```
module top_tb();
   reg clk = 1'b0;

   wire [1:0] pwm;
   wire [11:0] vec;


   top dut(.clk(clk), .pwm(pwm), .vec(vec));

   always
     #5 clk = ~clk;

   initial begin
      $dumpfile("dump.vcd");
      $dumpvars();
      #10000000 $finish();
   end

endmodule

module top
   (input         clk,
    output [12:1] vec,
    output [2:1]  pwm);

   wire [4:1] pwm_tab [1:2];

   sequencer seq(.clk(clk), .vec(vec), .pwm(pwm_tab));
   pwm pwm_gen(.clk(clk), .pwm_in(pwm_tab), .pwm(pwm));

endmodule

module sequencer
  (input         clk,
   output [12:1] vec,
   output [4:1]  pwm [1:2]);

   parameter OP_WAIT = 1'b0;
   parameter OP_JUMP = 1'b1;

   reg fetch = 1'b1;

   wire opcode;
   wire [9:1] addr_or_cycles;
   wire [9:1] daddr;
   wire [2:1] UNUSED;
   wire [32:1] dout;
   wire [9:1]  next_addr = (opcode == OP_JUMP ? addr_or_cycles : (daddr + 1));
   rom ucode(.clk(clk), .en(fetch), .addr(next_addr), .daddr(daddr), .dout(dout));
   assign {pwm[2], pwm[1], vec, UNUSED, opcode, addr_or_cycles} = dout;

   wire cout;
   prescaler prs(.clk(clk), .cout(cout), .reset(fetch));
   wire finished;
   counter ctr(.clk(clk), .en(cout), .threshold(addr_or_cycles), .reset(fetch), .fin
ished(finished));

   always @(negedge clk) begin
     case (opcode)
       OP_WAIT : begin
          if (finished)
            fetch <= 1'b1;
          else if (fetch)
            fetch <= 1'b0;
       end
       OP_JUMP : begin
          fetch = ~fetch;
       end
     endcase
   end

endmodule
```

```
module rom
  #(parameter FILE = "memory.hex",
    parameter WIDTH = 32,
    parameter DEPTH = 9)
  (input                clk,
   input                en,
   input [DEPTH:1]      addr,
   output reg [DEPTH:1] daddr,
   output reg [WIDTH:1] dout);

   reg [WIDTH:1] mem [0:((1<<DEPTH)-1)];

   initial
     $readmemh(FILE, mem);

   always @(posedge clk) begin
      if (en) begin
         daddr <= addr;
         dout <= mem[addr];
      end

   end

endmodule

module counter
  #(parameter WIDTH = 9)
   (input           clk,
    input           en,
    input [WIDTH:1] threshold,
    input           reset,
    output reg      finished);

   reg [WIDTH:1] ctr;

   always @(posedge clk)
     if (reset) begin
       {ctr, finished} <= {{WIDTH{1'b0}}, 1'b0};
     end else if (en) begin
        if (ctr >= threshold)
          finished <= 1'b1;
        else
          ctr <= ctr + 1;
     end

endmodule

module prescaler
  #(parameter WIDTH = 10)
   (input  clk,
    input  reset,
    output reg cout);

   reg [WIDTH:1] ctr = 0;

   always @(posedge clk)
     if (!reset)
       {cout, ctr} <= ctr + 1;
     else
       {cout, ctr} <= {1'b0, {WIDTH{1'b0}}};

endmodule

module pwm
  #(parameter PWM_WIDTH = 4)
   (input        clk,
    input [PWM_WIDTH:1]  pwm_in [1:2],
    output [2:1] pwm);

   wire en;

   reg [PWM_WIDTH:1] cnt;
   prescaler #(.WIDTH(2)) div(.clk(clk), .reset(1'b0), .cout(en));

   always @(posedge clk) begin
      if (en)
        cnt <= cnt + 1;
   end

   wire [PWM_WIDTH:1] cnt_180 = cnt + {(PWM_WIDTH-1){1'b1}};

   assign pwm[1] = pwm_in[1] > cnt;
   assign pwm[2] = pwm_in[2] > cnt_180;
endmodule
```