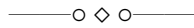


ISTITUTO TECNICO INDUSTRIALE STATALE “ENRICO MATTEI”

Elettronica e Telecomunicazioni

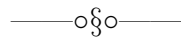
URBINO



NOTE SULLA SINTASSI DEL LINGUAGGIO VERILOG HDL

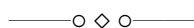
CON ESEMPI ED ESERCIZI

Prima parte



Autore: Ing. M. Zandri

12 settembre 2009



Indice

1	Concetti generali	4
1.1	Moduli	4
2	Azioni procedurali e reti combinatorie	7
2.1	begin ... end	9
2.2	@	10
2.3	#	10
2.4	if .. then .. else	10
2.5	case	12
2.6	always	13
2.7	initial	15
2.8	wait	15
2.9	Esempi	15
3	Azioni procedurali e reti sequenziali	19
3.1	Assegnamento bloccante e non bloccante	20
3.2	Dedurre un latch	22
3.3	Esempi	23

Sommario

Questi appunti sono relativi ad un linguaggio HDL di descrizione delle reti logiche combinatorie e sequenziali, noto con il nome di Verilog©. Il suo utilizzo in un corso di Sistemi Automatici Elettronici per studenti degli Istituti Tecnici Industriali rappresenta sicuramente una novità, ma si pone anche con continuità rispetto al III anno di corso, poiché presenta costrutti sintattici assai simili al linguaggio “C”.

L'autore inoltre, ritiene che vista la diffusione di sistemi elettronici dedicati con logiche programmabili, la conoscenza di questo linguaggio di descrizione dell'hardware, costituisce un bagaglio culturale indispensabile per il futuro perito elettronico.

Se fino a pochi anni fa sistemi di sviluppo basati su VHDL© o Verilog© avevano un costo inaccessibile al singolo utilizzatore, la tendenza attuale è quella di fornire dei software di sviluppo, simulazione e programmazione a costo nullo, seppure con lievi limitazioni, che non influenzano né il processo di apprendimento, né la fase di sperimentazione. Le stesse schede dimostrative hanno oramai una valenza didattica, con un costo spesso inferiore ai 200 €, che le rendono facilmente acquistabili da qualunque istituto scolastico.

I linguaggi VHDL e Verilog di solito sono implementati entrambi su tutti i compilatori HDL, perciò la scelta tra i due può derivare anche solo da criteri estetici. In realtà il VHDL ha una struttura formale più dettagliata e rigida ed è molto utilizzato nei paesi europei; fu sviluppato nella seconda metà degli anni '80 dal Dipartimento della Difesa americano e ricalca il rigore formale di un altro linguaggio di programmazione sviluppato nello stesso ambiente: ADA.

Il Verilog è invece più usato negli Stati Uniti e in Asia ed è la standardizzazione di un linguaggio sviluppato da una azienda americana (Automated Integrated Design Systems) per descrivere e simulare circuiti logici complessi.

Pur essendo rigoroso e altrettanto funzionale del VHDL, presenta un formalismo più simile al linguaggio C e per questo si pone in una visione di continuità con gli argomenti di programmazione, svolti durante il III anno del corso di Sistemi Automatici Elettronici.

La scelta di queste note è caduta su Verilog per il semplice motivo che chi vi si avvicina per la prima volta trova un linguaggio non troppo difforme da altri linguaggi ad alto livello, come C, o FORTRAN e per questo si trova in un ambiente più familiare.

Ciò tuttavia non deve distrarre dal fatto che si sta usando un linguaggio di descrizione per circuiti e non un linguaggio di programmazione per algoritmi, pertanto si dovrà avere in mente il funzionamento fisico del circuito stesso.

1 Concetti generali

Quando si parla di un linguaggio di descrizione hardware, occorre avere in mente la tipologia di circuito che si vuole realizzare.

Benché questo linguaggio sia per taluni aspetti simile al linguaggio C, in realtà il punto di vista è completamente differente. In un linguaggio di programmazione le differenti azioni sono eseguite in maniera consequenziale e pertanto la posizione all'interno del programma influenza lo svolgimento dello stesso.

In un linguaggio di descrizione dell'hardware invece, si **describe** uno o più circuiti logici, i quali possono **funzionare in parallelo e/o indipendentemente**.

Si evidenzia quindi che il punto di vista è del tutto differente: alcuni di questi circuiti possono funzionare in parallelo, altri possono agire a seguito di segnali prodotti dai precedenti circuiti. È evidente quindi che mentre un linguaggio come il C riesce solo a simulare dei processi consequenziali, un linguaggio di descrizione hardware descrive dei processi paralleli.

1.1 Moduli

Un modulo è l'entità di base del linguaggio, che rappresenta anche l'entità di base del circuito logico che si vuole realizzare, ovvero la funzione combinatoria che si vuole ottenere.

Come esempio la seguente funzione:

$$f(x_1, x_2) := y = x_1 * x_2 \quad (1)$$

che rappresenta l'operazione di AND logico tra due ingressi x_1 e x_2 può ben rappresentare un modulo.

Il linguaggio Verilog mette a disposizione il seguente costrutto per realizzare quanto appena descritto:

```
module and2
  (
    input wire x1, x2,
    output wire y
  ) ;

  assign y = x1 & x2;
```

```
endmodule
```

Il modulo richiede un nome che lo definisce (*and2*) ed una lista di parametri che sono i suoi segnali di ingresso e di uscita. Il termine `wire` che può essere anche omesso, indica che si tratta di segnali fisici che andranno in qualche modo realizzati nel circuito logico.

Successivamente alla dichiarazione dei segnali di ingresso ed uscita, si descrive la funzione logica che la rete realizzerà. Un modulo non deve necessariamente contenere una sola funzione logica: ne può contenere un numero arbitrario.

Si supponga di voler realizzare 2 funzioni logiche sugli stessi ingressi x_1 e x_2 : la prima effettua lo AND e la seconda lo OR. La descrizione tramite Verilog diventa:

```
module and2_e_or2
(
    input wire x1, x2,
    output wire y, z
) ;

    assign y = x1 & x2;
    assign z = x1 | x2;
```

```
endmodule
```

L'ordine delle due funzioni è del tutto irrilevante, perché le due funzioni vengono realizzate come due circuiti separati ed indipendenti, che tuttavia condividono gli stessi ingressi.

Questa maniera di descrivere funzioni logiche tramite il linguaggio è detta **descrizione funzionale o comportamentale** con chiaro riferimento dei termini utilizzati.

La definizione della funzione logica $y = f(x_0, x_1)$ è detto *assegnamento continuo*.

Si supponga di definire i seguenti 2 moduli di base:

```
module and2
(
    input wire x1, x2,
    output wire y
```

```

    ) ;

    assign y = x1 & x2;
endmodule
e
module or2
    (
        input wire x1, x2,
        output wire y
    ) ;

    assign y = x1 | x2;
endmodule

```

Il linguaggio Verilog permette di usare dei moduli di base all'interno di altri moduli, in questa semplice maniera:

```

module and2_e_or2
    (
        input wire a1, a2,
        output wire b, c
    ) ;

    and2 istanza_and2 (.x1(a1), .x2(a2), .y(b));
    or2 istanza_or2 (.x1(a1), .x2(a2), .y(c));

endmodule

```

Questo tipo di definizione del modulo prende il nome di **descrizione strutturale**, perché il modulo *and2_e_or2* non contiene le funzioni logiche, ma semplicemente delle istanze di altri moduli.

Occorre prestare attenzione alla corrispondenza tra segnali di ingresso e di uscita:

.x1(a1)

significa che il segnale x_1 del modulo **and2** è collegato al segnale a_1 del modulo **and2_e_or2**. La sintassi per istanziare un componente (o un modulo) già definito è piuttosto semplice:

- il primo campo è il nome del modulo che si vuole istanziare e che è già stato definito altrove;
- il secondo campo è il nome attuale dell'istanza;
- il terzo campo è la corrispondenza sulla lista dei segnali.

Se l'ordine dei segnali, come definito nei moduli di base è rispettato, l'assegnamento può essere semplificato in questo modo:

```
...
and2 istanza_and2 (a1, a2, b);
or2  istanza_or2  (a1, a2, c);
...
```

A conclusione del paragrafo si vuole sottolineare come un linguaggio di descrizione dell'hardware permette due tipologie di descrizione e cioè quella comportamentale e quella strutturale.

2 Azioni procedurali e reti combinatorie

Volendo realizzare la funzione logica di un multiplexer 2 a 1, si dovrebbe scrivere l'equazione combinatoria che descrive il suo funzionamento e poi realizzarla tramite circuiti logici. Ne risulta la seguente equazione minimizzata:

$$y = s * i_1 + \bar{s} * i_0 \quad (2)$$

e pertanto la descrizione in Verilog diventa:

```
module mux2a1(
    input  wire i0, i1, s,
    output wire y
);

    assign y = s & i1 + /s & i0;

endmodule
```

È evidente come in un caso semplice sia possibile descrivere il circuito in questa maniera, ma al crescere degli ingressi le equazioni minimizzate diventano sempre più complesse da sintetizzare a mano.

Il linguaggio Verilog mette a disposizione dei costrutti che prendono il nome di procedurali e assomigliano molto ai tradizionali linguaggi di programmazione, come il C.

All'interno di questi costrutti le azioni vengono svolte in maniera consequenziale. Un modulo pertanto può contenere sia equazioni (o funzioni) logiche, che azioni procedurali.

Occorre prestare attenzione al fatto che **le azioni procedurali non sono necessariamente ottenuti tramite logica sequenziale.**

Tornando all'esempio del multiplexer, una descrizione potrebbe essere la seguente: *se l'ingresso di selezione è 0 allora l'uscita è identica all'ingresso x_0 , altrimenti l'uscita è identica all'ingresso x_1 .*

I costrutti che lasciano al software di sintesi il compito di realizzare le equazioni sono:

```
if ... then ... else
oppure
case
```

Tuttavia questi costrutti non possono essere utilizzati all'interno del modulo direttamente, ma devono essere inclusi in un blocco procedurale.

Il blocco procedurale è definito dalla parola chiave:

```
always@
```

Tutto ciò non è ancora sufficiente, perché per ottenere una corretta realizzazione delle funzioni combinatorie, il compilatore deve conoscere anche l'elenco dei segnali che devono attivare l'esecuzione del blocco procedurale.

Nel caso del multiplexer, trattandosi di una rete puramente combinatoria e senza retroazione, tutti i segnali concorrono all'attivazione del blocco procedurale.

Il simbolo * significa che il blocco procedurale è sensibile a tutti i segnali che vengono utilizzati al suo interno, e quindi ai segnali s , x_0 e x_1 .

C'è da tenere ben presente che quanto scritto **non** è una parte di programma scritta in un linguaggio simile al C, ma è **la descrizione di un circuito combinatorio**. Pertanto il compilatore e successivamente il sintetizzatore utilizzeranno quanto scritto per realizzare un circuito logico, secondo quanto già visto, ad

```

module mux2(
    input wire x0, x1, s
    output reg y
) ;

always@*
    if (s == 0)
        y = x0;
    else
        y = x1;
endmodule

```

Figura 1: *Modulo multiplexer 2 a 1*

esempio con il metodo delle mappe di Karnaugh.

Occorre effettuare alcune precisazioni sulla sintassi utilizzata; l'uscita y non è stata definita come `wire` ma come `reg`.

differenza tra
`wire` e `reg`.

Infatti se l'uscita deriva da un blocco procedurale occorre usare tale termine, che tuttavia non significa che per l'uscita si utilizza un registro, ovvero un insieme di flip-flop. L'uscita è e resta di tipo combinatorio.

Anche l'assegnamento ad un segnale di tipo `reg` avviene con il simbolo `=` e non con la parola chiave `assign`.

2.1 `begin ... end`

Analogamente al linguaggio C, ogni comando procedurale può essere seguito da una sola istruzione; se si vogliono raggruppare più istruzioni bisogna racchiuderle in un gruppo, che in questo caso è delimitata dalle parole riservate: `begin` e `end`. L'uso di questi due delimitatori introduce un costrutto procedurale, all'interno del quale è possibile avere dei comandi consequenziali e dei comandi in "parallelo".

Naturalmente è possibile avere comandi esterni al blocco procedurale, come visto al par. §1.1 e questi sono sempre eseguiti in maniera "parallela", perché vanno pensati come implementazioni circuitali di equazioni logiche.

2.2 @

Questo simbolo definisce una lista di parametri sensibili, cioè variabili il cui cambiamento dello stato logico produce un aggiornamento delle equazioni all'interno del blocco procedurale.

La lista di parametri si esprime attraverso un elenco racchiuso in una parentesi tonda: (*parm1*, *parm2*, ...). La presenza di più parametri nella lista equivale ad un OR logico tra gli stessi.

Se si vogliono introdurre tutti i parametri ed evitare di scrivere una lista troppo lunga, si può usare il simbolo *.

La lista dei parametri sensibili, definisce quali segnali generano un evento; quest'ultimo va ad aggiornare il simulatore e come conseguenza si ha il ricalcolo delle equazioni presenti nei blocchi procedurali. È da notare che le equazioni esterne ai blocchi procedurali sono sempre aggiornate in maniera automatica, eventualmente possono ricevere dei comandi di ritardo.

2.3

Costituisce l'assegnamento di un ritardo temporale nell'esecuzione di una assegnamento in una equazione logica. Il numero che segue il simbolo indica la durata in termini di finestre temporali, del ritardo con cui verrà aggiornata l'equazione logica.

2.4 if .. then .. else

La sintassi generale del costrutto if è il seguente:

```
if (condizioni)
    ...
    ...
else if (condizioni2)
    ...
    ...
else
    ...
    ...
```

dove *condizione* è una o più condizioni logiche concatenate dai soliti operatori. Nell'utilizzo del costrutto `if` occorre prestare attenzione ad una caratteristica del compilatore di Verilog: *se in un blocco procedurale esistono differenti percorsi ed uno o alcuni di essi non assegnano valori alle variabili di uscita, allora la variabile mantiene il suo valore precedente, ovvero conserva lo stato.* Questo comportamento, se applicato ad un circuito puramente combinatorio, lo trasforma in un circuito con stati interni, ovvero sequenziale.

Tornando all'esempio di fig. 1 se si fosse scritto:

```
module mux2(  
    input wire x0, x1, s  
    output reg y  
);  
  
always@*  
    if (s == 0)  
        y = x0;  
endmodule
```

Figura 2: Modulo multiplexer 2 a 1 con assegnamento di stato interno

si sarebbe generata una rete sequenziale piuttosto che una rete combinatoria, poiché lo stato della variabile di uscita *y* non è definito nel ramo `else`, in quanto tale ramo manca. Il compilatore deduce che lo stato della variabile *y* va mantenuto e quindi pone un flip-flop prima di far uscire il segnale *y*. Sicuramente si tratta di un grave errore logico.

La soluzione è nel rimettere il ramo `else` oppure assegnare il valore di uscita prima di eseguire il costrutto `if`.

Pertanto il compilatore Verilog assegna stati interni a funzioni combinatorie in cui non tutti gli ingressi sono stati definiti; ciò significa che funzioni combinatorie sono trasformate in funzioni sequenziali.

Un comportamento analogo è presente anche nel costrutto `case`, che è riconducibile ad una catena di `if`.

```

module mux2(
    input wire x0, x1, s
    output reg y
) ;
always@*
    y = x1;    //assegno l'uscita
    if (s == 0)
        y = x0; //la modifico se necessario
endmodule

```

Figura 3: *Modulo multiplexer 2 a 1 con assegnamento di variabile*

2.5 case

La sintassi del costrutto `case` è molto simile a quella del corrispondente in C.

```

case (case-expr1)
[item]:
    begin
        [procedural statement1] ;
        [procedural statement1] ;
        . . .
    end
...
...
default:
    begin
        [procedural statement1] ;
        [procedural statement1] ;
        . . .
    end
endcase

```

Se l'utilizzo di questo costrutto serve a descrivere una rete combinatoria, occorre ricordare che le variabili di uscita vanno assegnate per tutti i possibili valori di

case-expr1.

Si debba realizzare un multiplexer 4 a 1 di cui sono utilizzati solo i due ingressi x_0 e x_1 utilizzando il costrutto appena visto. Il blocco procedurale diventa quello della fig. 4.

```
module mux2(  
    input wire x0, x1, x2, x3,  
    input wire [1:0] s,  
    output reg y  
);  
always@*  
    case (s)  
        2'b00:  
            y = x0;  
        2'b01:  
            y = x1;  
        default:  
            y = x0;  
    endcase  
endmodule
```

Figura 4: Modulo multiplexer 4 a 1 con utilizzo di 2 soli ingressi

Il termine `default` fa comprendere al compilatore che i casi 10 e 11 permettono di assegnare all'uscita y l'ingresso x_0 e quindi nessuno stato interno è generato automaticamente. Se si fosse dimenticato il ramo di `default` il compilatore avrebbe aggiunto un flip-flop.

2.6 always

Un blocco procedurale definito con questa parola chiave, viene eseguito ripetutamente e costituisce quindi un *processo*, ovvero una entità funzionale definita.

All'interno di un modulo possono essere definiti un numero arbitrario di costrutti procedurali, tramite la parola chiave `always@`.

Un errore che occorre evitare è che la stessa variabile sia assegnata in differenti

costrutti `always@`. Il sintetizzatore produrrebbe una stessa uscita pilotata da differenti circuiti logici e ciò produrrebbe sicuramente dei conflitti elettrici.

È sicuramente da evitare di scrivere il pezzo di codice riportato in (fig. 5). Il

```
reg y;
reg a , b , clear ;
always @*
    if ( clear )
        y = 1'b0 ;

always @*
    y = a & b ;
```

Figura 5: *Assegnamento alla stessa variabile in 2 blocchi procedurali*

primo blocco assegna ad y il valore logico 0 se `clear` è a livello logico alto, e contemporaneamente assegna ad y l'AND logico tra a e b . Siccome i due blocchi sono realizzati come circuiti separati si evidenzia immediatamente un conflitto elettrico. In questi casi il sintetizzatore può emettere un messaggio di errore ed interrompere il processo di sintesi.

Sicuramente sintetizzabile è la seguente funzionalità logica:

```
reg y;
reg a , b , clear ;
always @*
    if ( clear )
        y = 1'b0 ;
    else
        y = a & b ;
```

Figura 6: *Assegnamento alla stessa variabile in 1 blocco procedurale*

Ciò che è stato analizzato finora ha permesso di ottenere solamente delle funzioni di tipo combinatorio. Si tratta ora di analizzare come ottenere delle funzioni

sequenziali, cioè come far comprendere al compilatore che si vogliono realizzare delle funzioni che hanno memoria dello stato.

Il costrutto `always` richiede una lista di parametri alle cui variazioni esso è sensibile. La forma completa è la seguente:

```
reg y;
reg a , b , clear ;
always @ (a, b)
    if ( clear )
        y = 1'b0 ;
    else
        y = a & b ;
```

In questo caso, il costrutto procedurale è eseguito ogni volta che avviene una transizione sui segnale `a` oppure `b`.

Analogamente al linguaggio C, dopo ogni comando procedurale si può mettere una sola istruzione, oppure un blocco di istruzioni racchiuse tra le parole `begin` e `end`.

2.7 initial

È del tutto identico al al comando precedente, ma viene eseguito **una sola volta**, all'inizio della simulazione.

2.8 wait

`wait` è un altro costrutto procedurale che permette di sospendere un processo finché non diventa vera la condizione che lo segue. La sintassi è:

```
wait (condizioni)
```

dove `condizioni` sono delle espressioni che forniscono un valore logico. Quindi `wait` non è sensibile agli eventi (transizioni di segnali), ma ai livelli dei segnali.

Un esempio si può trovare nel prossimo paragrafo.

2.9 Esempi

1. *Descrivere tramite Verilog HDL un multiplexer 8 a 1.*

Un multiplexer 8 a 1 può essere descritto usando un metodo procedurale tramite costrutto `case`.

```

module mux8a1(
    input wire [7:0] x,
    input wire [2:0] s,
    output reg y
) ;
always@*
    case (s)
        3'b000:
            y = x[0];
        3'b001:
            y = x[1];
        ...
        ...
        3'b111:
            y = x[7]
    endcase
endmodule

```

2. *Descrivere la seguente funzione logica: $y = \sum(0, 1, 4, 5, 7)$*

Si ricorda che la notazione utilizzata è quella del formalismo somme di prodotti (sp) e quindi i numeri indicano i mintermini che rappresentano lo 1 logico della funzione.

```

module f8ingressi(
    input wire [2:0] x,
    output reg y
) ;
always@*
    case (x)
        3'b000, 3'b001, 3'b100, 3'b101, 3'b111:
            y = 1'b1;
        default:
            y = 1'b0;
    endcase
endmodule

```

Si può notare che la funzione è puramente combinatoria e che quindi tutti i possibili casi del costrutto vanno coperti, altrimenti il compilatore deduce degli stati interni. L'uso del termine **default** permette di porre a 0 i maxtermini, mentre è possibile raggruppare le combinazioni degli ingressi che producono i mintermini.

3. *Fornire una descrizione tramite Verilog di un flip flop di tipo Set Reset.*

Occorre innanzi tutto notare che le logiche programmabili non presentano al loro interno dei flip flop SR dedicati, poiché di utilizzo sicuramente meno diffuso rispetto agli altri tipi di flip flop. Per questo la sintesi di tale circuito va effettuata direttamente tramite equazioni logiche, usando l'assegnamento continuo. Una possibile soluzione, usando 2 NOR a 2 ingressi è la seguente:

```

module ffSR(
    input wire s, r,
    output wire q, notq
) ;
assign q = ~(r | notq);
assign notq = ~(s | q)

endmodule

```

Siccome la struttura interna della FPGA è costituita non da singole funzioni logiche, ma da LUT (look up table), di solito a 4 ingressi, questa realizzazione può costare ben 2 LUT, perché il software di sintesi non ha a disposizione funzioni più elementari. Si deve tener conto che con una LUT si può ottenere una funzione combinatoria a 4 ingressi, sicuramente più complessa del NOR2 e pertanto far sintetizzare flip-flop SR è un grosso spreco di risorse.

4. *Uso della funzione wait.*

È possibile usare questo costrutto per sincronizzare 2 o più processi, definiti da 2 o più costrutti `always`.

```

module syncProc(
    input wire [2:0] x,
    output reg y, z
);

reg sincronismo;

// primo processo
always@*
    case (x)
        3'b000, 3'b001, 3'b100, 3'b101, 3'b111:
            begin
                y = 1'b1;
                sincronismo = 0;
            end
        default:

```

```

        begin
            y = 1'b0;
            sincronismo = 1;
        end
    endcase

    // secondo processo sincronizzato al primo
    always          // sempre attivo senza lista di parametri sensibili
        z = x[0] | x[1];    // questa uscita è inizializzata
    begin
        wait (sincronismo)
            z = x[0] * x[1];
    end
endmodule

```

Il `wait` controlla il valore logico della variabile `sincronismo` e se è bassa, il processo si iberna, finché la stessa variabile non assume il valore alto. A quel punto l'uscita `z` viene aggiornata con l'operazione di AND tra i 2 ingressi. È evidente che l'uscita `z` va assegnata prima del comando `wait`, altrimenti durante l'ibernazione l'uscita sarebbe indefinita e quindi verrebbe associata ad un latch.

3 Azioni procedurali e reti sequenziali

In Verilog non esistono costrutti particolari che permettono di dedurre degli stati interni; il compilatore utilizza una forma differente di assegnamento all'interno del blocco procedurale `always@` per ottenere questo comportamento. Se si vuole sintetizzare un flip flop di tipo D si scrive:

```
module D_ff(  
    input wire clk, d,  
    output reg q );  
  
    always@ (posedge clk)  
        // da qui deduce che si vuole un flip-flop  
        q <= d;  
endmodule
```

Si nota immediatamente che la variabile di uscita q non è assegnata con segno `=`, ma con segno `<=`.

La differenza tra le due scritture risiede nell'istante temporale in cui le espressioni sono valutate ed assegnate; nel primo caso l'espressione è valutata ed assegnata immediatamente alla variabile di sinistra; nel secondo caso l'espressione è valutata, ma assegnata alla variabile di sinistra al prossimo evento di campionamento esattamente come avviene in un flip-flop di tipo D. La prima tipologia di assegnamento è detta *bloccante*, mentre la seconda *non bloccante*.

assegnamento
bloccante e
non

Il termine `posedge` all'interno della lista dei parametri sensibili, significa che del segnale `clk` verrà considerata la sola transizione dallo 0 logico allo 1 logico. Esiste anche la controparte negata che è `negedge`.

Un flip flop con reset attivo alto si può descrivere come riportato in fig. 7.

È possibile che in un modulo coesistano parti procedurali e parti comportamentali, cioè ci siano più istanze del costrutto `always@` e più istanze di assegnamento diretto, cioè funzioni combinatorie.

Questa parte verrà approfondita nel prossimo paragrafo dove si tratteranno le macchine a stati, dette anche automi e ne verrà fornita una descrizione hardware.

Si anticipa solo il fatto che esiste una relazione molto stretta tra macchine a stati ed algoritmi di programmazione e pertanto avere a disposizione un linguaggio di descrizione hardware facilita la sintesi e la verifica delle funzionalità.

```

module ff(
    input wire d, ck, reset
    output reg q
) ;
always@(posedge clk)
    if (reset)
        q <= 1'b0;    //uscita a "0"
    else
        q <= d;      //uscita che segue l'ingresso
endmodule

```

Figura 7: *Descrizione di flip flop con reset sincrono*

3.1 Assegnamento bloccante e non bloccante

Occorre precisare cosa si intende per assegnamento bloccante e non. Il simulatore Verilog esegue questa operazione usando un procedimento ad eventi discreti, cioè le equazioni vengono valutate a precisi istanti di tempo. Quando si trova ad analizzare un costrutto non bloccante, esamina l'espressione a destra del simbolo `<=` e alla fine dell'intervallo temporale assegna il valore calcolato alla variabile a sinistra di `<=`. Ciò accade in parallelo per tutte le istruzioni che presentano questo tipo di assegnamento e quindi la valutazione di ciascuna non agisce da blocco sulla valutazione delle altre.

Il costrutto bloccante invece, assegna il risultato dell'espressione a destra di `=` alla variabile di sinistra immediatamente (se non è specificato esplicitamente un ritardo), ma non permette di eseguire un'altra istruzione finché quella in esecuzione non è terminata. Avendo più istruzioni in sequenza, con alcune di esse che presentano dei ritardi, nell'assegnamento bloccante lo statment successivo è eseguito solo dopo che il precedente è terminato.

Si può portare come esempio un semplice circuiti che effettua il controllo di 2 ingressi ad 1 bit e restituisce 1 se sono identici, altrimenti 0.

In questo caso si applicano entrambe i tipi di assegnamento, bloccante e non bloccante.

```

module eql-block (
    input wire i0, i1,
    output reg eq );

```

```

reg p0, P1;

always @(i0,il) // only i0 and il in sensitivity list
            // the order of statements is important
begin
    p0 = ~i0 & ~il;
    p1 = i0 & il;
    eq = p0 | p1;
end
endmodule

```

Questo primo esempio è stato realizzato con l'assegnamento bloccante; molto importante è l'ordine degli assegnamenti, perché se una variabile non è assegnata viene dedotto un flip-flop. Si analizzi allora l'esempio seguente, in cui `eq` è valutata senza che `p0` e `p1` siano assegnati: viene indotto un flip flop per ciascuna variabile.

```

module eql-block (
    input wire i0, i1,
    output reg eq );

reg p0, P1;

always @(i0,il)

begin
    eq = p0 | p1; // ATTENZIONE: vengono indotti 2 flip flop
    p0 = ~i0 & ~il;
    p1 = i0 & il;
end
endmodule

```

Questo problema è presente perché gli assegnamenti bloccanti rispettano strettamente l'ordine con cui sono elencati.

Si analizza ora l'uso dell'assegnamento non bloccante, in cui le espressioni vengono valutate all'inizio della finestra temporale dell'evento e assegnate alla fine della stessa. Qui i flip flop sono indotti per definizione stessa del comportamento dell'assegnamento stesso.

```

module eql-non-block (
    input wire i0, i1,
    output reg eq );

    reg p0, p1;

    always @(i0,i1,p0,p1) // i0, i1, p0 e p1 sono nella lista sensibile
                        // l'ordine dei comandi non è importante
    begin
        p0 <= ~i0 & ~i1;
        p1 <= i0 & i1;
        eq <= p0 | p1;
    end
endmodule

```

Si può notare che nella lista sensibile ci sono anche i segnali `p0` e `p1`; il comportamento è il seguente: quando si ha un cambiamento sui segnali `i0` e `i1` il costrutto `always@` viene eseguito e vengono valutate le espressioni a destra dei simboli `<=`. Naturalmente `p0` e `p1` mantengono i valori precedenti. Alla fine della finestra temporale vengono aggiornati `p0`, `p1` e `eq`. Siccome `p1` e `p0` sono nella lista sensibile, allora `always@` si riattiva e alla fine della seconda finestra temporale `eq` assume il valore esatto.

A conclusione si può notare che l'assegnamento bloccante permette di ottenere il risultato di `eq` nello stesso istante in cui viene eseguito il blocco `always@`, mentre con l'assegnamento non bloccante occorre attendere 2 finestre temporali prima di avere il risultato finale.

Per questo le **logiche combinatorie** andrebbero descritte con **assegnamenti bloccanti**.

3.2 Dedurre un latch

È possibile dedurre un *latch*, cioè un elemento di memoria non sensibile alla transizione, come il flip-flop D, tramite un blocco procedurale con assegnamento bloccante.

Per ottenere questo risultato è necessario ricordare quando detto al par. § 2.4: *se un ramo di un if non prevede un assegnamento all'uscita, viene dedotto un latch*.

Quindi è possibile agire in questa maniera:

```

module latch(
    input wire d, en

```

```

        output reg q
    ) ;

    always@(d, en)
        if (en)
            q = d;
endmodule

```

L'analisi è semplice: se accade una variazione sui segnali `d` oppure `en`, il blocco `always` viene eseguito e si valuta il livello logico di `en`. Se è alto, allora l'uscita riceve il valore dell'ingresso, mentre se è basso, non viene specificata l'uscita. Ciò significa che viene mantenuto il valore precedente per `q` ovvero viene generato un elemento di memoria sensibile al livello e non al fronte.

3.3 Esempi

Descrivere un registro costituito da 4 flip flop di tipo D con reset attivo alto e asincrono.

```

module reg4(
    input wire [3:0] d,
    input wire ck, reset,
    output reg [3:0] q
) ;
    always@(posedge clk, reset)
        if (reset)
            q <= 0;          //4 uscite a "0"
        else
            q <= d;          //uscita che segue l'ingresso
endmodule

```