Amount of hours:
Javier DeVelasco Orio : 3h
Saad Ouassil Allak : 3h

Unfortunately, we haven’t tested our implementation the the FPGA

```
module dma_des (
    input           [1:0]  encryptDecrypt,
    input logic     [63:0] key,
    input logic     [63:0] data_in,
    output logic    [63:0] data_out
);

byte pc1[55:0]; // permutation table
assign pc1 = {57,49,41,33,25,17,9,
            1,58,50,42,34,26,18,
            10,2,59,51,43,35,27,
            19,11,3,60,52,44,36,
            63,55,47,39,31,23,15,
            7,62,54,46,38,30,22,
            14,6,61,53,45,37,29,
            21,13,5,28,20,12,4};


logic [55:0] key_plus;
always_comb
begin : key_plus_comb
    for(int i=0; i<56; i++) begin
        key_plus[i]=key[pc1[i]];
    end
end

logic [27:0] C[16:0];
logic [27:0] D[16:0];

assign C[0]=key_plus[55:28];
assign D[0]=key_plus[27:0];

always_comb
begin: DC_comb
    for(int i=2; i<17;i++)begin
        C[i] = {C[i-1]<<i[27:1],C[i-1][0]};
        D[i] = {D[i-1]<<i[27:1],D[i-1][0]};
    end
end

byte pc2[47:0];
assign pc2 ={14,17,11,24,1,5,
            3,28,15,6,21,10,
            23,19,12,4,26,8,
            16,7,27,20,13,2,
            41,52,31,37,47,55,
            30,40,51,45,33,48,
            44,49,39,56,34,53,
            46,42,50,36,29,32};

logic [47:0] keys[16:1];
always_comb
begin :key_comb
    logic [55:0] tmp;
    for(int i=1; i<17; i++)begin
        for(int j=0; j<48; j++)begin
            tmp = {C[i],D[i]};
            keys[i][j]=tmp[pc2[j]];
        end
    end
end

byte ip[63:0];
assign ip=  {58,50,42,34,26,18,10,2,
            60,52,44,36,28,20,12,4,
            62,54,46,38,30,22,14,6,
            64,56,48,40,32,24,16,8,
            57,49,41,33,25,17,9,1,
```

```
            59,51,43,35,27,19,11,3,
            61,53,45,37,29,21,13,5,
            63,55,47,39,31,23,15,7};
logic[63:0] data_in_ip;
always_comb
begin: data_in_permutation
    for(int i=0; i<64; i++)begin
        data_in_ip[i] = data_in[ip[i]];
    end
end
// for encryption
logic[31:0] LE[16:0];
logic[31:0] RE[16:0];
// for decryption
logic[31:0] RD[16:0];
logic[31:0] LD[16:0];
assign LE[0] = data_in_ip[63:32];
assign LD[0] = data_in_ip[63:32];
assign RE[0] = data_in_ip[31:0];
assign RD[0] = data_in_ip[31:0];

function sb;
    input integer i;
    input logic [5:0]b;

    logic[3:0] S1[4][16];
    logic[3:0] S2[4][16];
    logic[3:0] S3[4][16];
    logic[3:0] S4[4][16];
    logic[3:0] S5[4][16];
    logic[3:0] S6[4][16];
    logic[3:0] S7[4][16];
    logic[3:0] S8[4][16];
    logic[3:0] S[8][4][16];
    integer bi = {b[5],b[0]};
    integer bj = b[4:1];

    S1 = {
    {14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7},
    {0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8},
    {4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0},
    {15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13}
    };
    S2= {
    {15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10},
    {3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5},
    {0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15},
    {13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9}
    };
    S3={
    {10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8},
    {13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1},
    {13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7},
    {1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12}
    };
    S4={
    {7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15},
    {13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9},
    {10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4},
    {3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14}
    };
    S5={
    {2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9},
    {14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6},
    {4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14},
    {11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3}
    };
    S6={
```

```
    {12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11},
    {10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8},
    {9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6},
    {4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13}
    };
    S7={
    {4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1},
    {13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6},
    {1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2},
    {6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12}
    };
    S8={
    {13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7},
    {1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2},
    {7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8},
    {2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11}
    };

    S = {S1,S2,S3,S4,S5,S6,S7,S8};
    sb = S[i-1][bi][bj];

endfunction;

function frk;
    input logic [31:0]r;
    input logic [47:0]k;
    byte E[47:0];
    logic [47:0] er;
    logic [47:0] tmp;
    logic [32:0] tmpf;
    byte P[31:0];
    logic [31:0]res;
    P = {
    16,7,20,21,
    29,12,28,17,
    1,15,23,26,
    5,18,31,10,
    2,8,24,14,
    32,27,3,9,
    19,13,30,6,
    22,11,4,25
    };
    E = {32,1,2,3,4,5,
        4,5,6,7,8,9,
        8,9,10,11,12,13,
        12,13,14,15,16,17,
        16,17,18,19,20,21,
        20,21,22,23,24,25,
        24,25,26,27,28,29,
        28,29,30,31,32,1};

    for(int i=0; i<48; i++)begin
        er[i]=r[E[i]];
    end
    tmp = er^k;
    for(int i=0; i<8;i++)begin
        tmpf[(i+1)*4-1-:4] = sb(8-i,tmp[(i+1)*6-1-:6]);
    end

    for (int i = 0; i<32; i++)begin
        res[i] = tmpf[P[i]];
    end
    frk = res;
endfunction

always_comb
begin: LR_comb
    for(int i = 1; i<17; i++) begin
```

```
        LE[i]=RE[i-1];
        LD[i]=RD[i-1];
        RE[i]=LE[i-1]^frk(RE[i-1],keys[i]);
        RD[i]=LD[i-1]^frk(RD[i-1],keys[17-i]);
    end
end


logic[63:0] encrypt;
logic[63:0] decrypt;
logic[63:0] encrypt_ip;
logic[63:0] decrypt_ip;
assign encrypt = {RE[16],LE[16]};
assign decrypt = {RD[16],LD[16]};

byte ip_1[63:0];
assign ip_1 = {
40,8,48,16,56,24,64,32,
39,7,47,15,55,23,63,31,
38,6,46,14,54,22,62,30,
37,5,45,13,53,21,61,29,
36,4,44,12,52,20,60,28,
35,3,43,11,51,19,59,27,
34,2,42,10,50,18,58,26,
33,1,41,9,49,17,57,25
};
always_comb
begin : IP_1_comb
    for(int i=0; i<64; i++) begin
        encrypt_ip[i]=encrypt[ip[i]];
        decrypt_ip[i]=decrypt[ip[i]];
    end
end

always_comb
begin
    case (encryptDecrypt)
        2'b00 : data_out = data_in;
        2'b01 : begin
                data_out = encrypt_ip;
                end
        2'b10 : begin
                data_out = decrypt_ip;
                end
        default: data_out = data_in;
    endcase
end
endmodule
```

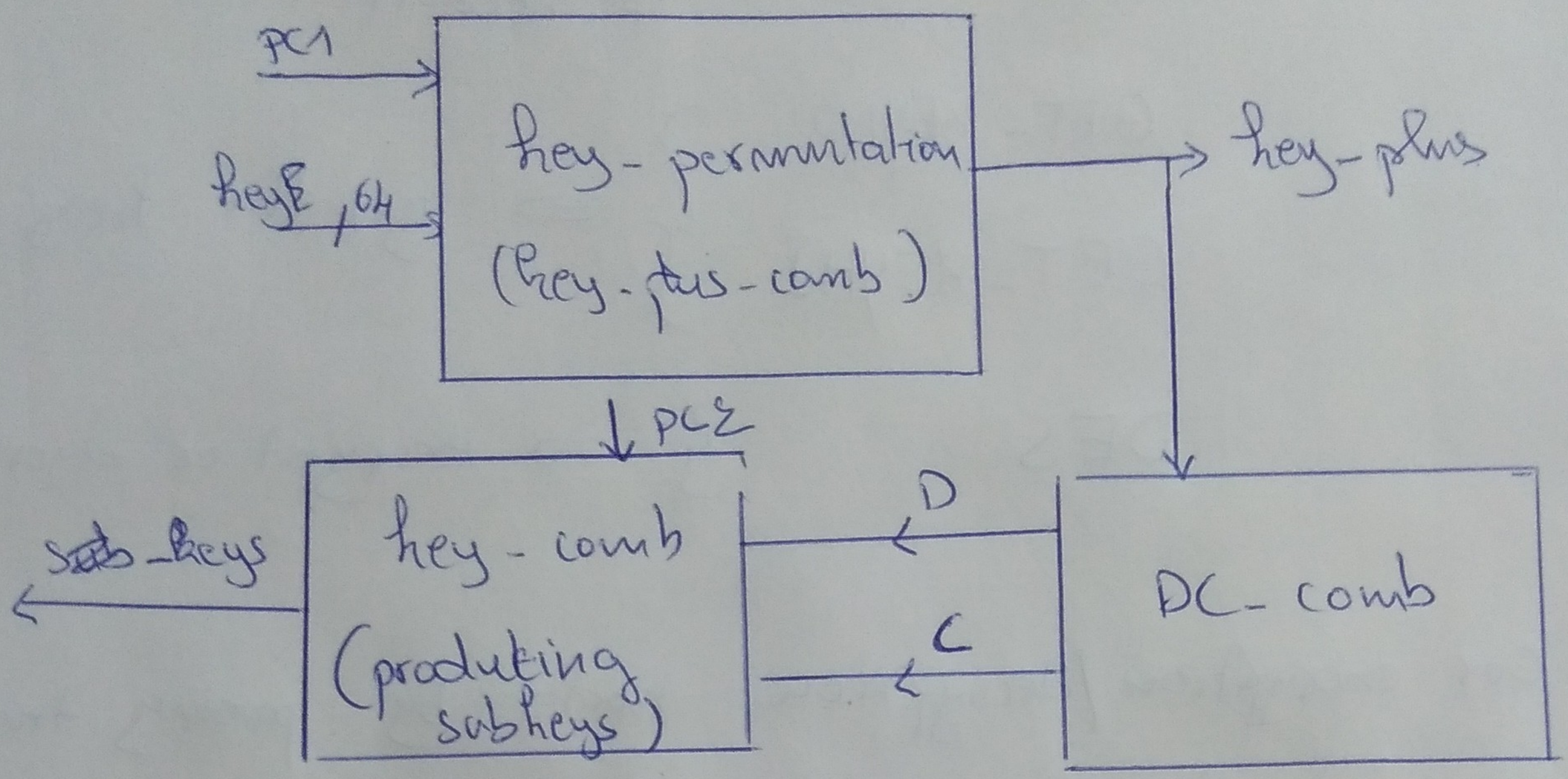
```
`include "mipsfpga_ahb_const.vh"

`define DMA_BASE_ADDR 27'hf98000
`define FIFO_SIZE 256

module mipsfpga_dma_engine (
output                      HCLK,
output                      HRESETn,
output logic    [ 31: 0]    HADDR,
output          [  2: 0]    HBURST,
output                      HMASTLOCK,
output          [  3: 0]    HPROT,
output          [  2: 0]    HSIZE,
output logic    [  1: 0]    HTRANS,
output logic    [ 31: 0]    HWDATA,
output logic                HWRITE,
input  logic    [ 31: 0]    HRDATA,
input                       HREADY,
input                       HRESP,

input                       CPU_CLK,
input                       CPU_RESETn,
input                       DMA_INTERRUPT,
output logic                CLEAR_START
);

//synchronize cpu and dma engine
assign HCLK = CPU_CLK;
assign HRESETn = CPU_RESETn;

assign HMASTLOCK = 0;
assign HPROT = 4'b0010;
assign HSIZE = 3'b010;
assign HBURST = 3'b000;

typedef enum { IDLE, GET_SIZE, GET_SRC, GET_DST, GET_DATA, GET_ED, DES, GET_KEYLO, GET_KEYH
I, SEND_DATA} state_t;
state_t dma_state;

logic [31:0] fifo [`FIFO_SIZE-1:0];
integer fifo_counter = 0;

logic [31:0] dma_src;
logic [31:0] dma_dst;
logic [31:0] dma_size;
logic [63:0] des_key;
logic [63:0] des_data_in;
logic [63:0] des_data_out;
logic [1:0] des_encryptDecrypt;

always @(posedge CPU_CLK, negedge CPU_RESETn)
begin
    if(~CPU_RESETn) begin
        dma_state <= IDLE;
        fifo_counter <= 0;

        dma_src <= 0;
        dma_dst <= 0;
        dma_size <= 0;

        HTRANS <= 2'b00;
        HWRITE <= 0;
        HWDATA <= 0;
        HADDR <= 0;

    end else begin
        CLEAR_START<=0;
        case (dma_state)
```

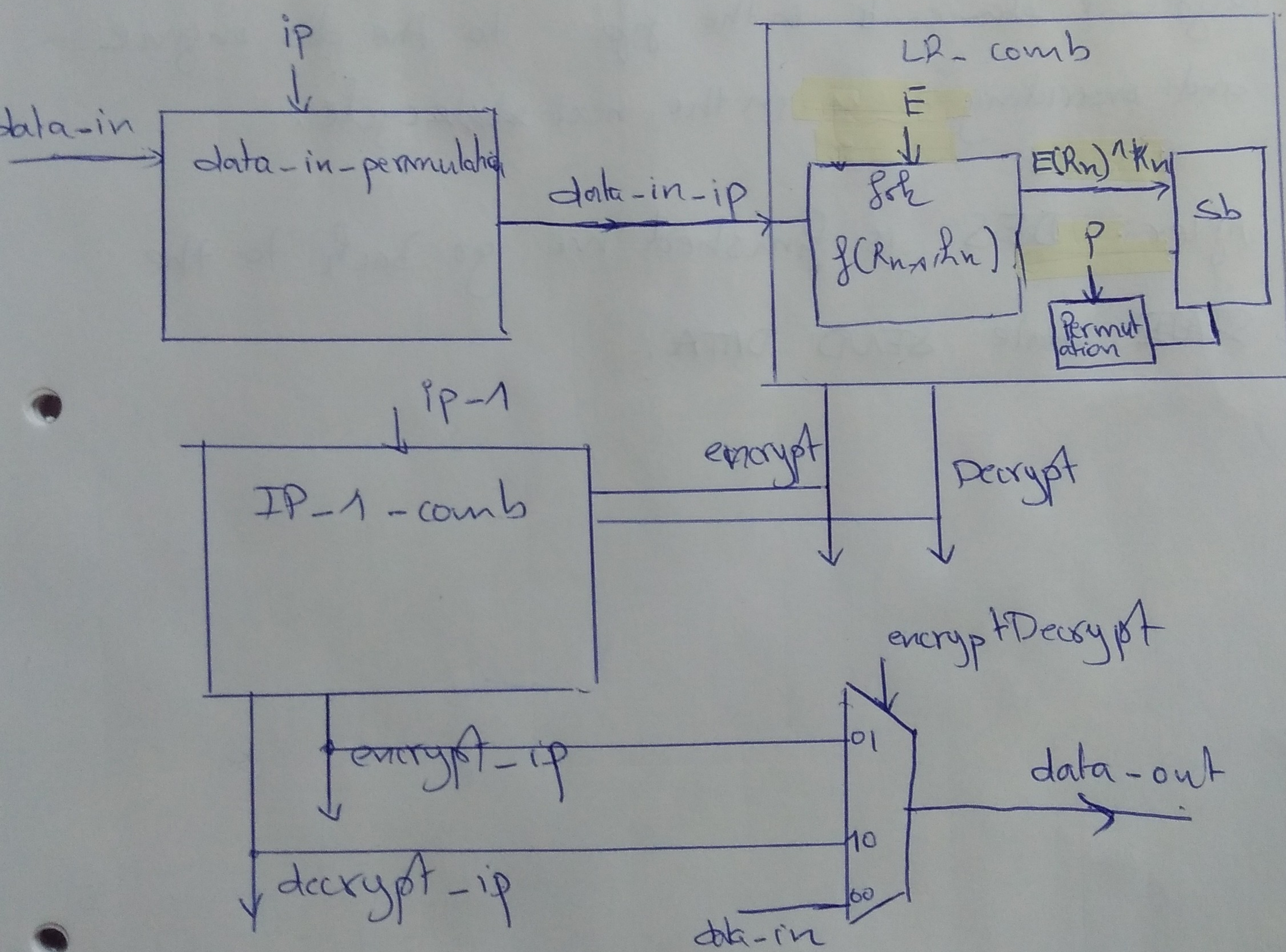
```
            IDLE :     begin
                           if(HREADY) begin
                               HWDATA<= 0;
                               HTRANS<= 2'b00;
                           end
                           if(DMA_INTERRUPT) begin
                               HTRANS <= 2'b10;
                               HADDR  <= {`DMA_BASE_ADDR ,`DMA_SIZE_ADDR};
                               if(HREADY) begin
                                   dma_state <= GET_SIZE;
                                   fifo_counter <= 0;
                                   CLEAR_START <= 1;
                               end

                           end
                       end
            GET_SIZE:  begin
                           if(~HRESP && HREADY) begin
                               dma_size <= HRDATA;
                               dma_state <= GET_SRC;
                               HADDR  <= {`DMA_BASE_ADDR ,`DMA_SRC_ADDR};
                           end
                       end
            GET_SRC:   begin
                           if(~HRESP && HREADY) begin
                               dma_src <= HRDATA;
                               dma_state <= GET_DST;
                               HADDR  <= {`DMA_BASE_ADDR ,`DMA_DST_ADDR};
                           end
                       end
            GET_DST:   begin
                           if(~HRESP && HREADY) begin
                               dma_dst <= HRDATA;
                               dma_state <= GET_ED;
                               HADDR  <= {`DMA_BASE_ADDR,`DMA_ED_ADDR};
                           end
                       end

            GET_ED:   begin
                           if(~HRESP && HREADY) begin
                               des_encryptDecrypt <= HRDATA;
                               if(HRDATA==0) begin
                                   dma_state <= GET_DATA;
                                   HADDR <= dma_src;
                               end else begin
                                   dma_state <= GET_KEYHI;
                                   HADDR <= {`DMA_BASE_ADDR, `DMA_KEYHI_ADDR};
                               end
                           end
                      end
            GET_KEYHI: begin
                        if(~HRESP && HREADY) begin
                            des_key[63:32] <= HRDATA;
                            dma_state <= GET_KEYLO;
                            HADDR <= {`DMA_BASE_ADDR, `DMA_KEYLO_ADDR};
                        end
                       end
            GET_KEYLO: begin
                        if(~HRESP && HREADY) begin
                            des_key[31:0] <= HRDATA;
                            dma_state <= GET_DATA;
                            HADDR <= dma_src;
                        end
                       end
            GET_DATA: begin
                           if(~HRESP && HREADY) begin
                               fifo[fifo_counter] <= HRDATA;
                               if(fifo_counter<dma_size-1) begin
```

```
                                    fifo_counter++;
                                    HADDR <= HADDR+4;
                                end else begin
                                    fifo_counter <= 0;
                                    if(des_encryptDecrypt==0) begin
                                        HADDR <= dma_dst;
                                        HWRITE <= 1;
                                        dma_state <= SEND_DATA;
                                    end else begin
                                        dma_state <= DES;
                                        HADDR <= 32'h0;
                                        des_data_in <= {fifo[0],fifo[1]};
                                        fifo_counter <= 2;
                                    end;
                                end
                            end
                        end
            DES:        begin
                            fifo[fifo_counter-2] = des_data_out[63:32];
                            fifo[fifo_counter-1] = des_data_out[31:0];
                            if(fifo_counter>=dma_size) begin
                                fifo_counter <= 0;
                                dma_state <= SEND_DATA;
                                HADDR <= dma_dst;
                                HWRITE <= 1;
                            end else begin
                                des_data_in <= {fifo[fifo_counter], fifo[fifo_counter+1]};
                                fifo_counter <= fifo_counter+2;
                            end
                        end

            SEND_DATA: begin
                            if(~HRESP && HREADY) begin
                                HWDATA <= fifo[fifo_counter];
                                if(fifo_counter<dma_size-1) begin
                                    fifo_counter++;
                                    HADDR <= HADDR+4;
                                end else begin
                                    fifo_counter <= 0;
                                    HADDR <= 0;
                                    HWRITE<= 0;
                                    dma_state <= IDLE;
                                end
                            end
                        end
        endcase

    end
end
dma_des  des_engine (
    des_encryptDecrypt,
    des_key,
    des_data_in,
    des_data_out
);

endmodule
```

```
`include "mipsfpga_ahb_const.vh"

module mipsfpga_ahb_dmaregs(
    input           HCLK,
    input           HRESETn,
    input  [  3: 0] HADDR,
    input  logic    [ 31: 0] HWDATA,
    input           HWRITE,
    input           HSEL,
    output logic    [ 31: 0] HRDATA,

    output          DMA_INTERRUPT, // asserted when start and size!=0
    input           CLEAR_START // asynchronous reset of the start
);

logic [31:0] dma_start;
logic [31:0] dma_dst;
logic [31:0] dma_size;
logic [31:0] dma_src;
logic [31:0] dma_keylo;
logic [31:0] dma_keyhi;
logic [1:0] dma_ed; //encrypt/decryp


always_ff @(posedge HCLK or negedge HRESETn)
begin
       if(CLEAR_START)begin
            dma_start <= 0;
       end
       if (~HRESETn) begin
            dma_start <= 0;
            dma_dst <= 0;
            dma_size <= 0;
            dma_src <= 0;
       end else if (HWRITE & HSEL) begin
            case (HADDR)
                `DMA_START_ADDR:    dma_start<=HWDATA;
                `DMA_SRC_ADDR :     dma_src<=HWDATA;
                `DMA_SIZE_ADDR:     dma_size<=HWDATA;
                `DMA_DST_ADDR :     dma_dst<=HWDATA;
                `DMA_KEYHI_ADDR:    dma_keyhi <= HWDATA;
                `DMA_KEYLO_ADDR:    dma_keylo <= HWDATA;
                `DMA_ED_ADDR :      dma_ed <= HWDATA[1:0];
            endcase
       end
end

always_comb
     case (HADDR)
        `DMA_START_ADDR:    HRDATA = dma_start;
        `DMA_SRC_ADDR :     HRDATA = dma_src;
        `DMA_SIZE_ADDR:     HRDATA = dma_size;
        `DMA_DST_ADDR :     HRDATA = dma_dst;
        `DMA_KEYLO_ADDR:    HRDATA = dma_keylo;
        `DMA_KEYHI_ADDR:    HRDATA = dma_keyhi;
        `DMA_ED_ADDR:       HRDATA = {30'h0 , dma_ed};
        default:            HRDATA = 32'h00000000;
     endcase

assign  DMA_INTERRUPT = (dma_start!=0) && (dma_size!=0);

endmodule
```

Our implementation of the des engine contains several combinatoral blocks; each block is modelizes one step of the algorithm. described on link provided





We extended our DMA engine with three more states: GED - ED : check if encryption/decryption is needed

GET-KEYLO
GET_KEYHI
} => get the key

DES } => encrypt or dycrypt

Our encryption/decryption works by passing two adjacent elements in the fifo to the des engine and overriding them in the next ~~state~~ cycle.

After DES is finished we go back to the ~~STATE~~ state SEND DATA.