alu_out は ALU の加算結果です。1bit 配線 nextcflag は ALU の演算結果の桁上げです。assign {nextcflag, alu_out} = alu_in + im; で alu_in+im の加算演算を

行い、4bit+4bit=5bit で 4bit の結果と 1bit の桁上げになるので、結果を alu_out、桁上げを nextcflag に送ります。

[ライトバック]

```
reg_a <= 4'b0; //  修正箇所
cflag <= 1'b0; //  修正箇所
```

リセットボタンが押されたとき、reg_a を 0 に cflag を 0 に初期化します。

```
reg_a <= (load_sel == 2'b00) ? alu_out : reg_a; //  修正箇所
cflag <= nextcflag; //  修正箇所
```

1 秒ごとに実行結果を書き込みます。

load_sel が 00 つまり opcode[7:6]が 00 ならば ALU の演算結果を reg_a に書き込みます。そうでない場合は reg_a はそのままです。nextcflag の値つまり ALU の桁上げを cflag に書き込みます。

[実行内容]

番地	命令	処理内容
0	00111010	即値 1010 と 0 を加算し reg_a に代入(MOV A, 1010)
1	00110001	即値 0001 と 0 を加算し reg_a に代入(MOV A, 0001)
2	10110001	opcode[7:6]が 10 なので処理内容は書き込まれない
3	10111010	opcode[7:6]が 10 なので処理内容は書き込まれない
4	00000101	reg_a+即値 0101 を reg_a に代入(ADD A, 0101) reg_a は 1 番地の命令で 0001 になっているので 0001+0101=0110 が reg_a に書き込まれる
5	00000010	reg_a+即値 0010 を reg_a に代入(ADD A, 0010) reg_a は 4 番地の命令で 0110 になっているので 0110+0010=1000 が reg_a に書き込まれる
6	00001101	reg_a+即値 1101 を reg_a に代入(ADD A, 1101) reg_a は 5 番地の命令で 1000 になっているので 1000+1101=10101 がの下位 0101 が reg_a に書き込まれ、 cflag が 1 になる
7	01000010	opcode[7:6]が 01 なので処理内容は書き込まれない
8	00100000	即値 0000 と 0 を加算し reg_a に代入(MOV A, 0000)
9-15	00100000	上記と同じ

このように動作しているか確認してください。

＜CPU3 入力 sw_in の実装＞

『IN A』命令(00010000)を実装します。CPU2に次の修正を加えてください。変更箇所は 46 行目だけです。rom2.txt はそのまま使います。

```
module cpu (
        input wire clock,
        input wire reset_n,
        input wire[3:0] sw_n,
        output wire [4:0] led_n,
        output wire [11:0] led2
);

wire [16:0] leddata;
assign led_n[4:0] = ~leddata[4:0];
assign led2 = leddata[16:5];

wire [3:0] sw_in;
assign sw_in = ~sw_n;

reg [3:0] pc = 4'b0000;
assign leddata[3:0] = pc;

reg [24:0] ct = 25'd0;

reg [7:0] rom[0:15];
initial $readmemb("rom2.txt", rom);

wire [7:0] opcode;
assign opcode = rom[pc];
assign leddata[12:5] = opcode;

// reg_a
reg [3:0] reg_a = 4'b0;
assign leddata[16:13] = reg_a;

// cflag
reg cflag = 1'b0;
```

```
assign leddata[4] = cflag;

// デコーダー
wire [1:0] alu_sel;
assign alu_sel = opcode[5:4];
wire [1:0] load_sel;
assign load_sel = opcode[7:6];
wire [3:0] im;
assign im = opcode[3:0];

// ALU
wire [3:0] alu_in;
assign alu_in = (alu_sel == 2'b00) ? reg_a : (alu_sel == 2'b10) ? sw_in : 4'b0000; //
修正箇所

wire [3:0] alu_out;
wire nextcflag;
assign {nextcflag, alu_out} = alu_in + im;

always @(posedge clock or negedge reset_n) begin
   if (!reset_n) begin
      ct <= 25'd0;
      pc <= 4'b0;
      reg_a <= 4'b0;
      cflag <= 1'b0;
   end else if (ct < 25'd2699_9999)
      ct <= ct + 1'b1;
   else begin
      ct <= 25'd0;
      pc <= pc + 1'b1;
      reg_a <= (load_sel == 2'b00) ? alu_out : reg_a;
      cflag <= nextcflag;
   end
end

endmodule
```

[sw_in の実装]

```
assign alu_in = (alu_sel == 2'b00) ? reg_a : (alu_sel == 2'b10) ? sw_in : 4'b0000; //
修正箇所
```

alu_sel(opcode[5:4])が 00 の場合に reg_a を ALU に入力するのはそのままです。そうでない場合、alu_sel が 10 の場合、sw_in を ALU に入力します。sw_in は入力ポートでスイッチの内容が読み込まれます。そうでない場合(01 と 11)は 0000 をALUに入力します。

[実行内容]

番地	命令	処理内容
0-7		CPU2 と同じ
8	00100000	sw_in と 0 を加算し reg_a に代入(IN A) スイッチの内容が reg_a に入ります。
9-15	00100000	上記と同じ

命令デコーダを修正したので、CPU2 から 8 番地から 15 番地の実行内容が変わります。スイッチの内容が reg_a に書き込まれます。

このように動作しているか確認してください。

考察と感想

情報Ⅰ FPGA その 5

年　　組　　番氏名

＜CPU4 reg_b と出力 reg_out の実装＞

B レジスタと OUT レジスタを実装し、『MOV A, B』命令(00010000)、『MOV B,A』命令(01000000)、『ADD B, Im』命令(0101xxxx)、『IN B』命令(01100000)、『MOV B, Im』命令(0111xxxx)、(未定義命令『OUT A』命令(10000000))、『OUT B』命令(10010000)、『OUT Im』命令(1011xxxx)を実装します。ここからは外部 led 上から 8 個を命令内容の出力ではなく OUT レジスタと B レジスタの出力に変更します。

次のコードを入力してください。

```
module cpu (
        input wire clock,
        input wire reset_n,
        input wire[3:0] sw_n,
        output wire [4:0] led_n,
        output wire [11:0] led2
);

wire [16:0] leddata;
assign led_n[4:0] = ~leddata[4:0];
assign led2 = leddata[16:5];

wire [3:0] sw_in;
assign sw_in = ~sw_n;

reg [3:0] pc = 4'b0000;
assign leddata[3:0] = pc;

reg [24:0] ct = 25'd0;

reg [7:0] rom[0:15];
initial $readmemb("rom4.txt", rom); //  修正箇所

wire [7:0] opcode;
assign opcode = rom[pc];
// assign leddata[12:5] = opcode; //  修正箇所
```

```
// reg_a
reg [3:0] reg_a = 4'b0;
assign leddata[16:13] = reg_a;

// cflag
reg cflag = 1'b0;
assign leddata[4] = cflag;

// reg_b
reg [3:0] reg_b = 4'b0;    //  修正箇所
assign leddata[12:9] = reg_b;    //  修正箇所

// reg_out
reg [3:0] reg_out = 4'b0; //  修正箇所
assign leddata[8:5] = reg_out; //  修正箇所

//  デコーダー
wire [1:0] alu_sel;
assign alu_sel = opcode[5:4];
wire [1:0] load_sel;
assign load_sel = opcode[7:6];
wire [3:0] im;
assign im = opcode[3:0];

// ALU
wire [3:0] alu_in;
assign alu_in = (alu_sel == 2'b00) ? reg_a : //  修正箇所
                (alu_sel == 2'b01) ? reg_b : //  修正箇所
                (alu_sel == 2'b10) ? sw_in : //  修正箇所
                                     4'b0000; //  修正箇所

wire [3:0] alu_out;
wire nextcflag;
assign {nextcflag, alu_out} = alu_in + im;
```

```
always @(posedge clock or negedge reset_n) begin
   if (!reset_n) begin
      ct <= 25'd0;
      pc <= 4'b0;
      reg_a <= 4'b0;
      reg_b <= 4'b0;   // 修正箇所
      reg_out <= 4'b0;   // 修正箇所
      cflag <= 1'b0;
   end else if (ct < 25'd2699_9999)
      ct <= ct + 1'b1;
   else begin
      ct <= 25'd0;
      pc <= pc + 1'b1;
      reg_a <= (load_sel == 2'b00) ? alu_out : reg_a;
      reg_b <= (load_sel == 2'b01) ? alu_out : reg_b;   // 修正箇所
      reg_out <= (load_sel == 2'b10) ? alu_out : reg_out;   // 修正箇所
      cflag <= nextcflag;
   end
end

endmodule
```

rom1.txt や rom2.txt と同じ場所に rom4.txt を作成し、次の内容にします。

```
00111010
00110001
10110001
10110101
01111011
01110100
01010101
01010001
01000000
01110101
00010000
10010000
01100000
```

```
01100000
01100000
01100000
```

[rom と外部 led]

```
initial $readmemb("rom4.txt", rom); //  修正箇所
```

rom4.txt を読み込むように変更しています。

```
// assign leddata[12:5] = opcode; //  修正箇所
```

外部 led に命令を出力していた箇所をコメントアウトして無効にします。

[B レジスタと OUT レジスタ]

```
// reg_b
reg [3:0] reg_b = 4'b0;    //  修正箇所
assign leddata[12:9] = reg_b;    //  修正箇所

// reg_out
reg [3:0] reg_out = 4'b0; //  修正箇所
assign leddata[8:5] = reg_out; //  修正箇所
```

4bit のフリップフロップ回路 reg_b と 4bit のフリップフロップ回路 reg_out を作り、reg_b を外部 led の上から 5 番目～8 番目、reg_out を外部 led の上から1番目～4 番目に出力しています。

[ALU]

```
assign alu_in = (alu_sel == 2'b00) ? reg_a : //  修正箇所
                (alu_sel == 2'b01) ? reg_b : //  修正箇所
                (alu_sel == 2'b10) ? sw_in : //  修正箇所
                                     4'b0000; //  修正箇所
```

ALU の入力を、alu_sel が 00(opcode[5:4]が 00)の場合は reg_a、01 の場合は reg_b、10 の場合は sw_in、そうでない場合(11 の場合)は 0000 にしています。

[ライトバック]

```
    reg_b <= 4'b0;    //  修正箇所
    reg_out <= 4'b0;    //  修正箇所
```

リセットボタンを押したときは reg_b、reg_out を 0 に初期化します。

```
reg_b <= (load_sel == 2'b01) ? alu_out : reg_b;   // 修正箇所
reg_out <= (load_sel == 2'b10) ? alu_out : reg_out;   // 修正箇所
```

load_sel が 01、つまり opcode[7:6]が 01 の場合、ALU の結果を reg_b に書き込みます。load_sel が 10、つまり opcode[7:6]が 10 の場合、ALU の結果を reg_out に書き込みます。

ALU とライトバックの修正で B レジスタと OUT レジスタに対する命令が実装されます。

[実行内容]

番地	命令	内容
0	00111010	即値 1010 を reg_a に書き込み(MOV A, 1010)
1	00110001	即値 0001 を reg_a に書き込み(MOV A, 0001)
2	10110001	即値 0001 を reg_out に書き込み(OUT 0001)
3	10110101	即値 0101 を reg_out に書き込み(OUT 0101)
4	01111011	即値 1011 を reg_b に書き込み(MOV B, 1011)
5	01110100	即値 0100 を reg_b に書き込み(MOV B, 0100)
6	01010101	即値 0101+reg_b を reg_b に書き込む(ADD B, 0101)reg_b は 5 番地の命令により 0100 なので、 0100+0101=1001 を reg_b に書き込む。
7	01010001	即値 0001+reg_b を reg_b に書き込む(ADD B, 0011) reg_b は 6 番地の命令により 1001 なので、 1001+0001=1010 を reg_b に書き込む。
8	01000000	reg_a+0000 を reg_b に書き込み(MOV B, A)
9	01110101	即値 0101 を reg_b に書き込む(MOV B, 0101)
10	00010000	reg_b+0000 を reg_a に書き込む(MOV A, B)
11	10010000	reg_b+0000 を reg_out に書き込む(OUT B)
12	01100000	sw_in+0000 を reg_b に書き込む(IN B)
13-15	01100000	上記と同じ

だいぶややこしいですが、このように動作しているか reg_a、reg_b、reg_out に対応する外部 led で確認してください。

＜CPU5 分岐命令の実装＞

『JMP Im』無条件分岐命令と、直前の命令が桁上げしてないならば分岐する『JNC Im』条件分岐命令を実装します。これで TD4 の基本命令をすべて実装し、順次処理以外の分岐処理と反復処理が実行できるようになります。

次のコードを入力してください。

```
module cpu (
        input wire clock,
        input wire reset_n,
        input wire[3:0] sw_n,
        output wire [4:0] led_n,
        output wire [11:0] led2
);

wire [16:0] leddata;
assign led_n[4:0] = ~leddata[4:0];
assign led2 = leddata[16:5];

wire [3:0] sw_in;
assign sw_in = ~sw_n;

reg [3:0] pc = 4'b0000;
assign leddata[3:0] = pc;

reg [24:0] ct = 25'd0;

reg [7:0] rom[0:15];
initial $readmemb("rominout.txt", rom);    //  修正箇所
// initial $readmemb("romled.txt", rom);    //  修正箇所
// initial $readmemb("romramen.txt", rom);    //  修正箇所

wire [7:0] opcode;
assign opcode = rom[pc];
// assign leddata[12:5] = opcode;

// reg_a
reg [3:0] reg_a = 4'b0;
assign leddata[16:13] = reg_a;

// cflag
reg cflag = 1'b0;
assign leddata[4] = cflag;
```

```
// reg_b
reg [3:0] reg_b = 4'b0;
assign leddata[12:9] = reg_b;

// reg_out
reg [3:0] reg_out = 4'b0;
assign leddata[8:5] = reg_out;

//  デコーダー
wire [1:0] alu_sel;
assign alu_sel = (opcode[7:6] == 2'b11) ? 2'b11 : opcode[5:4]; //  修正箇所
wire [1:0] load_sel;
assign load_sel = opcode[7:6];
wire [3:0] im;
assign im = opcode[3:0];

// ALU
wire [3:0] alu_in;
assign alu_in = (alu_sel == 2'b00) ? reg_a :
                (alu_sel == 2'b01) ? reg_b :
                (alu_sel == 2'b10) ? sw_in :
                                     4'b0000;

wire [3:0] alu_out;
wire nextcflag;
assign {nextcflag, alu_out} = alu_in + im;

// BRANCH
wire [3:0] next_pc; //修正箇所
assign next_pc = (load_sel == 2'b11 && (opcode[4] == 1'b1 || cflag == 1'b0)) ?
alu_out : pc + 1'b1; //修正箇所

always @(posedge clock or negedge reset_n) begin
   if (!reset_n) begin
      ct <= 25'd0;
```

```
        pc <= 4'b0;
        reg_a <= 4'b0;
        reg_b <= 4'b0;
        reg_out <= 4'b0;
        cflag <= 1'b0;
    end else if (ct < 25'd2699_9999)
        ct <= ct + 1'b1;
    else begin
        ct <= 25'd0;
        pc <= next_pc; //  修正箇所
        reg_a <= (load_sel == 2'b00) ? alu_out : reg_a;
        reg_b <= (load_sel == 2'b01) ? alu_out : reg_b;
        reg_out <= (load_sel == 2'b10) ? alu_out : reg_out;
        cflag <= nextcflag;
    end
end

endmodule
```

rominout.txt と romled.txt と romramen.txt の 3 つを用意しました。それぞれ rom1.txt 等と同じ場所に作成してください。

[rominout.txt]

```
10111111
01100000
10010000
11110001
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```

```
00000000
00000000
```

[romled.txt]

```
10110011
10110110
10111100
10111000
10111000
10111100
10110110
10110011
10110001
11110000
00000000
00000000
00000000
00000000
00000000
00000000
```

[romramen.txt]

```
10110111
00000001
11100001
00000001
11100011
10110110
00000001
11100110
00000001
11101000
10110000
10110100
00000001
11101010
```

```
10111000
11111111
```

[rom]

```
initial $readmemb("rominout.txt", rom);    //  修正箇所
// initial $readmemb("romled.txt", rom);    //  修正箇所
// initial $readmemb("romramen.txt", rom);    //  修正箇所
```

とりあえず、rominout.txt を読み込むようにしています。後で romled.txt や romramen.txt を実行する場合はコメントアウトする行を変更してください(rominout.txt の行頭に//を入れて、使う行の行頭の//を削除する)。

[命令デコーダ]

```
assign alu_sel = (opcode[7:6] == 2'b11) ? 2'b11 : opcode[5:4]; //  修正箇所
```

分岐命令(opcode[7:6]が 11)の場合、ALU への入力を 0000 に固定します。代わりに opcode[5:4]を条件分岐か無条件分岐かの指定に使います。

[次の pc の設定]

```
// BRANCH
wire [3:0] next_pc; //修正箇所
assign next_pc = (load_sel == 2'b11 && (opcode[4] == 1'b1 || cflag == 1'b0)) ?
alu_out : pc + 1'b1; //修正箇所
```

分岐命令の場合 ALU の出力は即値 im+0000 なので即値 IM が出力されます。

4bit の配線 next_pc に対して、無条件分岐命令『JMP』(opcode[7:4]=1111)か条件分岐命令『JNC』(opcode[7:4]=1110)で直前の命令で桁上げが発生していない場合(cflag が 0)、分岐して次の pc を ALU の出力にします。分岐命令でない場合や条件分岐命令『JNC』だけど直前の命令で桁上げが発生している場合、通常通り pc を+1 します。

(load_sel == 2'b11 && (opcode[4] == 1'b1 || cflag == 1'b0))の load_sel == 2'b11 は分岐命令 opcode[7:6]=11 の場合真になります。(opcode[4] == 1'b1 || cflag == 1'b0) は、opcode[4]が 1、つまり無条件分岐命令であるか、opcode[4]が 0、つまり条件分岐命令だけど cflag が 0 である場合に真になります。これらを&&で並べることで、分岐命令が真でかつ opcode[1]が1または cflag が0の場合、真になります。このようにして無条件分岐命令の場合と条件分岐命令で桁上げが発生していない場合に分岐するようにしています。

[実行結果]

[rominout.txt]

番地	命令	内容
0	10111111	即値 1111+0000 を reg_out に出力(OUT 1111)
1	01100000	sw_in+0000 を reg_b に書き込む(IN B)
2	10010000	reg_b+0000 を reg_out に出力(OUT B)
3	11110001	1 番地に無条件分岐 (JMP 0001) 常に分岐するので 4 番地以降を実行することはない
4-15	00000000	reg_a+0000 を reg_a に書き込む(ADD A, 0000) この命令は実行されない

最初に外部 led の上から 1 番目～4 番目に 1111 を表示して、以降はスイッチの内容を reg_b に書き込み、それを外部 led の上から 1 番目～4 番目に表示し、これを繰り返します。内蔵 led のアドレスが 0001、0010、0011 を繰り返しているのを確認してください。

[romled.txt]

番地	命令	内容
0	10110011	即値 0011+0000 を reg_out に出力(OUT 0011)
1	10110110	即値 0110+0000 を reg_out に出力(OUT 0110)
2	10111100	即値 1100+0000 を reg_out に出力(OUT 1100)
3	10111000	即値 1000+0000 を reg_out に出力(OUT 1000)
4	10111000	即値 1000+0000 を reg_out に出力(OUT 1000)
5	10111100	即値 1100+0000 を reg_out に出力(OUT 1100)
6	10110110	即値 0110+0000 を reg_out に出力(OUT 0110)
7	10110011	即値 0011+0000 を reg_out に出力(OUT 0011)
8	10110001	即値 0001+0000 を reg_out に出力(OUT 0001)
9	11110000	0 番地に無条件分岐 (JMP 0000) 常に分岐するので 10 番地以降を実行することはない
10-15	00000000	この命令は実行されない

外部 led 上から 1 番目～4 番目が表示されます。命令通り動いているか確認してください。

[romramen.txt]

番地	命令	内容
0	10110111	即値 0111 を reg_out に出力(OUT 0111)
1	00000001	reg_a+0001 を reg_a に書き込む(ADD A, 0001)

2	11100001	1 番地の命令で桁上げが発生していない (cflag が 0)ならば 1 番地に分岐、そうでないならば pc+1 をして 3 番地へ 1 番地の命令を 15 回繰り返すことになります
3	00000001	reg_a+0001 を reg_a に書き込む(ADD A, 0001)
4	11100011	桁上げが発生していないならば 3 番地に分岐、そうでないならば 5 番地へ
5	10110110	即値 0110 を reg_out に出力(OUT 0110)
6	00000001	reg_a+0001 を reg_a に書き込む(ADD A, 0001)
7	11100110	桁上げが発生していないならば 6 番地に分岐、そうでないならば 7 番地へ
8	00000001	reg_a+0001 を reg_a に書き込む(ADD A, 0001)
9	11101000	桁上げが発生していないならば 8 番地に分岐、そうでないならば 10 番地へ
10	10110000	即値 0000 を reg_out に出力(OUT 0000)
11	10110100	即値 0100 を reg_out に出力(OUT 0100)
12	00000001	reg_a+0001 を reg_a に書き込む(ADD A, 0001)
13	11101010	桁上げが発生していないならば 12 番地に分岐、そうでないならば 14 番地へ
14	10111000	即値 1000 を reg_out に出力(OUT 1000)
15	11111111	15 番地へ分岐(無限ループ)

カップラーメンタイマーです。途中色々 led が表示されますが、最終的に 3 分間経過すると上から 4 番目の led のみが点灯します。どうしてそう動作するかは考えてみてください。

考察と感想
