Generator semnal PWM
Documentatie

Cioroiu Silvia, Grasu Costin-Alexandru, Tancu Miruna

November 2025

Cuprins

1 Bridge de comunicatie

1.1 Comportamentul protocolului SPI

Protocolul SPI este un protocol de comunicație sincronă, in care masterul controlează întregul schimb de date. În modul CPOL=0, CPHA=0, comportamentul semnalelor este următorul:

- Masterul transmite bitul pe MOSI pe **frontul descrescător**
- Slave-ul citește bitul de pe MOSI pe **frontul crescător**, moment în care datele sunt valide.
- Slave-ul actualizează ieșirea MISO tot pe **frontul descrescător**, pentru ca masterul să o poată citi pe frontul crescător.
- Comunicarea rămâne activă doar cât timp `CS_N` este **LOW** adica 0.
- Transferul se face MSB-first

1.2 Logica utilizată în proiect şi justificare

În implementarea noastră, semnalul `SCLK` nu provine dintr-un domeniu de clock extern, ci este generat în interiorul FPGA-ului şi este sincron cu `CLK`. Acest lucru modifică modul în care tratăm semnalele şi simplifică arhitectura.

1.2.1 1. Fără necesitatea sincronizării (CDC)

Într-un design tipic SPI Slave, `SCLK`, `MOSI` şi `CS_N` sunt semnale asincrone şi trebuiesc sincronizate.

În cazul nostru, `SCLK` este deja generat sincron cu `clk`, astfel că:

- nu sunt necesare registre de sincronizare,
- logica poate utiliza direct fronturile lui `SCLK` în interiorul aceluiaşi domeniu de clock.

Consecinţă: design-ul devine mai simplu şi mai eficient, deoarece `SCLK` este tratat ca un semnal intern perfect aliniat cu `clk`.

1.2.2 2. Detectarea fronturilor lui SCLK

În implementarea noastră nu a fost nevoie să memorăm valoarea anterioară a lui `SCLK`, deoarece folosim direct sensibilitatea la front a blocurilor `always`. Astfel:

- blocul `always @(posedge sclk)` se execută automat la fiecare front crescător,
- blocul `always @(negedge sclk)` se execută la fiecare front descrescător.

Acest lucru este posibil deoarece `SCLK` este un semnal sincron, deci putem folosi direct posedge și negedge.

Motivaţie: Pentru a respecta protocolul:

- citirea bitului MOSI trebuie făcută la frontul crescător,
- scrierea bitului MISO trebuie făcută la frontul descrescător.

1.2.3 3. Registru separat pentru recepție și transmisie

Design-ul folosește două registre:

- un registru `r_slave` în care se acumulează bit cu bit ceea ce primește modulul pe MOSI,
- un registru `r_master` care conține byte-ul ce urmează a fi transmis pe MISO.

Motivaţie: Separarea registre-lor evită conflictele:

- în timp ce `r_slave` acumulează un byte,
- `r_master` poate transmite simultan altul.

1.2.4 4. Contorul de biți și generarea unui byte complet

Pentru fiecare front crescător al lui `SCLK`, contorul crește cu 1. La fiecare 8 biți:

- byte-ul complet este transferat către logica internă,
- se generează un puls `byte_sync`,
- contorul revine la 0.

Motivaţie: Logica internă primește o notificare exact când un byte este valid.

1.2.5 5. Controlul prin semnalul `CS_N`

Când `CS_N` devine HIGH(1):

- transferul este oprit,
- contorul se resetează,
- registrul `r_master` este reîncărcat,
- MISO revine la bitul MSB al următorului byte ce urmează să fie transmis.

1.2.6 6. Avantajele arhitecturii folosite

- Nu necesită sincronizare , deoarece `SCLK` și `CLK` sunt din același domeniu.
- Detectarea fronturilor devine foarte precisă, oferind o implementare robustă a protocolului SPI.
- Design simplu.

1.3 Implementarea în cod

Implementarea modulelor respectă structura logică descrisă anterior. Pe scurt, funcţionarea poate fi rezumată astfel:

- Semnalele `SCLK`, `MOSI` şi `CS_N` sunt sincronizate pe clock-ul intern al FPGA-ului.
- Pe frontul crescător al lui SCLK, bitul de pe MOSI este introdus în registrul de recepţie, iar contorul este incrementat.
- După recepţia a 8 biţi, registrul `r_slave` este transferat către ieşirea `data_in`, iar `byte_sync` este activat pentru un singur ciclu de clock.
- Pe frontul descrescător al lui SCLK, se actualizează bitul de ieşire MISO, folosind registrul de transmisie master.
- La începutul fiecărui nou byte (când contorul revine la zero), registrul de transmisie este încărcat cu valoarea `data_out`.
- Când `CS_N` devine HIGH, transferul se încheie, contorul este resetat, iar MISO revine la valoarea iniţială.

2 Decodorul de instrucțiuni

2.1 Descriere Generală

Modulul `instr_dcd` (Instruction Decoder) acționează ca un translator între bridge-ul de comunicație SPI și blocul de regiștri. Acesta primește secvențe de octeți de la SPI și le interpretează ca comenzi de citire sau scriere în registrele perifericului, generând semnalele de control necesare pentru accesarea acestora.

Protocolul de comunicare este structurat pe două faze distincte: o fază de *setup* (configurare) urmată de o fază de *date* (execuție), fiecare constând dintr-un singur octet.

2.2 Interfața Modulului

2.2.1 Semnale de sistem

Nume Port	Direcție	Descriere
`clk`	Input	Ceasul sistemului pentru sincronizare.
`rst_n`	Input	Reset asincron (Active Low). Resetează toate registrele interne și ieșirile.

Table 1: Semnale de sistem - Decoder

2.2.2 Interfața către SPI Bridge

Nume Port	Direcție	Descriere
`byte_sync`	Input	Puls de sincronizare generat de bridge la finalul recepției unui octet complet.
`data_in`	Input	Octetul receptionat de la master prin SPI.
`data_out`	Output	Octetul care va fi transmis înapoi către master la operații de citire.

Table 2: Semnale SPI - Decoder

2.2.3 Interfața către Blocul de Regiștri

Nume Port	Direcție	Descriere
`read`	Output	Semnal de control pentru operație de citire din regiștri.
`write`	Output	Semnal de control pentru operație de scriere în regiștri.
`addr`	Output	Adresa efectivă de 6 biți a registrului accesat (după procesarea bitului High/Low).
`data_write`	Output	Octetul care va fi scris în registru la operații de scriere.
`data_read`	Input	Octetul citit din registru, care va fi transmis înapoi către master.
`high_low`	Output	Semnal auxiliar (nefolosit în implementarea finală).

Table 3: Semnale către Regiștri - Decoder

2.3 Protocolul de Comunicare

Fiecare tranzacție de comunicare constă din exact două octeți transmiși secvențial de către master:

2.3.1 Faza de Setup (Primul Octet)

Primul octet conține informațiile necesare pentru decodificarea comenzii:

Bit	Nume	Semnificație
7	Read/Write	`1` = Operație de scriere (WRITE) `0` = Operație de citire (READ)
6	High/Low	`1` = Octetul superior [15:8] al registrului `0` = Octetul inferior [7:0] al registrului
5:0	Address	Adresa de bază a registrului (6 biți = 64 adrese posibile)

Table 4: Structura octetului de Setup

Exemplu: Octetul `0x83` = `1000_0011` înseamnă:

- Bit 7 = 1 → Operație de WRITE
- Bit 6 = 0 → Parte LOW a registrului
- Biți 5:0 = 3 → Adresa 0x03 (registrul COMPARE1)

2.3.2 Faza de Date (Al Doilea Octet)

După decodificarea setup-ului, al doilea octet reprezintă:

- Pentru **WRITE**: valoarea care va fi scrisă în registru.
- Pentru **READ**: acest octet este ignorat, iar răspunsul (valoarea citită) este transmis pe `data_out`.

2.4 Descrierea Funcționării Detaliate

Modulul implementează o mașină de stări cu două stări principale, controlată de registrul intern `faza`:

2.4.1 Starea 0: Faza de Setup

Când `byte_sync` devine activ și `faza = 0`:

1. Se extrag și se memorează bitii din `data_in`:
   - `rw_bit` ← `data_in[7]` (tip operație)
   - `hl_bit` ← `data_in[6]` (parte registru)
   - `saved_addr` ← `data_in[5:0]` (adresa de bază)
2. Se trece în `faza = 1` (faza de date).
3. Semnalele de control `read` și `write` sunt dezactivate.

2.4.2 Starea 1: Faza de Date

Când `byte_sync` devine activ și `faza = 1`:

1. **Calculul adresei efective:**
   - Dacă `hl_bit = 0`: `addr` ← `saved_addr` (octet LOW)
   - Dacă `hl_bit = 1`: `addr` ← `saved_addr + 1` (octet HIGH)

   Acest mecanism permite accesarea registrelor de 16 biți în două tranzacții de câte 8 biți.
2. **Execuția operației:**
   - Dacă `rw_bit = 1` (WRITE):
     - Se activează `write = 1`
     - Se transmite `data_write` ← `data_in`
   - Dacă `rw_bit = 0` (READ):
     - Se activează `read = 1`
     - Se capturează `data_out` ← `data_read`
3. Se revine la `faza = 0` pentru următoarea comandă.

2.4.3 Gestionarea Semnalelor de Control

Pentru a preveni executarea întâmplătoare a comenzilor, semnalele `read` și `write` sunt activate doar în ciclul de ceas în care `byte_sync` este activ și se execută faza de date. În rest, acestea sunt menținute la 0.

2.5 Exemplu de Funcționare

Scenariu: Scrierea valorii `0x1234` în registrul COMPARE1 (adresa 0x03, 16 biți).

Tranzacția 1 - Octet LOW:

1. Master trimite `0x83` (WRITE, LOW, addr=0x03)
2. Decoder: `faza=0`, salvează `rw_bit=1`, `hl_bit=0`, `saved_addr=0x03`
3. Master trimite `0x34`
4. Decoder: `faza=1`, calculează `addr=0x03`, activează `write=1`, `data_write=0x34`
5. Registrul COMPARE1[7:0] ← `0x34`

Tranzacția 2 - Octet HIGH:

1. Master trimite `0xC3` (WRITE, HIGH, addr=0x03)
2. Decoder: `faza=0`, salvează `rw_bit=1`, `hl_bit=1`, `saved_addr=0x03`
3. Master trimite `0x12`
4. Decoder: `faza=1`, calculează `addr=0x04`, activează `write=1`, `data_write=0x12`
5. Registrul COMPARE1[15:8] ← `0x12`

Rezultat final: COMPARE1 = `0x1234`

3 Blocul de Regiștrii

3.1 Descriere Generală

Modulul `regs` implementează spațiul de memorie al perifericului, oferind o interfață standardizată pentru stocarea și accesarea parametrilor de configurare. Acest bloc conține registre de diferite dimensiuni (1 bit, 8 biți, 16 biți) și tipuri de acces (R/W, R, W), fiecare având o funcție specifică în controlul numărătorului și generatorului PWM.

Arhitectura este optimizată pentru adresare pe octeți, permitând accesarea registrelor largi (16 biți) în două tranzacții consecutive de câte 8 biți.

3.2 Interfața Modulului

3.2.1 Semnale de sistem

Nume Port	Direcție	Descriere
`clk`	Input	Ceasul sistemului. Toate actualizările registrelor sunt sincrone.
`rst_n`	Input	Reset asincron (Active Low). Inițializează toate registrele la valori implicite.

Table 5: Semnale de sistem - Registri

3.2.2 Interfața către Decoder

Nume Port	Direcție	Descriere
`read`	Input	Semnal de comandă pentru citire. Când este activ, valoarea registrului specificat este plasată pe `data_read`.
`write`	Input	Semnal de comandă pentru scriere. Când este activ, valoarea de pe `data_write` este stocată în registrul specificat.
`addr`	Input	Adresa registrului accesat (6 biți = 64 locații posibile).
`data_write`	Input	Octetul care va fi scris în registru.
`data_read`	Output	Octetul citit din registru.

Table 6: Semnale de comunicare cu Decoder

3.2.3 Interfața către Counter și PWM

Nume Port	Direcție	Descriere
`counter_val`	Input	Valoarea curentă a numărătorului (pentru registrul read-only COUNTER_VAL).
`period`	Output	Perioada semnalului PWM (16 biți).
`en`	Output	Semnal de activare a numărătorului.
`count_reset`	Output	Semnal de resetare sincronă a numărătorului.
`upnotdown`	Output	Direcția de numărare (1=crescător, 0=descrescător).
`prescale`	Output	Factor de divizare a frecvenței (8 biți).
`pwm_en`	Output	Activare generare PWM.
`functions`	Output	Configurare mod PWM (aliniere, tip).
`compare1`	Output	Prag de comparare primar (16 biți).
`compare2`	Output	Prag de comparare secundar (16 biți).

Table 7: Semnale de control către module funcționale

3.3 Harta Registrelor

Modulul implementează următoarea hartă de adrese:

Nume	Adresă	Acces	Lățime	Descriere
PERIOD	0x00-01	R/W	[15:0]	Perioada numărătorului în ciclii de ceas.
COUNTER_EN	0x02	R/W	1	Activare numărător (1=activ, 0=oprit).
COMPARE1	0x03-04	R/W	[15:0]	Valoare de comparare pentru schimbarea stării PWM.
COMPARE2	0x05-06	R/W	[15:0]	Valoare de comparare secundară (mod nealiniat).
COUNTER_RESET	0x07	W	1	Reset numărător (auto-clear după 1 ciclu).
COUNTER_VAL	0x08-09	R	[15:0]	Valoarea curentă a numărătorului (read-only).
PRESCALE	0x0A	R/W	[7:0]	Exponent prescaler ($2^{prescale}$).
UPNOTDOWN	0x0B	R/W	1	Direcție numărare (1=up, 0=down).
PWM_EN	0x0C	R/W	1	Activare ieșire PWM.
FUNCTIONS	0x0D	R/W	[1:0]	Bit 0: aliniere (0=stânga, 1=dreapta) Bit 1: mod (0=aliniat, 1=nealiniat).

Table 8: Harta completă a registrelor

3.4 Descrierea Funcționării Detaliate

3.4.1 Logica de Scriere (Write Logic)

Scrierea în regiștri este implementată într-un bloc secvențial sincron cu ceasul. Când semnalul `write` este activ, un bloc `case` decodifică adresa și actualizează registrul corespunzător:

- **Registre de 16 biți:** Sunt mapate pe două adrese consecutive. De exemplu, PERIOD:
  - `0x00` → `period_reg[7:0]` (octet LOW)
  - `0x01` → `period_reg[15:8]` (octet HIGH)
- **Registre de 1 bit:** Doar bitul 0 al octetului scris este stocat. De exemplu:

```
COUNTER_EN: counter_en_reg <= data_write[0];
```

- **Registre write-only:** COUNTER_RESET nu poate fi citit; orice tentativă de citire returnează 0.

Mecanism de Auto-Clear:
Registrul `COUNTER_RESET` implementează un comportament special specificat în documentație: după ce este setat la 1, se golește automat la ciclul următor de ceas. Această funcționalitate este implementată printr-o verificare continuă în blocul secvențial:

```
if (counter_reset_reg == 1'b1) begin
    counter_reset_reg <= 1'b0;
end
```

Astfel, numărătorul primește un puls de reset de exact un ciclu de ceas.

3.4.2 Logica de Citire (Read Logic)

Citirea este implementată într-un bloc combinațional (`always @(*)`) pentru a oferi un răspuns instant:

- Când `read = 1`, un bloc `case` selectează registrul specificat de `addr` și plasează valoarea sa pe `data_read`.
- **Padding pentru registre mici:** Registrele de 1 bit sunt extinse la 8 biți prin adăugarea de zero-uri:

```
COUNTER_EN: data_read = {7'b0, counter_en_reg};
```

- **Registru read-only:** COUNTER_VAL nu este stocat în modulul de regiștri, ci este citit direct din semnalul de intrare `counter_val`:

```
0x08: data_read = counter_val[7:0];   // LOW byte
0x09: data_read = counter_val[15:8];  // HIGH byte
```

Aceasta permite citirea în timp real a valorii numărătorului fără sincronizare suplimentară.

- **Adrese nedefinite:** Orice acces la o adresă neimplementată returnează `0x00`.

3.4.3 Ieșiri Continue către Module Funcționale

Valorile registrelor sunt expuse permanent către modulele Counter și PWM prin atribuiri continue (`assign`):

```
assign period = period_reg;
assign en = counter_en_reg;
assign compare1 = compare1_reg;
// ... etc
```

Acest mecanism permite modulelor consumatoare să observe instantaneu orice modificare a configurației, fără a fi nevoie de semnale de sincronizare suplimentare. Efectul este similar cu un "tablou de bord" permanent actualizat.

3.5 Considerații de Design

3.5.1 Sincronizare și Timing

- Toate registrele sunt actualizate sincron cu `clk`, eliminând riscurile de metastabilitate.
- Logica de citire este combinațională pentru a minimiza latența de acces.
- Ieșirile `assign` propagă modificările în același ciclu de ceas în care registrul este actualizat.

3.5.2 Gestionarea Resetului

La activarea semnalului `rst_n`, toate registrele sunt inițializate la valori sigure:

- Registre de configurare → `0x0000`
- Semnale de enable → `0` (oprit)
- Rezultat: sistem în stare inactivă și sigură după reset.

3.5.3 Modularitate

Separarea clară între:

- Logica de acces (write/read),
- Stocarea efectivă (registre),
- Interfața către module funcționale (assign),

permite modificarea ușoară a hărții de regiștri fără a afecta restul sistemului.

4 Numărătorul

4.1 Descriere Generală

Modulul `counter` reprezintă inima perifericului generator de PWM, oferind baza de timp necesară funcționării acestuia. Deoarece ceasul sistemului (`clk`) are o frecvență fixă, perifericul necesită un mecanism flexibil pentru a controla durata perioadei semnalului PWM și rezoluția acestuia.

Arhitectura modulului este compusă din două elemente funcționale:

1. **Prescaler-ul:** Un divizor de frecvență programabil care generează un semnal de validare (*tick*) la intervale specifice de timp.
2. **Numărătorul Principal:** Un registru pe 16 biți care evoluează (crescător sau descrescător) doar la primirea semnalului de la prescaler.

4.2 Interfața Modulului

Semnalele de intrare și ieșire sunt descrise în tabelul de mai jos:

4.3 Descrierea funcționării detaliate

Funcționarea modulului se bazează pe interacțiunea dintre un contor intern de prescalare și contorul principal expus către restul sistemului.

4.3.1 Prescaler

Pentru a obține perioade lungi ale semnalului PWM fără a mări dimensiunea numărătorului principal, folosim un prescaler exponențial. Valoarea țintă a prescaler-ului este calculată hardware prin operația de deplasare logică la stânga (*shift left*), implementând formula $2^{\texttt{prescale}}$:

$$\texttt{prescale\_target} = 1 \ll \texttt{prescale} \tag{1}$$

Mecanismul de Tick: Un contor intern (`prescale_counter`) se incrementează la fiecare ciclu de ceas. Când acesta atinge valoarea `prescale_target - 1`, modulul generează un semnal intern de tip puls numit `prescale_tick`.

Nume Port	Direcție	Lățime	Descriere
`clk`	Input	1	Ceasul sistemului. Toate tranzițiile sunt sincrone cu acest ceas.
`rst_n`	Input	1	Reset asincron global (Active Low). Aduce registrele la valoarea 0.
`count_val`	Output	16	Valoarea curentă a numărătorului, utilizată de generatorul PWM pentru comparare.
`period`	Input	16	Valoarea maximă de numărare (TOP). Definește perioada semnalului PWM.
`en`	Input	1	Semnal de activare. Dacă este 0, numărătorul principal îngheață la valoarea curentă, iar prescaler-ul este resetat.
`count_reset`	Input	1	Reset sincron. Resetează forțat valoarea numărătorului la 0 la următorul front de ceas.
`upnotdown`	Input	1	Direcția de numărare: 1 = Crescător (0 → Period) 0 = Descrescător (Period → 0)
`prescale`	Input	8	Factorul de divizare a frecvenței (exponent al lui 2).

Table 9: Interfața modulului Counter

- Dacă `prescale` = 0, ținta este 1, deci tick-ul se generează la fiecare ciclu (frecvență maximă).
- Dacă `prescale` = 2, ținta este 4, deci tick-ul se generează o dată la 4 cicli.

4.3.2 Logica Numărătorului Principal

Numărătorul principal (`count_val`) este actualizat **doar** atunci când semnalul `prescale_tick` este activ. Acest lucru asigură sincronizarea perfectă cu ceasul sistemului, indiferent de factorul de divizare.

Comportamentul de numărare depinde de intrarea `upnotdown`:

- **Mod Crescător (`upnotdown` = 1):** Numărătorul evoluează de la 0 la `period`. Când `count_val` este egal cu `period`, la următorul tick valid, valoarea se resetează la 0 (Overflow).

$$0 \rightarrow 1 \rightarrow ... \rightarrow \texttt{period} \rightarrow 0...$$

- **Mod Descrescător (`upnotdown` = 0):** Numărătorul evoluează de la valoarea curentă în jos. Când `count_val` atinge 0, la următorul tick valid, se reîncarcă cu valoarea din `period` (Underflow).

$$... \rightarrow 1 \rightarrow 0 \rightarrow \texttt{period} \rightarrow ...$$

4.3.3 Control și Resetare

Modulul implementează o ierarhie strictă a semnalelor de control:

1. **Reset Asincron (`rst_n`):** Are cea mai mare prioritate. Când este activ (0), toate registrele sunt șterse instantaneu.
2. **Reset Sincron (`count_reset`):** Dacă este activ, aduce numărătorul principal la 0 pe frontul crescător al ceasului. De asemenea, resetează prescaler-ul pentru a realinia faza semnalului.
3. **Enable (`en`):**
   - Dacă `en = 0`: Prescaler-ul este ținut în reset (valoarea 0), ceea ce blochează generarea tick-urilor. Numărătorul principal își păstrează valoarea curentă ("îngheață").
   - Dacă `en = 1`: Prescaler-ul începe să numere, permițând actualizarea numărătorului principal.

5 Generatorul de PWM

5.1 Descriere Generală

Modulul `pwm_gen` este responsabil pentru sinteza efectivă a semnalului digital modulat în durată (PWM - Pulse Width Modulation). Acesta primește starea curentă a numărătorului (baza de timp) și configurația definită în regiștri, comparându-le continuu pentru a decide starea logică a ieșirii `pwm_out`.

Acest semnal este destinat controlului dispozitivelor externe (LED-uri, motoare, drivere), iar flexibilitatea sa este asigurată de multiplele moduri de operare (aliniere stânga, dreapta sau definită între două praguri).

5.2 Interfața Modulului (Porturi)

5.2.1 Semnale de sistem

Nume Port	Direcție	Descriere
`clk`	Input	Ceasul sistemului. Folosit pentru a sincroniza ieșirea `pwm_out`, evitând glitch-urile.
`rst_n`	Input	Reset asincron (Active Low). Aduce ieșirea în starea 0.

Table 10: Semnale de sistem PWM

5.2.2 Intrări de configurare și stare

Nume Port	Direcție	Descriere
`pwm_en`	Input	Bit de activare a ieșirii PWM. Acționează ca un întrerupător principal (”Master Switch”).
`period`	Input	Perioada semnalului. Folosită pentru a detecta momentul de resetare (wrap-around) al numărătorului.
`functions`	Input	Registru de configurare a modului: Bit 0: Aliniere (0=Stânga, 1=Dreapta) Bit 1: 0=Aliniat, 1=Nealiniat.
`compare1`	Input	Valoare de prag principală. Determină factorul de umplere (Duty Cycle) sau momentul comutării.
`compare2`	Input	Valoare de prag secundară. Folosită doar în modul ”Nealiniat” pentru a marca sfârșitul pulsului activ.
`count_val`	Input	Valoarea curentă a numărătorului (primită de la modulul `counter`).

Table 11: Semnale de configurare PWM

5.2.3 Ieșire

Nume Port	Direcție	Descriere
pwm_out	Output	Semnalul PWM final, sintetizat și sincronizat cu ceasul sistemului.

Table 12: Semnale de ieșire PWM

5.3 Descrierea funcționării detaliate

O provocare majoră în proiectarea acestui modul a fost lipsa semnalului de direcție (upnotdown) la intrare, deși comportamentul PWM depinde de momentul începerii unui nou ciclu. Soluția adoptată este o arhitectură "bazată pe evenimente" (*Event-Based*), care permite funcționarea corectă indiferent dacă numărătorul urcă sau coboară.

5.3.1 Detecția începutului de ciclu

Pentru a inițializa corect starea semnalului PWM la începutul fiecărei perioade, modulul trebuie să detecteze momentul de "wrap-around" al numărătorului. Acest lucru se realizează prin monitorizarea istorică a valorii count_val:

- Se folosește un registru intern count_val_prev pentru a memora valoarea din ciclul anterior.
- Se generează un semnal intern cycle_start_event activ pentru un ciclu de ceas dacă:
  - **Overflow:** Valoarea anterioară era period și valoarea curentă este 0.
  - **Underflow:** Valoarea anterioară era 0 și valoarea curentă este period.

De asemenea, se generează semnalele compare1_event și compare2_event prin compararea directă a valorii curente cu pragurile setate.

5.3.2 Logica Next-State (Combinațională + Secvențială)

Logica de generare a semnalului este implementată în două etape pentru a asigura stabilitatea și a preveni comportamentele nedorite:

A. Logica Combinațională (always @(*))
Calculează starea viitoare (pwm_out_next) pe baza stării curente, a evenimentelor detectate și a modului de operare.

Gestionarea semnalului pwm_en: Dezactivarea PWM-ului trebuie să "lase blocată linia în starea în care se afla". Acest lucru este implementat prin feedback implicit:

```
pwm_out_next = pwm_out_reg; // Default: menține starea anterioară
if (pwm_en) begin
    // ... logica de modificare a stării doar dacă en este activ
end
```

B. Logica Secvențială (`always @(posedge clk)`)
Actualizează registrul de ieșire `pwm_out_reg` cu valoarea calculată, sincron cu ceasul sistemului. Această separare elimină glitch-urile care ar putea apărea în logica combinațională pură.

5.3.3 Moduri de operare implementate

Comportamentul este dictat de registrul `functions` și de evenimentele de comparare:

1. **Aliniere Stânga (`functions` = 2'b00):**
    - La `cycle_start_event`: Ieșirea devine **1** (începe activ).
    - La evenimentul `compare1`: Ieșirea comută în **0**.
2. **Aliniere Dreapta (`functions` = 2'b01):**
    - La `cycle_start_event`: Ieșirea devine **0** (începe inactiv).
    - La evenimentul `compare1`: Ieșirea comută în **1**.
3. **Nealiniat (`functions` = 2'b1x):**
    - La `cycle_start_event`: Ieșirea devine **0**.
    - La evenimentul `compare1`: Ieșirea comută în **1**.
    - La evenimentul `compare2`: Ieșirea comută în **0**.