Generator semnal PWM

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

23 noiembrie 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 domeniul de ceas `clk`, sincronizeaza semnalele `sclk` si `cs_n` si implementeaza un registru de shiftare pentru liniile `miso`/`mosi`.
- `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(
    // peripheral clock signals
    input clk,
    input rst_n,
    // SPI master facing signals
    input sclk,
    input cs_n,
    input miso,
    output mosi,
    // peripheral signals
    output pwm_out
);
```

In interior, am declarat fire pentru:

- magistrala de registre: `read`, `write`, `addr`, `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:

- `spi_bridge` primeste direct semnalele SPI si genereaza `mosi`.
- `instr_dcd` preia octeti logici (`data_in`) si semnal de sincronizare (`byte_sync`) si genereaza accesul la registre.
- `regs` mapeaza adresele pe registrii interni si traduce operatiile de pe bus in semnale catre numarator si PWM.
- `counter` realizeaza numararea propriu-zisa cu prescaler.
- `pwm_gen` produce semnalul `pwm_out`.

Structurarea in module separate permite inlocuirea usoara a unei componente (de exemplu numaratorul) fara a afecta restul perifericului, atata timp cat interfata ramane aceeasi.

4 Modulul `spi_bridge`

4.1 Sincronizarea semnalelor SPI

`spi_bridge` lucreaza in domeniul de ceas `clk` si trebuie sa sincronizeze semnalele asincrone `sclk` si `cs_n`. Am folosit doua registre pe 2 biti:

```
reg [1:0] sclk_sync;
reg [1:0] cs_sync;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        sclk_sync <= 2'b00;
        cs_sync <= 2'b11;
    end else begin
        sclk_sync <= {sclk_sync[0], sclk};
        cs_sync <= {cs_sync[0], cs_n};
    end
end
```

Pe baza acestor registre definesc fronturile relevante ale lui SCLK si starea lui CS:

```
wire sclk_rise = (sclk_sync[1] == 1'b0) && (sclk_sync[0] == 1'b1);
wire sclk_fall = (sclk_sync[1] == 1'b1) && (sclk_sync[0] == 1'b0);
wire cs_active = (cs_sync[0] == 1'b0);
```

Aceasta abordare elimina metastabilitatea si permite detectarea fronturilor in mod robust in raport cu `clk`.

4.2 Registrul de shiftare

Pentru date folosesc un registru de shiftare de 8 biti si un registru pentru iesire:

```
reg [7:0] shift_reg;
reg mosi_reg;

assign mosi = mosi_reg;
```

Logica principala:

- cand `rst_n` este activat, se golesc registrele;
- cand `cs_active` este 0 (CS inactiv), resetez `shift_reg` si `mosi_reg`;
- pe front crescator de `sclk` (detectat in domeniul `clk`) eșantionez `miso` si shiftez in registru;
- pe front descrescator actualizez `mosi_reg` cu bitul cel mai semnificativ al registrului.

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        shift_reg <= 8'h00;
        mosi_reg <= 1'b0;
    end else begin
        if (!cs_active) begin
            shift_reg <= 8'h00;
            mosi_reg <= 1'b0;
        end else begin
            if (sclk_rise) begin
                shift_reg <= {shift_reg[6:0], miso};
            end
            if (sclk_fall) begin
                mosi_reg <= shift_reg[7];
            end
        end
    end
end
```

In aceasta versiune `spi_bridge` realizeaza sincronizarea si shiftarea bitilor, lasand interpretarea octetilor si generarea semnalelor de control catre `instr_dcd`.

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 ST_SETUP = 1'b0;
localparam ST_DATA = 1'b1;

reg state;
```

Intrarea principala este `byte_sync`, care marcheaza faptul ca un nou octet `data_in` este disponibil din interfata SPI. In starea `ST_SETUP` interpretez octetul ca pe un header, iar in starea `ST_DATA` ca pe octetul de date.

5.2 Identificarea registrelor pe 16 biti

Pentru ca o parte din registii perifericului sunt pe 16 biti, am introdus o functie care decide daca adresa de baza corespunde unui astfel de registru:

```
function is_16b_reg;
    input [5:0] a;
    begin
        is_16b_reg = (a == 6'h00) || (a == 6'h03) ||
                     (a == 6'h05) || (a == 6'h08);
    end
endfunction
```

In faza de setup, octetul primit are formatul:

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

Daca este un registru pe 16 biti, adresa efectiva se obtine din:

- adresa de baza + 0 pentru partea low (LSB);
- adresa de baza + 1 pentru partea high (MSB).

5.3 Generarea semnalelor read/write

In logica secventiala, la fiecare `byte_sync`:

- in starea `ST_SETUP`:
  - memorez bitul 7 in `curr_rw` (1 = write, 0 = read);
  - calculez adresa efectiva `addr_reg`;
  - daca este comanda de citire, ridic `read` pentru a citi din banca de registre in urmatorul ciclu;
  - trec in starea `ST_DATA`.
- in starea `ST_DATA`:
  - pentru write: copiez `data_in` in `data_write`, ridic `write` un ciclu si nu trimit nimic util pe `data_out`;
  - pentru read: copiez valoarea `data_read` (pregatita in faza de setup) in `data_out`, care va fi transmisa prin SPI.

Semnalele `read` si `write` sunt pulsuri de un singur ciclu de `clk`, initializate implicit la 0 la fiecare ciclu pentru a evita scrierile multiple.

6 Modulul `regs`

6.1 Maparea adreselor pe registrii interni

Modulul `regs` implementeaza banca de registre conform tabelului din enunt. Am folosit un singur bloc `always @(posedge clk or negedge rst_n)` in care tratez atat scrierile, cat si citirile.

La scriere, in functie de `addr`, actualizez registrii:

- PERIOD: `0x00` (LSB), `0x01` (MSB);
- COUNTER_EN: `0x02`, folosesc doar bitul 0;
- COMPARE1: `0x03` (LSB), `0x04` (MSB);
- COMPARE2: `0x05` (LSB), `0x06` (MSB);
- COUNTER_RESET: `0x07`, write-only, bitul 0;
- COUNTER_VAL: `0x08`/`0x09`, read-only (ignor orice write);
- PRESCALE: `0x0A`;
- UPNOTDOWN: `0x0B`, bitul 0;
- PWM_EN: `0x0C`, bitul 0;

- FUNCTIONS: `0x0D`, folosesc doar bits [1:0].

Fragment ilustrativ pentru scriere:

```
if (write) begin
    case (addr)
        6'h00: period[7:0] <= data_write;
        6'h01: period[15:8] <= data_write;
        6'h02: en <= data_write[0];
        6'h03: compare1[7:0] <= data_write;
        6'h04: compare1[15:8]<= data_write;
        // ...
        6'h0D: functions[1:0]<= data_write[1:0];
        default: ;
    endcase
end
```

La citire, semnalul `data_read` este multiplexat in functie de adresa:

```
if (read) begin
    case (addr)
        6'h00: data_read <= period[7:0];
        6'h01: data_read <= period[15:8];
        6'h02: data_read <= {7'h00, en};
        // ...
        6'h08: data_read <= counter_val[7:0];
        6'h09: data_read <= counter_val[15:8];
        // ...
        default: data_read <= 8'h00;
    endcase
end
```

Pentru registrele write-only (de exemplu COUNTER_RESET) intorc 0 la citire.

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 `reset_cnt` pe 2 biti:

- cand se scrie un 1 la adresa `0x07`, setez `count_reset <= 1` si `reset_cnt <= 2`;
- la fiecare ciclu in care `reset_cnt` este diferit de 0 il decrementez;
- cand ajunge la 1, in urmatorul ciclu pun `count_reset <= 0`.

Astfel, numaratorul vede un impuls de reset de lungime fixa, fara ca software-ul sa fie nevoit sa scrie explicit 0 inapoi.

7 Modulul `counter`

7.1 Valoarea contorului si prescalerul

Numaratorul pastreaza starea curenta in registrul `count_val_reg` (expus la exterior ca `count_val`) si foloseste un contor intern de prescalare `presc_cnt`:

```
reg [15:0] count_val_reg;
reg [15:0] presc_cnt;

assign count_val = count_val_reg;
```

Prescalerul este interpretat conform specificatiei din tema: `PRESCALE = n` inseamna ca numaratorul intern se incrementeaza la fiecare `2^n` cicluri de `clk`. Am implementat acest lucru calculand o limita:

```
wire [15:0] prescale_limit =
    (prescale[3:0] == 4'd0) ? 16'd0 :
    ((16'd1 << prescale[3:0]) - 16'd1);
```

Atunci cand `presc_cnt` ajunge la `prescale_limit`, il resetez si fac un "tick" al numaratorului principal. In rest, doar incrementez `presc_cnt`.

7.2 Numararea sus/jos si tratarea capetelor

In blocul secvential principal tratez, in ordinea prioritatilor:

1. reset-ul global `rst_n`;
2. semnalul `count_reset` (rescrie imediat contorul la 0 si reseteaza prescalerul);
3. activarea `en` (daca numaratorul este oprit, valorile raman neschimbate).

Cand prescalerul a ajuns la limita:

- daca `upnotdown = 1`, numar in sus:
  - la `count_val_reg >= period` revin la 0 (overflow explicit);
  - altfel incrementez cu 1.
- daca `upnotdown = 0`, numar in jos:
  - la 0 sar la `period` (underflow explicit);
  - altfel decrementez cu 1.

Aceasta implementare respecta cerinta de numarare intre 0 si `period` si permite atat moduri de numarare in sus, cat si in jos.

8 Modulul `pwm_gen`

8.1 Decodarea modurilor din `functions`

Modulul `pwm_gen` genereaza `pwm_out` pe baza valorilor `compare1`, `compare2`, `period`, `count_val` si a modului selectat in `functions[1:0]`:

- `functions[1] = 0`: mod **aliniat**;
- `functions[1] = 1`: mod **nealiniat**;
- `functions[0]` este folosit doar in mod aliniat:
  - 0 = aliniat la stanga;
  - 1 = aliniat la dreapta.

In cod am extras doua semnale auxiliare:

```
wire align_mode = (functions[1] == 1'b0);
wire right_align = (functions[0] == 1'b1);
```

8.2 Logica PWM pentru fiecare mod

Iesirea este implementata intr-un singur bloc secvential sincron pe `clk`:

```
reg pwm_reg;
assign pwm_out = pwm_reg;
```

Comportamentul este:

- daca `rst_n` este 0, `pwm_reg` este resetat la 0;
- daca `pwm_en` este 0, semnalul ramane in starea curenta (nu este fortat la 0 sau 1, ci "inghetata" la ultima valoare);
- daca `pwm_en` este 1:
  - in mod aliniat la stanga:

    $$pwm\_reg = 1 \text{ daca } count\_val < compare1,\ 0 \text{ altfel;}$$

    adica semnalul este 1 de la inceputul perioadei pana la valoarea `compare1`.
  - in mod aliniat la dreapta:

    $$pwm\_reg = 1 \text{ daca } count\_val \geq period - compare1$$

    adica semnalul este 1 spre finalul perioadei, pe o durata egala cu `compare1`.
  - in mod nealiniat:

    $$pwm\_reg = 1 \text{ daca } compare1 \leq count\_val < compare2$$

    si 0 in rest. Aceasta corespunde unui "puls" plasat intre cele doua valori de comparare (presupun `compare1 < compare2`, conform temei).

Am ales o implementare simpla, pur combinationala in raport cu `count_val` si registrata doar la frontul de ceas, ceea ce evita glitch-uri la iesire.

9 Concluzii

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

- interfata de comunicatie (modulul `spi_bridge`);
- decodarea protocolului (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/nealiniat si aliniere stanga/dreapta (modulul `pwm_gen`).