// Будет РК по Verilog лмао, в районе второй лабы правда так что пока не ссы
Продолжаем описывать конвейерный сумматор (до define wth(stage) - было в прошлый раз)

```
module pipelined_adder #(parameter w = 128, s = 4)(
    input clk, rstn, cin, valid_op1, valid_op2;
    input [w-1:0] op1, op2;
    output reg [w-1:0] res;
    output reg valid
);
localparam [s * 32 - 1:0] stage_width =
    {32'd34, 32'd32, 32'd32, 32'd30} // Размерность стадий, сумма 128
define wth(stage) stage_widths[32 * stage +:32]

function integer base;
    input integer stage;
    begin
        base = 0;
        for (stage = stage; stage > 0; stage - stage - 1) begin
            base = base + wth(stage + 1)
        end
    end
endfunction

define idx(stage) base(stage) +: wth(stage) // Конструкция, задающая итоговую "вертикальную" ширину
reg [w-1:0] stage_reg [0:s-1];
wire[w-1:0] stage_comb [0:s-1];
reg [w-1:0] stage_op1 [0:s-1];
reg [w-1:0] stage_op2 [0:s-1];
reg [s-1:0] valid_reg;
reg [s:0] c_reg;
wire [s:0] c_wire;
wire [s-1:0] f; // Бесполезная херня, нигде не используется, но добьёт длину до норм значений, что оптимизирует конструкцию г
integer i;
genvar k;
// В схеме по сути будет 2 ресета - один глобальный, "невидимый", происходящий при включении. Второй - с передаваемым нами зн

inintial begin
    for (i = 0; i < s; i = i + 1) begin
        stage_reg[i] <= {w{1'b0}};
        valid_reg[i] <= 1'b0;
        stage_op1[i] <= {w{1'b0}};
        stage_op2[i] <= {1{1'b0}};
        res <= {w{1'b0}};
    end
    c_reg <= {(s+1)}{1'b0}
end

always @(*) begin
    stage_op1[0] <= op1;
    stage_op2[0] <= op2;
    c_reg[0] <= cin;
end

generate
    for (k = 0; k < s; k = k + 1) begin
        assign {c_comb[k + 1], stage_comb[k][`idx(k)], f[k]} = {1'b0, stage_op1[k][`idx(k)], c_reg[k]} + {1'b0, stage_op2[k]
    end
    always @(posedge clk) begin
        if (~rstn)
            for (i = 0; i < s; i = i + 1)
                stage_reg[i][`idx(k)] <= {w{1'b0}}
        else begin
            stage_reg[0][`idx(k)] <= stage_comb[0][`idx(k)];
            for (i = 1; i < s; i = i + 1) begin
                if (i == k)
                    stage_reg[i][`idx(k)] <= stage_comb[i][`idx(k)];
                else
                    stage_reg[i][`idx(k)] <= stage_reg[i - 1][`idx(k)];
            end
        end
    end
endgenerate

always @(posedge clk) begin
    if (!rstn) begin
        valid <= 1'b0;
        res <= {w{1'b0}};
        valid_reg <= {s{1'b0}};
        for (i = 1; i < s; i = i + 1) begin
            stage_op1[i] <= {w{1'b0}};
            stage_op2[i] <= {w{1'b0}};
            c_reg[i] <= 1'b0;
        end
        c_reg[s] <= 1'b0;
    end else begin
        valid_reg[0] <= valid_op1 & valid_op2;
        for (i = 1; i < s; c = i + 1) begin
            valid_reg[i] <= valid_reg[i - 1];
            c_reg[i] <= c_comb[i];
            stage_op1[i] <= stage_op1[i - 1];
            stage_op2[i] <= stage_op2[i - 1];
        end
        res <= stage_res[s - 1];
        valid <= valid_reg[s - 1];
    end
end
endmodule
```

^ Вот эту пиздоту разгонять на лабе

Вторая лаба

Идея второй лабы - автомат УУ
Устройство управления МК релизовано не схемой а программой.

"Напоминает программирование 🤙"
© Попов 2025

Микрокод по сути представляет собой автоматы и огромное количество переходов в нужные адреса на каждом такте
На второй лабе будем как раз писать микрокод под микропрограммный автомат (который тоже сами напишем).
Сначала Hello World потом по regex'ам по вариантам