Documentație tema: Generator semnal PWM

Atanasiu Ștefan, Piloiu Raul-Ispas, Stoica Lorena-Elena

26 noiembrie 2025

Rezumat

Acest document prezintă detaliile de implementare pentru un generator de semnal PWM (Pulse Width Modulation) controlat printr-o interfață serială SPI. Arhitectura este modulară, separând logica de comunicație, decodificarea instrucțiunilor, stocarea configurației și generarea efectivă a semnalului. Documentația se concentrează pe soluțiile tehnice adoptate pentru a asigura sincronizarea între domenii de ceas, gestionarea stărilor și manipularea semnalelor digitale.

Cuprins

1 Introducere

Proiectul implementează un periferic hardware descris în limbajul Verilog, capabil să genereze semnale PWM configurabile. Sistemul este împărțit în cinci module principale interconectate prin modulul de top (`top.v`):

- **SPI Bridge**: Gestionează comunicația fizică serială.
- **Instruction Decoder**: Interpretează comenzile primite.
- **Registers**: Stochează configurația sistemului.
- **Counter**: Asigură baza de timp.
- **PWM Generator**: Formează semnalul de ieșire.

2 Descrierea Implementării Modulelor

2.1 SPI Bridge (`spi_bridge.v`)

Interfața SPI a fost implementată ținând cont de faptul că ceasul perifericului (`clk`) și ceasul SPI (`sclk`) sunt asincrone sau pot avea faze diferite.

2.1.1 Sincronizarea Domeniilor de Ceas (CDC)

Pentru a evita problemele de setup/hold time, nu am utilizat `sclk` direct ca ceas pentru registrele interne. În schimb, am implementat un mecanism de *oversampling* și detecție de fronturi folosind ceasul sistemului (`clk`):

```
// Sincronizare double-flop
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        sclk_sync_r1 <= 1'b0;
        sclk_sync_r2 <= 1'b0;
    end else begin
        sclk_sync_r1 <= sclk;
        sclk_sync_r2 <= sclk_sync_r1;
    end
end
```

Astfel, generăm semnale interne `sclk_posedge` și `sclk_negedge` care sunt pulsuri de durata unui ciclu de `clk`. Aceasta garantează că toată logica internă rămâne sincronă cu `clk`.

2.1.2 Logica de Transfer

Protocolul este implementat astfel:

- **Recepție (MOSI):** Datele sunt eșantionate pe frontul descrescător detectat al SCLK.
- **Transmisie (MISO):** Datele sunt schimbate pe frontul crescător detectat al SCLK.

La finalul a 8 biți transferați, se generează semnalul `byte_sync`, care activează decodorul de instrucțiuni.

2.2 Instruction Decoder (`instr_dcd.v`)

Decodorul funcționează ca o mașină de stări finite (FSM) cu două stări principale: `S_INSTR` (faza de comandă) și `S_DATA` (faza de date).

2.2.1 Particularități ale FSM

- **Anticiparea Citirii:** În starea `S_INSTR`, dacă bitul de R/W indică o citire, semnalul `read` este activat imediat combinatoric, iar adresa este setată. Acest lucru permite modulului de regiștri să furnizeze datele valide (`data_read`) înainte de următorul front de ceas SPI, asigurând că `spi_bridge` are datele corecte pregătite pentru transmisie în faza `S_DATA`.
- **Adresarea:** Se extrage adresa de bază din instrucțiune, iar logica internă decide dacă accesează octetul superior sau inferior în funcție de bitul 6 al comenzii.

2.3 Blocul de Regiștri (`regs.v`)

Acest modul gestionează stocarea parametrilor. O particularitate importantă a implementării este gestionarea resetării software a contorului.

2.3.1 Mecanismul Self-Clearing pentru Reset

Registrul `COUNTER_RESET` (adresa 0x07) este de tip "write-only" și declanșează o acțiune, nu o stare persistentă. Am implementat un mecanism de auto-curățare:

```
if (write && addr == 6'h07) begin
    count_reset_pulse_r <= 1'b1;
end else if (count_reset_pulse_r == 1'b1) begin
    count_reset_pulse_r <= 1'b0;
end
```

Această logică asigură că semnalul de reset către contor este activ timp de exact un ciclu de ceas, prevenind blocarea contorului în starea de reset dacă software-ul nu scrie explicit '0' înapoi.

2.4 Numărătorul (`counter.v`)

Numărătorul este inima sistemului și include un etaj de prescalare pentru divizarea frecvenței.

2.4.1 Logica de Prescalare

În loc să folosim un ceas divizat (ceea ce nu este recomandat în FPGA/ASIC), am folosit un semnal de "Enable" derivat intern. Variabila `prescaler_count` incrementează la fiecare ciclu de ceas. Doar când aceasta atinge valoarea din registrul `prescale`, contorul principal (`count_val`) este actualizat. Formula implicită este: $T_{update} = T_{clk} \times (PRESCALE + 1)$.

2.4.2 Numărare Bidirecțională

S-a implementat logica de *overflow* și *underflow*:

- **Up:** Când atinge valoarea `period`, se resetează la 0.
- **Down:** Când atinge 0, se încarcă cu valoarea `period`.

2.5 Generatorul PWM (`pwm_gen.v`)

Acest modul este pur combinațional la nivel logic, dar ieșirea este înregistrată (registered output) pentru a evita glitch-urile pe linia de ieșire.

2.5.1 Moduri de Funcționare

Implementarea folosește biții din registrul `functions` pentru a multiplexa comportamentul:

- **Mod Nealiniat (Standard):** Compară `count_val` doar cu `compare1`. Ieșirea comută când contorul depășește pragul. Polaritatea este decisă de bitul de aliniere.
- **Mod Dual-Compare:** Ignoră biții de aliniere clasică și generează pulsul activ strict în intervalul [`compare1`, `compare2`). Această implementare permite generarea unor pulsuri poziționate arbitrar în cadrul perioadei, nu doar la începutul sau sfârșitul acesteia.

3 Concluzii

Soluția propusă respectă arhitectura impusă, aducând contribuții specifice în zona de robustețe a semnalelor (sincronizarea SPI cu ceasul sistemului) și ușurința în utilizare (reset-ul contorului cu auto-clear). Codul este modular, parametrizabil prin regiștri și pregătit pentru integrarea într-un sistem embedded mai larg.