Documentatie proiect generator de semnal PWM

Cismaru Adrian, Gomboș Eliza, Halanduț Alexandru

Noiembrie 2025

1 Implementare `spi_bridge.v`

1.1 Mod general de funcţionare

- Modulul `spi_bridge` implementează un SPI slave simplificat care permite comunicarea între un master SPI şi logica internă a sistemului.
- Funcţionează conform protocolului SPI Mode 0 (CPOL=0, CPHA=0): datele MOSI sunt eşantionate pe frontul crescător al `sclk`, iar datele MISO sunt transmise pe frontul descrescător.
- Transferul de date este MSB-first, atât la recepţie cât şi la transmisie.
- La recepţia fiecărui byte complet, este generat un impuls de un ciclu `clk` pe semnalul `byte_sync`, indicând faptul că `data_in` este valid.
- Sunt suportate transferuri consecutive pe durata unui `cs_n` activ, fără pierdere de date.

1.2 De ce am ales această implementare

- Separarea clară a domeniilor de ceas: `sclk` pentru logica SPI şi `clk` pentru perifericul intern.
- Utilizarea unui buffer dedicat (`data_buffer`) pentru stabilitatea datelor recepţionate.
- Generarea robustă a semnalului `byte_sync` prin sincronizare între domenii de ceas folosind un mecanism de toggle.
- Permite transferuri SPI continue, păstrând simplitatea implementării.

1.3 Explicaţia implementării pas cu pas

Regiştri interni

- `shift_in [7:0]`: registru de shift utilizat pentru recepţia biţilor de pe linia MOSI.

- `data_buffer [7:0]`: buffer care reţine un byte complet recepţionat, asigurând stabilitatea semnalului `data_in`.
- `shift_out [7:0]`: registru de shift pentru transmiterea datelor pe MISO.
- `bit_cnt [2:0]`: contor de biţi pentru recepţia unui byte complet.
- `byte_tgl`: semnal care togglează la fiecare byte recepţionat complet.
- `t_s1`, `t_s2`: registre de sincronizare ale semnalului `byte_tgl` în domeniul `clk`.
- `cs_q`: registru utilizat pentru detectarea frontului descrescător al semnalului `cs_n`.

Recepţia datelor de pe MOSI

- Pe fiecare front crescător al semnalului `sclk`, bitul prezent pe MOSI este shiftat în registrul `shift_in`.
- Contorul `bit_cnt` este incrementat pentru fiecare bit recepţionat.
- Când `bit_cnt` ajunge la valoarea 7, un byte complet a fost recepţionat.
- Byte-ul complet este copiat în `data_buffer`, iar semnalul `byte_tgl` este inversat (toggle).
- Dacă `cs_n` devine inactiv, contorul de biţi este resetat.

Transmiterea datelor pe MISO

- Frontul descrescător al semnalului `cs_n` este detectat folosind registrul `cs_q`.
- La începutul fiecărui cadru SPI, registrul `shift_out` este încărcat cu valoarea `data_out`.
- Pe fiecare front descrescător al `sclk`, registrul `shift_out` este shiftat, iar bitul MSB este transmis pe MISO.
- Această metodă respectă cerinţele SPI Mode 0 pentru transmiterea datelor.

Generarea semnalului `byte_sync`

- Semnalul `byte_tgl` este generat în domeniul `sclk` la finalul fiecărui byte recepţionat.
- Acesta este sincronizat în domeniul `clk` prin două registre consecutive (`t_s1`, `t_s2`).
- Semnalul `byte_sync` este obţinut prin operaţia XOR dintre `t_s1` şi `t_s2`.
- Rezultatul este un impuls de un ciclu `clk`, generat la fiecare byte recepţionat complet.

Gestionarea semnalului `cs_n` şi resetarea

- Resetul asincron iniţializează toate registrele interne.
- Când `cs_n` este inactiv, contorul de biţi este resetat, iar transferul SPI este considerat încheiat.
- La următoarea activare a lui `cs_n`, modulul este pregătit pentru un nou transfer.

2 Implementare `instr_dcd.v`

2.1 Mod general de funcţionare

- Modulul `instr_dcd` realizează decodarea comenzilor SPI primite de la un master şi interacţionează cu un registru periferic.
- Datele sunt transmise în pachete de 8 biţi, primul bit indicând tipul comenzii (read sau write), iar restul de 6 biţi reprezentând adresa registrului.
- Pentru comenzile de tip write, modulul aşteaptă un payload ulterior pentru a-l scrie in registru.
- Pentru comenzile de tip read, modulul pulsează semnalul de read şi pune pe `data_out` conţinutul registrului citit.
- Funcţia este realizata printr-o maşina de stări finite (FSM) cu trei stări: `IDLE`, `WAIT_FOR_DATA` şi `SEND_READ_BYTE`.

2.2 De ce am ales această implementare

- Am ales o implementare bazată pe FSM pentru a avea control clar asupra fluxului de date şi pentru a sincroniza corect semnalele de read/write cu semnalul `byte_sync`.
- Dezactivarea implicită a pulsurilor `read` şi `write` în fiecare ciclu evită generarea de semnale false.
- Separarea logicii secvenţiale de cea combinaţională asigură predictibilitate în schimbarea stării şi respectarea sincronizării cu ceasul.
- Implementarea permite o extensibilitate uşoară pentru comenzi suplimentare sau protocoale SPI mai complexe.

2.3 Explicaţie implementare pas cu pas

Regiştri interni

- `cmd_is_write`: indică dacă comanda curentă este write (1) sau read (0).
- `addr_reg [5:0]`: stochează adresa registrului ţinta.
- `data_write_reg [7:0]`: reţine payload-ul pentru operaţiunea write.
- `out_reg [7:0]`: reţine datele ce vor fi transmise către master.
- `read_reg` şi `write_reg`: semnale de puls pentru read şi write generate pe un ciclu de ceas.

Recepţionarea datelor de la master

- În starea `IDLE`, la detectarea semnalului `byte_sync`, se preia primul byte.
- Bitul 7 indică tipul comenzii, iar biţii [5:0] reprezintă adresa.
- Pentru comenzi read, se pulsează `read_reg` şi se trece in starea `SEND_READ_BYTE`.
- Pentru comenzi write, se trece în starea `WAIT_FOR_DATA` pentru a aştepta payload-ul.

Preluarea payload-ului pentru write

- În starea `WAIT_FOR_DATA`, la următorul `byte_sync`, se stochează payload-ul în `data_write_reg`.
- Se pulsează `write_reg` pentru un ciclu pentru a scrie datele în registru.
- După scriere, FSM-ul revine în starea `IDLE`.

Transmiterea datelor pentru read

- În starea `SEND_READ_BYTE`, la `byte_sync` se plasează conţinutul registrului citit (`data_read`) în `out_reg`.
- Datele sunt astfel puse pe `data_out` pentru master.
- După transmitere, FSM-ul revine în starea `IDLE`.

Generarea pulsurilor de control

- Semnalele `read_reg` şi `write_reg` sunt activate doar un ciclu de ceas.
- Această abordare previne blocarea semnalelor şi asigură sincronizarea corectă cu masterul.

Logica FSM combinaţională

- Determină următoarea stare pe baza stării curente şi a semnalului `byte_sync`.
- Asigură tranziţii corecte pentru comenzile read şi write.

3 Implementare regs.v

3.1 Mod general de functionare

- Modulul `regs` ofera o interfata intre software (prin adrese de registri) si module hardware precum counter si PWM.
- Permite citirea si scrierea de registre prin semnalele `read`, `write`, `addr`, `data_write`, `data_read`.
- Gestioneaza si actualizeaza valori interne precum: `period`, `compare1`, `compare2`, `prescale`, `en`, `count_reset`, `upnotdown`, `pwm_en`, `functions`.
- Sustine scrierea secventiala (pe clk) si citirea combinatorie (asynchronous).
- Suporta resetarea hardware completa (`rst_n`) sau reset temporar (`count_reset`) pentru contor.

3.2 Adresare si harta de registre

- `0x00`, `0x01`: perioada (low si high)
- `0x02`: activare contor
- `0x03`, `0x04`: compare1 (low si high)
- `0x05`, `0x06`: compare2 (low si high)
- `0x07`: reset temporar pentru contor (scriere cu 1)
- `0x08`, `0x09`: valoarea curenta a contorului (doar citire)
- `0x0A`: prescaler
- `0x0B`: directie de numarare (up/down)
- `0x0C`: activare semnal PWM
- `0x0D`: functii suplimentare (doar bitii [1:0])

3.3 Logica de scriere

- Se actualizeaza registrele interne doar cand `write` este activ.
- Fiecare adresa are un efect specific: de exemplu, `ADDR_PERIOD_L` scrie partea joasa a perioadei.
- Semnalul `count_reset` este generat doar pentru un singur ciclu, daca se scrie 1 in adresa 0x07.
- La reset hardware (`!rst_n`), toate registrele revin la 0.

3.4 Logica de citire

- In functie de adresa primita, se trimite valoarea corespunzatoare pe `data_read`.
- Valorile de iesire sunt actuale si combinate: nu sunt intarziate.
- Daca se cere o adresa invalida, se returneaza 0x00.

3.5 Utilizare in sistem

- Modulul poate fi controlat din software prin scrierea de date in registre specifice.
- Permite citirea in timp real a valorii contorului pentru monitorizare sau sincronizare.
- Ofera flexibilitate pentru configurarea PWM-ului, fara interventie hardware.

4 Implementare `counter.v`

4.1 Mod general de functionare

- Genereaza o **baza de timp** incrementand/decrementand un contor pe 16 biti.
- Viteza de avans este controlata de **prescaler**: avans la fiecare $2^{prescale}$ cicluri de `clk`.
- Se opreste/porneste cu `en`, se reseteaza cu `count_reset` (doar contorul) sau cu `rst_n` (reset hardware).
- Cand ajunge la capat, face **wrap** controlat de `period`:
  - **UP**: $0, 1, ..., period \rightarrow 0 \rightarrow ..., 0, 1, ..., period \rightarrow 0 \rightarrow ...$
  - **DOWN**: $period, ..., 1, 0 \rightarrow period \rightarrow ...period, ..., 1, 0 \rightarrow period \rightarrow ...$

4.2 De ce am ales aceasta implementare

- **Prescaler pe puteri ale lui 2**: nu e costisitor din punct de vedere hardware (registru + comparatie cu $2^N - 1$), este stabil si usor de temporizat.
- **Wrap sincron pe `period`**: deoarece comparatia se evalueaza pe frontul lui `clk` comportamentul este determinist, fara glitch-uri.
- **Doua tipuri de reset**: `rst_n` (hardware) readuce imediat in starea initiala; `count_reset` realiniaza doar contorul/prescalerul, fara a atinge alte registre.

- **Freeze cu `en=0`**: opreste numararea si permite schimbari sigure din software; apoi repornire.

4.3 Explicatie implementare pas cu pas

Registri interni

- `cnt_r[15:0]`: valoarea curenta a contorului (iese pe `count_val`).
- `ps_cnt[8:0]`: prescalerul (numara pana la limita).

Limita prescalerului (`ps_lim`)

- Calculam $2^{prescale} - 1$.
- Cand `prescale = 0` $\Rightarrow$ `ps_lim = 0` (tick la fiecare `clk`).
- Pentru valori ≥ 8, facem *clamp* la 255 (divide by 256) ca sa nu depasim latimea prescalerului.

Generarea "tick"-ului

- Cat timp `en = 1`, `ps_cnt` urca de la 0 la `ps_lim`.
- Cand `ps_cnt == ps_lim` $\Rightarrow$ *tick* $= 1$ un ciclu; apoi `ps_cnt` revine la 0.

Avansul contorului (`cnt_r`)

- Se face **doar** la *tick*.
- Daca `upnotdown = 1` (UP):
  - Daca `cnt_r` $\geq$ `period` $\Rightarrow$ wrap la 0.
  - Altfel `cnt_r` $\leftarrow$ `cnt_r` $+ 1$.
- Daca `upnotdown = 0` (DOWN):
  - Daca `cnt_r` $= 0 \Rightarrow$ wrap la `period`.
  - Altfel `cnt_r` $\leftarrow$ `cnt_r` $- 1$.

Reseturi

- `!rst_n` sau `count_reset` pun `cnt_r = 0` si `ps_cnt = 0` (realiniere curata).

Enable

- `en = 0` $\Rightarrow$ nu se mai numara; `ps_cnt` si `cnt_r` raman neschimbate.

5 Implementare `pwm_gen.v`

5.1 Mod general de funcţionare

- Modulul `pwm_gen` generează un semnal PWM pe baza valorii externe `count_val`, a perioadei programabile `period` şi a pragurilor `compare1` / `compare2`.
- Logica PWM utilizează direct valoarea contorului, fără a introduce o fază internă separată.
- Modul de funcţionare este selectat prin `functions[1:0]`:
  - **Aliniat** (`functions[1]` = 0):
    - **stânga** (`functions[0]` = 0): PWM activ de la `count_val`=0 până la `compare1`;
    - **dreapta** (`functions[0]` = 1): PWM activ de la `compare1` până la `period`.
  - **Interval** (`functions` = 2'b10): PWM activ în intervalul [`compare1`, `compare2`), dacă `compare1 < compare2`.
- Modificările de configuraţie sunt aplicate doar în condiţii sigure, pentru a evita glitch-uri pe ieşire.
- Semnalul `pwm_en` controlează actualizarea ieşirii PWM; când este 0, ieşirea rămâne în ultima stare.

5.2 De ce am ales aceasta implementare

- **Snapshot al configuraţiei**: parametrii activi sunt copiaţi în registre interne doar la început de ciclu, la oprirea contorului sau când PWM este dezactivat.
- **Compatibilitate cu contoare externe**: logica PWM funcţionează direct pe `count_val`, fără presupuneri despre direcţia sau tipul contorului.
- **Validare praguri**: valorile `compare1` şi `compare2` sunt limitate la `period`.
- **Separare clară între configuraţie şi ieşire**: `pwm_en` permite modificarea sigură a configuraţiei fără efecte tranzitorii pe ieşire.

5.3 Explicaţia implementării pas cu pas

Regiştri interni

- `period_act`: copie activă a perioadei PWM.
- `c1_act, c2_act`: copii active ale pragurilor `compare1` şi `compare2`, limitate la `period`.

- `func_act[1:0]`: copierea activă a modului de funcţionare selectat.
- `prev_count_val`: valoarea anterioară a contorului, utilizată pentru detectarea opririi acestuia.
- `pwm_q`: registrul care reţine starea ieşirii PWM.

Detectarea condiţiilor de actualizare

- `start_of_cycle` este activ când `count_val` este 0, indicând începutul unui ciclu PWM.
- `counter_stopped` este adevărat când valoarea contorului nu se modifică între două cicluri consecutive de ceas.
- `cfg_update` este adevărat dacă:

$$\neg\texttt{pwm\_en} \vee \texttt{start\_of\_cycle} \vee \texttt{counter\_stopped}.$$

Actualizarea configuraţiei active

- Când `cfg_update` este activ, registrele interne sunt actualizate cu valorile de intrare.
- Pragurile `compare1` şi `compare2` sunt limitate la `period` pentru a preveni comparaţii invalide.

Ferestre PWM (logică combinaţională)

- **Aliniat stânga**: PWM este activ pentru $\texttt{count\_val} \leq \texttt{compare1}$, cu excepţia cazului `compare1 = 0`, când ieşirea este forţată la 0.
- **Aliniat dreapta**: PWM este activ pentru $\texttt{count\_val} \geq \texttt{compare1}$.
- **Interval**: PWM este activ dacă `compare1 < compare2` şi

$$\texttt{compare1} \leq \texttt{count\_val} < \texttt{compare2}.$$

Calcul final şi control ieşire

- Dacă `compare1 = compare2`, ieşirea PWM este forţată la 0, indiferent de modul selectat.
- `pwm_calc` reprezintă rezultatul combinational al ferestrei PWM.
- Pe frontul pozitiv al lui `clk`, dacă `pwm_en` este activ, `pwm_q` este actualizat cu `pwm_calc`.
- Când `pwm_en` este 0, `pwm_q` îşi păstrează valoarea anterioară.
- Ieşirea `pwm_out` este conectată direct la `pwm_q`.

6 Modificare `top.v`

- La sectiunea pentru `spi_bridge` am inversat numele porturilor `mosi/miso` deoarece, in varianta initiala, perifericul nu primea MOSI corect si nu furniza MISO corect, fapt pentru care am avut probleme la trecerea testelor din testbench.