Documentație tehnică detaliată
Implementare PWM Generator

Constantin Călin-Ștefan, Șerban Dan-Adrian, Vasile Corneliu - 333AA

24.11.2025

1 Introducere

Acest document include descrierea pas-cu-pas și explicațiile aferente implementării modulelor `counter.v`, `instr_dcd.v`, `regs.v` `pwm_gen.v`, `spi_bridge.v`, `top.v` realizate în cadrul acestui proiect. Se omit explicațiile privind arhitectura modulului, acestea regăsindu-se printre materialele oferite.

2 Modulele și explicațiile lor

2.1 0) top.v — Interconectare module + pipeline pentru semnale SPI (versiune actualizată)

Scopul modulului

Modulul `top.v` are rolul de integrare a tuturor blocurilor: `spi_bridge`, `instr_dcd`, `regs`, `counter`, `pwm_gen`. În varianta actualizată, `top.v` mai introduce și un **pipeline de 1 ciclu** pentru semnalele de byte provenite din SPI, astfel încât decodarea să se facă pe valori stabile în domeniul `clk`.

Cod relevant

```
// raw SPI byte interface (din spi_bridge)
wire byte_sync_w;
wire [7:0] data_in_w;

// catre spi_bridge (pentru citiri SPI)
wire [7:0] data_out;

// pipeline in domeniul clk
reg byte_sync;
reg [7:0] data_in;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        byte_sync <= 1'b0;
        data_in <= 8'h00;
    end else begin
        byte_sync <= byte_sync_w;
        if (byte_sync_w)
            data_in <= data_in_w;
    end
end

spi_bridge i_spi_bridge (
    .clk(clk),
    .rst_n(rst_n),
    .sclk(sclk),
    .cs_n(cs_n),
    .mosi(miso), // TB: master->slave
    .miso(mosi), // TB: slave->master
    .byte_sync(byte_sync_w),
    .data_in(data_in_w),
    .data_out(data_out)
);

instr_dcd i_instr_dcd (
```

```
    .clk(clk),
    .rst_n(rst_n),
    .byte_sync(byte_sync),
    .data_in(data_in),
    .data_out(data_out),
    .read(read),
    .write(write),
    .addr(addr),
    .data_read(data_read),
    .data_write(data_write)
);
```

Explicație pas-cu-pas

1. **Instanțierea modulelor și traseul datelor**
   - `spi_bridge` transformă fluxul serial SPI în octeți (`data_in_w`) și generează un puls (`byte_sync_w`) la finalizarea fiecărui byte.
   - `instr_dcd` interpretează octeții și generează semnale de acces către `regs` (`read`, `write`, `addr`, `data_write`).
   - `regs` păstrează valorile programabile (period, compare1, compare2, functions, etc.) și furnizează `data_read` pentru citiri.
   - `counter` produce `count_val` în funcție de `period`, `prescale` și `upnotdown`.
   - `pwm_gen` generează `pwm_out` folosind `count_val` și comparatoarele.
2. **De ce este necesar pipeline-ul pentru `byte_sync` și `data_in`?**
   - `spi_bridge` produce `byte_sync_w` în domeniul `clk` (prin CDC intern), însă `data_in_w` este capturat concomitent cu acest puls.
   - Dacă `instr_dcd` ar folosi direct `byte_sync_w` și `data_in_w` în același ciclu, pot apărea situații de timing/ordine în care decoderul vede `byte_sync` activ înainte ca `data_in` să fie stabil.
   - Pipeline-ul de 1 ciclu garantează că atunci când `byte_sync` este 1, `data_in` a fost deja latched și este stabil pentru decodare.
3. **Funcționarea pipeline-ului**
   - `byte_sync <= byte_sync_w;` propagă pulsul cu întârziere de un ciclu.
   - `if (byte_sync_w) data_in <= data_in_w;` salvează octetul exact atunci când este declarat valid de SPI bridge.
   - Rezultatul: `instr_dcd` lucrează pe `byte_sync` și `data_in` curățate și stabile.
4. **Conectarea semnalului `data_out` (citiri SPI)**
   - `data_out` este ieșirea din `instr_dcd` către `spi_bridge`.
   - La o citire SPI, `instr_dcd` pune valoarea `data_read` în `data_out`, iar `spi_bridge` o transmite serial pe MISO.
   - Astfel, traseul pentru citire este: `regs.data_read` → `instr_dcd.data_out` → `spi_bridge` → MISO.
5. **Diferența principală față de varianta inițială**

- În varianta inițială, legăturile dintre SPI și decoder nu erau făcute complet (ex. `byte_sync` neconectat, `data_in` nealimentat), ceea ce făcea ca registrele să nu fie programate corect.
- În varianta actualizată, `top.v` este responsabil explicit de conectarea acestor semnale și de stabilizarea lor, obținând o funcționare deterministă.

2.2 1) counter.v — Prescaler, Update, Wrap (explicație detaliată)

Explicație pas-cu-pas

1. **Declarațiile:**

```
reg [31:0] presc_cnt;
reg [31:0] presc_limit;
```

   - `presc_cnt` (32b) este contorul intern care numără ciclurile de ceas înainte de a actualiza `count_val`.
   - `presc_limit` (32b) reține pragul la care prescaler-ul declanșează actualizarea numărătorului principal.

2. **Blocul combinatorial** `always @(*):` calculează `presc_limit`.

```
always @(*) begin
    if (prescale >= 31)
        presc_limit = 32'hFFFFFFFF;
    else
        presc_limit = (1 << prescale);
    if (presc_limit == 0) presc_limit = 1;
end
```

   - `if (prescale >= 31) presc_limit = 0xFFFFFFFF;` - Motiv: `prescale` este un octet; shift-ul `(1 << prescale)` nu este definit/sigur pentru exponent mare. Aici limităm la o valoare pentru a evita overflow-uri.
   - `else presc_limit = (1 << prescale);` - Interpretează `prescale` ca exponent: prescaler = $2^{prescale}$.
   - `if (presc_limit == 0) presc_limit = 1;` - Protecție suplimentară: asigurăm că pragul nu devine 0 (evită update continuu).

3. **Blocul sincron** `always @(posedge clk ...):`
   - Functionalitățile de reset și enable setează valorile dorite la 0.
   - Modul activ(en):
     - Se verifică counter-ul prescaler-ului în comparație cu limita sa:
       - Pentru `presc_limit = 1` (prescale=0) se intră imediat deoarece `0+1 >= 1` adevărat → se actualizează la fiecare ciclu.
       - Pentru `presc_limit = 4` (prescale=2), `presc_cnt` va avea valorile 0,1,2,3 → la 3: `3+1 >=4` → actualizare (deci la fiecare 4 cicluri).
     - Dacă pragul e atins, `presc_cnt` se resetează și `count_val` este incrementat/decrementat în funcție de `upnotdown`, cu regula de wrap:
       - la up: dacă `count_val >= period` → punem 0 (wrap).

```
if (upnotdown) begin
    // count up
    if (count_val >= period) begin
        // overflow -> wrap to 0
        count_val <= 16'h0000;
    end else begin
        count_val <= count_val + 1'b1;
    end
end
```

  - la down: dacă `count_val == 0` → punem `period` (wrap).
- Dacă pragul nu e atins, incrementăm doar `presc_cnt`.

2.3 2) instr_dcd.v — FSM pentru înregistrări SPI

Explicație pas-cu-pas

1. **Stări:**
   - `S_SETUP`: așteaptă byte-ul de setup.
   - `S_DATA`: așteaptă byte-ul de date și efectuează operația.
2. **SETUP byte parsing:**
   - `rw <= data_in[7]`: memorăm dacă e write(1) sau read(0).
   - `base_addr <= data_in[5:0]`: adresă de bază.
3. **DATA phase:**
   - Dacă `rw==1` (write): punem `data_write=data_in` și pulsam `write`.
   - Dacă `rw==0` (read): citim `data_read` din `regs` și o plasăm în `data_out`; pulsăm `read`.
   - read-ul își pregătește `data_out` încă din CMD phase (ca SPI să aibă byte-ul disponibil pentru transferul următor)
   - `read_stb/write_stb` există ca să țină strobul vizibil un ciclu complet

```
S_CMD: begin
  rw <= data_in[7];
  latched_addr <= data_in[5:0];
  addr <= data_in[5:0];

  if (!data_in[7]) begin
    data_out <= data_read;
    read_stb <= 1'b1;
  end
  state <= S_DATA;
end
...

S_DATA: begin
  addr <= latched_addr;
  if (rw) begin
    data_write <= data_in;
    write_stb <= 1'b1;
  end
  state <= S_CMD;
end
```

4. **Sincronizare / timing:** `byte_sync` vine de la `spi_bridge` după recepția completă a octetului; astfel decoderul operează sincron și generează semnale pentru `regs`.

2.4 3) regs.v — Harta registrelor și logica de acces

Maparea adreselor

```
// Mapare (addresses in hex):
// 0x00 -> PERIOD LSB
// 0x01 -> PERIOD MSB
// 0x02 -> COUNTER_EN (bit0)
// 0x03 -> COMPARE1 LSB
// 0x04 -> COMPARE1 MSB
// 0x05 -> COMPARE2 LSB
// 0x06 -> COMPARE2 MSB
// 0x07 -> COUNTER_RESET (write-only)
// 0x08 -> COUNTER_VAL LSB (read-only)
// 0x09 -> COUNTER_VAL MSB (read-only)
// 0x0A -> PRESCALE
// 0x0B -> UPNOTDOWN (bit0)
// 0x0C -> PWM_EN (bit0)
// 0x0D -> FUNCTIONS
```

Explicații esențiale

- **Implementare sincronă:** toate registrele sunt flip-flop-uri care se actualizează pe `posedge clk`.
- **WRITE handling:**
  - Când `write` e activ, un `case(addr)` decide ce byte să actualizeze.
  - Pentru registre pe 16 biți, LSB și MSB scrise separat actualizează părți din același registru (ex: `period[7:0]` și `period[15:8]`).
  - `COUNTER_RESET` la adresă 0x07 generează un puls setând `count_reset<=1` pentru acel ciclu.
- **READ combinatorial:**
  - `data_read` se calculează permanent din addr.
  - decoderul poate încărca `data_out` imediat.

2.5 4) pwm_gen.v — Generare PWM (explicație detaliată, versiune actualizată)

Cod relevant

```
reg [15:0] count_next;

always @(*) begin
    if (count_val >= period)
        count_next = 16'h0000;
    else
        count_next = count_val + 16'h0001;
end

always @(posedge clk or negedge rst_n) begin
```

```
    if (!rst_n) begin
        pwm_out <= 1'b0;
    end else if (!pwm_en) begin
        pwm_out <= 1'b0;
    end else if (compare1 == compare2) begin
        pwm_out <= 1'b0;
    end else begin
        case (functions[1:0])
            2'b00: pwm_out <= (compare1 != 16'h0000) && (count_next <= compare1);
            2'b01: pwm_out <= (count_next >= compare1);
            2'b10: pwm_out <= (count_next >= compare1) && (count_next < compare2);
            default: pwm_out <= 1'b0;
        endcase
    end
end
```

Explicație pas-cu-pas

1. **Scopul modulului:** `pwm_gen` generează semnalul `pwm_out` în funcție de:
   - `count_val` (valoarea curentă a contorului),
   - `period` (valoarea de wrap/terminal count),
   - `compare1` și `compare2`,
   - `functions[1:0]` (modul PWM),
   - `pwm_en` (enable).
2. **Calculul lui `count_next` (corecție de fază):**

```
always @(*) begin
    if (count_val >= period)
        count_next = 16'h0000;
    else
        count_next = count_val + 16'h0001;
end
```

   - `count_next` reprezintă valoarea **anticipată** a contorului pentru următorul pas.
   - Motivul: `counter.v` și `pwm_gen.v` sunt ambele evaluate pe `posedge clk`.
   - Dacă PWM ar folosi doar `count_val`, deciziile PWM apar „decalate" cu 1 tick (off-by-one).
   - Folosind `count_next`, PWM se aliniază cu valorile pe care le „vede" efectiv testarea la fiecare tick.
3. **Comportamentul la reset și când PWM este dezactivat:**
   - La reset (`!rst_n`) se forțează `pwm_out=0`.
   - Când `pwm_en=0`, se forțează tot `pwm_out=0` (nu se păstrează starea veche).

```
if (!rst_n) begin
    pwm_out <= 1'b0;
end else if (!pwm_en) begin
    pwm_out <= 1'b0;
end
```

4. **Caz limită:** `compare1 == compare2` **(fereastră de lățime zero):**

```
else if (compare1 == compare2) begin
    pwm_out <= 1'b0;
end
```

   - Dacă `compare1==compare2`, intervalul „între comparatoare" are lățime 0.
   - În acest caz, definim PWM ca fiind mereu 0 pentru a evita impulsuri nedorite.
   - Această regulă clarifică funcționarea pentru cazul-limită și previne generarea unei „linii" de un tick din cauza comparatoarelor egale.

5. **Selecția modului PWM (functions[1:0]):**

   - `2'b00` – ALIGN_LEFT
   - `2'b01` – ALIGN_RIGHT
   - `2'b10` – RANGE_BETWEEN_COMPARES
   - `default` – ieșire 0

6. **Mod 2'b00: ALIGN_LEFT (high de la început până la compare1):**

```
2'b00: pwm_out <= (compare1 != 16'h0000) && (count_next <= compare1);
```

   - PWM este **HIGH** pentru `count_next` în intervalul `[0, compare1]`.
   - Condiția `(compare1 != 0)` este un caz special:
     - dacă `compare1=0`, se dorește ca PWM să fie 0 (nu un singur tick HIGH la start).

7. **Mod 2'b01: ALIGN_RIGHT (high de la compare1 până la sfârșitul perioadei):**

```
2'b01: pwm_out <= (count_next >= compare1);
```

   - PWM este **HIGH** pentru `count_next` în intervalul `[compare1, period]`.
   - Practic: cu cât `compare1` este mai mare, cu atât duty-cycle scade.

8. **Mod 2'b10: RANGE BETWEEN COMPARES (high doar între compare1 și compare2):**

```
2'b10: pwm_out <= (count_next >= compare1) && (count_next < compare2);
```

   - PWM este **HIGH** doar în intervalul `[compare1, compare2)`.
   - Observație: limita superioară este **exclusivă** (`< compare2`), ceea ce produce o fereastră cu lungime `compare2-compare1`.

9. **Default:**

```
default: pwm_out <= 1'b0;
```

   - Pentru valori neacceptate ale `functions[1:0]`, ieșirea este forțată la 0.

2.6 5) spi_bridge.v — Interfață SPI Slave (versiune actualizată)

Scopul modulului

Modulul `spi_bridge` implementează o interfață SPI slave compatibilă cu:

- MSB first
- CPOL = 0
- CPHA = 0

și realizează transferul sigur al octeților între domeniul `sclk` (SPI) și domeniul `clk` (periferic intern).

Cod relevant

```
// toggle when a full byte is received (sclk domain)
reg byte_toggle_s;

// synchronize toggle into clk domain
reg toggle_ff1, toggle_ff2;

always @(posedge clk or negedge rst_n) begin
    toggle_ff1 <= byte_toggle_s;
    toggle_ff2 <= toggle_ff1;
    byte_sync <= (toggle_ff2 ^ toggle_ff1);
    if (toggle_ff2 ^ toggle_ff1)
        data_in <= byte_buf_s;
end
```

Explicație pas-cu-pas

1. **Separarea domeniilor de ceas**

   - `sclk` este ceasul SPI, complet asincron față de `clk`.
   - Recepția și transmiterea SPI se realizează în **domeniul sclk**.
   - Semnalele `byte_sync` și `data_in` sunt sincronizate în **domeniul clk**.

   Această separare elimină condițiile de metastabilitate și face modulul robust chiar dacă `sclk` este apropiat sau mai rapid decât `clk`.

2. **Inițializare la selecția slave-ului (CS low)**

   - La tranziția `cs_n: 1 -> 0`:
     - `bitcnt_s` este setat la 7 (MSB first),
     - `shift_out_s` este încărcat cu `data_out`,
     - `miso` este setat la `data_out[7]`.
   - Astfel, primul bit MISO este stabil înainte de primul front rising al lui `sclk`.

3. **Recepția datelor (MOSI) pe rising edge**

```
always @(posedge sclk) begin
    shift_in_s[bitcnt_s] <= mosi;
    if (bitcnt_s == 0) begin
        byte_buf_s <= {shift_in_s[7:1], mosi};
        byte_toggle_s <= ~byte_toggle_s;
```

```
        bitcnt_s <= 3'd7;
    end else begin
        bitcnt_s <= bitcnt_s - 1'b1;
    end
end
```

- MOSI este citit pe frontul rising al lui `sclk` (CPHA=0).
- Bitul este salvat în poziția corespunzătoare `bitcnt_s`.
- La recepția completă a unui octet:
  - octetul este stocat în `byte_buf_s`,
  - `byte_toggle_s` este inversat (toggle).

4. **Transmiterea datelor (MISO) pe falling edge**

```
always @(negedge sclk) begin
    shift_out_s <= {shift_out_s[6:0], 1'b0};
    miso <= shift_out_s[6];
end
```

- MISO este actualizat pe falling edge.
- Acest lucru garantează că bitul este stabil înainte de următorul rising edge, când masterul îl va citi.

5. **Detectarea recepției unui octet complet (byte_sync)**

```
byte_sync <= (toggle_ff2 ^ toggle_ff1);
```

- `byte_toggle_s` este sincronizat prin două flip-flop-uri (`toggle_ff1`, `toggle_ff2`).
- Orice schimbare a toggle-ului produce un impuls `byte_sync` de un ciclu `clk`.
- Această tehnică este cunoscută ca **toggle-based CDC**.

6. **Capturarea sigură a octetului în domeniul clk**

```
if (byte_sync)
    data_in <= byte_buf_s;
```

- La detectarea impulsului `byte_sync`, octetul finalizat este copiat în `data_in`.
- De aici înainte, datele sunt stabile și pot fi utilizate de `instr_dcd`.

7. **Avantajele față de implementarea inițială**

- Elimină complet dependența de `prev_sclk` în domeniul `clk`.
- Nu mai există sampling asincron al lui `sclk`.
- Funcționează corect indiferent de raportul dintre `clk` și `sclk`.
- Respectă bune practici de CDC (Clock Domain Crossing).

Explicație detaliată

1. **Inițializare și activare (CS low):**
   - La tranzitia CS high->low, `active` devine 1, `shift_out<=data_out`, `bitcnt=7` și `miso=MSB`.
   - Asta pregătește transmisia MSB-first și asigură că MISO este stabil înainte de primul front de ceas.
2. **Rising edge (sample MOSI):**
   - Detectăm rising cu `prev_sclk==0 && sclk==1`.
   - Pe rising citim bitul MOSI și îl plasăm în poziția `bitcnt`.
   - Dacă e ultimul bit (bitcnt==0), construim octetul finalizat, publicăm `data_in` și pulsăm `byte_sync`.
   - Resetăm `bitcnt` pentru următorul octet și reîncărcăm `shift_out` cu noul `data_out`.
3. **Falling edge (update MISO):**
   - Detectăm falling cu `prev_sclk==1 && sclk==0`.
   - Actualizăm `shift_out` (shift-left) și scriem pe `miso` bitul următor (`shift_out[6]`).
   - Această schemă schimbă MISO pe falling, lăsând bitul stabil pe rising pentru ca masterul să-l citească.
4. **CS high (idle/reset):**
   - Resetăm `active` și registrii locali, reîncărcăm `shift_out` cu `data_out` pentru o următoare sesiune.

3 Diferențe față de implementarea inițială

Această secțiune evidențiază modificările majore realizate față de implementarea inițială, împreună cu motivația tehnică pentru fiecare schimbare. Scopul principal al acestor modificări a fost obținerea unei funcționări deterministe, robuste la diferențe de ceas și compatibile cu testbench-ul furnizat.

3.1 1) top.v — Corectarea interconectării și introducerea pipeline-ului

Implementare inițială:

- Semnalele `byte_sync` și `data_in` nu erau conectate între `spi_bridge` și `instr_dcd`.
- `instr_dcd` primea un `byte_sync` neconectat, deci FSM-ul de decodare nu era niciodată activat.
- Nu exista niciun mecanism de stabilizare a octetului recepționat înainte de decodare.

Implementare actualizată:

- `byte_sync` și `data_in` sunt conectate explicit.
- Se introduce un pipeline de 1 ciclu în domeniul `clk` pentru aceste semnale.
- Decoderul lucrează exclusiv pe date stabile și sincronizate.

Motivație: Fără această modificare, registrele nu erau programate corect prin SPI, iar modulele dependente (counter, pwm_gen) rămâneau în stări default.

—

3.2 2) spi_bridge.v — Separare clară de domenii de ceas (CDC)

Implementare inițială:

- Detectarea fronturilor lui `sclk` se făcea în domeniul `clk` folosind `prev_sclk`.
- Semnalul `sclk` era eșantionat asincron, riscând metastabilitate.
- Funcționarea depindea de relația de frecvență dintre `clk` și `sclk`.

Implementare actualizată:

- Recepția și transmiterea SPI se fac în domeniul `sclk`.
- Se utilizează o tehnică de sincronizare bazată pe toggle pentru transferul evenimentelor în domeniul `clk`.
- Se elimină complet eșantionarea asincronă a lui `sclk`.

Motivație: Noua implementare este robustă la orice raport între `clk` și `sclk` și respectă bune practici de proiectare pentru CDC (Clock Domain Crossing).

—

3.3 3) instr_dcd.v — Simplificarea protocolului SPI

Implementare inițială:

- Byte-ul de comandă conținea un bit `highlow` pentru selectarea MSB/LSB.
- Adresa efectivă era calculată ca `base\_addr + highlow`.
- Protocolul SPI presupunea acces pe jumătăți de registru.

Implementare actualizată:

- Adresa din byte-ul de comandă este utilizată direct.
- Fiecare adresă corespunde unui octet din `regs`.
- FSM-ul este redus la două faze simple: CMD și DATA.

Motivație: Testbench-ul furnizat accesează registrele la nivel de octet și nu utilizează bitul high/low. Simplificarea elimină conversii inutile și potențiale erori de adresare.

—

3.4 4) pwm_gen.v — Trecerea de la logică bazată pe toggle la logică comparativă

Implementare inițială:

- PWM-ul era generat prin toggle la evenimente (`count==0`, `count==compare1`, `count==compare2`).
- Necesita variabile de stare (`last_count_was_zero`).
- Comportamentul era sensibil la cazuri-limită (ex: `compare1=0`, `compare1=compare2`).

Implementare actualizată:

- PWM-ul este o funcție pur combinatorie de `count_next`.
- Se elimină complet stările interne.
- Se definesc explicit cazurile-limită (compare egal, compare1=0).

Motivație: Noua abordare este mai simplă, deterministă și ușor de verificat, producând exact duty-cycle-ul așteptat de testbench.

—

3.5 5) Impact general

- Sistemul funcționează corect pentru toate modurile PWM testate.
- Comportamentul este independent de detalii de timing necontrolabile.
- Codul este mai ușor de întreținut și extins.