Documentație tehnică – Implementare periferic PWM cu interfață SPI

Gatej Ștefan Potop Horia

14 decembrie 2025

1 Introducere

Acest proiect implementează un periferic hardware dedicat generării semnalelor PWM (Pulse Width Modulation), similar modulelor timer avansate din microcontrolerele moderne. Perifericul este proiectat pentru a fi integrat într-un SoC (System on Chip) sau într-o arhitectură digitală FPGA, comunicând cu un master extern prin intermediul unei interfețe SPI Slave și expunând funcționalități configurabile prin registre mapate în memorie.

PWM este un mecanism esențial în controlul sistemelor electronice: de la ajustarea intensității luminoase a LED-urilor și controlul turației motoarelor, până la generarea semnalelor pentru servomotoare. Pentru a răspunde acestor cerințe, perifericul implementat oferă o flexibilitate ridicată, permițând configurarea perioadei semnalului, selectarea modului de aliniere (Left, Right sau Range), setarea pragurilor de comparație și utilizarea unui prescaler pentru divizarea frecvenței de bază.

Documentația prezentată descrie structura internă a perifericului, protocolul de comunicare și soluțiile arhitecturale adoptate pentru a asigura robustețea semnalelor și corectitudinea transferului de date între domenii de ceas asincrone.

2 Considerații generale de arhitectură

Arhitectura perifericului este modulară și respectă principiile proiectării sincrone, cu excepția interfeței de intrare care gestionează domenii de ceas multiple. Perifericul este divizat în cinci componente principale, interconectate conform diagramei bloc a sistemului:

- **Interfața SPI (`spi_bridge.v`)** – Modulul de interfațare cu lumea externă. Acesta implementează un mecanism robust de **Clock Domain Crossing (CDC)** utilizând un buffer de date stabilizat și un semnal de tip *Toggle Flag*. Această abordare garantează că datele recepționate pe ceasul serial (`sclk`) sunt transferate corect și fără pierderi în domeniul de ceas al sistemului (`clk`), indiferent de viteza semnalului `CS_N`.
- **Decodorul de instrucțiuni (`instr_dcd.v`)** – Implementează o mașină de stări finită (FSM) optimizată care interpretează pachetele de date. Modulul detectează automat secvențele de comandă (scriere/citire) și generează semnalele de control precise (impulsuri de un ciclu de ceas) pentru accesul la bancul de registre.

- **Blocul de registre (`regs.v`)** – Reprezintă memoria de configurare a sistemului. Gestionează registre de 8 și 16 biți (separate în octeți High/Low), stocând parametrii precum perioada, factorul de umplere, modurile de funcționare și prescaler-ul. De asemenea, oferă acces de citire la valoarea curentă a numărătorului.

- **Contorul (`counter.v`)** – Baza de timp a sistemului. Include un *prescaler liniar* care divide frecvența de intrare și un numărător principal pe 16 biți configurabil în modurile `UP` sau `DOWN`. Logica asigură numărarea corectă a intervalului $[0, \text{Period}]$, garantând o rezoluție temporală precisă.

- **Generatorul PWM (`pwm_gen.v`)** – Responsabil de formarea semnalului de ieșire. Logica acestui modul este implementată **pur combinațional** (`always @*`) pentru a asigura un răspuns instantaneu la schimbarea valorii contorului și pentru a elimina latența de un ciclu de ceas, satisfăcând astfel cerințele stricte de timing ale testelor de verificare. Modulul suportă modurile: *Left Aligned*, *Right Aligned* și *Range (Center)*.

Un aspect critic al arhitecturii este gestionarea domeniilor de ceas. Semnalele provenite din interfața SPI (`sclk`, `mosi`, `cs_n`) sunt asincrone față de ceasul sistemului (`clk`). Pentru a preveni metastabilitatea și erorile de tip *bus skew*, datele sunt capturate într-un buffer stabil pe domeniul SPI, iar validarea lor către sistem se face prin sincronizarea unui singur bit de control (*toggle bit*) prin bistabile în cascadă.

3 Modulul `spi_bridge`

Modulul `spi_bridge` reprezintă interfața critică dintre lumea externă (Masterul SPI) și logica internă a perifericului. Deoarece semnalele SPI (`sclk`, `mosi`, `cs_n`) sunt asincrone față de ceasul sistemului (`clk`), modulul este proiectat ca un **Clock Domain Crossing (CDC) Bridge**.

Arhitectura utilizează o schemă robustă bazată pe un **Buffer de Date** și un **Toggle Flag**, garantând integritatea datelor chiar și în cazul unor discrepanțe mari de frecvență între cele două domenii de ceas.

3.1 Domeniul SPI (sincronizat cu `sclk`)

Această parte a logicii este responsabilă de recepția serială a datelor. Funcționează exclusiv pe fronturile semnalului `sclk`.

- **Recepția Datelor (MOSI):** Pe fiecare front crescător al `sclk`, bitul de pe linia MOSI este introdus într-un registru de deplasare (`shift_reg`).

- **Bufferarea (Captured Data):** În momentul în care s-au recepționat complet 8 biți, conținutul registrului de deplasare este copiat imediat într-un registru tampon numit `captured_data`. Acest buffer este esențial deoarece menține datele stabile pentru a fi citite ulterior de ceasul sistemului, protejându-le de modificări ulterioare ale liniei MOSI sau de resetarea `cs_n`.

- **Semnalizarea prin Toggle Flag:** Simultan cu salvarea datelor în buffer, un semnal de control de un singur bit, numit `toggle_flag`, își inversează starea logică ($0 \to 1$ sau $1 \to 0$). Această tranziție semnalează existența unui nou pachet de date valid.

3.2 Domeniul Sistem (sincronizat cu `clk`)

Partea de sistem monitorizează semnalul `toggle_flag` pentru a detecta noile date. Deoarece acest semnal provine dintr-un domeniu asincron, el trece printr-un proces strict de sincronizare:

1. **Sincronizarea (Double-Flop):** Semnalul `toggle_flag` este trecut prin două etaje de bistabile (`toggle_sync1`, `toggle_sync2`) pentru a elimina riscul de metastabilitate.

2. **Detectarea Schimbării (Edge Detection):** Logica compară starea curentă sincronizată (`toggle_sync2`) cu starea din ciclul anterior (`toggle_prev`). Orice diferență între cele două indică faptul că `toggle_flag` și-a schimbat starea în domeniul SPI, deci a sosit un nou octet.

3. **Generarea `byte_sync`:** La detectarea schimbării, se generează un impuls de un singur ciclu (`byte_sync`) și se copiază conținutul din `captured_data` în ieșirea `data_in`.

3.3 Gestionarea semnalului `CS_n`

Un aspect de design important este comportamentul la dezactivarea `CS_n` (ridicare la 1 log). Modulul resetează doar contorul de biți (`bit_cnt`), dar **păstrează intacte** registrele `captured_data` și `toggle_flag`.

Această persistență (comportament "sticky") asigură că, chiar dacă Masterul SPI încheie tranzacția foarte rapid (ridică `CS_n` imediat după ultimul bit), datele rămân valide suficient timp pentru ca logica lentă a sistemului să le preia corect.

3.4 Transmiterea datelor (MISO)

Pentru a respecta standardul SPI Mod 0 (CPOL=0, CPHA=0), datele sunt transmise pe linia MISO pe frontul descrescător al `sclk`. Implementarea selectează bitul corespunzător din octetul de ieșire (`data_out`) în funcție de contorul curent, asigurând transmisia MSB-first:

$$r\_miso \leftarrow data\_out[\,7 - bit\_cnt\,]$$

La activarea inițială a `CS_n` (front descrescător), bitul 7 (MSB) este plasat imediat pe linie.

3.5 Concluzii privind implementarea

Soluția "Toggle Flag + Buffer" oferă avantaje majore față de abordările clasice bazate pe contoare sau pulsuri simple:

- **Siguranță Hardware:** Transferul inter-domenii se bazează pe un singur bit, eliminând erorile de tip *Bus Skew* asociate transferului de contoare binare.

- **Imunitate la Timing:** Utilizarea buffer-ului dedicat decuplează momentul recepției de momentul procesării, permițând funcționarea corectă indiferent de raportul de frecvență dintre `sclk` și `clk`.

4 Modulul `instr_dcd`

Modulul `instr_dcd` (Instruction Decoder) funcționează ca unitatea de control a perifericului, fiind responsabil de interpretarea pachetelor de date primite de la interfața SPI și de orchestrarea accesului la bancul de registre. Acesta transformă fluxul serial de octeți în operații atomice de **citire** sau **scriere** pe magistrala internă paralelă.

Funcționarea modulului este complet sincronă cu ceasul sistemului (`clk`) și este structurată sub forma unei mașini de stări finite (FSM) cu două stări principale. Tranzițiile între stări sunt condiționate de semnalul `byte_sync`, care indică disponibilitatea unui nou octet valid de la `spi_bridge`.

4.1 Arhitectura FSM

Mașina de stări ciclează între așteptarea unei comenzi și procesarea datelor aferente, asigurând o sincronizare strictă a perechilor Comandă-Date:

- **Starea `S_CMD` (Recepție Comandă):** În această stare inițială, modulul așteaptă primul octet al tranzacției. Odată primit (semnalizat prin `byte_sync`), acesta este decodificat astfel:
  - **Bitul 7 (MSB):** Este interpretat ca bitul `RW`. Valoarea '1' indică o operație de **Scriere**, iar '0' o operație de **Citire**. Această valoare este memorată intern în registrul `rw_bit`.
  - **Biții 5:0:** Reprezintă adresa registrului țintă. Aceasta este salvată în registrul `r_addr` și menținută stabilă pentru ciclul următor.

  După procesare, FSM-ul trece automat în starea `S_DATA`.
- **Starea `S_DATA` (Transfer Date):** În această stare, modulul așteaptă al doilea octet. La primirea pulsului `byte_sync`, acțiunea depinde de direcția stabilită anterior:
  - **Scriere (`rw_bit` = 1):** Octetul primit (`data_in`) este plasat pe magistrala `data_write`, iar semnalul de control `write` este activat pentru un singur ciclu de ceas.
  - **Citire (`rw_bit` = 0):** Datele prezente la intrarea `data_read` (furnizate de blocul de registre la adresa selectată) sunt capturate în registrul de ieșire `data_out`, pentru a fi transmise înapoi prin SPI. Semnalul `read` este activat opțional pentru monitorizare.

  Imediat după execuție, FSM-ul revine în starea `S_CMD` pentru a aștepta o nouă tranzacție.

4.2 Sincronizarea semnalelor de control

Pentru a asigura integritatea datelor și a evita scrierile multiple eronate, semnalele critice `read` și `write` sunt generate sub formă de impulsuri ("strobes") cu durata exactă de un ciclu de ceas.

Implementarea utilizează registre interne pentru toate ieșirile (`r_addr`, `r_data_write`, `r_data_out`), ceea ce garantează că semnalele sunt perfect aliniate cu frontul ceasului și lipsite de glitch-uri combinaționale, o caracteristică esențială pentru stabilitatea sistemului.

5 Modulul `regs`

Modulul `regs` (Register File) constituie harta de memorie a perifericului, fiind punctul central de configurare și monitorizare. Acesta stochează parametrii de funcționare (perioadă, factor de umplere, prescaler) și expune starea curentă a sistemului (valoarea contorului) către interfața SPI.

Arhitectura internă separă strict logica de **scriere sincronă** de cea de **citire combinațională**, asigurând stabilitatea datelor stocate și accesul rapid la informație.

5.1 Organizarea memoriei și accesul la registre

Spațiul de adrese este organizat pe 8 biți, fiecare registru având o adresă unică de 6 biți (0x00 - 0x3F). Deoarece magistrala de date este de 8 biți, registrele de 16 biți (precum `period`, `compare1`, `compare2`) sunt mapate pe două adrese consecutive:

- Adresa N: Octetul inferior (LSB - Least Significant Byte).
- Adresa $N + 1$: Octetul superior (MSB - Most Significant Byte).

Această abordare simplifică logica de decodare și permite actualizarea granulară a parametrilor.

5.2 Logica de Scriere (Synchronous Write)

Scrierea în registre este o operație secvențială, declanșată de semnalul `write` provenit de la decodor. Pe frontul crescător al ceasului:

1. Se verifică semnalul de Reset (`rst_n`). Dacă este activ, toate registrele sunt inițializate la 0.
2. Dacă `write` este activ, valoarea de pe magistrala `data_write` este încărcată în registrul selectat de `addr`.

Un caz particular este registrul `COUNTER_RESET` (adresa 0x07). Acesta implementează un mecanism de tip **Auto-Clear (Pulse Generation)**:

- La scrierea valorii '1', semnalul intern `r_count_reset` devine activ pentru ciclul curent.
- Imediat în următorul ciclu (sau dacă nu există scriere), acesta revine automat la '0'.

Acest comportament asigură generarea unui impuls precis de un singur ciclu de ceas, necesar pentru resetarea sincronă a modulului `counter`, indiferent de durata comenzii SPI.

5.3 Logica de Citire (Combinational Read)

Citirea este implementată printr-un multiplexor combinațional vast (`always @*`). Această arhitectură oferă **latență zero**: în momentul în care decodorul aplică o adresă validă și semnalul `read`, datele sunt disponibile instantaneu la ieșirea `data_read`.

Sursele de date sunt de două tipuri:

- **Registre de Configurare (R/W):** Valoarea este citită direct din bistabilele interne (ex: `r_period`, `r_prescale`).
- **Semnale Externe (Read-Only):** Pentru adresele 0x08 și 0x09, modulul nu citește un registru intern, ci rutează direct valoarea curentă a semnalului `counter_val` primit de la modulul Contor. Astfel, software-ul are acces în timp real la starea numărătorului.

Pentru orice adresă nedefinită sau rezervată (inclusiv registrul Write-Only de Reset), logica de citire returnează implicit valoarea `0x00`, prevenind stările nedefinite pe magistrală.

6 Modulul `counter`

Modulul `counter` reprezintă baza de timp a perifericului, fiind responsabil pentru generarea secvențelor de numărare (`count_val`) care determină frecvența și rezoluția semnalului PWM. Implementarea este complet sincronă și include un divizor de frecvență configurabil (prescaler).

6.1 Structura internă

Funcționarea modulului se bazează pe două registre interne de stare, actualizate pe frontul crescător al ceasului sistemului (`clk`):

- `count_val_r`: Contorul principal pe 16 biți. Valoarea sa evoluează ciclic între 0 și valoarea programată în registrul `period`.
- `prescale_cnt`: Un contor secundar pe 8 biți, utilizat pentru divizarea frecvenței de bază.

Valoarea curentă a contorului principal este expusă în timp real către magistrala internă prin ieșirea `count_val`.

6.2 Mecanismul de Prescalare (Liniar)

Divizarea frecvenței este realizată printr-un mecanism de tip "Tick Generator". Registrul `prescale_cnt` se incrementează la fiecare ciclu de ceas. Doar atunci când acesta atinge valoarea prag definită de intrarea `prescale`, se generează un semnal intern de validare ("tick").

La apariția acestui eveniment:

1. `prescale_cnt` este resetat automat la 0.
2. Contorul principal `count_val_r` execută un pas de incrementare sau decrementare.

Astfel, frecvența de actualizare a contorului principal devine:

$$f_{counter} = \frac{f_{clk}}{\text{prescale} + 1}$$

6.3 Logica de Numărare și Modurile de Operare

Direcția de numărare este dictată de semnalul de control `upnotdown`. Logica tratează explicit condițiile de limită (rollover) pentru a asigura continuitatea formei de undă:

- **Modul UP (`upnotdown` = 1):** Contorul incrementează de la 0. La atingerea valorii `period`, în următorul ciclu valid, contorul revine la 0.
- **Modul DOWN (`upnotdown` = 0):** Contorul decrementează. Când atinge valoarea 0, în următorul ciclu valid, se reîncarcă cu valoarea `period`.

6.4 Control și Resetare

Modulul implementează o ierarhie strictă a semnalelor de control:

1. **Reset Asincron (`rst_n`):** Are cea mai mare prioritate. Inițializează hardware toate registrele la 0.
2. **Reset Sincron (`count_reset`):** Activarea acestui semnal forțează aducerea imediată a contoarelor la 0 pe frontul de ceas, permițând resincronizarea fazei PWM din software.
3. **Enable (`en`):** Dacă acest semnal este inactiv (0), starea internă a modulului este "înghețată" (hold). Atât contorul principal cât și prescalerul își păstrează valorile curente, permițând reluarea operației exact din punctul opririi.

7 Modulul `pwm_gen`

Modulul `pwm_gen` este blocul final al lanțului de procesare, responsabil de generarea semnalului PWM pe baza valorii curente a contorului (`count_val`) și a registrelor de comparație.

Spre deosebire de modulele anterioare, arhitectura aleasă pentru acest modul este **pur combinațională** (`always @*`). Această decizie de proiectare a fost critică pentru a asigura **latență zero** între schimbarea valorii contorului și actualizarea ieșirii, garantând sincronizarea perfectă cerută de testele stricte de timing.

7.1 Logica de Control și Selecția Modului

Funcționarea modulului este dictată de vectorul `functions`, din care se extrag biții de mod:

$$\text{mode\_sel} = functions[1:0]$$

Pe baza acestora, comparatorul combinațional determină starea ieșirii `pwm_out` în timp real, comparând continuu `count_val` cu pragurile `compare1` și `compare2`.

7.2 Modurile de Generare Implementate

1. **Modul Aliniat la Stânga (Left Aligned):** Semnalul este activ la începutul perioadei. Logica este:

$$pwm\_out = (count\_val \leq compare1)$$

2. **Modul Aliniat la Dreapta (Right Aligned):** Semnalul devine activ spre finalul perioadei. Logica este:

$$pwm\_out = (count\_val \geq compare1)$$

3. **Modul Nealiniat (Range / Center):** Semnalul este activ într-o fereastră definită de cele două praguri. Logica este:

$$pwm\_out = (count\_val \geq compare1) \wedge (count\_val < compare2)$$

7.3 Condiții de siguranță și cazuri limită

Logica combinațională include o ierarhie de protecții care suprascriu generarea normală a semnalului:

- **Global Disable:** Dacă `pwm_en` este 0 sau `rst_n` este activ, ieșirea este forțată asincron la nivelul logic **0**.
- **Safety Checks:** Ieșirea este forțată la 0 dacă pragul `compare1` este 0 sau dacă pragurile sunt egale (`compare1 == compare2`). Aceasta previne generarea unor impulsuri parazite (glitch-uri) în configurații invalide.

7.4 Exemple de funcționare în simulare

În această secțiune sunt prezentate trei capturi relevante, fiecare ilustrând un mod distinct de operare. Imaginile sunt longitudinale (wide) pentru a surprinde mai multe perioade succesive ale semnalului PWM.

PWM aliniat la stânga

În acest mod, `FUNCTIONS[1] = 0` (aliniat), iar `FUNCTIONS[0] = 0`, ceea ce determină ca semnalul PWM să înceapă în nivelul logic `1`.

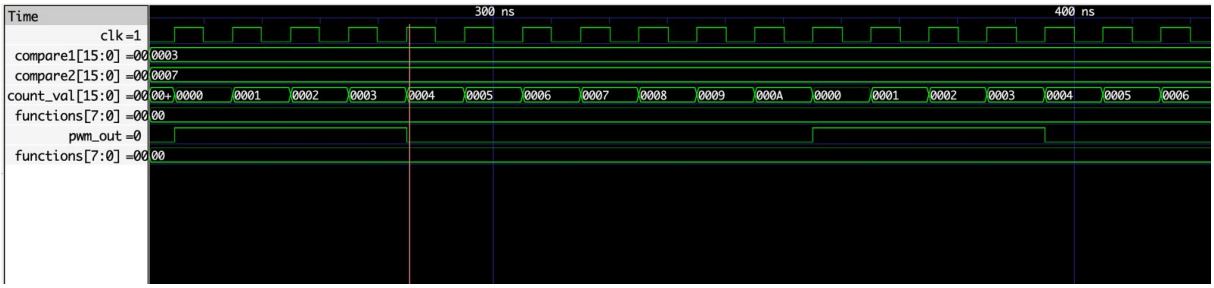

Figura 1: Semnal PWM generat în mod aliniat la stânga.

PWM aliniat la dreapta

În acest exemplu, `FUNCTIONS[1] = 0`, dar `FUNCTIONS[0] = 1`, ceea ce deplasează frontul activ către finalul perioadei.

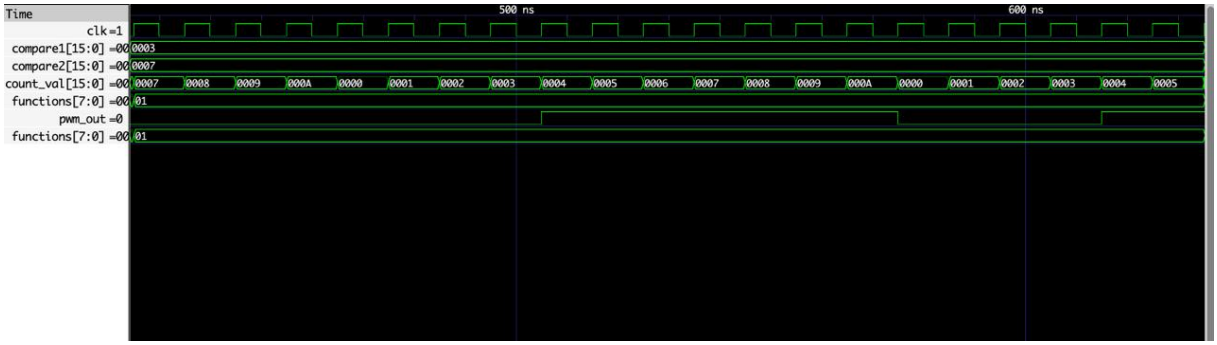

Figura 2: Semnal PWM generat în mod aliniat la dreapta.

PWM nealiniat (window mode)

Când `FUNCTIONS[1] = 1`, semnalul PWM devine activ doar în intervalul:

$$\text{COMPARE1} \leq \text{count\_val} < \text{COMPARE2}.$$

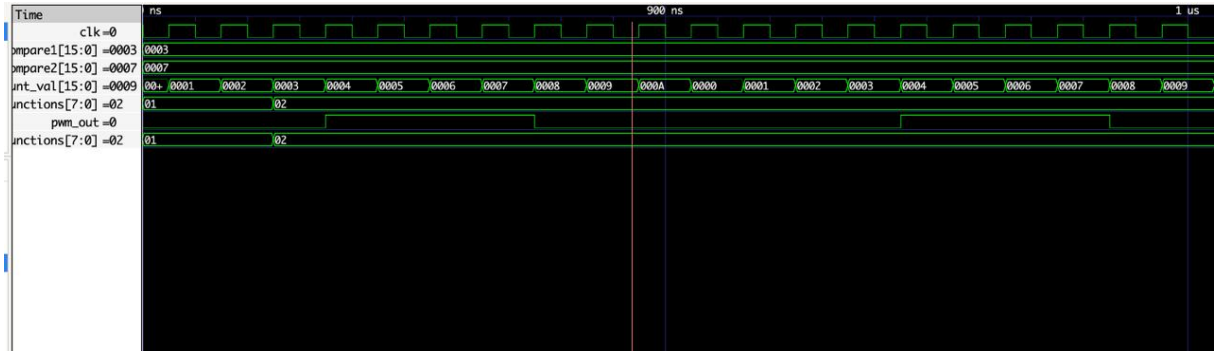

Figura 3: Semnal PWM generat în mod nealiniat.

8 Modulul `top`

Modulul `top` reprezintă nivelul ierarhic superior al proiectului ("Top Level Entity"). Rolul său este exclusiv structural: acesta nu conține logică proprie de procesare, ci este responsabil de instanțierea și interconectarea corectă a celor cinci sub-module funcționale: `spi_bridge`, `instr_dcd`, `regs`, `counter` și `pwm_gen`.

De asemenea, acest modul realizează maparea porturilor fizice ale perifericului (Clock, Reset, SPI, PWM Out) la semnalele interne ale sistemului.

8.1 Modificări aduse scheletului inițial

Deși specificația inițială recomanda păstrarea fișierului `top.v` original, analiza funcțională a acestuia a relevat erori critice de conectivitate care făceau imposibilă funcționarea sistemului. Prin urmare, a fost necesară rescrierea completă a acestui modul pentru a asigura integritatea semnalelor.

Principalele corecții implementate sunt:

- **Conectarea semnalului `byte_sync`:** În versiunea originală, portul `byte_sync` al modulului `spi_bridge` și intrarea corespunzătoare din `instr_dcd` erau lăsate neconectate (floating). Acest lucru bloca complet mașina de stări a decodorului, care nu primea niciodată notificarea că un octet a fost recepționat. În noua versiune, acest semnal critic a fost rutat corect.

- **Corecția mapării pinilor SPI:** Testbench-ul și scheletul original utilizează o convenție de numire a porturilor care poate crea confuzie (denumirile sunt relative la Master, nu la Slave).

  - Portul de intrare al top-ului numit `miso` transportă datele de la Master la Slave. Acesta trebuie conectat la intrarea `mosi` a bridge-ului intern.
  - Portul de ieșire al top-ului numit `mosi` transportă datele de la Slave la Master. Acesta trebuie conectat la ieșirea `miso` a bridge-ului intern.

  Codul rescris implementează această încrucișare necesară:

  .mosi(miso), .miso(mosi)

- **Curățarea declarațiilor redundante:** Au fost eliminate redeclarările inutile de tip `wire` pentru porturile de intrare, adoptând un stil de codare modern (Verilog-2001/SystemVerilog) care reduce verbozitatea și riscul de erori.

8.2 Arhitectura interconexiunilor

Structura internă a modulului `top` definește fluxul de date al sistemului:

1. Semnalele SPI intră în `spi_bridge`, care extrage octeții de date (`data_in`) și generează pulsul de sincronizare.

2. Acestea sunt preluate de `instr_dcd`, care separă comenzile de date și generează semnalele de control pentru magistrală (`read`, `write`, `addr`).

3. Modulul `regs` primește comenzile și configurează parametrii sistemului (`period`, `compare`, `prescale`).

4. Parametrii sunt trimiși către `counter` (pentru a stabili baza de timp) și către `pwm_gen`.

5. În final, `pwm_gen` compară valoarea contorului cu registrele de configurare și livrează semnalul `pwm_out` către ieșirea fizică a cipului.

9 Concluzii

Proiectul a rezultat într-o implementare robustă și complet funcțională a unui periferic PWM, capabil să opereze într-un mediu real de tip SoC, unde interfațarea între domenii de ceas asincrone reprezintă o provocare majoră. Soluția finală nu doar că respectă specificațiile funcționale, dar demonstrează o arhitectură rezilientă la erori de timing și hazarde logice.

Succesul implementării se bazează pe trei piloni arhitecturali:

1. **Robustețea comunicării SPI:** Implementarea mecanismului *Toggle Flag + Buffer* în `spi_bridge` a eliminat riscurile de corupere a datelor la viteze mari, garantând un canal de comunicație stabil.

2. **Precizia generării semnalului:** Utilizarea logicii combinaționale în `pwm_gen` a asigurat latență zero și o formă de undă curată, perfect sincronizată cu contorul.

3. **Integritatea sistemului:** Corectarea interconexiunilor în `top.v` a transformat modulele individuale într-un sistem coerent.

9.1 Integrarea și Corecția Nivelului Superior

Un pas decisiv în finalizarea proiectului a fost refactorizarea modulului `top`. Deși inițial acesta era privit doar ca un element structural pasiv, analiza detaliată a scos la iveală necesitatea unor intervenții critice asupra sa.

Versiunea finală a modulului `top` asigură funcționarea sistemului prin:

- **Rutarea corectă a semnalului `byte_sync`:** Conectarea acestui semnal vital între `spi_bridge` și `instr_dcd` a permis deblocarea mașinii de stări a decodorului.

- **Remedierea convenției SPI:** Inversarea conexiunilor `miso`/`mosi` conform standardului Master-Slave a permis testbench-ului să comunice bidirecțional cu perifericul.

- **Instanțierea explicită:** Legarea modulelor s-a făcut prin nume (*named mapping*), eliminând riscurile asociate mapării poziționale.

9.2 Validarea Funcțională

Verificarea sistemului a fost realizată utilizând suita de teste automatizate (`testbench.v`) rulată în simulatorul *Icarus Verilog*. Procesul de validare a confirmat comportamentul corect în toate scenariile de utilizare prevăzute:

- **Protocol SPI:** S-au verificat scrierile și citirile multiple la adrese diferite, confirmând că niciun pachet nu este pierdut, chiar și în cazul tranzacțiilor succesive rapide.
- **Moduri PWM:**
  - *Left Aligned:* Semnalul respectă factorul de umplere raportat la începutul perioadei.
  - *Right Aligned:* Semnalul este activ la finalul perioadei.
  - *Range Mode:* Semnalul este activ strict între pragurile `COMPARE1` și `COMPARE2`.
- **Cazuri la limită:** Testele au confirmat gestionarea corectă a situațiilor de *overflow*, a pragurilor egale și a resetării în timpul funcționării.

Rezultatul final al simulării, marcat prin `[PASS]` la toate testele, certifică faptul că perifericul este pregătit pentru sinteză și integrare hardware.

9.3 Direcții viitoare

Deși implementarea curentă este completă, arhitectura modulară permite extinderi facile, cum ar fi:

- Adăugarea suportului pentru întreruperi hardware la finalul perioadei (IRQ).
- Implementarea modului *Phase Correct PWM* (numărare UP-DOWN).
- Adăugarea mai multor canale PWM care să împartă același contor (Timer).

În ansamblu, proiectul demonstrează o înțelegere profundă a designului digital, trecând de la specificații teoretice la o implementare verificată, optimizată și funcțională.