Documentație tehnică: Generator de semnal PWM

1. Implementarea interfeței de comunicare SPI (spi_bridge.v, instr_dcd.v)

Modulul spi_bridge se află între masterul SPI (din exterior: microcontroller, CPU etc.) și logica internă a generatorului PWM. Acesta are două roluri foarte importante:

1. să primească biți seriali pe MOSI și să îi prezinte intern în data_in; în momentul în care data_in este complet, byte_sync devine 1 și informează logica generatorului PWM că are date disponibile pentru procesare;
2. să ia câte un bit din data_out și să îl trimită prin MISO la master până se termină cei 8 biți.

Logica modulului este una simplă. Aceasta include, pe lângă datele din schelet, un counter care ține cont de câți biți au fost primiți de la master și doi registri care memorează informațiile din MOSI, respectiv data_out, și le transmit în data_in, respectiv în MISO.

Modulul instr_dcd este un decodor de instrucțiuni și se află între spi_bridge și regs. Acesta e un pic mai complicat. În esență, el așteaptă primul byte_sync să fie 1 și atunci intră în perioada de setup. Dacă ultimul bit din data_in (data_in[7]) este 1, atunci avem operație de write, iar dacă este 0, avem operație de read. Dacă data_in[6] este 1, atunci instrucțiunea se referă la partea MSB a registrului de 16 biți ([15:8]), iar dacă este 0, se referă la partea LSB ([7:0]). Ultimii biți data_in[5:0] reprezintă adresa registrului la care se face operația.

În faza de setup, modulul doar decodează această instrucțiune și o memorează intern (tipul operației, high/low și adresa). Dacă este o operație de read, în același moment modulul trimite un puls pe semnalul read către regs, iar valoarea citită din registru (data_read) este pusă direct în data_out, pentru a fi trimisă mai departe spre SPI la următoarea transmisie.

La a doua instanță în care byte_sync = 1, intrăm în faza de date. Aici, dacă operația este write (RW = 1), byte-ul primit în data_in reprezintă efectiv valoarea care trebuie scrisă în registru, iar modulul pune această valoare în data_write și trimite și un puls write către

regs pentru a confirma scrierea. Dacă operația este read (RW = 0), acest al doilea byte nu se folosește la nimic și este doar un byte dummy trimis de master ca să existe ceas pentru transmisia SPI; valoarea care trebuia returnată fusese deja pregătită în faza de setup în data_out.

La finalul acestor două faze, modulul revine în starea inițială și așteaptă noul byte_sync de la spi_bridge pentru următoarea instrucțiune.

2. Logica de gestiune a registrelor (regs.v)

Modulul regs joacă rolul esențial de interfață de control și status între o magistrală de date periferică de 8 biți (specificată de semnalele data_write, data_read și addr) și blocurile funcționale interne (Contorizare și PWM, care operează pe 16 biți).

Implementarea acestui modul a vizat trei aspecte critice ale arhitecturii sistemului, aspecte care au necesitat decizii specifice, distincte de simpla mapare a registrelor.

1. Gestiunea datelor. Accesul la registrele de 16 biți pe o magistrală de 8 biți

O particularitate a acestei implementări este configurare registrelor interne de 16 biți (period, compare1, compare2 etc.) utilizând o interfață periferică de 8 biți, astfel, rezultând o disparitate. Această disparitate impune o soluție de serializare a datelor.

Alegerea implementată: **selectorul de octet prin bitul de adresă addr[5]**.

Am ales să dedicăm bitul cel mai semnificativ al adresei fizice de 6 biți (addr[5] denumit in regs.v ca hl_bit) rolului de selector de octet pentru registrele de 16 biți, în locul utilizării a două adrese consecutive. Când hl_bit = 0, transferul vizează LSB-ul, iar când hl_bit = 1 se vizează MSB-ul.

Decodificarea Adresei (5 biți): Adresa fundamentală a registrului este identificată de addr[4:0] .

Justificarea Arhitecturală a Selecției prin addr[5]:

Această metodă oferă o modalitate elegantă de a configura registrele:

- **Simplificarea Logicii:** Logica de decodificare din interiorul blocului case al logicii secvențiale este simplificată, deoarece fiecare registru de 16 biți necesită o singură ramură case (ex: 5'h00 pentru r_period), iar diferențierea LSB/MSB se realizează printr-un simplu if/else intern bazat pe hl_bit.
- **Eficiența Spațiului de Adresare:** Dacă am fi utilizat două adrese consecutive, am fi consumat rapid spațiul de adresare disponibil de 64 de locații (limitat la 5 biți pentru a

menține consistența structurii). Prin utilizarea addr[5] ca selector, obținem accesul complet la registrul de 16 biți menținând o singură intrare în tabelul de adrese pentru Contor (ex: 0x00).

- **Apropierea de Standarde Periferice:** Această abordare emulează modul în care microcontrolerele moderne gestionează accesul la registrii mari (ex: timer/counter-uri de 16/32 de biți) prin porturi de 8 biți, unde secvența de scriere (LSB urmat de MSB) este esențială pentru a asigura coerența datelor la nivelul blocului de execuție internă.

Deși scrierea completă a unui registru de 16 biți necesită două cicluri de transfer SPI, valoarea de 16 biți este asamblată și menținută sincron în registrul intern (r_period[7:0] <= data_write; r_period[15:8] <= data_write;). Acest lucru asigură că, în orice moment, Contorul vede o valoare completă, coerentă, chiar dacă actualizarea ei a fost realizată pe parcursul a două transferuri separate.

2. Disocierea Logicii de citire (combinațională) de cea de scriere (secvențială)

Modulul regs.v este structurat în mod intenționat folosind două blocuri always distincte pentru a gestiona operațiile de citire și scriere, deși ambele accesează aceleași registre interne (r_period, r_compare1, etc.). Această disociere este o practică standard în designul hardware pentru a optimiza performanța și a garanta sincronizarea.

A. Logica de scriere (Secvențială - Sincronă)

Logica de scriere este implementată într-un bloc secvențial (always @(posedge clk or negedge rst_n)).

- **Scop:** Stocarea permanentă și sigură a stării (configurației) perifericului.
- **Mecanism:** Operațiile de scriere (r_period <= ...) se realizează numai pe frontul crescător al ceasului principal (clk).
- **Justificare:**

  Stabilitate: Asigură că actualizarea registrelor (schimbarea stării) are loc într-un moment de timp bine definit (marginea ceasului). Fără această sincronizare, schimbările de date ar putea duce la stări tranzitorii, instabile (condiții de cursă sau glitches) în interiorul sistemului.

  Control: Blocul răspunde doar când semnalul de control write (generat de instr_dcd.v) este activ, garantând că registrele se actualizează doar când o tranzacție SPI validă a sosit.

B. Logica de citire (Combinațională - Asincronă)

Logica de citire este implementată într-un bloc combinațional (always @(*)), care produce datele de ieșire (data_read) imediat.

- **Scop:** Furnizarea rapidă a valorii stocate către masterul SPI.
- **Mecanism:** Ieșirea data_read este o simplă funcție logică (un multiplexor mare) a intrărilor read, addr și a valorilor registrelor interne.

  data_read = f(read, addr, r_period, r_compare1, …)

- **Justificare:**

  Latență Zero: SPI Masterul așteaptă datele de citire (MISO) în cel de-al doilea ciclu al tranzacției. Pentru a nu introduce întârzieri inutile (latență), logica combinațională accesează direct valoarea registrului (care este stabilă până la următorul front de ceas) și o furnizează imediat. Nu este nevoie să așteptăm un ciclu de ceas suplimentar.

  Evitarea Erorilor: Dacă citirea ar fi fost secvențială, ar fi fost necesar un registru

suplimentar de ieșire și o logică complexă de temporizare. Logica combinațională simplifică radical interfața de citire-înapoi.

Separarea se reflectă direct în structura Verilog a modulului regs.v:

Aspect	**Logica de Scriere (Sequential)**	**Logica de Citire (Combinational)**
Bloc Verilog	always @(posedge clk or negedge rst_n)	always @(*)
Ieșiri Controlate	Registrele interne (r_period, r_compare1, etc.)	Ieșirea de date (data_read)
Dependență	Depinde de frontul ceasului (clk)	Depinde de schimbarea oricărei intrări (asincron)
Rol Principal	Păstrează starea (Memorie)	Preia starea (Interfață)

Această arhitectură este esențială pentru a asigura o funcționare stabilă și un răspuns rapid la cererile de citire din partea SPI Masterului.

3. Mecanismul de contorizare (counter.v)

Modulul utilizează un contor ascendent de 16 biți (r_count) care se incrementează sincron cu frontul pozitiv al ceasului (clk). Această alegere de design (Up-Counter) conduce la un semnal PWM de tip Edge-Aligned, unde toate impulsurile încep simultan la momentul de resetare al contorului (r_count = 0), iar durata lor (Duty Cycle) este determinată de momentul unic de oprire (comparare).

- **Implementare Cheie:**

```
always @(posedge clk or negedge rst_n) begin
   if (!rst_n)
      r_count <= 16'h0;
   else if (count_reset || (r_count == r_period))
      r_count <= 16'h0;
   else
      r_count <= r_count + 1'b1;
end
```

A. Mecanismul prescaler

Cea mai semnificativă caracteristică arhitecturală a acestui design este includerea unui **Prescaler programabil** (Divizor de Frecvență), gestionat de input-ul prescale (8 biți). Prescaler-ul (registrul r_prescale_count) acționează ca un ceas "mai lent", care generează un impuls count_tick la o frecvență divizată cu P + 1 (unde P este valoarea din prescale). Modulul reduce încărcarea (overhead) la nivelul main counter-ului (r_count_val), care se incrementează/decrementează doar la frontul semnalului count_tick, nu la fiecare ciclu de clk.

- **Implementare Cheie:**

```
if (count_reset || (r_count_val == period && upnotdown == 1'b1)) begin
    r_prescale_count <= 8'h00;
end
else if (r_prescale_count == prescale) begin
   r_prescale_count <= 8'h00; // Reset prescaler counter to 0
end
else begin
   r_prescale_count <= r_prescale_count + 1'b1;
end
```

Condiția de mai sus asigură că nu doar resetarea software (count_reset), ci și rollover-ul contorului principal (când r_count_val atinge period în modul Up-count) resetează imediat Prescaler-ul. Această sincronizare este importantă pentru a garanta că noul ciclu al contorului principal (care începe de la 0) este perfect aliniat cu prima "bătaie" (count_tick) a Prescaler-ului, eliminând orice latență sau eroare de fază acumulată la începutul fiecărui ciclu de numărare.

B. Flexibilitatea contorului principal (Numărarea bidirecțională)

Contorul principal (r_count_val, 16 biți) este cel care definește durata ciclului de temporizare, controlat de period. Logica sa este complexă, deoarece trebuie să gestioneze cele două moduri de operare distincte.

Modulul gestionează două tipuri de Rollover (întoarcere la începutul ciclului), care sunt asimetrice și depind de semnalul upnotdown.

1. Up-Counting Mode (upnotdown = 1'b1)

- **Comportament:** Contorul se incrementează de la 0 până la period.
- **Rollover:** La atingerea valorii period, contorul se resetează la 0.
- **Ciclu:** Generază o formă de undă tipică Edge-Aligned, utilă pentru PWM standard. Durata ciclului este period + 1 (de la 0 la period).

2. Down-Counting Mode (upnotdown = 1'b0)

- **Comportament:** Contorul decrementează de la o valoare (period) până la 0.
- **Rollover (The Differentiator):** Când contorul atinge 0, acesta nu se resetează, ci se reîncarcă cu valoarea period (r_count_val <= period;).
- **Aplicație:** Acest mod este esențial pentru generarea de forme de undă Center-Aligned (sau Symmetrical) PWM, unde contorul numără alternativ în sus și în jos, creând o simetrie perfectă în jurul punctului central.

4. Generarea PWM (pwm_gen.v)

Modulul pwm_gen.v este etapa de execuție a întregului sistem, având rolul de a transforma variabilele de control stocate în regs.v și semnalul de sincronizare (contorul) generat de counter.v în semnalul final de Modulare a Lățimii Impulsului (PWM). Acesta implementează logica de comparație crucială, unde valoarea curentă a contorului este comparată continuu cu pragurile (compare1, compare2) pentru a determina momentul exact al tranzițiilor de la LOW la HIGH (sau invers) ale semnalului de ieșire. Designul acestui modul este conceput pentru a suporta moduri multiple de operare (de exemplu, *edge-aligned* sau *center-aligned* PWM) și pentru a asigura că modificările parametrilor de la nivelul registrelor sunt aplicate fără întreruperi, garantând astfel un control precis și granular al ciclului de lucru (*duty cycle*) al aplicației.

Modulul pwm_gen primește valoarea curentă a contorului (count_val) ca intrare, în loc să o gestioneze intern. Această separare logică a funcției de contorizare de logica de generare a impulsului permite reutilizarea modulului de contorizare (counter.v) pentru multiple instanțe de PWM sau pentru funcții de sincronizare adiacente.

Ieșirea finală (pwm_out) este legată de un registru de stare intern (r_pwm_out). Toate deciziile logice sunt finalizate cu actualizarea acestui registru, care este reîmprospătat exclusiv pe frontul crescător al ceasului (posedge clk). Această metodă asigură că orice tranziție a semnalului PWM (LOW to HIGH sau invers) este perfect sincronizată cu ciclul de ceas, garantând o precizie maximă (Cycle-Accurate Precision) a lățimii impulsului.

A. Abordarea de dezvoltare

Logica internă a modulului pwm_gen.v este o descriere a hardware-ului (HDL).

- **Logică Secvențială Sincronă:** Ieșirile PWM sunt gestionate într-un bloc always @(posedge clk) pentru a asigura tranziții stabile, perfect sincronizate cu ceasul sistemului.
- **Set/Reset:** Logica definește explicit, în timp real, condițiile de **SET** (ridicare la 1) și **RESET** (coborâre la 0) pentru fiecare canal, bazându-se pe evenimente de comparație a contorului.

În plus, modulul utilizează Center-Aligned (aliniat la centru) folosește o numărare triunghiulară (0 la max_val și înapoi la 0). Impulsul PWM este generat simetric în jurul punctului central.

B. Logica modurilor de operare (Mode Register - mode_reg[1:0])

Cei doi biți din mode_reg dictează comportamentul semnalului de ieșire (pwm_outN) în funcție de valoarea curentă a contorului (count) comparată cu valoarea de referință (compareN).

Modurile de operare sunt definite de combinațiile binare:

01: Modul 0 (Edge-Aligned, Non-Inversat - Clasic)

- **Comportament:** Contorul numără de la 0 la max_val și se resetează la 0.
- **Logica:**
  - **SET:** Ieșirea (pwm_outN) este setată la **HIGH (1)** când contorul atinge 0.
  - **RESET:** Ieșirea este resetată la **LOW (0)** când count devine egal sau mai mare decât compareN.

10: Modul 1 (Edge-Aligned, Inversat)

- **Comportament:** Similar cu Modul 0, dar cu polaritatea inversată.
- **Logica:**
  - **SET:** Ieşirea (pwm_outN) este setată la **HIGH (1)** când count atinge valoarea compareN.
  - **RESET:** Ieşirea este resetată la **LOW (0)** când contorul atinge max_val (sau 0, în funcție de implementarea exactă a contorului, de obicei la resetarea contorului).

11: Modul 2 (Center-Aligned, Non-Inversat - Avansat)

- **Comportament:** Contorul numără triunghiular (0 -> max_val -> 0). Impulsul PWM este centrat pe perioada semnalului.
- **Logica:**
  - **SET:** Ieşirea (pwm_outN) este setată la **HIGH (1)** când contorul se află sub valoarea compareN **în ambele faze** (ascendentă și descendentă).
  - **RESET:** Ieşirea este resetată la **LOW (0)** când count devine egal cu compareN (în faza ascendentă) sau când count ajunge la o valoare care inversează logica de comparație (în faza descendentă).

00: Modul 3 (Output Disabled)

- **Comportament:** Indiferent de valoarea contorului (count) și a pragurilor de comparație (compare1, compare2), ambele ieșiri PWM sunt forțate la o stare predefinită, de obicei LOW (0), pentru a opri funcționarea modulului.
- **Logica:**
  - **SET:** Ieşirea nu este setată niciodată la **HIGH (1)**.
  - **RESET**: Ieşirea este forțată permanent la **LOW (0)**.

De asemenea, modul gestionează stabil cazurile limită de 0% și 100% a Duty Cycle-ului fără a produce impulsuri scurte sau glich-uri.

Condiție	**Valoare compareN (N =1 sau N=2)**	**Ieşire pwm_out**	**Abordarea**
0% Duty Cycle	compareN = 0	Constant LOW	Logica include o excepție: dacă compareN este citit ca zero, ieșirea este forțată la 0.
100% Duty Cycle	compareN = max_val	Constant HIGH	Dacă compareN este egal cu valoarea maximă a contorului, ieșirea este forțată la 1.