Modellazione comportamentale

▼ Creatore originale: @Gianbattista Busonera

La modellazione comportamentale è ottenuta tramite blocchi `always`, i quali definiscono una sorta di processi, ovvero funzioni le cui istruzioni sono eseguite in sequenza, ma nello stesso istante.

Per fare un paragone, un blocco `always` è una specie di chiamata ricorsiva che viene effettuata ogni qual volta i parametri "sensibili" (che devono essere `reg` o `wire`) del blocco `always` vengono modificati al di fuori del blocco `always` stesso.

Con tale paradigma si possono modellare sia circuiti combinatori, sia sequenziali.

Ecco la sintassi di un blocco `always`:

```
always @(<elenco_sensibilità>) begin
    ...
end

// Quando uno o più segnali presenti nell'elenco di sensibilità
// cambia valore, il blocco always viene eseguito.
```

⚠️ Tutte le uscite di un blocco `@always` devono essere di tipo `reg`:

- le assegnazioni usano solo "=" o "<=";
- mantengono memoria dell'ultima assegnazione, al contrario dei `wire` (che, in uscita, sono frutto di assegnazioni "continue").

Il tipo `reg` assomiglia a un registro, ma non ne rappresenta necessariamente uno.

Dentro i blocchi `always` possono essere utilizzati i costrutti `if` e `case`, grazie al fatto che le operazioni vengono eseguite in ordine.

Always con circuiti combinatori

La caratteristica dei circuiti combinatori è tale per cui l'uscita dipende, in ogni istante, dal valore degli ingressi e, pertanto, il blocco `always` sarà del tipo:

```
always @(ingresso1, ingresso2, ..., ingressoN) begin
    ...
end

// L'elenco di sensibilità è dato dagli ingressi presenti nel circuito combinatorio.
```

Per essere sicuri di valutare tutti gli ingressi in una logica combinatoria si può anche utilizzare tale sintassi:

```
always @(*) begin // Viene eseguito sulla variazione di:
    out1 = a+b+c+d; // sulla variazione di uno tra a,b,c,d
    out2 = f*e+3; // sulla variazione di uno tra f ed e.
end

// Il blocco always viene eseguito ogni qual volta che un
// secondo membro di un'assegnazione cambia valore.
```

▼ Esempio - Porta AND

Si realizzi la porta AND in figura, tramite il blocco `always`.

Porta AND

```
reg y; // uscita sempre di tipo reg
wire a,b; // ingressi possono essere wire o reg.
always @(a or b) begin
    y = a&b;
end
```

Tale codice è identico a scrivere il seguente codice:

```
reg y; // uscita sempre di tipo reg, mantiene memoria dell'ultima assegnazione
wire a,b; // ingressi possono essere wire o reg.
always @(*)
   y = a&b; // con una sola riga di codice posso evitare di usare begin e end
```

Esso è ancora uguale a:

```
wire y, a,b; // il wire presuppone un'assegnazione continua che esiste sempre!
assign y = a & b; // quì possiamo usare un wire come uscita!

// Reg si potrebbe usare come secondo membro di un'operazione
// di assign ma non come primo membro
```

Always con circuiti sequenziali

Nel caso di circuiti sequenziali è necessario far sì che il blocco `always` sia attivato solo durante il fronte di salita/discesa del clock tramite le parole chiave:

- `posedge`: fronte di salita;
- `negedge`: fronte di discesa.

```
always @(posedge clock) // esempio 1
always @(negedge clock or reset) // esempio 2, FF attivo su fronte di discesa
// con reset asincrono

// Il clock è un input, non una parola chiave (wire o input che sia)
```

▼ Esempio - Porta AND + FF

Quando arriva il fronte di salita del clock, si vuole che un flip flop memorizzi l'AND tra due ingressi a e b.

```
reg a,b,ff;
always @(posedge clock) // ogni fronte di salita del clock voglio che il ff memorizzi
   ff = a & b;
```

Assegnazioni bloccanti

⚠️ È importante ricordare che, all'interno del corso, utilizzeremo solo l'assegnazione bloccante.

Le assegnazioni bloccanti hanno un effetto immediato (come in C):

```
always @(a or b) begin // 1
   a = b;              // 2
   b = a;              // 3
end                   // 4

// Qualora a o b (o entrambi) abbiano una variazione di valore,
// il blocco always NON verrà eseguito.
```

1. supponiamo che, durante la prima esecuzione del blocco `always` , si abbia `a = 0` e `b = 1`, come in tabella;

	1° riga di codice	2° riga di codice	3° riga di codice	4° riga di codice
a	0	1	1	1
b	1	1	1	1

2. alla seconda riga di codice otteniamo `a = b` e, quindi, `a` passa da 0 a 1, mentre `b` mantiene il suo valore precedente (pari a 1);
3. alla terza riga di codice, `a` mantiene il suo valore precedente (1) e si esegue `b = a`, ma `a = 1`, quindi `b = a = 1`;
4. si raggiunge la fine del blocco `always`: `a` e `b` mantengono i loro valori precedenti (1 e 1).

⚠️ Ci si potrebbe chiedere come mai non si rientri nel blocco `always` , dato che `a` ha cambiato valore.

Questo non avviene perché il blocco `always` viene rieseguito se e solo se `a` o `b`, o addirittura entrambi subiscono variazioni, ma esse devono avvenire al di fuori del blocco `always` stesso.

Visto che le variazioni ai parametri di sensibilità (`a` e `b`) sono solo interne al blocco `always` , quest'ultimo non viene nuovamente eseguito!

Assegnazioni non bloccanti

Le assegnazioni non bloccanti avvengono solo al termine dell'esecuzione del blocco `always` e, in tal senso, comporterebbero una nuova esecuzione del blocco `always` nel caso di variazione.

```
always @(a or b) begin // 1
   a <= b;          // 2 assegna a = b alla fine del blocco always
   b <= a;          // 3 assegna b = a alla fine del blocco always
end                 // 4

// qualora a o b (o entrambi) abbiano una variazione di
// valore, il blocco always verrà eseguito.
```

1. supponiamo che, durante la prima esecuzione del blocco `always`, si abbia `a = 0` e `b = 1`, come in tabella;

1° esecuzione	1° riga di codice	2° riga di codice	3° riga di codice	4° riga di codice	Valore futuro
a	0	0	0	1	1
b	1	1	1	0	0

2. viene eseguita l'istruzione `a <= b` e, quindi, il valore futuro di `a` sarà pari al valore corrente di `b` (1) solo alla fine del blocco `always`;
3. viene eseguita l'istruzione `b <= a` e, quindi, il valore futuro di `b` sarà pari al valore corrente di `a` (0) solo alla fine del blocco `always`;
4. raggiunta la fine del blocco `always` vengono assegnati i valori futuri di `a` e `b`.

⚠️ Poiché `a` e `b` sono variate al di fuori del blocco `always` e sono nella lista di sensibilità, il blocco si riattiva e viene nuovamente eseguito.

2° esecuzione	1° riga di codice	2° riga di codice	3° riga di codice	4° riga di codice	Valore futuro
a	1	1	1	0	0
b	0	0	0	1	1

Poiché, nuovamente, `a` e `b` sono variate e si trovano nella lista di sensibilità, il blocco `always` viene nuovamente eseguito (indefinitivamente, come in figura).

	1ª esecuzione				2ª esecuzione				3ª esecuzione				4ª esecuzione				5ª esecuzione				6ª esecuzione				7ª esecuzione				8ª esecuzione				9ª esecuzione				
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	Futuro
a	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	
b	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	

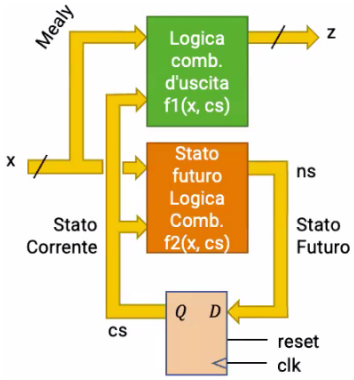
Tipicamente questo comportamento non è desiderato, anche se somiglia a un clock

▼ Esempio - Realizzazione di un circuito sequenziale Mealy (FSM)

Come noto, una macchina di Mealy è tale per cui il valore dell'uscita dipende sia dal valore corrente degli ingressi, sia dallo stato corrente.

Si vuole, quindi, realizzare una macchina di Mealy come in figura.

Si noti che il FF-D è attivo sul fronte di salita.



Macchina di Mealy

```
module MealyFSM (z, clk, reset, x);
   output reg z;
   input clk, reset, x;
   // si poteva anche fare output z; reg z;
   reg ns; // next state (sono come wire ma necessariamente devono essere dichiarati reg)
   reg cs; // current state (come wire ma serve dichiararlo come reg poichè output).

   /// LE REG SONO NECESSARIE PER USARLE COME "uscita" dei blocchi always

   // GESTIAMO LA VARIAZIONE DEGLI STATI (dipendente  da clock e reset)
   always @(posedge clock // attivo sul fronte di salita del clock
                or
                posedge reset) // reset asincrono: se reset passa da 0→1 allora si entra
   begin: stateReg // etichetta utile per documentazione, nome del blocco always
      if (reset)
         cs = 0;
      else
         cs = ns; // current state = next state... aggiorno lo stato corrente
   end
   always @(x or cs) begin: outputFunction
      // la lista di sensibilità è data dall'ingresso e dallo stato corrente.
      z = f1(x,cs); // l'uscita dipende da ingresso e stato corrente.
   end
```

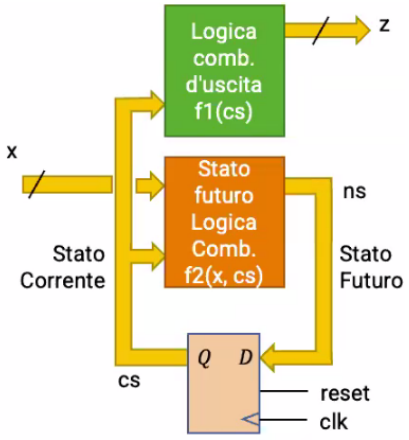
```
    always @(x or cs) begin: nextStateTransition
        ns = f2(x, cs); // il next state dipende da ingresso e stato corrente
endmodule
```

▼ Esempio - Realizzazione di un circuito sequenziale Moore (FSM)

Come noto, una macchina di Moore è tale per cui il valore dell'uscita dipende solo dallo stato corrente.

Si vuole, quindi, realizzare una macchina di Moore come in figura.

Si noti che il FF-D è attivo sul fronte di salita e che il reset è sincrono (cioè deve arrivare insieme al clock).



Macchina di Moore

```
module MealyFSM (z, clk, reset, x);
    output z;
    input clk, reset, x;
    reg ns; // next state (sono come wire ma necessariamente devono essere dichiarati reg)
    reg cs; // current state (come wire ma serve dichiararlo come reg poichè output).
    reg z;
    /// LE REG SONO NECESSARIE PER USARLE COME "uscita" dei blocchi always

    // GESTIAMO LA VARIAZIONE DEGLI STATI (dipendente  da clock)
    always @(posedge clock) // vogliamo un reset sincrono!
    begin: stateReg
        if (reset) // Il reset deve arrivare durante la transizione del clock!
            cs = 0;
        else begin
            cs = ns;
        end
    end
    always @(cs) begin: outputFunction
        // la lista di sensibilità è data dal solo stato corrente.
        z = f1(cs); // l'uscita dipende da ingresso e stato corrente.
    end
```

```
    always @(x or cs) begin: nextStateTransition
        ns = f2(x, cs); // il next state dipende da ingresso e stato corrente
endmodule
```