最終報告　レポート D

1029 32 4054　竹田原俊介　6 月 6 日

設計を担当した、コンポーネント：

program_counter,control,sign_extension,sign_ext_im,alu,SEG_SEL,szcv_register,simple

全体の設計図は下の図 0 です。

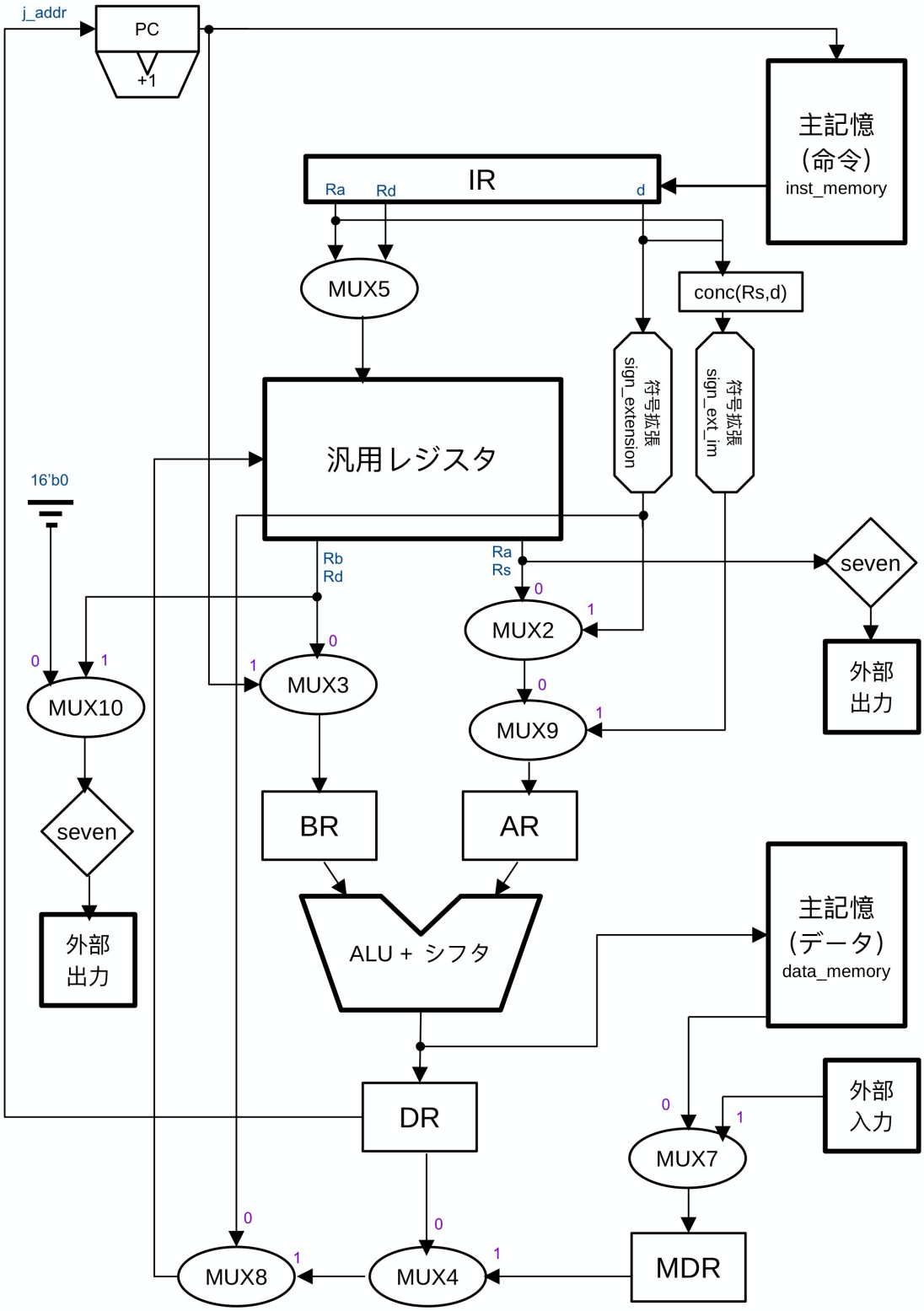

j_addr
PC
+1
主記憶
(命令)
inst_memory
IR
Ra
Rd
d
MUX5
conc(Rs,d)
符号拡張
sign_extension
符号拡張
sign_ext_im
汎用レジスタ
16'b0
Rb
Rd
Ra
Rs
seven
MUX2
外部
出力
MUX10
MUX3
MUX9
seven
BR
AR
外部
出力
ALU + シフタ
主記憶
(データ)
data_memory
外部
入力
DR
MUX7
MUX8
MUX4
MDR

図 0

❶program_counter

```
1   module program_counter(clock, rst,j_flag,j_addr,phase,  pc_out);
2      input       clock;
3      input       rst;
4      input j_flag;
5      input[15:0]j_addr;
6      input [2:0]phase;
7
8
9      output   [15:0] pc_out;
10
11     reg   [15:0] pc_out;
12     reg [2:0] counter;
13
14     always @(posedge clock or negedge rst  ) begin
15        if(rst==0)begin
16           pc_out<=16'b0000000000000000;
17
18        end
19        else begin
20           if(phase==3'b100)begin  //100 wo 011
21              if(j_flag==1)begin
22                 pc_out<=j_addr+16'b0000000000000001; //j_addr ha PC+ext_d nanode +1 gahituyou
23              end
24              else begin
25              pc_out   <= pc_out+16'b0000000000000001;//pc_inwo16'b1nikaeta
26              end
27           end
28        end
29     end
```

図 1

図 1 は、program_counter のコードです。

まず、プログラムカウンタの外部仕様は、入力が clock,rst,j_flag,j_addr,phase,pc_out です。clock,rst,j_flag が 1 ビットで、j_addr,pc_out は 1 6 ビットです。

プログラムカウンタを出力する役割を担っています。

内部仕様としては、clock は、クロックの役割をしており、rst は押されると、プログラムカウンタを 0 にするというリセットの役割です。j_flag は、無条件分岐命令や条件分岐命令が起こる際に、出力するプログラムカウンタの値を j_addr に 1 を加えたものになるようにする役割です。元のプログラムカウンタに 1 を加えたり、j_addr に 1 を加えてそれをプログラムカウンタとして代入する作業は、phase が 4 の時に起こるように設定しています。pc _out は命令フェッチ用の主記憶 inst_memory と mux3 に入力されていきます。

次に性能評価ですが、LUT 数は 3 3 で、clock to output times は 7.771 です。最大動作周波数は、314.66MHz で、クリティカルパスは、pc_out[4]から pc_out[15]です。

❷sign_extention

```
1
2   module sign_extension(d, result);
3       input [7:0] d;
4       output  [15:0] result;
5
6
7
8       assign result = { {8{d[7]}}, d };
9
10
11  endmodule
```

図 2

図 2 は sign_extention のコードです。

この sign_extention の外部仕様は、入力が d,出力が result です。

d が 8 ビットで、result が 1 6 ビットです。 8 ビットの d を拡張する役割を担っています。

ロード命令、即値ロード命令、無条件分岐命令、条件分岐命令の際に、使用されます。

内部仕様としては、result は、d の最上位ビットを 8 つ並べたものと、d を結合するようにしています。この result は、mux2 と mux8 に入力されていきます。

❸control

```
module control(phase,
            s,z,c,v,
            instruction,
            aluc_e,
            ar_e,br_e,dr_e,mdr_e,ir_e, // enablers
                reg_e, // signal for all non-general registers --> 同期
            genr_w,
            //pc_e,
            mem_e, mem_w,
            jump,m2_s,m3_s,m4_s,m5_s, m6_s, m7_s, m8_s,m9_s,out_s,hlt,szcv_s,
                alu_instruction);
    input [2:0] phase;
    input s, z, c, v;
    input [15:0] instruction;
    output reg  aluc_e,
            ar_e,br_e,dr_e,mdr_e,ir_e,
                reg_e,
            genr_w,
            mem_e, mem_w,
            jump,m2_s,m3_s,m4_s, m5_s, m6_s, m7_s, m8_s,m9_s,out_s,szcv_s;
     output reg hlt;
    output [5:0] alu_instruction; // ALU制御部へ

     wire [1:0] op = instruction[15:14];
     wire [2:0] r1 = instruction[13:11];
     wire [2:0] r2 = instruction[10:8];
     wire [3:0] alu_op = instruction[7:4];
     reg [4:0] command;

     // set the value of alu_instruction depending on the type of instruction
     // if(op==2'b11) alu_instruction <= { instruction[15:14], instruction[7:4] };
     // else alu_instruction <=  {instruction[15:10]};
     assign alu_instruction = (op==2'b11) ? { instruction[15:14], instruction[7:4] } : {instruction[15:10]};
```

図 3

```
always @(*) begin
    // set the value of "command" depending on the instruction
    case(op)
        2'b11: command <= {1'b0,alu_op}; // ALU
        2'b00: command <= 5'b10000; //LD  r[Ra]=*(r[Rb]+sign_ext(d))
        2'b01: command <= 5'b10001; //ST  *(r[Rb]+sign_ext(d))=r[Ra]
        2'b10: begin
            case(r1)
                3'b000: command <= 5'b10010; //LI r[Rb]=sign_ext(d)
                3'b100: command <= 5'b10011; //B PC=PC+1+sign_ext(d)
                3'b111: begin
                    case(r2)
                        3'b000: begin
                              case(Z)
                                 1'b1:command<=5'b10100; //BE PC=PC+1+sign_ext(d)
                                 1'b0:command<=5'b11000;  //11000 wo 10100
                                 default:command<=5'b11000;
                              endcase
                        end
                        3'b001: begin
                              case(S^V)
                                 1'b1:command<=5'b10101;  //BLT PC=PC+1+sign_ext(d)
                                 1'b0:command<=5'b11000;  //11000 wo 10101
                                 default:command<=5'b11000;
                              endcase
                        end
                        3'b010: begin
                              case(Z||(S^V))
                                 1'b1:command<=5'b10110;  //BLE PC=PC+1+sign_ext(d)
                                 1'b0:command<=5'b11000;  //11000 wo 10110
                                 default:command<=5'b11000;
                              endcase
                        end
                        3'b011: begin
                              case(Z)
                                 1'b1:command<=5'b11000;   //11000 wo 10111
                                 1'b0:command<=5'b10111;  //BNE PC=PC+1+sing_ext(d)
                                 default:command<=5'b11000;
                              endcase
                        end
                        default:command<=5'b11000;
                    endcase
                end
                default:command<=5'b11000;

            endcase
        end
        default:command<=5'b11000;
    endcase
```

図 4

```
// alu control unit signal
if(phase==3'b000 || command==5'b01100 || command==5'b01101 || command==5'b01111
|| command== 5'b10010) begin
    aluc_e <= 0;
end else begin
    aluc_e <= 1;
end

// ar signal
if(command== 5'b00000 || command==5'b00001 || command==5'b00010 || command==5'b00011
|| command==5'b00100 || command==5'b00101 || command==5'b01101 || command==5'b10000
|| command==5'b10001 || command==5'b10011 || command==5'b10100 || command==5'b10101
|| command==5'b10110 || command==5'b10111||command==5'b00110) begin
    ar_e <= 1;
end else begin
    ar_e <= 0;
end

// br signal
if(phase==3'b000 || command==5'b01100 || command==5'b01101 || command==5'b01111
|| command==5'b10010) begin
    br_e <= 0;
end else begin
    br_e <= 1;
end

// dr signal
if(phase==3'b000 || command==5'b00101 ||  command==5'b01100 || command==5'b01101
|| command==5'b01111 || command==5'b10010) begin
    dr_e <= 0;
end else begin
    dr_e <= 1;
end

// mdr signal
if(command==5'b01100 || command==5'b10000) begin
    mdr_e <= 1;
end else begin
    mdr_e <= 0;
end

// ir signal
if(phase==3'b000 || command==5'b01111) begin
    ir_e <= 0;
end else begin
    ir_e <= 1;
end

// the clock for all the registers
if(phase==3'b000 || command==5'b01111) begin
    reg_e <= 0;
end else begin
    reg_e <= 1;
end

// memory read
if(phase==3'b000 || command==5'b00101 || command==5'b00110 || command==5'b01111) begin
```

図 5

```
        // memory read
        if(phase==3'b000 || command==5'b00101 || command==5'b00110 || command==5'b01111) begin
            mem_e <= 0;
        end else begin
            mem_e <= 1;
        end

        // jump signal
        if(command==5'b10011 || command==5'b10100 || command==5'b10101 || command==5'b10110 ||
        command==5'b10111)begin
            jump <= 1;
        end else begin
            jump <= 0;
        end

        // mux2 selector
        if(command==5'b01000 || command==5'b01001 || command==5'b01010 || command==5'b01011
        || command==5'b10000 || command==5'b10001 || command==5'b10011 || command==5'b10100
        || command==5'b10101 || command==5'b10110 || command==5'b10111) begin
            m2_s <= 1;
        end else begin
            m2_s <= 0;
        end

        // mux3 selector
        if(command==5'b10011 || command==5'b10100 || command==5'b10101 || command==5'b10110
        || command==5'b10111) begin
            m3_s <= 1;
        end else begin
            m3_s <= 0;
        end

        // mux4 selector
        if(command==5'b01100 || command==5'b10000) begin
            m4_s <= 1;
        end else begin
            m4_s <= 0;
        end

        // mux5 selector
        if(phase==3'b000 || command==5'b00101 || command==5'b01101 || command==5'b01111
        || command==5'b10000 || command==5'b10001 || command==5'b10011 || command==5'b10100
        || command==5'b10101 || command==5'b10110 || command==5'b10111) begin
            m5_s <= 0;
        end else begin
            m5_s <= 1;
        end

        // mux6 selector
        if(command==5'b10001) begin
            m6_s <= 1;
        end else begin
            m6_s <= 0;
```

図 6

```
191          // mux6 selector
192          if(command==5'b10001) begin
193              m6_s <= 1;
194          end else begin
195              m6_s <= 0;
196          end
197
198          // mux7 selector
199          if(command==5'b01100) begin
200              m7_s <= 1;
201          end else begin
202              m7_s <= 0;
203          end
204
205          // mux8 selector
206          if(command==5'b10010) begin
207              m8_s <= 1;s
208          end else begin
209              m8_s <= 0;
210          end
211
212          // mux9 selector
213          if((instruction[3]==1'b1)&&(command==5'b00000||command==5'b00001||command==5'b00010||
214          command==5'b00011||command==5'b00100||command==5'b00101||command==5'b00110))begin
215              m9_s<=1;
216          end else begin
217              m9_s<=0;
218          end
219
220          // output signal (for 7SEG LED)
221          if(command==5'b01101) begin // OUT命令
222              out_s <= 1;
223          end else begin
224              out_s <= 0;
225          end
226
227          // halt flag
228          if(command==5'b01111) begin //HALT命令
229              hlt <= 1;
230          end else begin
231              hlt <= 0;
232          end
233
234          // 汎用レジスタに書き込む
235        if(phase==3'b101 && (command==5'b00000 || command==5'b00001 || command==5'b00010
236          || command==5'b00011 || command==5'b00100 || command==5'b01000 || command==5'b01001
237          || command==5'b01010 || command==5'b01011 || command==5'b01100 || command==5'b10000
238          || command==5'b10010||command==5'b00110))begin
239           genr_w<=1;
240        end else begin
241           genr_w<=0;
242        end
243
244          // メモリに書き込む
245        if(phase==3'b101 && command==5'b10001)begin
246           mem_w<=1;
247        end else begin
```

図 7

```
243
244          // メモリに書き込む
245        if(phase==3'b101 && command==5'b10001)begin
246           mem_w<=1;
247        end else begin
248           mem_w<=0;
249        end
250
251        if(phase==3'b101&&(command==5'b00000 || command==5'b00001 || command==5'b00010
252          || command==5'b00011 || command==5'b00100 || command==5'b00101 || command==5'b00110
253          || command==5'b01000 || command==5'b01001 || command==5'b01010 || command==5'b01011
254          ))begin
255           szcv_s<=1'b1;
256        end else begin
257           szcv_s<=1'b0;
258        end
259      end // always の end
260
261  endmodule
```

図 8

図３から図８は、control のコードです。
まず、control の外部仕様は、入力が phase,S,Z,C,V,instruction で、出力が aluc_e,ar_e,br_e,dr_e,mdr_e,ir_e,genr_w,mem_e,mem_w,jump,m2_s,m3_s,m4_s,m5_s,m7_s,m8_s,m9_s,out_s,hlt,szcv_s,alu_instruction です。
phase が４ビット、instruction が１６ビットで、alu_instruction が６ビットです。それ以外の信号は１ビットです。
内部仕様としては、まず、それぞれの命令を command の数値に割り当てていきます。
そして、その command の値、phase の値を条件として、それぞれの信号を決めていきます。
まず、aluc_e という alu_control_unit を行うかどうかの信号は、フェーズが０００つまり、初期状態の時と、input 命令と、output 命令と、hlt 命令と ＬＩ 命令の時は、alu_control_unit は動かさないようにしています。
ar_e に読み込むかどうかという信号は、
ADD,SUB,AND,OR,XOR,CMP,MOV,OUT,LD,ST,B,BE,BLT,BLE の命令の時に、１を出力するようにします。
br _e という br レジスタに読み込むかどうかという信号は、phase が 0 の時、IN,OUT,LI の時は、動かないようにし、それ以外では、動くようにしています。
dr _e という dr レジスタに読み込むかどうかという信号は、phase が 0 の時、CMP,IN,HLT,LI の時、動かないようにし、それ以外では、動くようにしています。
mdr _e という mdr レジスタに読み込むかどうかという信号は、IN,LD の時のみ動くようにしています。
ir _e という ir レジスタに読み込むかどうかという信号は、phase が 0 の時、HLT の時は、動かないようにしています。
jump というプログラムカウンタに、jump_addr を採用するかどうかを判別させるための信号は、B,BE,BLT,BLE,BNE という分岐命令の時に 1 を出力するように設定しています。
mux_2 については、シフト演算、ST,分岐命令の際に、d を拡張した値を、それ以外の時は、汎用レジスタの値を出力するように信号を与えています。
mux_3 については、分岐命令の時に、PC の値を、それ以外の時は、汎用レジスタの値を出力するように信号を与えています。
mux_4 については、IN,LD の時のみ、mdr の値を出力し、それ以外の時は、dr の値を出力するように信号を与えています。
mux_5 については、phase が 0 の時、CMP,OUT,HLT,LD,ST,条件分岐の時、汎用レジスタへの書き込みアドレスとして、Rs もしくは Ra を出力し、それ以外の時は、書き込みアドレスとして,Rd もしくは Rb を出力するように、信号を与えています。
mux_7 については、ＩＮ 命令のときは、外部入力を、それ以外の時は、主記憶のデータを出力するように、信号を与えています。

mux_8 については、LI 命令の時は、拡張した d の値を、それ以外の場合は、mux4 の値を出力するように信号を与えています。この mux8 の値は、汎用レジスタの書き込みデータとして入力されていきます。
mux_9 については、もし、シフト計算以外の演算命令の時で、d フィールドの最上位ビットが 1 の時は、レジスタの内容の代わりに、Rs と d フィールドの残り 3 ビットを結合したものを ar に出力するための信号を与えています。これは、拡張機能としての即値オペランドの強化のために追加されたマルチプレクサです。
out_s については、OUT 命令の際に、seven 関数に入った数字を LED に光らせるという信号です。OUT 命令以外の時は、LED を全く光らせないという役割があります。
hlt については、HLT 命令の時は、hlt 信号を出し、CPU を停止させるという役割をします。
genr_w については、phase が 5 の時かつ、演算命令、LD,LI の際に、汎用レジスタに書き込む許可を与える信号です。
mem_w については、phase が 5 の時かつ、ST の時、主記憶（data_memory）に書き込む許可を与える信号です。
szcv_s については、演算命令で、phase が 5 の時、信号を出すもので、この信号が出されると、szcv のフラグが更新されます。

性能評価としては、LUT 数は、５２で、RR における遅延時間が 18.145 です。
クリティカルパスは、Z から mdr_e です。

❹sign_ext_im
下図 9 が sign_ext_im のコードです。
これは Rs と d フィールドの残り 3 ビットを結合したものを１６ビットに拡張し、m 9 に出力するために作られた関数です。
この sign_ext_im の外部仕様は、入力は、d,出力は、result で、d が１１ビット、result は、１６ビットです。
内部仕様としては、d の最上位ビットを１０個並べて、その後ろに d[10:8]を並べ、その後に、d[2:0]を並べたものを result に代入するというものです。この d には、Rs と 8 ビットの d フィールドを並べたものが入力されていきます。

```
1   module sign_ext_im(d,result);
2      input[10:0]d;
3      output[15:0]result;
4
5      assign result={{10{d[10]}},{d[10:8]},{d[2:0]}};
6   endmodule
```

図 9

❺alu

```
module alu( opcode,d, alu_in_a, alu_in_b, alu_out, S,Z,C,V);

    input [3:0] opcode;
    input [15:0] alu_in_a;
    input  [15:0] alu_in_b;
    input [3:0] d;
    output [15:0] alu_out;
    output S;
    output Z;

    output C;
    output V;

    wire [16:0] SUM;
    wire [16:0] SUB;
    wire [15:0] AND;
    wire [15:0] OR;
    wire [15:0] XOR;
    wire c;
    assign c=alu_in_b[15];

    function [15:0]SLR;
       input [15:0] alu_in_b;  //a wo b ni kaeta

       input [3:0] D; // d 4 bit takes values 0 to 16
       begin
          case(D)
             4'b0000:SLR=alu_in_b[15:0];    // a wo b ni kaeta
             4'b0001:SLR={alu_in_b[14:0], alu_in_b[15:15]};
             4'b0010:SLR={alu_in_b[13:0], alu_in_b[15:14]};
             4'b0011:SLR={alu_in_b[12:0], alu_in_b[15:13]};
             4'b0100:SLR={alu_in_b[11:0], alu_in_b[15:12]};
             4'b0101:SLR={alu_in_b[10:0], alu_in_b[15:11]};
             4'b0110:SLR={alu_in_b[9:0], alu_in_b[15:10]};
             4'b0111:SLR={alu_in_b[8:0], alu_in_b[15:9]};
             4'b1000:SLR={alu_in_b[7:0], alu_in_b[15:8]};
             4'b1001:SLR={alu_in_b[6:0], alu_in_b[15:7]};
             4'b1010:SLR={alu_in_b[5:0], alu_in_b[15:6]};
             4'b1011:SLR={alu_in_b[4:0], alu_in_b[15:5]};
             4'b1100:SLR={alu_in_b[3:0], alu_in_b[15:4]};
             4'b1101:SLR={alu_in_b[2:0], alu_in_b[15:3]};
             4'b1110:SLR={alu_in_b[1:0], alu_in_b[15:2]};
             4'b1111:SLR={alu_in_b[0:0], alu_in_b[15:1]};
          endcase
       end
    endfunction


    function [15:0]SRA;
       input [15:0] alu_in_b;
       input [3:0] D;
       input c;
       begin
          case(D)
             4'b0000:SRA=alu_in_b[15:0];
             4'b0001:SRA={c,alu_in_b[15:1]};
             4'b0010:SRA={c,c,alu_in_b[15:2]};
```

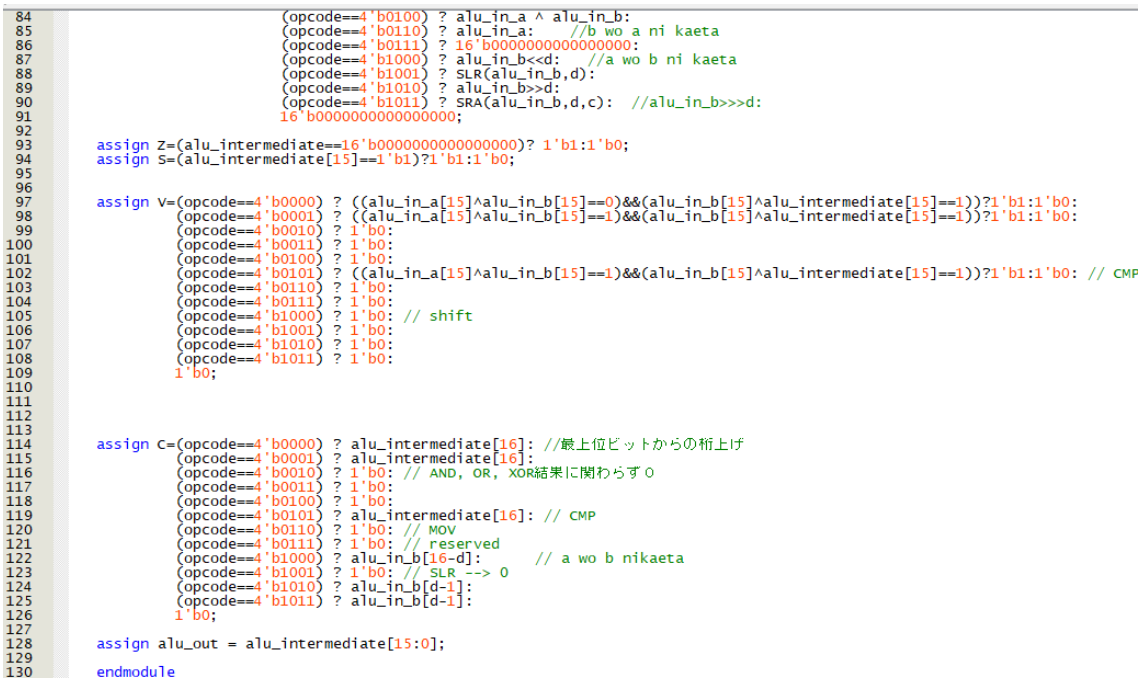
図１０

```
49      function [15:0]SRA;
50          input [15:0] alu_in_b;
51          input [3:0] D;
52          input c;
53          begin
54              case(D)
55                  4'b0000:SRA=alu_in_b[15:0];
56                  4'b0001:SRA={c,alu_in_b[15:1]};
57                  4'b0010:SRA={c,c,alu_in_b[15:2]};
58                  4'b0011:SRA={c,c,c,alu_in_b[15:3]};
59                  4'b0100:SRA={c,c,c,c,alu_in_b[15:4]};
60                  4'b0101:SRA={c,c,c,c,c,alu_in_b[15:5]};
61                  4'b0110:SRA={c,c,c,c,c,c,alu_in_b[15:6]};
62                  4'b0111:SRA={c,c,c,c,c,c,c,alu_in_b[15:7]};
63                  4'b1000:SRA={c,c,c,c,c,c,c,c,alu_in_b[15:8]};
64                  4'b1001:SRA={c,c,c,c,c,c,c,c,c,alu_in_b[15:9]};
65                  4'b1010:SRA={c,c,c,c,c,c,c,c,c,c,alu_in_b[15:10]};
66                  4'b1011:SRA={c,c,c,c,c,c,c,c,c,c,c,alu_in_b[15:11]};
67                  4'b1100:SRA={c,c,c,c,c,c,c,c,c,c,c,c,alu_in_b[15:12]};
68                  4'b1101:SRA={c,c,c,c,c,c,c,c,c,c,c,c,c,alu_in_b[15:13]};
69                  4'b1110:SRA={c,c,c,c,c,c,c,c,c,c,c,c,c,c,alu_in_b[15:14]};
70                  4'b1111:SRA={c,c,c,c,c,c,c,c,c,c,c,c,c,c,c,alu_in_b[15]};
71              endcase
72          end
73      endfunction
```

図１１

```
75      wire shift;
76      wire [16:0] alu_intermediate;
77
78      assign ADD={1'b0, alu_in_a }+{1'b0, alu_in_b};
79      assign shift=8*d[3]+4*d[2]+2*d[1]+d[0];
80      assign alu_intermediate=(opcode==4'b0000) ? alu_in_a +alu_in_b:
81                              (opcode==4'b0001) ? alu_in_b-alu_in_a:
82                              (opcode==4'b0010) ? alu_in_a & alu_in_b:
83                              (opcode==4'b0011) ? alu_in_a | alu_in_b:
84                              (opcode==4'b0100) ? alu_in_a ^ alu_in_b:
85                              (opcode==4'b0110) ? alu_in_a:    //b wo a ni kaeta
86                              (opcode==4'b0111) ? 16'b0000000000000000:
87                              (opcode==4'b1000) ? alu_in_b<<d:   //a wo b ni kaeta
88                              (opcode==4'b1001) ? SLR(alu_in_b,d):
89                              (opcode==4'b1010) ? alu_in_b>>d:
90                              (opcode==4'b1011) ? SRA(alu_in_b,d,c):  //alu_in_b>>>d:
91                              16'b0000000000000000;
92
93      assign Z=(alu_intermediate==16'b0000000000000000)? 1'b1:1'b0;
94      assign S=(alu_intermediate[15]==1'b1)?1'b1:1'b0;
95
96
97      assign V=(opcode==4'b0000) ? ((alu_in_a[15]^alu_in_b[15]==0)&&(alu_in_b[15]^alu_intermediate[15]==1))?1'b1:1'b0:
98               (opcode==4'b0001) ? ((alu_in_a[15]^alu_in_b[15]==1)&&(alu_in_b[15]^alu_intermediate[15]==1))?1'b1:1'b0:
99               (opcode==4'b0010) ? 1'b0:
100              (opcode==4'b0011) ? 1'b0:
101              (opcode==4'b0100) ? 1'b0:
102              (opcode==4'b0101) ? ((alu_in_a[15]^alu_in_b[15]==1)&&(alu_in_b[15]^alu_intermediate[15]==1))?1'b1:1'b0: // CMP
103              (opcode==4'b0110) ? 1'b0:
104              (opcode==4'b0111) ? 1'b0:
105              (opcode==4'b1000) ? 1'b0: // shift
106              (opcode==4'b1001) ? 1'b0:
107              (opcode==4'b1010) ? 1'b0:
108              (opcode==4'b1011) ? 1'b0:
109              1'b0;
110
111
112
113
114     assign C=(opcode==4'b0000) ? alu_intermediate[16]: //最上位ビットからの桁上げ
115              (opcode==4'b0001) ? alu_intermediate[16]:
116              (opcode==4'b0010) ? 1'b0: // AND, OR, XOR結果に関わらず０
117              (opcode==4'b0011) ? 1'b0:
118              (opcode==4'b0100) ? 1'b0:
119              (opcode==4'b0101) ? alu_intermediate[16]: // CMP
120              (opcode==4'b0110) ? 1'b0: // MOV
121              (opcode==4'b0111) ? 1'b0: // reserved
122              (opcode==4'b1000) ? alu_in_b[16-d]:      // a wo b nikaeta
123              (opcode==4'b1001) ? 1'b0: // SLR --> 0
124              (opcode==4'b1010) ? alu_in_b[d-1]:
125              (opcode==4'b1011) ? alu_in_b[d-1]:
126              1'b0:
```

図１２



```
                        (opcode==4'b0100) ? alu_in_a ^ alu_in_b:
                        (opcode==4'b0110) ? alu_in_a:    //b wo a ni kaeta
                        (opcode==4'b0111) ? 16'b0000000000000000:
                        (opcode==4'b1000) ? alu_in_b<<d:   //a wo b ni kaeta
                        (opcode==4'b1001) ? SLR(alu_in_b,d):
                        (opcode==4'b1010) ? alu_in_b>>d:
                        (opcode==4'b1011) ? SRA(alu_in_b,d,c):  //alu_in_b>>>d:
                        16'b0000000000000000;

    assign Z=(alu_intermediate==16'b0000000000000000)? 1'b1:1'b0;
    assign S=(alu_intermediate[15]==1'b1)?1'b1:1'b0;


    assign V=(opcode==4'b0000) ? ((alu_in_a[15]^alu_in_b[15]==0)&&(alu_in_b[15]^alu_intermediate[15]==1))?1'b1:1'b0:
             (opcode==4'b0001) ? ((alu_in_a[15]^alu_in_b[15]==1)&&(alu_in_b[15]^alu_intermediate[15]==1))?1'b1:1'b0:
             (opcode==4'b0010) ? 1'b0:
             (opcode==4'b0011) ? 1'b0:
             (opcode==4'b0100) ? 1'b0:
             (opcode==4'b0101) ? ((alu_in_a[15]^alu_in_b[15]==1)&&(alu_in_b[15]^alu_intermediate[15]==1))?1'b1:1'b0: // CMP
             (opcode==4'b0110) ? 1'b0:
             (opcode==4'b0111) ? 1'b0:
             (opcode==4'b1000) ? 1'b0: // shift
             (opcode==4'b1001) ? 1'b0:
             (opcode==4'b1010) ? 1'b0:
             (opcode==4'b1011) ? 1'b0:
             1'b0;



    assign C=(opcode==4'b0000) ? alu_intermediate[16]: //最上位ビットからの桁上げ
             (opcode==4'b0001) ? alu_intermediate[16]:
             (opcode==4'b0010) ? 1'b0: // AND, OR, XOR結果に関わらず０
             (opcode==4'b0011) ? 1'b0:
             (opcode==4'b0100) ? 1'b0:
             (opcode==4'b0101) ? alu_intermediate[16]: // CMP
             (opcode==4'b0110) ? 1'b0: // MOV
             (opcode==4'b0111) ? 1'b0: // reserved
             (opcode==4'b1000) ? alu_in_b[16-d]:      // a wo b nikaeta
             (opcode==4'b1001) ? 1'b0: // SLR --> 0
             (opcode==4'b1010) ? alu_in_b[d-1]:
             (opcode==4'b1011) ? alu_in_b[d-1]:
             1'b0;

    assign alu_out = alu_intermediate[15:0];

    endmodule
```

図１３

図１０から図１３までが alu のコードです。
前回の中間報告から変更した部分は、図１１と図１２です。
まず、図１１では、ＳＲＡのための関数を定義しています。
c には、alu_in_b の最上位ビットが入ります。そして、シフトの数と c の数が同じになるように、ＳＲＡに代入しています。
次に、図１２ですが、変更しているのは、Ｖの部分です。
オーバーフローが起こるのは、加算と減算の時だけなので、それ以外の部分は、全て０を代入しています。
まず、加算ですが、オーバーフローが起こるのは、正の数と正の数を足した時か、負の数と負の数を足した時のみなので、alu_in_a と alu_in_b の排他的論理和が０になるという条件が必要です。さらに、正の数同士を足して、負の数が出てきた場合と、負の数同士を足して正の数が出た時に、オーバーフローが起こるので、alu_in_b の最上位ビットと加算の結果の最上位ビットの排他的論理和が１という条件が必要です。よって、このようなコードになります。
次に減算ですが、オーバーフローが起こるのは、正の数から負の数を引いた時か、負の数から正の数を引いた時であるので、alu_in_a と alu_in_b の排他的論理和が１になるという

条件が必要です。さらに、正の数から負の数を引いた時、負の数が出たり、負の数から正の数を引いた時、正の数が出た場合に、オーバーフローが起こっているので、引かれる数と減算結果の排他的論理和が 1 になるという条件が必要です。よって、このようなコードになります。

性能評価としては、LUT 数が 379 で、RR における遅延時間が 20.024 です。
クリティカルパスは d[0]から Z です。

❻SEG_SEL

```
1   module SEG_SEL(in,seg_sel,seg_sel_1,seg_sel_2,seg_sel_3,seg_sel_4,seg_sel_5,seg_sel_6,seg_sel_7);
2      input [2:0]in;
3      output seg_sel,seg_sel_1,seg_sel_2,seg_sel_3,seg_sel_4,seg_sel_5,seg_sel_6,seg_sel_7;
4
5      assign seg_sel=(in==3'b000)? 1'b1:1'b0;
6      assign seg_sel_1=(in==3'b001)? 1'b1:1'b0;
7      assign seg_sel_2=(in==3'b010)? 1'b1:1'b0;
8      assign seg_sel_3=(in==3'b011)? 1'b1:1'b0;
9      assign seg_sel_4=(in==3'b100)? 1'b1:1'b0;
10     assign seg_sel_5=(in==3'b101)? 1'b1:1'b0;
11     assign seg_sel_6=(in==3'b110)? 1'b1:1'b0;
12     assign seg_sel_7=(in==3'b111)? 1'b1:1'b0;
13  endmodule
```

図１４

図１４が SEG_SEL のコードです。
このＳＥＧ_ＳＥＬの外部仕様は、入力が in で、出力が seg_sel,seg_sel_1,seg_sel_2,seg_sel_3, seg _sel_4,seg_sel_5,seg_sel_6,seg_sel_7 です。In が 3 ビットで、それ以外が 1 ビットです。in の値によって、出力が変わります。この seg_sel の信号の番号により、LED の光る場所が決まります。
内部仕様としては、in が 0 の時は、seg_sel が 1、in が 1 の時は、seg_sel_1 が 1、in が 2 の時は、seg_sel_2 が 1、in が 3 の時は、seg_sel_3 が 1、in が 4 の時は、seg_sel_4 が 1、in が 5 の時は、seg_sel_5 が 1、in が 6 の時は、seg_sel_6 が 1、in が 7 の時は、seg_sel_7 が 1 というように、代入しています。

❼szcv_register

```
1   module szcv_register(reg_e, reg_write_en, reg_in, reg_out);
2      input            reg_e;
3       input            reg_write_en;
4      input      [3:0] reg_in;
5      output    reg [3:0] reg_out;
6
7      always @(posedge reg_e) begin
8         if(reg_write_en) begin
9               reg_out <= reg_in;
10              end
11     end
12  endmodule
```

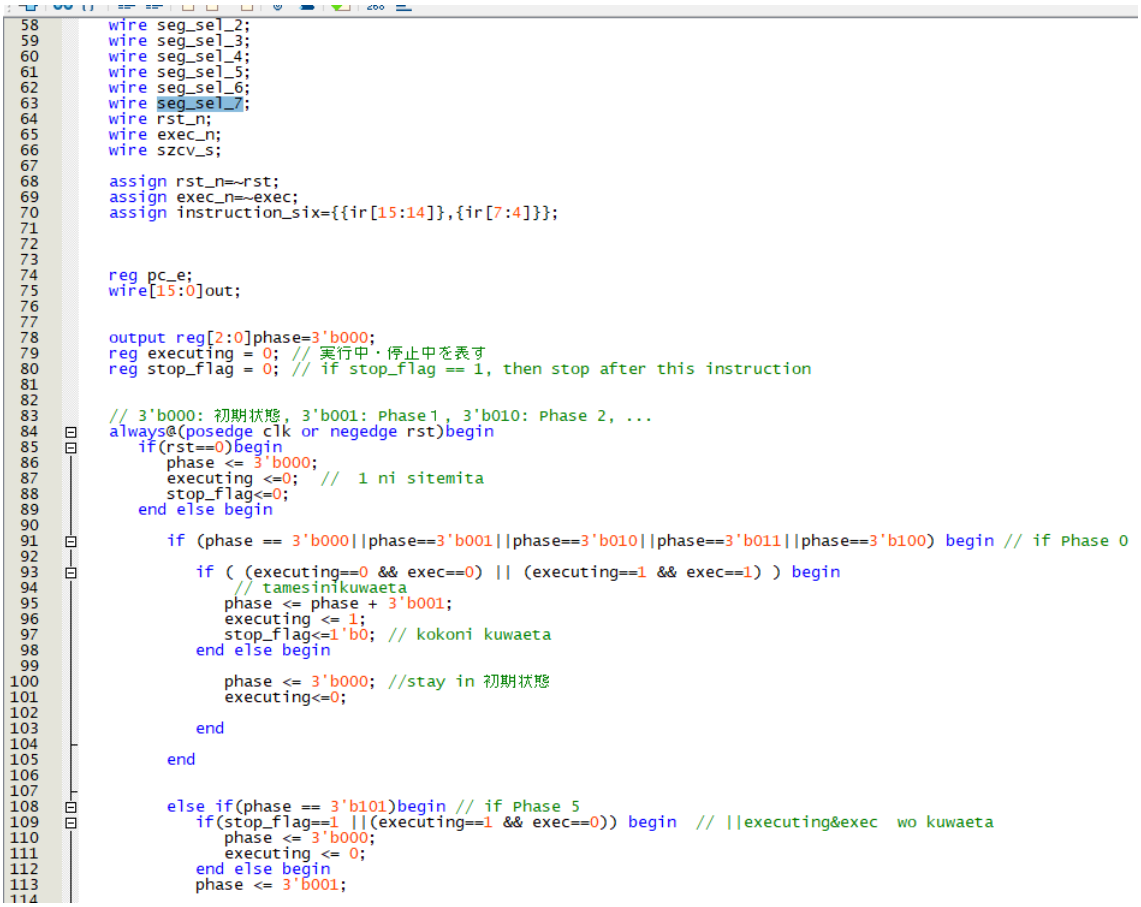
図１５

図１５がszcv_registerのコードです。
このszcv_registerの外部仕様は、入力がreg_e,reg_write_en,reg_inで出力がreg_outです。reg _e,reg_write _enが１ビットで、reg_in,reg_outが４ビットです。
aluの出力であるS,Z,C,Vを次のＳＺＣＶの更新時まで保持するという役割があります。内部仕様としては、reg_eがclockの役割をし、もし、reg_write_enが１の時に、reg_outにreg_inを代入します。

❽simple

```
3   module simple(clk,rst,exec,in,out,out2,out3,out4,seg_out,seg_out_2,seg_sel,seg_sel_1,
4   seg_sel_2,seg_sel_3,seg_sel_4,seg_sel_5,seg_sel_6,seg_sel_7, phase);
5       input clk;
6       input rst;
7       input exec;
8
9       input[15:0]in;
10      output[15:0]out;
11      output[15:0]out2;
12      output[15:0]out3;
13      output[31:0]out4;
14      output[31:0]seg_out;
15      output[31:0]seg_out_2;
16      output seg_sel;
17      output seg_sel_1;
18      output seg_sel_2;
19      output seg_sel_3;
20      output seg_sel_4;
21      output seg_sel_5;
22      output seg_sel_6;
23      output seg_sel_7;
24
25
26      wire aluc_e, ar_e,br_e,dr_e,mdr_e,ir_e,s,z,c,v,jump,
27      mem_e,mem_w,m2_s,m3_s,m4_s,m5_s,m6_s,m7_s,m8_s,m9_s,out_s,hlt,reg_write,reg_read;
28      wire [3:0] ALU_Cnt; //alu opcode
29      wire[5:0] instruction_six;
30      wire [15:0] ar; //AR content
31      wire[15:0] br; //BR content
32      wire[15:0] dr; //DR content
33      wire[15:0] mdr; //MDR content
34      wire[15:0] ir; //ir content
35      wire[15:0] pc; //
36      wire[15:0] pc_inc; //pc+1
37      wire[15:0] m2;
38      wire[15:0] m3;
39      wire[15:0] m4;
40      wire[15:0] m5;
41
42      wire[15:0] m7;
43      wire[15:0] m8;
44      wire[15:0] m9;
45      wire[15:0] m10;
46      wire[15:0] mem_out1; //meireifech
47      wire[15:0] mem_out2; //roadmeirei P4
48      wire[15:0] exd;
49      wire[15:0] exd_im;
50      wire[15:0] re0;
51      wire[15:0] re1;
52      wire[15:0] pc_out;
53      wire [15:0] address;
54      wire [15:0] alu_out;
55      wire[3:0] Flag;
56      wire seg_sel;
57      wire seg_sel_1;
```

図１６



```
58      wire seg_sel_2;
59      wire seg_sel_3;
60      wire seg_sel_4;
61      wire seg_sel_5;
62      wire seg_sel_6;
63      wire seg_sel_7;
64      wire rst_n;
65      wire exec_n;
66      wire szcv_s;
67
68      assign rst_n=~rst;
69      assign exec_n=~exec;
70      assign instruction_six={{ir[15:14]},{ir[7:4]}};
71
72
73
74      reg pc_e;
75      wire[15:0]out;
76
77
78      output reg[2:0]phase=3'b000;
79      reg executing = 0; // 実行中・停止中を表す
80      reg stop_flag = 0; // if stop_flag == 1, then stop after this instruction
81
82
83      // 3'b000: 初期状態, 3'b001: Phase 1 , 3'b010: Phase 2, ...
84      always@(posedge clk or negedge rst)begin
85         if(rst==0)begin
86            phase <= 3'b000;
87            executing <=0;  //  1 ni sitemita
88            stop_flag<=0;
89         end else begin
90
91            if (phase == 3'b000||phase==3'b001||phase==3'b010||phase==3'b011||phase==3'b100) begin // if Phase 0
92
93               if ( (executing==0 && exec==0) || (executing==1 && exec==1) ) begin
94                  // tamesinikuwaeta
95                  phase <= phase + 3'b001;
96                  executing <= 1;
97                  stop_flag<=1'b0; // kokoni kuwaeta
98               end else begin
99
100                  phase <= 3'b000; //stay in 初期状態
101                  executing<=0;
102
103               end
104
105            end
106
107
108            else if(phase == 3'b101)begin // if Phase 5
109               if(stop_flag==1 ||(executing==1 && exec==0)) begin  // ||executing&exec  wo kuwaeta
110                  phase <= 3'b000;
111                  executing <= 0;
112               end else begin
113               phase <= 3'b001;
114
```

図１７

```
115
116             end
117         end
118         else begin
119             phase <= phase + 3'b001;
120         end
121
122         if(hlt==1'b1)begin
123             stop_flag<=1;
124         end
125         //stop_flag<=hlt;
126
127
128
129          //kokoniarunoha exec tekini mazui
130
131
132     end
133 end
134 control controls(.phase(phase),.S(Flag[3]),.Z(Flag[2]),.C(Flag[1]),
135 .V(Flag[0]),.instruction(ir),.aluc_e(aluc_e),.ar_e(ar_e)
136 ,.br_e(br_e),.dr_e(dr_e),.mdr_e(mdr_e),.ir_e(ir_e),.genr_w(genr_w)
137 ,.mem_e(mem_e)
138 ,.mem_w(mem_w),.jump(jump) ,.m2_s(m2_s),.m3_s(m3_s),.m4_s(m4_s)
139 ,.m5_s(m5_s),.m7_s(m7_s),.m8_s(m8_s),.m9_s(m9_s),.out_s(out_s),.hlt(hlt),.szcv_s(szcv_s),
140 alu_instruction(alu_instruction));
141 //MEI wo ir nikaeta
142
143 seven sev(.in(re0),.signal(out_s),.out(seg_out));  //out_s wo 1'b1   ar wo re0
144  //re0 wo kaeta  mem_out1 wo pc_out
145
146 seven sev2(.in(m10),.signal(out_s),.out(seg_out_2));
147
148 SEG_SEL(.in(ir[3:1]),.seg_sel(seg_sel),.seg_sel_1(seg_sel_1),.seg_sel_2(seg_sel_2)
149 ,.seg_sel_3(seg_sel_3),.seg_sel_4(seg_sel_4),.seg_sel_5(seg_sel_5),.seg_sel_6(seg_sel_6)
150 ,.seg_sel_7(seg_sel_7));
151
152 szcv_register(.reg_e(clk),.reg_write_en(szcv_s),.reg_in({S,Z,C,V}),.reg_out(Flag));
153
154
155 register_16 IR(.reg_e(clk), .reg_write_en(ir_e), .reg_in(mem_out1) //MEI wo mem_out1
156 , .reg_out(ir)); //ir_e wo 1'b1
157
158 register_16 AR(.reg_e(clk), .reg_write_en(ar_e), .reg_in(m9)
159 , .reg_out(ar));
160
161 register_16 BR(.reg_e(clk), .reg_write_en(br_e), .reg_in(m3)
162 , .reg_out(br));
163
164 register_16 DR(.reg_e(clk), .reg_write_en(dr_e), .reg_in(alu_out)
165 , .reg_out(dr));
166
167 register_16 MDR(.reg_e(clk),.reg_write_en(mdr_e),.reg_in(m7)
168 ,.reg_out(mdr));
169
170 register_general registerfile(.clk(clk),.rst(rst),
171 .reg_write_en(genr_w)   //reg_e wo genr_w nisita
```

図１８

```
    ,.reg_write_dest(m5),.reg_write_data(m8),.reg_read_addr_1(ir[13:11])
    ,.reg_read_data_1(re0),.reg_read_addr_2(ir[10:8]),.reg_read_data_2  //MEI wo ir nisita
    (re1));

    alu_control_unit aluconu(.alu_control_unit_e(clk)
    ,.instruction_six(instruction_six),.ALU_Cnt(ALU_Cnt));

    alu alu_0( .opcode(ALU_Cnt),.d(ir[3:0])
    ,. alu_in_a(ar), .alu_in_b(br), .alu_out(alu_out), .s(s),.z(z)
    ,.c(c),.v(v));   //ar wo 3 br wo1  ALU_Cnt wo ir[7:4]

    ram01 inst_memory(.data(16'b0),.wren(1'b0),.address(pc_out)
    ,.clock(clk),.q(mem_out1));

    //ram01 inst_memory(.data(16'b0),.wren(1'b0),.address(pc_out),.clock(clk),.q(mem_out1));

    ram02 data_memory(.data(re0),.wren(mem_w),.
    address(alu_out),.clock(clk),.q(mem_out2));




    program_counter pc_0(.clock(clk),.rst(rst),.j_flag(jump)
    ,.j_addr(dr),.phase(phase),.pc_out(pc_out));

    sign_extension siex(.d(ir[7:0]),.result(exd)); //ir[7:0] wo 8'b00001111

    sign_ext_im(.d({{ir[13:11]},{ir[7:0]}}),.result(exd_im));


    multiplexer_16 m2_0(.mux_s(m2_s),.mux_in_a(re0),.mux_in_b(exd)
    ,.mux_out(m2));

    multiplexer_16 m3_0(.mux_s(m3_s),.mux_in_a(re1),.mux_in_b(pc_out)
    ,.mux_out(m3));

    multiplexer_16 m4_0(.mux_s(m4_s),.mux_in_a(dr),.mux_in_b(mdr)
    ,.mux_out(m4));

    multiplexer_16 m5_0(.mux_s(m5_s),.mux_in_a(ir[13:11]),.mux_in_b(ir[10:8]) //MEI wo ir nisita
    ,.mux_out(m5));



    multiplexer_16 m7_0(.mux_s(m7_s),.mux_in_a(mem_out2),.mux_in_b(in)
    ,.mux_out(m7));

    multiplexer_16 m8_0(.mux_s(m8_s),.mux_in_a(m4),.mux_in_b(exd)
    ,.mux_out(m8));  //m8_s ga 1 ni nattenai

    multiplexer_16 m9_0(.mux_s(m9_s),.mux_in_a(m2),.mux_in_b(exd_im),.mux_out(m9));

    multiplexer_16 m10_0(.mux_s(ir[10]),.mux_in_a(16'b0000000000000000),.mux_in_b(re1),.mux_out(m10));

    assign out=mem_out1;
    assign out2=pc_out; //br wo re1
    assign out3=Flag[2];
```

図１９

図１６から図１９が simple のコードです。この simple というのは、全ての部品をつなげるためのコードです。

simple の外部仕様としては、入力が clk,rst,exec,in で、出力は、seg_out,seg_out_2,seg_sel,seg_sel_1,seg_sel_2,seg_sel_3,seg_sel_4,seg_sel_5,seg_sel_6,seg_sel_7

です。(out,out2,out3,out4 は、デバック用)

clk,rst,exec は、１ビットで、in は１６ビットで、seg_out,seg_out_2 は、３２ビットです。

seg_sel,seg_sel_1,seg_sel_2,seg_sel_3,seg_sel_4,seg_sel_5,seg_sel_6,seg_sel_7 は１ビットで

す。
clk はクロック、rst はリセット、exec は停止起動、in は外部入力の値を表しています。
seg _out,seg_out_2 はＬＥＤへの表示を表します。
always 文の説明をします。まず、リセットボタンが押されると、フェーズカウンタを 0 にし、executing を 0 にし、stop_flag を 0 にします。この executing は、実行中であることを表しており、stop_flag は、１になると、cpu は停止します。
リセットが押されていない時を説明します。
初期状態とフェーズ１～４の時、停止状態で起動ボタンを押した場合と、実行中で停止ボタンを押していない場合は、フェーズカウンタを１進めて、executing に 1 を代入し、stop_flag に 0 を代入します。それ以外の場合は、初期状態にして、executing に 0 を代入します。
フェーズ 5 の場合、stop_flag が 1 の時と、実行中に停止ボタンが押された時には、初期状態に戻し、executing に 1 を代入します。
それ以外の場合は、フェーズカウンタを 1 にします。
そして、hlt 信号が 1 の場合は、stop_flag を 1 にします。
以上が always 文の説明です。
always 文の後は、図 0 の設計図のようにつながるようなコードを書いてあります。
まず、control では、フェーズカウンタとＳＺＣＶ信号と ir の内容を入れることで、それぞれの信号を決めています。
seven sev は、レジスタの中身をＬＥＤに表示させるためのもので、seven sev2 は、拡張機能として、OUTPUT の際に、2 つ目のレジスタも表示できるように追加したものです。
SEG_SEL は、d フィールドの内容によって、8 つのうち、どこに表示させるかを決めるためのものです。
szcv_register は、S,Z,C,V を次の演算命令まで保持させる役割があり、この出力の Flag は control に入力されていきます。
alu_control_unit は、ir[15:14]と ir[7:4]を結合したものから、alu の演算種類を決めるので、それを入力しています。
ram 01　　inst_memory には、データを書き込む必要がないので,wren には 0 が data には０が入力されています。また、プログラムカウンタの値をアドレスとして入力しています。
ram 02　　data_memory には、データとして,レジスタの中身 re0、アドレスとして、alu_out(r[Rb]+sign_ext(d))を入力しています。
program_counter の j_addr には、alu_out(PC+sign_ext(d))を入力しています。
mux 10 についてですが、d フィールドの最下位ビットが 1 の時,レジスタの中身を seven に入力し、0 の時、0 を seven に入力しています。（図１９では、ir[10]となっていますが、正しくは ir[0]です。）

性能評価としては、LUT 数が 1154 で、最大周波数が 65.29MHz です。clock to output times が 15.263 です。

❾考察および感想

元々のプログラムカウンタでは、フェーズカウンタが 5 の時に 1 増やすということができなかったため、フェーズカウンタを入力するようにコードを変更しました。そのようにすることで、フェーズ 5 のタイミングでプログラムカウンタを更新することができるようになっています。

主記憶では、address に pc_out と mem_w の 2 種類があるので、inst_memory と data_memory の 2 つを用意しましたが、これは、pc_out と mem_w をマルチプレクサで選ぶようにすれば、一つの主記憶で収めることができたと考えられます。

control では、当初、命令ごとに、信号を全て羅列して代入していたのですが、それだと、ラッチが生まれたので、信号ごとに if 文を書くコードに変更しました。

szcv_register についてですが、当初はこの部品を作っていなかったのですが、その場合、szcv が alu の演算を行うタイミングであるフェーズ 3 のタイミングで切り替わってしまい、control における出力の信号がフェーズ 3 で変わってしまうという問題点が生まれたので、フェーズ 1 のタイミングかつ演算命令のタイミングのみレジスタの中身を入れ替えるというような部品を作りました。

本実験を通して、コンパイルが上手くいっているとしても、シミュレーション や実機上では、上手くいかないことが多々あるということを学びました。これは、verilog HDL が文法上甘いということが影響していると思います。間違ったことを書いているのに、コンパイルは上手くいくということがよくあったので、そこの間違いをシミュレーションで正すという作業が真新しいものであったので、慣れるまで苦労しました。