Generator semnal PWM

Dumitrascu Andrei-Mihnea, Milu Catalin-Constantin, Madar Gabriel

14 decembrie 2025

1 Introducere

Scopul modulului este implementarea unui periferic configurabil prin SPI care genereaza un semnal PWM (Pulse Width Modulation) programabil din software. Procesorul (masterul SPI) poate citi si scrie registre interne ale perifericului pentru a configura perioada, factorul de umplere, directia de numarare si modul de aliniere al semnalului PWM.

Tema pune deja la dispozitie arhitectura generala a perifericului. In aceasta documentatie nu reiau pe larg enuntul, ci accentuez **deciziile de implementare** in Verilog pentru fiecare submodul: `top`, `spi_bridge`, `instr_dcd`, `regs`, `counter` si `pwm_gen`.

2 Arhitectura de ansamblu

La nivel de blocuri, structura respecta diagrama din enunt:

- `spi_bridge` lucreaza in doua domenii de ceas: `sclk` pentru operatiile de shiftare SPI si `clk` pentru sincronizarea cu restul perifericului.
- `instr_dcd` primeste octeti (`data_in`, `byte_sync`) si genereaza operatiile de citire/scriere in banca de registre.
- `regs` contine toti registrii de configurare (PERIOD, COMPARE1, COMPARE2, PRESCALE, UPNOTDOWN, PWM_EN, FUNCTIONS etc.) si expune semnalele necesare catre numarator si generatorul PWM.
- `counter` implementeaza un numarator cu prescaler, care numara intre 0 si `period`, in sus sau in jos, si genereaza `count_val`.
- `pwm_gen` compara `count_val` cu valorile `compare1`/`compare2` si cu modul de functionare din `functions`, pentru a produce `pwm_out`.

Modulul `top` instantiataza toate aceste blocuri si conecteaza interfata externa (ceas, reset, SPI, iesire PWM) la logica interna.

3 Modulul `top`

Modulul `top` este impus de scheletul de cod al temei si nu poate fi modificat la nivel de antet. El defineste interfata globala a perifericului si toate legaturile dintre submodule:

```
module top(
    input clk,
    input rst_n,
    input sclk,
    input cs_n,
    input mosi,
    output miso,
    output pwm_out
);
```

In interior, am declarat fire pentru:

- magistrala de registre: `read`, `write`, `addr`, `high_byte`, `data_read`, `data_write`;
- configuratia numaratorului: `period`, `en`, `count_reset`, `upnotdown`, `prescale`, `counter_val`;
- configuratia PWM: `pwm_en`, `functions`, `compare1`, `compare2`.

Submodulele sunt instantiatate explicit si conectate prin semnale interne, permitand inlocuirea usoara a oricarei componente fara a afecta restul perifericului.

4 Modulul `spi_bridge`

4.1 Arhitectura cu doua domenii de ceas

Implementarea `spi_bridge` foloseste o abordare cu doua domenii de ceas distincte:

- Domeniul `sclk`: pentru operatiile de shiftare RX/TX
- Domeniul `clk`: pentru sincronizarea cu restul perifericului

Aceasta separare este esentiala pentru respectarea protocolului SPI (CPOL=0, CPHA=0) si pentru evitarea problemelor de metastabilitate.

4.2 Domeniul SCLK - Receptie (rising edge)

Pe frontul crescator al `sclk` esantionam bitul de pe linia `mosi` si il shiftam in registrul de receptie:

```
always @(posedge sclk or negedge rst_n) begin
    if (!rst_n) begin
        bit_cnt <= 3'd0;
        rx_shift <= 8'd0;
        rx_data_sclk <= 8'd0;
        byte_done_sclk <= 1'b0;
    end else if (cs_n) begin
        bit_cnt <= 3'd0;
    end else begin
        rx_shift <= {rx_shift[6:0], mosi};

        if (bit_cnt == 3'd7) begin
            rx_data_sclk <= {rx_shift[6:0], mosi};
            byte_done_sclk <= ~byte_done_sclk;
            bit_cnt <= 3'd0;
        end else begin
            bit_cnt <= bit_cnt + 1'b1;
        end
    end
end
```

Observati ca folosim un mecanism de "toggle" pentru semnalul `byte_done_sclk` in loc de un puls. Acest lucru simplifica sincronizarea intre domeniile de ceas.

4.3 Domeniul SCLK - Transmisie (falling edge)

Pe frontul descrescator al `sclk` actualizam linia `miso` cu urmatorul bit de transmis:

```
always @(negedge sclk or negedge rst_n) begin
    if (!rst_n) begin
        tx_shift <= 8'd0;
        miso_reg <= 1'b0;
    end else if (cs_n) begin
        miso_reg <= 1'b0;
    end else begin
        if (bit_cnt == 3'd0) begin
            tx_shift <= data_out;
            miso_reg <= data_out[7];
        end else begin
            miso_reg <= tx_shift[7];
            tx_shift <= {tx_shift[6:0], 1'b0};
        end
    end
end
```

4.4 Sincronizarea cu domeniul CLK

Pentru a transfera datele din domeniul `sclk` in domeniul `clk`, folosim un sincronizator pe 3 registre si detectam tranzitia semnalului toggle:

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        byte_done_sync1 <= 1'b0;
        byte_done_sync2 <= 1'b0;
        byte_done_sync3 <= 1'b0;
        rx_data_latched <= 8'd0;
        byte_sync_reg <= 1'b0;
    end else begin
        byte_done_sync1 <= byte_done_sclk;
        byte_done_sync2 <= byte_done_sync1;
        byte_done_sync3 <= byte_done_sync2;

        if (byte_done_sync2 != byte_done_sync3) begin
            rx_data_latched <= rx_data_sclk;
        end

        byte_sync_reg <= (byte_done_sync2 != byte_done_sync3);
    end
end
```

Semnalul `byte_sync` este intarziat cu un ciclu pentru a garanta ca datele sunt stabile cand decodorul de instructiuni le citeste.

5 Modulul `instr_dcd`

5.1 FSM pe doi pasi: setup + data

`instr_dcd` implementeaza protocolul pe 2 octeti descris in enunt printr-un automat cu doua stari:

```
localparam STATE_SETUP = 1'b0;
localparam STATE_DATA  = 1'b1;

reg state;
```

Intrarea principala este `byte_sync`, care marcheaza faptul ca un nou octet `data_in` este disponibil din interfata SPI.

5.2 Decodarea comenzii

In faza de setup, octetul primit are formatul:

`[7] = R/W, [6] = High/Low, [5:0] = adresa`

Campurile sunt extrase si memorate pentru utilizare in faza de date:

```
STATE_SETUP: begin
    is_write <= data_in[7];
    high_byte_reg <= data_in[6];
    reg_addr <= data_in[5:0];

    if (!data_in[7]) begin
        read_reg <= 1'b1;
    end
end
```

Observati ca pentru operatiile de citire, semnalul `read` este generat imediat in faza de setup, astfel incat `data_read` sa fie disponibil la timp pentru transmisia pe MISO.

5.3 Generarea semnalelor read/write

Semnalele `read` si `write` sunt pulsuri de un singur ciclu de `clk`, resetate la 0 in fiecare ciclu pentru a evita scrierile multiple:

```
always @(posedge clk or negedge rst_n) begin
    read_reg <= 1'b0;
    write_reg <= 1'b0;

    if (byte_sync) begin
        // ... generare pulsuri
    end
end
```

6 Modulul `regs`

6.1 Maparea adreselor pe registrii interni

Modulul `regs` implementeaza banca de registre conform tabelului din enunt. Fiecare registru are un bloc `always` dedicat pentru scriere:

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        period_reg <= 16'h0000;
    end else if (write && addr == ADDR_PERIOD) begin
        if (high_byte)
            period_reg[15:8] <= data_write;
        else
            period_reg[7:0] <= data_write;
    end
end
```

Aceasta abordare permite scrieri pe 8 biti pentru registrii de 16 biti, folosind bitul `high_byte` pentru a selecta partea MSB sau LSB.

6.2 Auto-clear pentru `COUNTER_RESET`

Conform enuntului, scrierea in registrul `COUNTER_RESET` trebuie sa produca un impuls de reset pentru numarator, apoi bitul sa revina singur la 0 dupa 2 cicluri de ceas. Am implementat aceasta cerinta folosind un contor intern pe 2 biti:

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        counter_reset_reg <= 1'b0;
        counter_reset_cnt <= 2'd0;
    end else begin
        if (write && addr == ADDR_COUNTER_RESET && data_write[0]) begin
            counter_reset_reg <= 1'b1;
            counter_reset_cnt <= 2'd2;
        end else if (counter_reset_cnt > 0) begin
            counter_reset_cnt <= counter_reset_cnt - 1'b1;
            if (counter_reset_cnt == 1) begin
                counter_reset_reg <= 1'b0;
            end
        end
    end
end
```

6.3 Multiplexorul de citire

Citirea registrilor este implementata combinational:

```
always @(*) begin
    case (addr)
        ADDR_PERIOD: begin
            if (high_byte)
                data_read_reg = period_reg[15:8];
            else
                data_read_reg = period_reg[7:0];
        end
        ADDR_COUNTER_EN:    data_read_reg = {7'b0, counter_en_reg};
        ADDR_COUNTER_VAL: begin
            if (high_byte)
                data_read_reg = counter_val[15:8];
            else
                data_read_reg = counter_val[7:0];
        end
        // ... alte registre
        default:            data_read_reg = 8'h00;
    endcase
end
```

7 Modulul `counter`

7.1 Prescalerul

Numaratorul foloseste un mecanism de prescalare bazat pe putere a lui 2. Pragul de prescalare este calculat astfel:

```
wire [7:0] prescale_threshold = (1 << prescale) - 1;
wire prescale_tick = (prescale_cnt == prescale_threshold);
```

Pentru `prescale = 0`, thresholdul este 0, deci numaratorul principal incrementeaza la fiecare ciclu de ceas. Pentru `prescale = n`, incrementeaza o data la fiecare 2^n cicluri.

7.2 Numararea sus/jos

In blocul secvential principal, cand prescalerul a ajuns la limita, actualizam numaratorul principal in functie de directia de numarare:

```
if (prescale_tick) begin
    prescale_cnt <= 8'd0;

    if (upnotdown) begin
        if (counter_reg >= period) begin
            counter_reg <= 16'd0;
        end else begin
            counter_reg <= counter_reg + 1'b1;
        end
    end else begin
        if (counter_reg == 16'd0) begin
            counter_reg <= period;
        end else begin
            counter_reg <= counter_reg - 1'b1;
        end
    end
end
```

Aceasta implementare asigura ca numaratorul cicleaza intre 0 si `period` inclusiv, indiferent de directie.

8 Modulul `pwm_gen`

8.1 Decodarea modurilor din `functions`

Modulul `pwm_gen` genereaza `pwm_out` pe baza modului selectat in `functions[1:0]`:

```
wire align_right = functions[0];
wire unaligned   = functions[1];
```

- `functions = 2'b00`: aliniat la stanga
- `functions = 2'b01`: aliniat la dreapta
- `functions = 2'b10`: nealiniat (intre compare1 si compare2)

8.2 Logica PWM pentru fiecare mod

Iesirea este implementata combinational cu prioritatea corecta a modurilor:

```
always @(*) begin
    if (!pwm_en) begin
        pwm_out_comb = 1'b0;
    end else if (unaligned) begin
        if (compare1 >= compare2) begin
            pwm_out_comb = 1'b0;
        end else if (count_val >= compare1 && count_val < compare2)
           begin
            pwm_out_comb = 1'b1;
        end else begin
            pwm_out_comb = 1'b0;
        end
    end else if (align_right) begin
        if (count_val >= compare1) begin
            pwm_out_comb = 1'b1;
        end else begin
            pwm_out_comb = 1'b0;
        end
    end else begin
        if (compare1 == 16'd0) begin
            pwm_out_comb = 1'b0;
        end else if (count_val <= compare1) begin
            pwm_out_comb = 1'b1;
        end else begin
            pwm_out_comb = 1'b0;
        end
    end
end
```

Observati tratarea cazului special pentru `compare1 == 0` in modul aliniat la stanga, care produce un duty cycle de 0%.

9 Concluzii

Implementarea propusa respecta arhitectura perifericului descrisa in tema si pune accent pe o separare clara intre:

- interfata de comunicatie cu doua domenii de ceas (modulul `spi_bridge`);
- decodarea protocolului pe 2 faze (modulul `instr_dcd`);
- banca de registre cu auto-clear pentru `COUNTER_RESET` (modulul `regs`);
- numaratorul parametrizabil cu prescaler si mod up/down (modulul `counter`);
- generatorul PWM cu moduri aliniat stanga/dreapta si nealiniat (modulul `pwm_gen`).

Toate cele 5 teste din testbench-ul furnizat trec cu succes, validand corectitudinea implementarii pentru toate modurile de functionare ale PWM-ului.