1 Lezione del 03-12-24

1.1 Interfacce parallele

1.1.1 Interfacce di ingresso senza handshake

Iniziamo a vedere le interfacce parallele di **ingresso** senza handshake. Queste scambiano interi byte col processore, attraverso un solo registro. Non c'è quindi nessuna linea di indirizzo, l'intero registro va direttamente al processore su una linea dati. Abbiamo poi la linea di select `/s` e la linea di I/O read `/ior`.

Il registro in uscita è forchettato da una tri-state in modo da mantenere ad alta impedenza l'uscita del registro RBR quando il processore non sta leggendo dall'interfaccia. Inoltre, non vorremo nemmeno che il registro RBR si trovi a leggere dati quando il processore non sta leggendo.

Possiamo quindi ricavare, dal select e l'I/O read attraverso una porta NOR, un segnale di enable sia per la porta tri state che per il registro: quando entrambi sono bassi, si legge dal lato dispositivo dell interfaccia all'interno del registro, e si restituisce l'uscita del registro al processore.

Notiamo che l'aggiornamento dell'RBR avviene al *fronte di salita* dell'enabler (quindi di `/ior`), quindi una volta sola per ogni lettura.

1.1.2 Interfacce di uscita senza handshake

L'interfaccia parallela di **uscita** senza handshake è simile: si ha sempre un solo registro, TBR, che memorizza le linee dati in entrata qundo sono entrambi bassi il select e l'I/O write (`/iow`), cosa ricavata attraverso un'altra porta NOR. Notiamo che in questo caso TBR campiona sul *fronte di discesa* dell'enabler (quindi di `/iow`), anziché di salita.

Vediamo la descrizione Verilog:

```
// un'interfaccia parallela di ingresso senza handshake su 8 bit
module parallel_in(s_, ior_, d7_d0, byte_in);
  input s_, ior_;
  output[7:0] d7_d0;
  input[7:0] byte_in;

  reg[7:0] RBR;

  wire e;
  assign e = {s_, ior_} == 2'B00;

  assign d7_d0 = e ? RBR : 8'BX;

  always @(posedge e) RBR <= byte_in;;
endmodule
```

1.1.3 Interfacce di ingresso/uscita senza handshake

Le intefacce di ingresso/uscita senza handshake si costruiscono unendo due interfacce, una di ingresso e una di uscita (e appunto dette **sottointerfacce** di ingresso e uscita), e selezionando l'interfaccia giusta attraverso un bit di indirizzo (`a0`). Il select per la selezione viene generato da una semplice rete combinatoria che prende in ingresso il `/s` globale e il bit di indirizzo, con tabella di verità: Dove `/si` è il select della sottointerfaccia di ingresso, e `/so` il select della sottointerfaccia di uscita. Come vediamo dall'esempio,

/s	/a0	/si	/so
1	-	1	1
0	0	0	1
0	1	1	0

solitamente si mette a indirizzo **pari** la porta di *ingresso*, e a indirizzo **dispari** la porta di *uscita*.

Notiamo che un montaggio alternativo si ottiene ignorando il bit di indirizzo e accedendo direttamente alle due sottointerfacce con un unico select. A questo punto sarà compito del processore discriminare fra `/ior` e `/iow` per selezionare la sottointerfaccia desiderata.

Vediamo la descrizione Verilog:

```
// un'interfaccia parallela di ingresso senza handshake su 8 bit
module parallel_in(s_, ior_, d7_d0, byte_in);
  input s_, ior_;
  output[7:0] d7_d0;
  input[7:0] byte_in;

  reg[7:0] RBR;

  wire e;
  assign e = {s_, ior_} == 2'B00;

  assign d7_d0 = e ? RBR : 8'BX;

  always @(posedge e) RBR <= byte_in;;
endmodule
```

1.1.4 Interfacce di ingresso/uscita

Possiamo combinare le interfacce senza handshake viste finora in un unica interfaccia di ingresso/uscita. In questo caso i registri RBR e TBR verranno unificati, a quanto visto dal processore, in un unico registro RTBR, mentre i registri di stato RSR e TSR verranno unificati in un unico RTSR.

In Verilog, mostrando un esempio con calcolo esplicito dei select (sulla base dell'LSB dell'offset) e uno a connessione diretta:

```
// un'interfaccia parallela di ingresso e uscita senza handshake su
// 8 bit

// a due buffer
module parallel_inout_2buf(s_, ior_, iow_, a0, d7_d0, byte_in,
                                                      byte_out);
  input s_, ior_, iow_;
  input a0;
  inout[7:0] d7_d0;
  input[7:0] byte_in;
  output[7:0] byte_out;

  wire sr_;
  assign sr_ = !({s_, a0} == 2'B00);

  wire sw_;
  assign sw_ = !({s_, a0} == 2'B01);

```

```
  parallel_in in (
    .s_(sr_), .ior_(ior_), .d7_d0(d7_d0), .byte_in(byte_in)
  );

  parallel_out out (
    .s_(sw_), .iow_(iow_), .d7_d0(d7_d0), .byte_out(byte_out)
  );
endmodule

// a buffer singolo
module parallel_inout_1buf(s_, ior_, iow_, d7_d0, byte_in, byte_out);
  input s_, ior_, iow_;
  inout[7:0] d7_d0;
  input[7:0] byte_in;
  output[7:0] byte_out;

  parallel_in in (
    .s_(s_), .ior_(ior_), .d7_d0(d7_d0), .byte_in(byte_in)
  );

  parallel_out out (
    .s_(s_), .iow_(iow_), .d7_d0(d7_d0), .byte_out(byte_out)
  );
endmodule
```

1.1.5 Interfacce di ingresso con handshake

Ricordiamo la visione funzionale delle interfacce di ingresso con handshake: dovremo avere i registri RBR e RSR, da cui ricaviamo il flag FI lato processore, mentre lato dispositivo dovremo avere sia le linee di ingresso dati che le linee di handshake, che assumiamo essere in forma `/dav` e `rfd`.

L'interfaccia si implementa quindi come una combinazione di una rete combinatoria, come avevamo visto per le interfacce senza handshake, per la generazione degli enable, e una rete sequenziale per la gestione degli handshake.

Il processore potrà accedere in lettura sia al RBR che al RSR: questo si discrimina attraeverso un bit di indirizzo (`a0`). Si ha quindi la rete combinatoria per la generazione degli enable, dove `eb` è l'enable del buffer e `es` è l'enable del registro di stato:

/s	/ior	/a0	/es	/eb
0	0	0	1	0
0	0	1	0	1
...			0	0

Vediamo la descrizione Verilog:

```
// un'interfaccia parallela di ingresso con handshake su 8 bit
module hs_parallel_in(clock, reset_,
                      s_, ior_, a0, dav_, rfd, d7_d0, byte_in);
  input clock, reset_;

  input s_, ior_;
  input a0;

  input dav_;
  output rfd;

  output[7:0] d7_d0;
```

```
13    input[7:0] byte_in;
14
15    wire e_b, e_s;
16    hs_parallel_in_comb comb (
17      .s_(s_), .ior_(ior_), .a0(a0),
18      .e_b(e_b), .e_s(e_s)
19    );
20
21    wire[7:0] rbr;
22    wire fi;
23
24    hs_parallel_in_seq seq (
25      .clock(clock), .reset_(reset_),
26      .dav_(dav_), .rfd(rfd), .e_b(e_b),
27      .byte_in(byte_in), .fi(fi), .rbr(rbr)
28    );
29
30    assign d7_d0 = e_b ? rbr:
31                   e_s ? {7'BZ, fi}:
32                   /*dc*/8'BZ;
33  endmodule
34
35  module hs_parallel_in_seq(clock, reset_,
36                            e_b, dav_, rfd, byte_in, fi, rbr);
37    input clock, reset_;
38
39    input e_b;
40
41    input dav_;
42    output rfd;
43
44    reg RFD;
45    assign rfd = RFD;
46
47    input[7:0] byte_in;
48
49    output fi;
50
51    reg FI;
52    assign fi = FI;
53
54    output[7:0] rbr;
55
56    reg[7:0] RBR;
57    assign rbr = RBR;
58
59    reg[1:0] STAR;
60    localparam
61      s0 = 2'B00,
62      s1 = 2'B01,
63      s2 = 2'B10,
64      s3 = 2'B11;
65
66    always @(reset_) if(reset_ == 0) #1 begin
67      RFD <= 1;
68      FI <= 0;
69      STAR <= s0;
70    end
71
72    always @(posedge clock) if(reset_ == 1) #3 begin
```

```
73      casex(STAR)
74        s0: begin
75          RFD <= 1;
76          RBR <= byte_in;
77          FI <= 0;
78          STAR <= (dav_ == 0) ? s1 : s0;
79        end
80        s1: begin
81          RFD <= 0;
82          STAR <= (dav_ == 1) ? s2: s1;
83        end
84        s2: begin
85          FI <= 1;
86          STAR <= (e_b == 1) ? s3 : s2;
87        end
88        s3: begin
89          STAR <= (e_b == 0) ? s0 : s3;
90        end
91      endcase
92    end
93  endmodule
94
95  module hs_parallel_in_comb(s_, ior_, a0, e_b, e_s);
96    input s_, ior_, a0;
97    output e_b, e_s;
98
99    assign {e_b, e_s} = ({s_, ior_, a0} == 3'B000) ? 'B01:
100                       ({s_, ior_, a0} == 3'B001) ? 'B10:
101                       /*         don't care       */ 'B00;
102 endmodule
```

Notiamo che, nella sintesi, si ritarda il clock dell'interfaccia rispetto a quello del processore per evitare condizioni di *corsa* all'interfaccia.

1.1.6 Interfacce di uscita con handhsake

L'interfaccia di uscita è realizzata in modo analogo: includiamo un registro TSR che contiene il flag FO lato processore, il registro TBR, e lato dispositivo le linee `/dav` e `rfd`, dove però è l'interfaccia, e non più il processore, a fare da produttore.

La struttura interna sarà del tutto simile a l'interfaccia di ingresso, con una parte combinatoria che si occupa degli enable e una parte sequenziale che si occupa degli handshake. La parte combinatoria obbedirà alla tabella di verità:

/s	/ior	/iow	/a0	/es	/eb
0	0	1	0	1	0
0	1	0	1	0	1
...				0	0

Notiamo che compare comunque la linea di uscita in quanto vogliamo leggere lo stato del TSR.

Vediamo la descrizione Verilog:

```
1  // un'interfaccia parallela di ingresso con handshake su 8 bit
2  module hs_parallel_out(clock, reset_,
3                         s_, ior_, iow_, a0, dav_, rfd, d7_d0, byte_out);
4    input clock, reset_;
5
6    input s_, ior_, iow_;
```

```
  input a0;

  output dav_;
  input rfd;

  inout[7:0] d7_d0;
  output[7:0] byte_out;

  wire e_b, e_s;
  hs_parallel_out_comb comb (
    .s_(s_), .ior_(ior_), .iow_(iow_), .a0(a0),
    .e_b(e_b), .e_s(e_s)
  );

  wire fo;

  hs_parallel_out_seq seq (
    .clock(clock), .reset_(reset_),
    .dav_(dav_), .rfd(rfd), .e_b(e_b),
    .byte_out(byte_out), .fo(fo), .d7_d0(d7_d0)
  );

  assign d7_d0 = e_s ? {3'BZ, fo, 4'BZ}:
                 /*dc*/8'BZ;
endmodule

module hs_parallel_out_seq(clock, reset_,
                           e_b, dav_, rfd, byte_out, fo, d7_d0);
  input clock, reset_;

  input e_b;

  output dav_;
  input rfd;

  reg DAV;
  assign dav_ = DAV;

  output[7:0] byte_out;

  input[7:0] d7_d0;

  output fo;

  reg FO;
  assign fo = FO;

  reg[7:0] TBR;
  assign byte_out = TBR;

  reg[1:0] STAR;
  localparam
    s0 = 2'B00,
    s1 = 2'B01,
    s2 = 2'B10,
    s3 = 2'B11;

  always @(reset_) if(reset_ == 0) #1 begin
    DAV <= 1;
    FO <= 1;
```

```
    STAR <= s0;
  end

  always @(posedge clock) if(reset_ == 1) #3 begin
    casex(STAR)
      s0: begin
        TBR <= d7_d0;
        FO <= 1;
        STAR <= (e_b == 1) ? s1 : s0;
      end
      s1: begin
        FO <= 0;
        STAR <= (e_b == 0) ? s2 : s1;
      end
      s2: begin
        DAV <= 0;
        STAR <= (rfd == 0) ? s3 : s2;
      end
      s3: begin
        DAV <= 1;
        STAR <= (rfd == 1) ? s0 : s3;
      end
    endcase
  end
endmodule

module hs_parallel_out_comb(s_, ior_, iow_, a0, e_b, e_s);
  input s_, ior_, iow_, a0;
  output e_b, e_s;

  assign {e_b, e_s} = ({s_, ior_, iow_, a0} == 4'B0010) ? 'B01:
                      ({s_, ior_, iow_, a0} == 4'B0101) ? 'B10:
                      /*            don't care          */ 'B00;
endmodule
```

1.1.7 Interfacce di ingresso/uscita con handshake

Possiamo combinare le interfacce con handhshake viste finora in un unica interfaccia di ingresso/uscita. Combineremo il registro di stato in un unico registro RTSR dotato dei flag FO e FI, e useremo un bit di indirizzo (`a0`) per distinguere fra le porte di ingresso e uscita.

In Verilog:

```
// un'interfaccia parallela di ingresso e uscita con handshake su 8
// bit
module hs_parallel_inout(clock, reset_,
                         s_, ior_, iow_, a0, d7_d0,
                         dav_in_, rfd_in,
                         dav_out_, rfd_out,
                         byte_in, byte_out);
  input clock, reset_;
  input s_, a0, ior_, iow_;

  input dav_in_;
  output rfd_in;

  output dav_out_;
  input rfd_out;

```

```
  inout[7:0] d7_d0;
  input[7:0] byte_in;
  output[7:0] byte_out;

  hs_parallel_in hs_in (
    .clock(clock), .reset_(reset_),
    .s_(s_), .ior_(ior_), .a0(a0),
    .dav_(dav_in_), .rfd(rfd_in),
    .d7_d0(d7_d0), .byte_in(byte_in)
  );

  hs_parallel_out hs_out (
    .clock(clock), .reset_(reset_),
    .s_(s_), .ior_(ior_), .iow_(iow_), .a0(a0),
    .dav_(dav_out_), .rfd(rfd_out),
    .d7_d0(d7_d0), .byte_out(byte_out)
  );
endmodule
```

1.2 Interfacce seriali

Le interfacce seriali trasmettono informazioni un bit a volta. Noi ne consideriamo una versione semplificata, l'interfaccia seriale start/stop.

Dal punto di vista fisico, un interfaccia seriale trasporta informazioni su due linee:

- La linea di **massa**, che funge da riferimento;
- La linea **segnale**, che porta appunto il segnale riferito a massa.

Dal punto di vista logico, invece, a interessarci sarà solo la linea segnale. Questa trasporta informazioni *half duplex*, cioè da un trasmettitore a un ricevitore (la comunicazione nelle due direzioni richiede quindi due linee).

Ora, se il segnale è rappresentato da una sequenza di **marking** (1, linea in tensione) e **spacing** (0, linea a massa) trasmessi con un periodo T, il problema diventerà sincronizzare trasmettitore e ricevitore in modo che possano comunicare efficientemente.

Quello che ci interesserà stabilire sarà quindi il **tempo di bit** T e la **trama** del segnale. Definiamo trama una sequenza di bit che rappresenta un frammento di comunicazione: iniziamo la trama con uno spacing (**bit di start**), e seguiamo poi con un numero che va da 5 a 8, stabilito in precedenza, di **bit utili** (solitamente LSB). Infine, trasmettiamo un marking per segnalare la fine della trama (**bit di stop**).

I bit di start e di stop rappresentano un **overhead**: una trama di n bit utili è lunga almeno $n+2$ per segnalare inizio e fine della trama. Avremo quindi che la velocità netta della linea è $\frac{n}{n+2}$, che è comunque più efficiente di usare un clock secondario o effettuare un handshake per ogni bit.

Converrebbe quindi usare n arbitrariamente lunghi: purtroppo questo è fattibile fino ad un certo limite superiore, in quanto i clock di trasmettitore e ricevitore saranno necessariamente leggermente differenti in frequenza, ergo sul lungo termine si andrebbe ad accumulare un'errore troppo grande.

Infatti, l'impedenza dalla linea di trasmissione determinerà uno "smussamento" del segnale, motivo per cui preferiremo campionare ogni bit trasmesso nella posizione più centrale possibile rispetto alla sua forma d'onda. Questo significherà che, posto T il periodo del clock del trasmettitore, e $T+\Delta T$ il periodo del clock del ricevitore, vorremo:

$$n \cdot \Delta T \leq \frac{T}{2} \implies \frac{\Delta T}{T} \leq \frac{1}{2n}$$

Uno standard di *interoperabilità* per le trasmissioni seriali è l EIA-RS232C, che fissa lo 0 logico a tensioni da 3V a 25V, e l'1 logico a tensione negativa da -25V a -3V.

Vediamo quindi un implementazione Verilog, usando un clock al ricevitore con periodo $T + \Delta T = \frac{T}{16}$ per trasmissioni di trame da 10 bit (8 bit utili).

Il trasmettitore sarà il seguente:

```
// un trasmettitore seriale
module serial_transmitter(clock, reset_, dav_, rfd, byte, txd);
  input clock, reset_;
  input dav_;
  output rfd;
  output[7:0] byte;
  output txd;

  reg[9:0] BUFFER;

  reg RFD;
  assign rfd = RFD;

  reg TXD;
  assign txd = TXD;

  reg[3:0] COUNT;

  reg[1:0] STAR;
  localparam
    s0 = 2'B00,
    s1 = 2'B01,
    s2 = 2'B10;

  parameter mark = 1'B1, start_bit = 1'B0, stop_bit = 1'B1;

  always @(reset_ == 0) #1 begin
    RFD <= 1;
    TXD = mark;
    STAR <= s0;
  end

  always @(posedge clock) if(reset_ == 1) #3 begin
    casex(STAR)
      s0 : begin
        RFD <= 1;
        COUNT <= 10;
        TXD <= mark;
        BUFFER <= {stop_bit, byte, start_bit};
        STAR <= (dav_ == 0) ? s1 : s0;
      end
      s1 : begin
        RFD <= 0;
        BUFFER <= {mark, BUFFER[9:1]};
        TXD <= BUFFER[0];
        COUNT <= COUNT - 1;
        STAR <= (COUNT == 1) ? s2 : s1;
      end
      s2 : begin
        STAR <= (dav_ == 1) ? s0 : s2;
      end
    endcase
  end
endmodule
```

Il ricevitore, invece, sarà il seguente:

```
// un ricevitore seriale
module serial_receiver(clock, reset_, dav_, /*rfd,*/ byte, rxd);
  input clock, reset_;
  output dav_;
  // input rfd;
  output[7:0] byte;
  input rxd;

  reg[7:0] BUFFER;
  assign byte = BUFFER;

  reg DAV;
  assign dav_ = DAV;

  reg[4:0] WAIT;
  reg[3:0] COUNT;

  reg[1:0] STAR;
  localparam
    s0 = 2'B00,
    s1 = 2'B01,
    s2 = 2'B10,
    s3 = 2'B11;

  parameter start_bit = 1'B0;

  always @(reset_ == 0) #1 begin
    DAV <= 1;
    STAR <= s0;
  end

  always @(posedge clock) if(reset_ == 1) #0 begin
    casex(STAR)
      s0 : begin
        DAV <= 1;
        COUNT <= 8;
        WAIT <= 23;
        STAR <= (rxd == start_bit) ? s2 : s0;
      end
      s1 : begin
        BUFFER <= {rxd, BUFFER[7:1]};
        COUNT <= COUNT - 1;
        WAIT <= 15;
        STAR <= (COUNT == 1) ? s3 : s2;
      end
      s2 : begin
        WAIT <= WAIT - 1;
        STAR <= (WAIT == 1) ? s1 : s2;
      end
      s3 : begin
        DAV <= 0;
        WAIT <= WAIT - 1;
        STAR <= (WAIT == 1) ? s0 : s3;
      end
    endcase
  end
endmodule
```

E l'intera interfaccia, ricavata a partire da un interfaccia di ingresso uscita parallela

con handshake (che approvigiona un trasmettitore e un ricevitore seriali):

```
// un'interfaccia seriale di ingresso e uscita
module serial_interface(clock, reset_,
                        s_, ior_, iow_, a0, d7_d0,
                        rxd, txd);
  input clock, reset_;
  input s_, a0, ior_, iow_;

  wire dav_in_;
  wire rfd_in;

  wire dav_out_;
  wire rfd_out;

  inout[7:0] d7_d0;
  wire[7:0] byte_in;
  wire[7:0] byte_out;

  input rxd;
  output txd;

  reg rec_clock;
  reg tra_clock;

  initial begin
    rec_clock = 0;
    tra_clock = 0;
  end

  always #1 rec_clock = ~rec_clock;
  always #16 tra_clock = ~tra_clock;

  hs_parallel_inout parallel (
    .clock(clock), .reset_(reset_),
    .s_(s_), .ior_(ior_), .iow_(iow_), .a0(a0), .d7_d0(d7_d0),
    .dav_in_(dav_in_), .rfd_in(rfd_in),
    .dav_out_(dav_out_), .rfd_out(rfd_out),
    .byte_in(byte_in), .byte_out(byte_out)
  );

  serial_receiver receiver (
    .clock(rec_clock), .reset_(reset_),
    .dav_(dav_in_), .byte(byte_in),
    .rxd(rxd)
  );

  serial_transmitter transmitter (
    .clock(tra_clock), .reset_(reset_),
    .dav_(dav_out_), .rfd(rfd_out), .byte(byte_out),
    .txd(txd)
  );
endmodule
```