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` realizate în cadrul acestui proiect. Se omit explicațiile privind arhitectura modulului, acestea regăsindu-se printre materialele oferite.

2 Modulele și explicațiile lor

2.1 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`.

2.2 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).
   - `highlow <= data_in[6]`: bit ce indică MSB/LSB.
   - `base_addr <= data_in[5:0]`: adresă de bază.
   - `addr <= base_addr + (highlow ? 1 : 0)`: transformăm în adresă de octet folosită în `regs`.
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`.

```
if (rw) begin
    data_write <= data_in;
    write <= 1'b1;
end else begin
    data_out <= data_read;
    read <= 1'b1;
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`.

2.3 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 handling:**
  - Când `read` e activ, `data_read` primește valoarea corespunzătoare adresei (de ex. 0x08/0x09 citesc `counter_val`).
  - Adresele nealocate returnează 0.

2.4 4) pwm_gen.v — Generare PWM (explicație detaliată)

Cod relevant

Explicație pas-cu-pas

1. **Initializare:** la reset `pwm_out=0`. `last_count_was_zero` se folosește pentru a detecta începutul perioadei.
2. **pwm_en=0:** În cazul acesta, outputul este blocat în starea în care se afla. Nu se intră în blocul else ce descrie funcționalitatea.

```
else if (unaligned) begin
    if (count_val == compare1) begin
        pwm_out <= 1'b1;
    end else if (count_val == compare2) begin
        pwm_out <= 1'b0;
    end
    last_count_was_zero <= (count_val == 16'h0000);
end else begin
    if (count_val == 16'h0000) begin
        pwm_out <= (align_bit == 1'b0) ? 1'b1 : 1'b0;
```

```
    end else if (count_val == compare1) begin
        pwm_out <= ~pwm_out;
    end
```

3. **Mod nealiniat:**
   - la `count_val==compare1`: setăm 1.
   - la `count_val==compare2`: setăm 0.
   - Aceasta creează o fereastră [compare1, compare2).
4. **Mod aliniat:**
   - la începutul perioadei (`count_val==0`): setăm starea inițială conform align (left -> 1, right -> 0).
   - la `count_val==compare1`: toggle `pwm_out`.

2.5 5) spi_bridge.v — Detalii pentru interfata SPI (explicație detaliată)

Cod relevant

```
    if (prev_sclk == 1'b0 && sclk == 1'b1) begin
        shift_in[bitcnt] <= mosi;
        if (bitcnt == 3'd0) begin
            data_in <= {shift_in[7:1], mosi};
            byte_sync <= 1'b1;
            bitcnt <= 3'd7;
            shift_out <= data_out;
            miso <= data_out[7];
        end else begin
            bitcnt <= bitcnt - 1'b1;
        end
    end

    if (prev_sclk == 1'b1 && sclk == 1'b0) begin
        shift_out <= {shift_out[6:0], 1'b0};
        miso <= shift_out[6];
    end
end else begin
    active <= 1'b0;
    bitcnt <= 3'd7;
    shift_in <= 8'h00;
    shift_out <= data_out;
    miso <= data_out[7];
end

prev_sclk <= sclk;
end
```

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.