하드웨어 시스템 설계 6주차 실습 보고서

2017-12751 컴퓨터공학부 이동학

Goal: Implement PE controller based on PE made in lab5

Code:

pe_tb

```
module pe_tb();
    parameter L_RAM_SIZE = 4;
    reg start;
    wire done;
    reg aclk;
    reg aresetn;
    wire [L_RAM_SIZE:0] rdaddr;
    reg [31:0] rddata;
    wire [31:0] wrdata;

    reg [31:0] in [0:31];

    initial begin
        aclk <= 0;
        start <= 0;
        aresetn <= 0;

        $readmemh("din.txt", in);
```

```
        #10;
        start <= 1;
        aresetn <= 1;
        #10;
        start <= 0;
        #3300;
        aresetn <= 0;
    end

    always #5 aclk <= ~aclk;

    always @(negedge aclk) begin
        rddata <= in[rdaddr];
    end

    pe_ctrl pe(
        .start(start),
        .done(done),
        .aclk(aclk),
        .aresetn(aresetn),
        .rdaddr(rdaddr),
        .rddata(rddata),
        .wrdata(wrdata)
    );
endmodule
```

pe_ctrl

```
module pe_ctrl#(
        parameter VECTOR_SIZE = 16,
        parameter L_RAM_SIZE = 4
    )
    (
        input start,
        output done,
        input aclk,
        input aresetn,
        output [L_RAM_SIZE:0] rdaddr,
        input [31:0] rddata,
        output reg [31:0] wrdata
    );

    wire [31:0] ain;
    wire [31:0] din;
    wire [L_RAM_SIZE-1:0] addr;
    wire we;
    wire global_we;
    wire valid;
    wire dvalid;
    wire [31:0] dout;
    reg [31:0] global_dout;
```

```
(* ram_style = "block" *) reg [31:0] global_buffer [0:VECTOR_SIZE-1];


reg [3:0] state, state_d;
wire LOAD_END;
wire CALC_END;
wire DONE_END;
localparam S_IDLE = 4'd0;
localparam S_LOAD = 4'd1;
localparam S_CALC = 4'd2;
localparam S_DONE = 4'd3;


always @(posedge aclk)
    if (!aresetn) begin
        state <= S_IDLE;
        state_d <= S_IDLE;
    end
    else begin
        state_d <= state;
        case (state)
            S_IDLE:
                state <= (start) ? S_LOAD : S_IDLE;
            S_LOAD:
                state <= (LOAD_END) ? S_CALC : S_LOAD;
            S_CALC:
                state <= (CALC_END) ? S_DONE : S_CALC;
```

```
                S_DONE:
                    state <= (DONE_END) ? S_IDLE : S_DONE;
                default:
                    state <= S_IDLE;
            endcase
        end

reg LOAD_FLAG;
wire LOAD_RESET = (!aresetn || LOAD_END);
wire LOAD_EN = (state == S_LOAD) && (state_d == S_IDLE);
always @(posedge aclk)
        if (LOAD_RESET)
            LOAD_FLAG <= 'd0;
        else
            if (LOAD_EN)
                LOAD_FLAG <= 'd1;
            else
                LOAD_FLAG <= LOAD_FLAG;

assign we = (LOAD_FLAG && rdaddr[L_RAM_SIZE]) ? 'd1 : 'd0;
assign global_we = (LOAD_FLAG && !rdaddr[L_RAM_SIZE]) ? 'd1 : 'd0;

always @(posedge aclk)
        if (global_we) global_buffer[addr] <= rddata;
        else global_dout <= global_buffer[addr];
```

```
assign din = we ? rddata   : 'd0;


reg CALC_FLAG;
wire CALC_RESET = (!aresetn || CALC_END);
wire CALC_EN = (state == S_CALC) && (state_d == S_LOAD);
always @(posedge aclk)
    if (CALC_RESET)
        CALC_FLAG <= 'd0;
    else
        if (CALC_EN)
            CALC_FLAG <= 'd1;
        else
            CALC_FLAG <= CALC_FLAG;


always @(posedge aclk)
    if (!aresetn)
        wrdata <= 'd0;
    else
        if (CALC_END)
            wrdata <= dout;
        else
            wrdata <= wrdata;


reg valid_pre, valid_reg;
always @(posedge aclk)
    if (!aresetn)
```

```
            valid_pre <= 'd0;
        else
            if (CALC_EN || dvalid)
                valid_pre <= 'd1;
            else
                valid_pre <= 'd0;

always @(posedge aclk)
        if (!aresetn)
            valid_reg <= 'd0;
        else if (CALC_FLAG)
            valid_reg <= valid_pre;

assign valid = valid_reg;
assign ain = valid ? global_dout : 'd0;

reg DONE_FLAG;
wire DONE_RESET = (!aresetn || DONE_END);
wire DONE_EN = (state == S_DONE) && (state_d == S_CALC);
always @(posedge aclk)
        if (DONE_RESET)
            DONE_FLAG <= 'd0;
        else
            if (DONE_EN)
                DONE_FLAG <= 'd1;
            else
```

```
                DONE_FLAG <= DONE_FLAG;

assign done = (state == S_DONE);

reg [31:0] CNT;

wire CNT_LD = (LOAD_EN || CALC_EN || DONE_EN);
wire CNT_EN = (LOAD_FLAG || (CALC_FLAG && dvalid) || DONE_FLAG);
wire CNT_RESET = (!aresetn);

always @(posedge aclk)
    if (CNT_RESET)
        CNT <= 'd0;
    else
        if (CNT_LD)
            CNT <= 'd0;
        else if (CNT_EN)
            CNT <= CNT + 1;

assign addr = (LOAD_FLAG) ? rdaddr[L_RAM_SIZE-1:0] :
                (CALC_FLAG) ? CNT :
                'd0;
    assign rdaddr = (LOAD_FLAG) ? CNT : 'd0;
assign LOAD_END = (LOAD_FLAG) && (CNT == 'd31);
assign CALC_END = (CALC_FLAG) && (CNT == 'd15) && dvalid;
assign DONE_END = (DONE_FLAG) && (CNT == 'd4);
```

```
    my_pe #(
        .L_RAM_SIZE(L_RAM_SIZE)
    ) pe (
        .aclk(aclk),
        .aresetn(aresetn && (state != S_DONE)),
        .ain(ain),
        .din(din),
        .addr(addr),
        .we(we),
        .valid(valid),
        .dvalid(dvalid),
        .dout(dout)
    );
endmodule
```

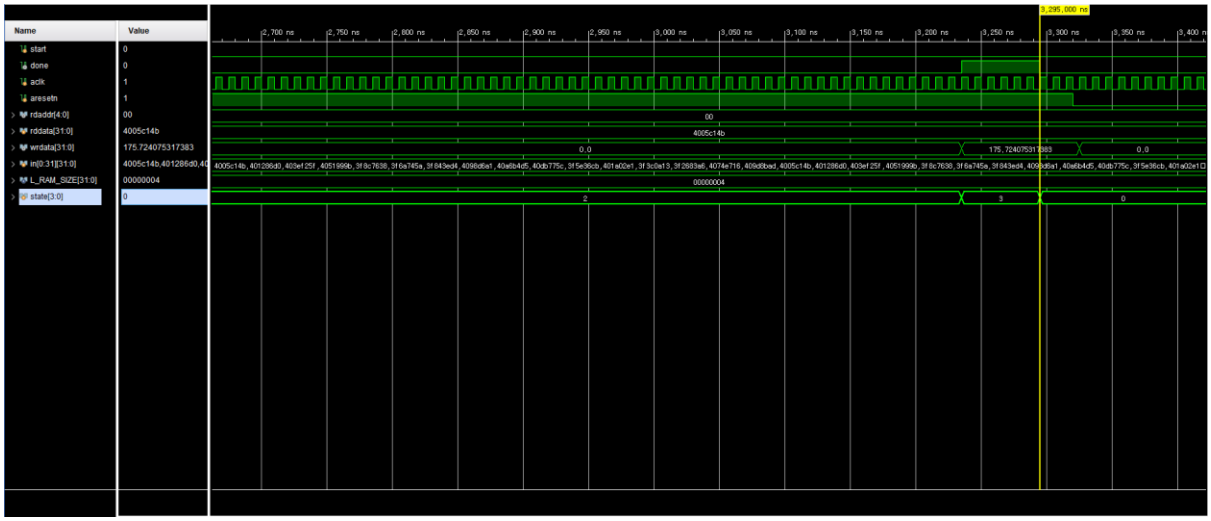
my_pe

```
module my_pe #(
    parameter L_RAM_SIZE = 6
    )
    (
    input aclk,
    input aresetn,
    input [31:0] ain,
    input [31:0] din,
```

```
input [L_RAM_SIZE-1:0] addr,
input we,
input valid,
output dvalid,
output [31:0] dout
);

(* ram_style = "block" *) reg [31:0] peram [0:2**L_RAM_SIZE - 1];

reg [31:0] psum;
wire [31:0] buffer;

always @(posedge aclk) begin
    if(we == 1) peram[addr] <= din;
    if(~aresetn) psum <= 0;
    if(dvalid == 1) psum = buffer;
end

assign dout = (dvalid == 1) ? buffer : 32'b0;

floating_point_0 mac(
    .aclk(aclk),
    .aresetn(aresetn),
    .s_axis_a_tdata(ain),
    .s_axis_a_tvalid(valid),
    .s_axis_b_tdata(peram[addr]),
```

```
        .s_axis_b_tvalid(valid),
        .s_axis_c_tdata(psum),
        .s_axis_c_tvalid(valid),
        .m_axis_result_tdata(buffer),
        .m_axis_result_tvalid(dvalid)
    );
endmodule
```

Result: pe_ctrl



Discussion: 제가 구현한 pe_ctrl 을 설명드리도록 하겠습니다.

1. FSM

```
reg [3:0] state, state_d;
wire LOAD_END;
wire CALC_END;
wire DONE_END;
localparam S_IDLE = 4'd0;
localparam S_LOAD = 4'd1;
localparam S_CALC = 4'd2;
localparam S_DONE = 4'd3;

always @(posedge aclk)
    if (!aresetn) begin
        state <= S_IDLE;
        state_d <= S_IDLE;
    end
    else begin
        state_d <= state;
        case (state)
            S_IDLE:
                state <= (start) ? S_LOAD : S_IDLE;
            S_LOAD:
                state <= (LOAD_END) ? S_CALC : S_LOAD;
            S_CALC:
                state <= (CALC_END) ? S_DONE : S_CALC;
            S_DONE:
                state <= (DONE_END) ? S_IDLE : S_DONE;
            default:
                state <= S_IDLE;
        endcase
    end
```

각 state 에 진입한 직후인지 판단하기 위해 state 와 state_d 두개의 변수를 만들었습니다. reset 하면 둘다 S_IDLE 이고 start 하거나 각 state 가 끝나면 다음 state 로 변합니다.

2. LOAD

```
reg LOAD_FLAG;
wire LOAD_RESET = (!aresetn || LOAD_END);
wire LOAD_EN = (state == S_LOAD) && (state_d == S_IDLE);
always @(posedge aclk)
    if (LOAD_RESET)
        LOAD_FLAG <= 'd0;
    else
        if (LOAD_EN)
            LOAD_FLAG <= 'd1;
        else
            LOAD_FLAG <= LOAD_FLAG;

assign we = (LOAD_FLAG && rdaddr[L_RAM_SIZE]) ? 'd1 : 'd0;
assign global_we = (LOAD_FLAG && !rdaddr[L_RAM_SIZE]) ? 'd1 : 'd0;

always @(posedge aclk)
    if (global_we) global_buffer[addr] <= rddata;
    else global_dout <= global_buffer[addr];

assign din = we ? rddata  : 'd0;
```

각 state 마다 flag, reset, en 의 3 개의 변수가 있습니다. flag 는 현재 state 가 무엇인지, reset 은 외부에서 reset 이 들어오거나 state 가 끝났는지, en 은 state 에 진입한 직후인지 판단합니다. load 상태일 때 rdaddr 가 0~15 면 global_we 를, 16~31 이면 we 를 활성화합니다. global_we 상태이면 global_buffer 에 rddata 의 값을 아니라면 global_dout 에 global_buffer 의 값을 저장합니다. we 상태이면 din 에 rddata 값을 저장합니다.

3. CALC

```
reg CALC_FLAG;
wire CALC_RESET = (!aresetn || CALC_END);
wire CALC_EN = (state == S_CALC) && (state_d == S_LOAD);
always @(posedge aclk)
    if (CALC_RESET)
        CALC_FLAG <= 'd0;
    else
        if (CALC_EN)
            CALC_FLAG <= 'd1;
        else
            CALC_FLAG <= CALC_FLAG;

always @(posedge aclk)
    if (!aresetn)
        wrdata <= 'd0;
    else
        if (CALC_END)
            wrdata <= dout;
        else
            wrdata <= wrdata;
```

계산이 끝나면 wrdata, 즉 출력 값에 dout 의 값을 저장합니다.

```
reg valid_pre, valid_reg;
always @(posedge aclk)
    if (!aresetn)
        valid_pre <= 'd0;
    else
        if (CALC_EN || dvalid)
            valid_pre <= 'd1;
        else
            valid_pre <= 'd0;

always @(posedge aclk)
    if (!aresetn)
        valid_reg <= 'd0;
    else if (CALC_FLAG)
        valid_reg <= valid_pre;

assign valid = valid_reg;
assign ain = valid ? global_dout : 'd0;
```

CALC 상태에 진입한 직후 혹은 dvalid 가 활성화 된 상태면 valid_pre 를 활성화하고, 그 다음 cycle 에 valid 를 활성화하여 ain 에 global_dout 의 값을 저장합니다.

4. DONE

```
reg DONE_FLAG;
wire DONE_RESET = (!aresetn || DONE_END);
wire DONE_EN = (state == S_DONE) && (state_d == S_CALC);
always @(posedge aclk)
    if (DONE_RESET)
        DONE_FLAG <= 'd0;
    else
        if (DONE_EN)
            DONE_FLAG <= 'd1;
        else
            DONE_FLAG <= DONE_FLAG;

assign done = (state == S_DONE);
```

DONE 상태에 진입하면 done 을 활성화합니다.

5. CTR

```
reg [31:0] CNT;

wire CNT_LD = (LOAD_EN || CALC_EN || DONE_EN);
wire CNT_EN = (LOAD_FLAG || (CALC_FLAG && dvalid) || DONE_FLAG);
wire CNT_RESET = (!aresetn);

always @(posedge aclk)
    if (CNT_RESET)
        CNT <= 'd0;
    else
        if (CNT_LD)
            CNT <= 'd0;
        else if (CNT_EN)
            CNT <= CNT + 1;

assign addr = (LOAD_FLAG) ? rdaddr[L_RAM_SIZE-1:0] :
              (CALC_FLAG) ? CNT :
              'd0;
assign rdaddr = (LOAD_FLAG) ? CNT : 'd0;
assign LOAD_END = (LOAD_FLAG) && (CNT == 'd31);
assign CALC_END = (CALC_FLAG) && (CNT == 'd15) && dvalid;
assign DONE_END = (DONE_FLAG) && (CNT == 'd4);
```

LD 는 각 상태에 진입한 직후인지, EN 은 CNT 가 enable 한 상황인지, RESET 은 리셋이 들어왔는지 판단합니다. RESET 이거나 LD 이면 CNT 를 0 으로 초기화하고 LOAD 와 DONE 은 매 cycle 마다, CALC 는 매 연산 마다 CNT 를 1 증가시킵니다. addr 은 LOAD 일땐 rdaddr 값을, CALC 일땐 CNT 값을 저장해줍니다. rdaddr 은 LOAD 일 때 CNT 값을 저장해줍니다. LOAD 는 32 cycle 후에, CALC 는 16 번 연산 후에, DONE 은 5cycle 후에 각 상태를 종료합니다.

그렇게 해서 LOAD 에 32+1 = 33cycle, CALC 에 16*18+1 = 289cycle, DONE 에 5+1 = 6cycle, 총 328 cycle 을 소모합니다.