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` este un SPI slave simplificat care permite comunicarea între un master SPI și perifericul intern al sistemului.
- Funcţionează conform protocolului SPI Mode 0 (CPOL=0, CPHA=0): MOSI este citit pe frontul crescător al SCLK, iar MISO este transmis pe frontul descrescător.
- Datele sunt preluate MSB-first, asigurând alinierea corectă cu masterul SPI.
- La recepţia fiecărui byte complet, semnalul `byte_sync` este generat ca impuls de un ciclu clk, indicând perifericului că datele sunt valide.
- Transferuri multiple sunt suportate pe durata unui CS activ, iar la ridicarea CS registrele de shift și contorul de biţi sunt resetate.

1.2 De ce am ales această implementare

- Simplitate și claritate didactică: codul este uşor de urmărit, fără complexitate inutilă.
- Sincronizare simplă între `sclk` şi `clk`: nu este necesar FIFO sau sincronizare complexă, deoarece ceasurile sunt sincrone.
- Shift registers separate pentru intrare (`shift_in`) şi ieşire (`shift_out`), claritatea codului şi respectarea MSB-first.
- Semnalul `byte_sync` derivat din `byte_done` folosind registrul `bd_d1`, generând un impuls sigur de un ciclu clk pentru periferic.
- Permite transferuri consecutive fără pierderi de date şi menţine implementarea didactică şi uşor de testat.

1.3 Explicaţie implementare pas cu pas

Regiştri interni

- `shift_in [7:0]`: registru de shift pentru datele recepţionate de pe linia MOSI.
- `shift_out [7:0]`: registru de shift pentru datele transmise pe MISO.
- `bit_count [2:0]`: contor care urmareşte câti biţi au fost recepţionaţi.
- `byte_done`: semnal intern care indică că s-a recepţionat un byte complet.
- `bd_d1`: registru intermediar pentru detectarea tranziţiei `byte_done` în domeniul `clk`.

Recepţia datelor de pe MOSI

- Pe frontul crescător al SCLK, se preia un bit de pe MOSI şi se adaugă la `shift_in` (MSB-first).
- Contorul `bit_count` este incrementat pentru fiecare bit recepţionat.
- Când `bit_count` ajunge la 7, `byte_done` devine 1, indicând că un byte complet a fost recepţionat.
- Dacă `cs_n` devine 1, contorul şi `byte_done` sunt resetate.

Transmiterea datelor pe MISO

- La începutul transferului (falling edge `cs_n`), `shift_out` se încarcă cu valoarea `data_out`.
- Pe fiecare front descrescător al `sclk`, registrul se shift-ează spre stânga şi bitul cel mai semnificativ (`shift_out[7]`) este transmis pe MISO.
- Această abordare asigură transmiterea corectă a datelor către master, bit cu bit, MSB-first.

Generarea semnalului `byte_sync`

- `byte_done` este generat pe `sclk`.
- Registrul `bd_d1` memorează valoarea anterioară a lui `byte_done` pe frontul pozitiv al ceasului periferic (`clk`).
- `byte_sync` este activat când `byte_done = 1` şi `bd_d1 = 0`, creând un impuls de un ciclu `clk` care indică perifericului că `data_in` este valid.

Gestionarea `cs_n` şi resetarea modulului

- Când `cs_n` devine 1, contorul `bit_count` şi registrul `shift_in` sunt resetate.

- `shift_out` se pregăteşte pentru următorul transfer încarcându-se cu `data_out` curent.
- Această metodă permite masterului să trimită mai multe byte-uri consecutive fără pierderi de date şi asigură consistenţa semnalului `byte_sync`.

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 functionare

- Genereaza **semnalul PWM** pe baza valorii `count_val` si a pragurilor programabile `compare1`/`compare2`, respectand `period`.
- Modurile sunt selectate prin `functions[1:0]`:
  - **Aliniat** (`functions[1]` $= 0$):
    - **stanga** (`functions[0]` $= 0$): PWM $= 1$ pentru faza $<$ `compare1`;
    - **dreapta** (`functions[0]` $= 1$): PWM $= 1$ pe ultimele `compare1` trepte din ciclu.
  - **Nealiniat** (`functions[1]` $= 1$): PWM $= 1$ in $[$`compare1`, `compare2`$)$ daca `compare1` $<$ `compare2`.
- Schimbarile de configuratie se aplica **la limita de ciclu** (overflow/underflow) sau cand `pwm_en` $= 0$, pentru a evita glitch-uri.
- `pwm_en` controleaza actualizarea iesirii: cand este 0, iesirea ramane in ultima stare (sau poate fi fortata 0).

5.2 De ce am ales aceasta implementare

- **Snapshot al configuratiei la inceput de ciclu**: asigura comportament determinist si fara pulsuri partiale cand registrele se modifica in timpul perioadei.
- **Faza locala independenta de directie**: folosesc o *faza* interna care avanseaza la fiecare pas al contorului; astfel, logica PWM nu depinde de UP/DOWN.
- **Validare praguri**: limitez `compare1`/`compare2` la `period` pentru a preveni comparatii invalide.
- **Separare clara intre configuratie si iesire**: `pwm_en` decupleaza actualizarea iesirii, util pentru schimbari in siguranta.

5.3 Explicatie implementare pas cu pas

Registri interni

- `period_act, comp1_act, comp2_act`: copii active ale `period`, `compare1`, `compare2`.
- `func_act[1:0]`: copierea activa a `functions[1:0]` (mod aliniat/nealiniat; stanga/dreapta).
- `phase[15:0]`: faza locala in cadrul ciclului, $0 \ldots$ `period_act`.

- `count_d[15:0]`: esantionul anterior al `count_val` (pentru a detecta pasul si wrap-ul).
- `pwm_q`: registrul de iesire care conduce `pwm_out`.

Detectia inceputului de ciclu

- Definim `cycle_wrap` drept trecerea **period** $\rightarrow$ **0** sau **0** $\rightarrow$ **period** intre doi pasi consecutivi.
- `tick_seen` este adevarat cand `count_val` s-a schimbat fata de `count_d` (pas al contorului).

Reimprospatarea configuratiei

- La `cycle_wrap` sau cand `pwm_en` $= 0$, se copiaza `period`, `compare1`, `compare2`, `functions[1:0]` in registrii activi si se reseteaza `phase` la 0.
- `compare1`/`compare2` sunt limitate la `period`.

Evolutia fazei

- Daca `tick_seen` $= 1$ si `phase` $<$ `period_act`, atunci `phase` $\leftarrow$ `phase` $+ 1$.
- La `cycle_wrap`, `phase` se reseteaza la 0.

Ferestre PWM (logica combinationala)

- **Aliniat stanga** (`func_act[1]`=0, `func_act[0]`=0): PWM $= 1$ daca `phase` $<$ `comp1_act`.
- **Aliniat dreapta** (`func_act[1]`=0, `func_act[0]`=1): PWM $= 1$ pentru ultimele `comp1_act` trepte:

$$\texttt{phase} \geq \max\{0, \texttt{period\_act} - \texttt{comp1\_act}\}.$$

- **Nealiniat** (`func_act[1]`=1): PWM $= 1$ in intervalul

$$[\texttt{comp1\_act}, \texttt{comp2\_act}) \quad \text{daca } \texttt{comp1\_act} < \texttt{comp2\_act}.$$

Control iesire

- `pwm_calc` este rezultatul combinational al ferestrei PWM in functie de modul selectat.
- Pe frontul lui `clk`: daca `pwm_en`=1, atunci `pwm_q` $\leftarrow$ `pwm_calc`; altfel `pwm_q` ramane neschimbat (sau poate fi fortat 0).
- `pwm_out` este conectat la `pwm_q`.