Descrierea modulului *SPI Bridge*

1. Scopul modulului

Modulul `spi_bridge` este un SPI Slave care conecteaza un Master extern cu logica interna a sistemului. Acesta asigura:

- conversia din serial in paralel pentru datele primite (MOS -> data_in)
- conversia din paralel in serial pentru datele transmise (data_out -> MISO)
- sincronizarea semnalelor asincrone SPI cu clock-ul sistemului (clk)

2. Interfata modulului

Semnale principale

- clk (input): ceasul de sistem, toata logica interna este sincronizata cu el
- rst_n (input): reset asincron, activ pe 0

Interfata SPI

- sclk (input): SPI clock de la Master
- cs_n (input): Chip Select activ pe 0
- mosi (input): Master Out Slave In
- miso (output): Master In Slave Out

Interfata interna

- byte_sync (output): puls de 1 ciclu clk cand un octet a fost receptionat complet
- data_in (output, 8 biti): octetul receptionat, valid cand byte_sync = 1
- data_out (input, 8 biti): octetul ce trebuie transmis catre Master

3. Functionare

3.1. Sincronizarea semnalelor (CDC)

Semnalele sclk, cs_n si mosi sunt asincrone fata de clk, de aceea sunt trecute prin doua registre de sincronizare pentru a preveni metastabilitatea. Fronturile lui sclk sunt detectate comparand starea curenta cu cea anterioara.

3.2. Receptia datelor (RX)

Datele sunt trimise pe frontul crescator al sclk sincronizat

Bitii sunt shiftati MSB-first intr-un registru intern

Un contor tine evidenta celor 8 biti

Dupa 8 biti, registrul se copiaza in data_in si byte_sync pulseaza

Cand cs_n revine High, contorul si receptia se reseteaza

3.3. Transmisia datelor (TX)

Cand cs_n este High, registrul de transmisie se incarca cu data_out

Pe fiecare front descrescator sclk sincronizat se trimite cate 1 bit pe MISO

Transmisia este MSB-first

Descrierea modulului *instruction decoder*

Semnale de intrare si de iesire

- `clk` - semnalul de ceas al sistemului
- `rst_n` - reset asincron activ pe 0
- `byte_sync` - semnalul care confirma primirea unui nou byte de la interfata SPI
- `data_in` - datele primite de la interfata SPI
- `data_out` - datele transmise catre interfata SPI
- `read` - enable citire din registrii
- `write` - enable scriere in registrii
- `addr` - adresa accesata in registrii
- `data_read` - datele citite din registrii
- `data_write` - datele scrise in registrii

Modulul *instruction decoder* implementeaza un decodor de instructiuni pentru interfata SPI. Acesta este responsabil pentru interpretarea secventelor de bytes primite si trimiterea datelor catre registri sau catre interfata SPI.

Modulul functioneaza in doua faze: faza de setup si faza de transmitere de date. Folosind o variabila interna `state` modulul memoreaza faza curenta (`0` pentru faza de setup si `1` pentru faza de date).

Interfata SPI trimite catre decodor semnalul `byte_sync` care anunta sosirea unui nou byte.

Faza de setup

In faza de setup, semnalul `data_in` este codificat astfel: bitul **7** semnifica tipul instructiunii ce urmeaza sa fie facuta (`1` pt write si `0` pt read), bitul **6** reprezinta zona din registru unde va fi facuta instructiunea (`1` pt [15:8] si `0` pt [7:0]), iar restul de biti **[5:0]** reprezinta adresa registrului.

Decodorul seteaza adresa registrului in functie de bitul **6** sau *high/low*. Acesta verifica daca adresa trimisa face parte dintr-un registru pe 16 biti sau de 8 biti.

In cazul in care adresarea se face pe unul dintre registrii `PERIOD`, `COMPARE1`, `COMPARE2` sau `COUNTER_VAL` (registrii pe 16 biti), decodorul de instructiuni seteaza adecvat adresa:

- pentru *high/low* `0` se pastreaza adresa de baza
- pentru *high/low* `1` se seteaza adresa de baza + 1

Faza de date

In faza de date, in functie de `write` si `read` (doar unul poate fi activ intr-un anumit moment dat) se realizeaza instructiunea precizata. Pentru `write` enabled, `data_write` ia valoarea trimisa in `data_in` si se realizeaza scrierea in registru, iar pentru `read` enabled i se atribuie semnalului `data_out` valoarea trimisa de registru prin `data_read`.

Descrierea modulului *regs*

Introducere

Modulul **regs.v** este organizat în:

- Logica de scriere a datelor în registru
- Logica de citire a datelor din regsitru
- Resetare
- Logica pentru counter reset

Regiștrii

În acest proiect, regiștrii sunt implementați prin flip-flop–uri D, cu dimensiuni multiplu de 8 biți pentru adresarea pe octeți. Pentru a putea reține stări și a face atribuiri, fiecare registru a fost asociat cu o variabilă de tipul reg:

- Variabile de tipul reg care reprezintă semnalele trimise către numărător (counter)

```
reg[15:0] reg_period;
reg reg_en;
reg reg_count_reset;
reg reg_upnotdown;
reg[7:0] reg_prescale;
```

- Variabile de tipul reg care reprezintă semnalele trimise catre generatorul de semnale de tip PWM (pwm_gen)

```
reg reg_pwm_en;
reg[7:0] reg_functions;
reg[15:0] reg_compare1;
reg[15:0] reg_compare2;
```

- Variabilă de tipul reg care reprezintă datele citite din registru și trimise către decodorul de instrucțiuni

```
reg[7:0] reg_data_read;
```

- Variabilă de tipul reg care contorizează ciclii de ceas pentru a asigura că registrul pentru counter_reset se golește după al doilea ciclu de ceas

```
reg[1:0] count_reset_cycles;
```

Variabilele de tipul reg sunt declarate în afara blocului always. Se atribuie porturilor modulului (variabile de tipul wire) valorile reținute în regiștrii prin atribuiri continue (assign):

```
assign period = reg_period;
assign en = reg_en;
assign count_reset = reg_count_reset;
assign upnotdown = reg_upnotdown;
assign prescale = reg_prescale;
assign pwm_en = reg_pwm_en;
assign functions = reg_functions;
assign compare1 = reg_compare1;
assign compare2 = reg_compare2;
assign data_read = reg_data_read;
```

Atribuirile continue (assign) pentru variabilele de tip wire sunt făcute în exteriorul blocului always, ce va reprezenta logica secvențială.

Logica secvențială

În modul, logica secvențială este reprezentată prin blocul always:

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

Starea bistabililor se modifică pe frontul crescător al semnalului de ceas (se adaugă posedge clk în lista de senzitivități), semnalul de reset este activ pe frontul negativ (se adaugă negedge rst_n în lista de senzitivități).

În interiorul blocului always este implementată logica de scriere și de citire a datelor și resetarea, se folosesc atribuiri non-blocante ("<=").

Reset

Atunci când semnalul de reset este activ, toate valorile sunt setate la 0 pentru a evita situația în care perifericul pornește cu valori nedeterminate.

Logica de scriere

Se verifică dacă are loc o instrucțiune de scriere (write = 1).

Scrierea datelor este organizată printr-o instrucțiune de decizie (case) în funcție de adresa trimisă de decodorul de instrucțiuni. Dacă adresa se regăsește în tabel, se va executa operația de scriere a datelor în registru, altfel adresa va fi ignorată. Registrul cu adresa 0x08 (counter val) va fi ignorat în operația de scriere, pentru ca accesul este doar pentru citire.

Deși un registru poate avea date utile mai mici decât 8 biți, fiecare adresă va reprezenta un byte. Pentru regiștrii cu dimensiunea de 16 biți (period, compare1, compare2, counter_val) sunt folosite două adrese diferite: o adresă pentru scrierea/citirea byte-ului în/din secțiunea LSB a regsistrului și o altă adresă pentru scrierea/citirea byte-ului în/din secțiunea MSB a regsistrului. Am considerat prima adresă asociată secțiunii LSB din registru și adresa următoare asociată secțiunii MSB.

```
6'h00: reg_period[7:0] <= data_write; //writes to the LSB section of the 16 bit register
6'h01: reg_period[15:8] <= data_write; //writes to the MSB section of the 16 bit register
```

Logica de citire

Se verifică dacă are loc o instrucțiune de citire (read = 1).

La fel ca în cazul operației de scriere, se folosește instrucțiunea de decizie case în funcție de adresa trimisă de decodorul de instrucțiuni. Dacă adresa se regăsește în tabel, se va executa operația de citire a datelor din registru, altfel va returna valoarea 0. Registrul cu adresa 0x07 (counter reset) va fi ignorat în operația de citire, pentru ca accesul este doar pentru scriere.

Logica pentru counter reset

La activarea semnalului counter reset, acesta trebuie să dureze exact două cicluri de ceas.

- **Implementarea:**

După scrierea în registru, se numără pentru câți ciclii de ceas este activ, contorul este inițializat cu 2.

Modulul regs este implementat astfel încât să fie sintetizabil, toate atribuirile asupra variabilelor de tipul reg sunt realizate în același bloc always, sincronizat cu ceasul și cu reset asincron.

Descrierea modulului *counter*

Semnale de intrare si iesire

- `clk` - semnalul de ceas al sistemului
- `rst_n` - reset asincron activ pe 0
- `count_val`- valoarea curenta a numaratorului
- `period` - valoarea maxima pe care o poate numara modulul

- `en` - enable counter
- `count_reset` - reset intern al counterului
- `upnotdown` - semnal pentru stabilirea directiei de numarare
- `prescale` - semnal pentru stabilirea numarului de cicluri de ceas care trebuie sa treaca pana la realizarea urmatoarei incrementari/decrementari

Modulul *counter* are ca scop oferirea unei baze de timp pentru semnalul PWM. Latimea numaratorului influenteaza direct rezolutia la care semnalul poate fi generat. Astfel, numarul de biti al semnalului de intrare `period` stabileste rezolutia PWM-ului.

Implementarea modulului *counter* permite numararea crescatoare, prin setarea intrarii `upnotdown` ca `1`, sau descrescatoare, prin setarea acesteia ca `0`. In cazul unui overflow, numaratorul reia numaratoarea de la `0` (pt. sens crescator) sau de la `period` (pt. sens descrescator).

De asemenea, numaratorul are ca sarcina si adaptarea frecventei semnalului PWM la nevoile oricarui caz, independent de clock-ul sistemului. Acest lucru se realizeaza prin **prescalare**. Modulul foloseste o variabila interna pentru a determina cate cicluri de ceas au trecut de la ultima incrementare/decrementare. La fiecare ciclu de ceas contorul intern este incrementat, iar cand acesta ajunge la valoarea `prescale` se realizeaza urmatoarea modificare a variabilei `count_val`.

Descrierea modulului *pwm_gen*

Introducere

Modulul **pwm_gen.v** este organizat în:

- `Logica de decodificare a modului de funcționare`
- `Logica combinațională` (determinarea stării următoare a semnalului PWM)
- `Logica secvențială` (memorarea stării curente a semnalului PWM)

Decodificarea modului de funcționare

În implementare se face decodificarea celor doi biți din functions pentru a determina modul de funcționare (aliniere la stânga/aliniere la dreapta/nealiniat).

```
wire is_aligned_left = (functions[1:0] == 2'b00);
wire is_aligned_right = (functions[1:0] == 2'b01);
wire is_unaligned = (functions[1] == 1'b1);
```

Logică combinațională

Se declară variabilele de tipul reg folosite pentru a reține starea curentă și starea următoare a semnalului PWM.

```
reg pwm_next_state;
reg pwm_current_state;
```

Se atribuie ieșirii pwm_out (variabilă de tipul wire) starea curentă prin atribuire continuă (assign). Când starea curentă își schimbă valoarea, acest lucru se va reflecta imediat în ieșire:

```
assign pwm_out = pwm_current_state;
```

Prin blocul always @(*) este implementată logica combinațională, prin care este determinată starea urmatoare a semnalului PWM. În acest bloc sunt folosite atribuiri blocante ("=").

Starea următoare este inițial setată la starea curentă, pentru a evita valori nedeterminate.

Se verifică dacă pwm_en este activ pentru ca semnalul PWM să fie generat:

```
if (pwm_en) begin
    ...
end
```

Dacă pwm_en este activ, starea următoare este determinată în funcție de modul de funcționare (aliniat la stânga/aliniat la dreapta/nealiniat) și de valorile compare1 și compare2 care se compară cu valoarea la care se află numărătorul (se consideră compare1 < compare2).

Dacă pwm_en nu este activ, starea următoare va ramâne aceeași cu starea curentă, semnalul PWM nu își modifică valoarea.

Logica secvențială

Logica secvențială, folosită pentru memorarea stării curente a semnalului PWM, este reprezentată prin blocul:

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

în blocul care modelează logică secvențială se folosesc atribuiri non-blocante ("<=").

Dacă resetul este activ, se setează ieșirea PWM la 0, altfel ieșirea este actualizată cu starea următoare (determinată de logica combinațională).

Pentru ca modulul să fie sintetizabil, nu se fac atribuiri asupra aceluiași registru în mai mult de un bloc always.