1 Lezione del 05-11-24

1.1 D-Latch trasparente

Introduciamo una nuova rete sequenziale dotata di due ingressi, d (data) e c (control), e un'uscita q. Il D-latch memorizza il bit in d quando c (**trasparenza**) vale 1. Quando c vale 0, invece, si dice che è in **conservazione**, ergo memorizza l'ultimo valore che d ha assunto quando c valeva 1.

La tabella di flusso di questa rete è la seguente, assunti in quest'ordine c e d:

	00	01	10	11	q
S_0	S_0	S_0	S_0	S_1	0
S_1	S_1	S_1	S_0	S_1	1

cioè quando si è in conservazione, qualsiasi valore di d viene ignorato e si memorizza il valore passato. Quando si è in trasparenza, invece, q si asegua a d.

Si può realizzare un D-latch attraverso un latch SR, con in ingresso una certa rete combinatoria. Quello che vogliamo fare è portare d e c in s e r, attraverso la tabella di verità:

c	d	s	r
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	0

Questo si sintetizza in $s = c \cdot d$ e $r = c \cdot \overline{d}$. Si ha che le porte AND che rappresentano le congiunzioni in questa rete combinatoria possono collassare con le porte AND che formavano la rete combinatoria del latch SR che permetteva preset e preclear.

In Verilog, l'implementazione è la seguente (trascurando le linee di reset, una versione che le contiene è comunque definita nella directory `/verilog`):

```
// un D-latch trasparente con preset e preclear
module d_latch(d, c, q, q_N);
  input d, c;
  output q, q_N;

  wire s, r;
  assign {s, r} == ({d, c} == 'B00) ? 'B00:
                   ({d, c} == 'B01) ? 'B01:
                   ({d, c} == 'B10) ? 'B00:
                 /*({d, c} == 'B11)?*/'B10;

  sr_latch latch (
    .s(s), .r(r),
    .q(q), .q_N(q_N)
  );
endmodule

// implementazione a porte logiche
module d_latch_p(d, c, q, q_N);
  input d, c;
  output q, q_N;

  wire s, r;
```

```
24    assign s = d & c;
25    assign r = ~d & c;
26
27    sr_latch latch (
28      .s(s), .r(r),
29      .q(q), .q_N(q_N)
30    );
31  endmodule
```

1.1.1 Pilotaggio del D-Latch

Nel pilotaggio del D-latch, dobbiamo assicurarci che d sia costante a cavallo della transizione di c da 1 a 0, in quanto potremmo finire per memorizzare dati ignoti (l'ultima cosa che il D-latch ha "visto" prima del reset di c). I tempi per cui d deve essere costante, rispettivamente **prima** e **dopo** della transizione di c, si dicono T_{setup} e T_{hold}, e sono dati di progetto.

1.1.2 Trasparenza

Quando il D-latch è in **trasparenza**. il suo ingresso è direttamente connesso, in **senso logico** (ci sono comunque ritardi nella logica delle reti), all'uscita. Per questo motivo, se q e d sono collegati in **retroazione negativa** (un feedback loop negato), si ha che con $c = 1$ abbiamo oscillazioni incontrollate, e che con $c = 0$ in q (cioè lo stato interno) resta un valore casuale (l'ultimo rilevato durante le oscillazioni casuali prima che c sia transito a 0).

Questo significa che il D-latch è una rete **trasparente**, cioè *la sua uscita cambia mentre la rete è sensibile alle variazioni di ingresso*. Questo significa che non possiamo memorizzare niente che sia funzione dell'uscita (saremmo nel caso della retroazine negativa di prima).

Poniamo di voler eseguire un'istruzione semplice come `INC %AX`. A livello hardware, questo significà connettere un registro (quindi una serie di D-latch) ad una rete combinatoria per l'incremento (probabilmente un half adder), e quindi l'uscita di questa rete di nuovo al D-latch. Quello che abbiamo essenzialmente creato è un ciclo di retrazione: il sistema devolverà velocemente in uno stato di oscillazione incontrollata.

1.2 D flip-flop

Il **D flip-flop** è una rete sequenziale **non trasparente** che si pone di risolvere i problemi di trasparenza del D-latch. Quello che vedremo nel dettaglio è il **positive edge-triggered D flip-flop**, che è una rete che si comporta come segue, sulla base degli ingressi d (data) e p, e l'uscita q: quando p ha un fronte di salita, memorizza d, *attendi* un determinato istante temporale e adegua l'uscita.

Possiamo concettualizzare il D flip-flop come composto, alla base, da un D-latch. Mettiamo a c, invece dell'ingresso p, il **generatore di impulso** P^+ sul fronte di salita di p. In uscita a q, poi, abbiamo un buffer Δ, che introduce ritardo. La proprietà fondamentale che desideriamo è:

$$\Delta > P^+$$

Questo significa che q si adegua al valore campionato di d soltanto *dopo* che la rete ha smesso di essere sensibile a d. È questa proprietà a rendere il D flip-flop una rete non trasparente.

In Verilog, possiamo rendere una rete di questo tipo attraverso l'operatore `posedge`:

```
// un D flip-flop pilotato sul fronte di salita
module d_flip_flop(d, p, q, q_N);
  input d, p;

  reg Q;
  output q, q_N;
  assign q = Q;
  assign q_N = ~q;

  wire q_l;

  d_latch latch (
    .d(d), .c(p),
    .q(q_l)
  );

  always @(posedge p) #2 Q <= q_l;
endmodule
```

1.2.1 Pilotaggio del D flip-flop

Innanzitutto, a cavallo del fronte di salita di p l'ingresso d deve rimanere costante, ergo si hanno gli stessi $_{setup}$ e T_{hold} del D-latch. Inoltre, si ha il ritardo di adeguamento dell'uscita, che denominiamo T_{prop} (dall'inglese *propagation*). Qui la diseguaglianza d prima si traduce come:

$$T_{prop} > T_{hold}$$

Si che l'uscita di un D-FF non oscilla mai, a differenza di quella del D-Latch: l'adeguamento avviene in modo "secco", sul fronte di salita, e di lì in poi fino a reset e successivo set di p, l'uscita q è in conservazione e ignora il comportamento di d.

1.2.2 Sintesi Master-Slave di un D flip-flop

Storicamente, un D flip-flop è stato realizzato attraverso un montaggio master/slave, attraverso due D-latch in cascata (di cui uno master, e l'altro chiaramente slave). Si invia quindi l'ingresso p allo slave, e il suo negato al master, e si fa passare la linea d prima dal master, poi dalla sua uscita all'ingresso del slave, e poi al q del D flip-flop. Si ha che negli stati:

- $p = 0$: **master** e in *trasparenza*, **slave** in *conservazione*;
- $p = 1$: **master** in *conservazione*, **slave** in *trasparenza*.

Quando p è a 0, lo slave è in conservazione, quindi la rete memorizza. Nel frattempo il master è in trasparenza, quindi reagisce al valore in entrata di d. Quando p transisce a 1, lo slavean automa in automaton theory passa in trasparenza, e quindi risponde a quello che esce dal master, che invece si trova in conservazione del valore che aveva un'attimo prima della transizione. Il risultato è un comportamento effettivamente analogo a quello della struttura a generatore di impulso e buffer vista prima.

Si possono avere problemi nel funzionamento transitorio dei due D-latch: per questo si agisce elettronicamente, sviluppando questi per commutare c a valori di tensione diversi. In particolare, vogliamo che in transizione di p da 1 a 0 lo slave conservi il valore prima che il master passi a trasparenza, quindi che c dello slave commuti prima di c del master.

Nella pratica, infine, si ha che la sintesi reale di un D flip-flop è fatta a partire da un latch SR, prima del quale si dispone una rete sequenziale asincrona la cui sintesi è fuori dagli scopi del corso.

1.3 Memorie RAM statiche

Esistono due tipi principali di memoria:

- **S-RAM**, costituite da matrici di D latch;
- **D-RAM**, realizzate in modo diverso (capacitori che bisognano di refresh periodico), che per adesso ingnoreremo.

Una riga di D latch rappresenta quindi una **locazione di memoria**, che può essere **letta** o **scritta** con apposite operazioni, strettamente **non simultanee**.
Una SRAM è presenta gli ingressi e le uscite:

- **Ingressi di indirizzo:** in numero sufficiente per indirizzare tutte le celle di memoria. Ad esempio, con 2^{23} celle di 4 bit, 23 ingressi;
- **Ingressi/uscite di dati:** che andranno forchettati con porte **tri-state**;
- **Memory read** e **memory write**, segnali attivi bassi;
- **Select**, segnale attivo basso che fa da **enabler**, in modo simile a quanto avevamo visto nei decoder.

Il comportamento che vogliamo dalla SRAM è il seguente:

/s	/mr	/mw	Azione
1	-	-	Nulla
0	1	1	Nulla
0	0	1	Lettura in corso
1	1	0	Scrittura in corso

In Verilog generativo, potremo quindi sintetizzare una semplice SRAM basata su D-Latch come segue:

```
// un banco di memoria RAM statica da 2^N locazioni da M bit
module nNbyM_sram
  #(parameter N = 4, parameter M = 4) // N: linee di indirizzo
                                      // M: linee di dati
  (s_, mr_, mw_, addr, data_in, data);

  input s_, mr_, mw_;
  input[N-1:0] addr;

  // sarebbe una tristate
  output[M-1:0] data; // emulata con due porte, data_in e data_out

  input[M-1:0] data_in;
  reg[M-1:0] data_out;

  wire b; // enabler della tristate
  wire c; // controllo dei latch

  assign b = ~s_ & ~mr_ & mw_; // select e memory read
```

```
  assign c = ~s_ & mr_ & ~mw_; // select e memory write

  // logica delle tristate
  // assign data_in = data;
  assign data = b ? data_out : {M{1'BZ}};

  // sarebbe un demultiplexer da 1 a 2^N
  reg[2**N-1:0] c_plex;
  always @(*) begin
      c_plex = {2**N{1'B0}};
      c_plex[addr] = c;
  end

  // sarebbe un multiplexer da 2^N a 1
  wire[2**N-1:0] data_out_plexes [0:M-1];
  integer i;
  always @(*) begin
    for(i = 0; i < M; i = i + 1) begin
      data_out[i] = data_out_plexes[i][addr];
    end
  end

  genvar j;
  genvar k;
  generate
    // banco di locazioni
    for(k = 0; k < 2**N; k = k + 1) begin: locations
      // singola locazione
      for(j = 0; j < M; j = j + 1) begin : latches
        d_latch latch (
          .d(data_in[j]), .c(c_plex[k]),
          .q(data_out_plexes[j][k])
        );
      end
    end
  endgenerate
endmodule
```

Ad esempio, una SRAM con locazioni da 4 bit e indirizzi su 24 bit (che fa $2^{23} = 2^3 \times 2^{20} = 8\text{M}$, cioè 8M nibble) si sintetizzerebbe impostando i parametri `N=4` e `M=23`.

1.3.1 Temporizzazione delle RAM statiche

Facciamo innanzitutto la divisione lettura/scrittura:

- **Lettura:** per fare una lettura bisogna dare il comando (attivo basso) di memory read (`/mr`), e impostare l'indirizzo di lettura. Il comando di select (`/s`) arriva in ritardo, e a quel punto, quando sia `/s` che `/mr` sono in conduzione, i multiplexer vanno a regime e si può fare una lettura sull'uscita dei dati. Infine, quando `/mr` torna a 1, i dati tornano ad alta impedenza, e l'indirizzo di lettura e la select possono assumere valori arbitrari.
- **Scrittura:** si ha che la scrittura è **distruttiva** (manda i D-latch in trasparenza). Bisogna quindi attendere che il select `/s` e gli indirizzi siano stabili prima di abbassare `mw` per dare il comando di scrittura (l'opposto di quanto avevamo fatto in lettura, qui vogliamo scrivere solo quando siamo sicuri di poterlo fare, ergo i multiplexer sono a regime). A questo punto, abbiamo che quando `mw` torna alto dobbiamo assicurarci che i dati in scrittura siano fermi, in quanto i multiplexer riportano

gli ingressi di controllo dei D-latch a 0 e l'indirizzo di lettura e la select possono, nuovamente, assumere valori arbitrari.

1.3.2 Montaggio di banchi di memoria

Vediamo come combinare più banchi di memoria per aumentare lo spazio di memoria indirizzabile.

- **Montaggio in parallelo:** prendiamo in considerazione due banchi di memoria da $8\text{M} \times 4$ bit, e vediamo come collegarli per formare un singolo banco di memoria da $8\text{M} \times 8$ bit, quindi raddoppiando la dimensione delle locazioni.

  – Per quanto riguarda gli **indirizzi di lettura**, basta inviare l'indirizzo ad entrambi i banchi, da cui preleveremo la parte *alta* e *bassa* della locazione;

  – Per quanto riguarda gli **ingressi/uscite di dati**, avremo che la combinazione delle linee sui due banchi, da 4 bit ciascuna, formano un singolo byte da 8 bit, ergo la locazione di memoria completa.

  Cioè in Verilog, ad esempio per creare un banco da 16×8 bit a partire da due da 16×4 bit:

```
// esempio di montaggio di banchi di RAM in parallelo
module parallel_sram(s_, mr_, mw_, addr, data_in, data);
  input s_, mr_, mw_;
  input[3:0] addr;

  output[7:0] data;
  input[7:0] data_in;

  nNbyM_sram #(.N(4), .M(4)) bank_0 (
    .s_(s_), .mr_(mr_), .mw_(mw_),
    .addr(addr), .data(data[7:4]), .data_in(data_in[7:4])
  );

  nNbyM_sram #(.N(4), .M(4)) bank_1 (
    .s_(s_), .mr_(mr_), .mw_(mw_),
    .addr(addr), .data(data[3:0]), .data_in(data_in[3:0])
  );
endmodule
```

- **Montaggio in serie:** prendiamo in considerazione due banchi di memoria da $8\text{M} \times 8$ bit, e vediamo come collegarli per formare un singolo banco di memoria da $16\text{M} \times 8$ bit, quindi raddoppiamdo il numero di locazioni.

  – Per quanto riguarda gli **indirizzi di lettura**, discrimina dal MSB dell'indirizzo se selezionare dal primo o dal secondo banco, che faranno quindi da parte *alta* e *bassa* dello spazio di memoria indirizzabile. Facciamo questo attraverso l'ingresso di select `/s`, che useremo per determinare altri due segnali di select `/sl` e `sh` (*select low* e *select high*), che a loro volta ci permettono di discriminare sulla base del MSD quale banco andiamo a selezionare (effettivamente rendere attivo);

  – Per quanto riguarda gli **ingressi/uscite di dati**, avremo che il banco attivo in un dato momento determina completamente le uscite. Potremmo pensare di dover inserire porte tri-state in uscita ai singoli banchi di memoria sulla linea

di ingresso/uscita, ma questo non è necessario: le `sl` e `sh` sono mutualmente esclusive, e quindi non si verificherà mai il caso in cui le linee di uscita di entrambi i banchi sono in conduzione contemporaneamente.

Cioè in Verilog, ad esempio per creare un banco da 32×4 bit a partire da due da 16×4 bit:

```
// esempio di montaggio di banchi di RAM in serie
module parallel_sram(s_, mr_, mw_, addr, data_in, data);
  input s_, mr_, mw_;
  input[4:0] addr;

  output[3:0] data;
  input[3:0] data_in;

  wire l_s_;
  wire r_s_;
  assign l_s_ = ~addr[4] | s_;
  assign r_s_ = addr[4] | s_;

  wire[3:0] data_0;
  wire[3:0] data_1;
  assign data = addr[4] ? data_0 : data_1;
  // per emulazione delle tristate, nella pratica bastano da sole

  nNbyM_sram #(.N(4), .M(4)) bank_0 (
    .s_(l_s_), .mr_(mr_), .mw_(mw_),
    .addr(addr[3:0]), .data(data_0), .data_in(data_in)
  );

  nNbyM_sram #(.N(4), .M(4)) bank_1 (
    .s_(r_s_), .mr_(mr_), .mw_(mw_),
    .addr(addr[3:0]), .data(data_1), .data_in(data_in)
  );
endmodule
```