7. Verilog

Rappresentazioni HDL

Per descrivere e progettare circuiti hardware, si usano appositi linguaggi di descrizione, detti hardware description languages (HDL). Il loro scopo è quello di fare l'analisi del circuito, per descriverne le funzioni logiche, mentre per la produzione e la sintesi ottimizzata del circuito vengono usati appositi strumenti CAD (Computer Aided Design).

I principali HDL sono:

- SystemVerilog
- VHDL 2008

In questo corso, si analizzerà solo l'uso del Verilog.

Da HDL ai circuiti

Gli HDL permettono di trasformare in modo semplice il codice in una *netlist* che descrive l'hardware (per esempio, una lista di porte ed i cavi che le connettono), sottoforma di file testuale o disegno schematico. Il sintetizzatore logico applicherà delle ottimizzazioni per ridurre la quantità di hardware necessario.

In genere è possibile anche eseguire simulazioni per fare debug del comportamento dei circuiti prima di produrli come hardware.

Quando si scrive codice verilog, è comodo pensare al sistema in termini di logica combinatoria, registri ed automi a stati finiti.

Sintassi verilog

I moduli in verilog

In verilog, un qualsiasi blocco hardware con input e output è chiamato modulo: per esempio, porte AND o multiplexer sono moduli hardware.

Possiamo dividere i moduli in due categorie:

- Moduli comportamentali: descrivono quello che un modulo fa
- Moduli strutturali: descrivono come il modulo è formato da moduli più semplici (per esempio, un decodificatore formato da porte AND)

Modulo comportamentale

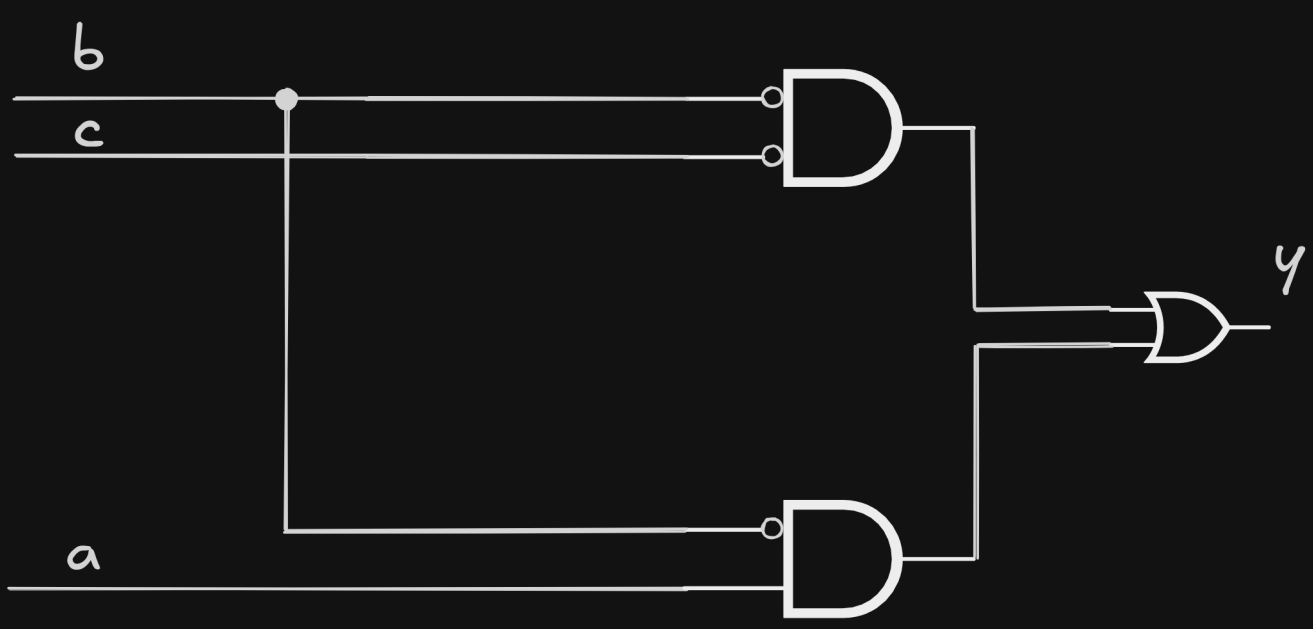
Diamo un'occhiata al seguente modulo ed analizziamo la sintassi del linguaggio:

```
module example(input logic a, b, c,
                           output logic y);
        assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
endmodule
```

Notiamo che la dichiarazione di un modulo è simile a quella di una funzione nei linguaggi di programmazione.

- `module` ed `endmodule` servono per indicare l'inizio e la fine del modulo che stiamo dichiarando
- `example` è il nome del modulo
- `~` è l'operatore NOT, `&` è l'operatore AND e `|` è l'operatore OR

La sintesi di questo modulo produrrebbe il seguente circuito già ottimizzato (le mappe di Karnaugh e la tavola di verità verificano che il circuito descritto è stato già ottimizzato in fase di sintesi):



a	b	c	y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

ab \ c	0	1
00	1	0
01	0	0
11	0	0
10	1	1

$y = \bar{a}\bar{b} + \bar{b}\bar{c}$

$\bar{a}\bar{b}$

$\bar{b}\bar{c}$

Alcune cose di cui tenere conto:

- Il linguaggio è Case sensitive (`reset` e `Reset` non sono lo stesso segnale)
- I nomi non possono iniziare con numeri (`2mux` non è un nome valido)
- Gli spazi bianchi sono ignorati
- Gli _ nella scrittura dei numeri sono ignorati (ex. 1010_1010 = 10101010)
- Si possono scrivere commenti

```
// commento su una riga
/* commento
multilinea*/
```

Modulo strutturale

Si possono definire moduli semplici per comporre un modulo più complesso (proprio come si possono definire funzioni che fanno cose semplici per usarle in funzioni che fanno cose più complesse).

```
module and3(input logic a, b, c,
                    output logic y);
        assign y = a & b & c;
endmodule
--------

module inv(input logic a,
                   output logic y);
        assign y = ~a;
endmodule
--------

module nand3(input logic a, b, c,
                         output logic y);
        /*
        n1 è un segnale interno: non è ne un input o un output, ma è
usato
        solo internamente al modulo, come se fosse una variabile locale
in
        un linguaggio di programmazione.
        */
        logic n1;
        and3 andgate(a,b,c,n1); // istanza del modulo and3
        inv inverter(n1, y); // istanza del modulo inv
```

Operatori bitwise, bus ed operatori di riduzione

```
module gates(input logic [3:0] a, b,
                         output logic [3:0] y1, y2, y3, y4, y5);
        /* Cinque porte logiche che agiscono
           su bus da 4 bit */
        assign y1 = a & b; // AND
        assign y2 = a | b; // OR
        assign y3 = a ^ b // XOR
        assign y4 = ~(a & b); // NAND
        assign y5 = ~(a | b); // NOR
endmodule
```

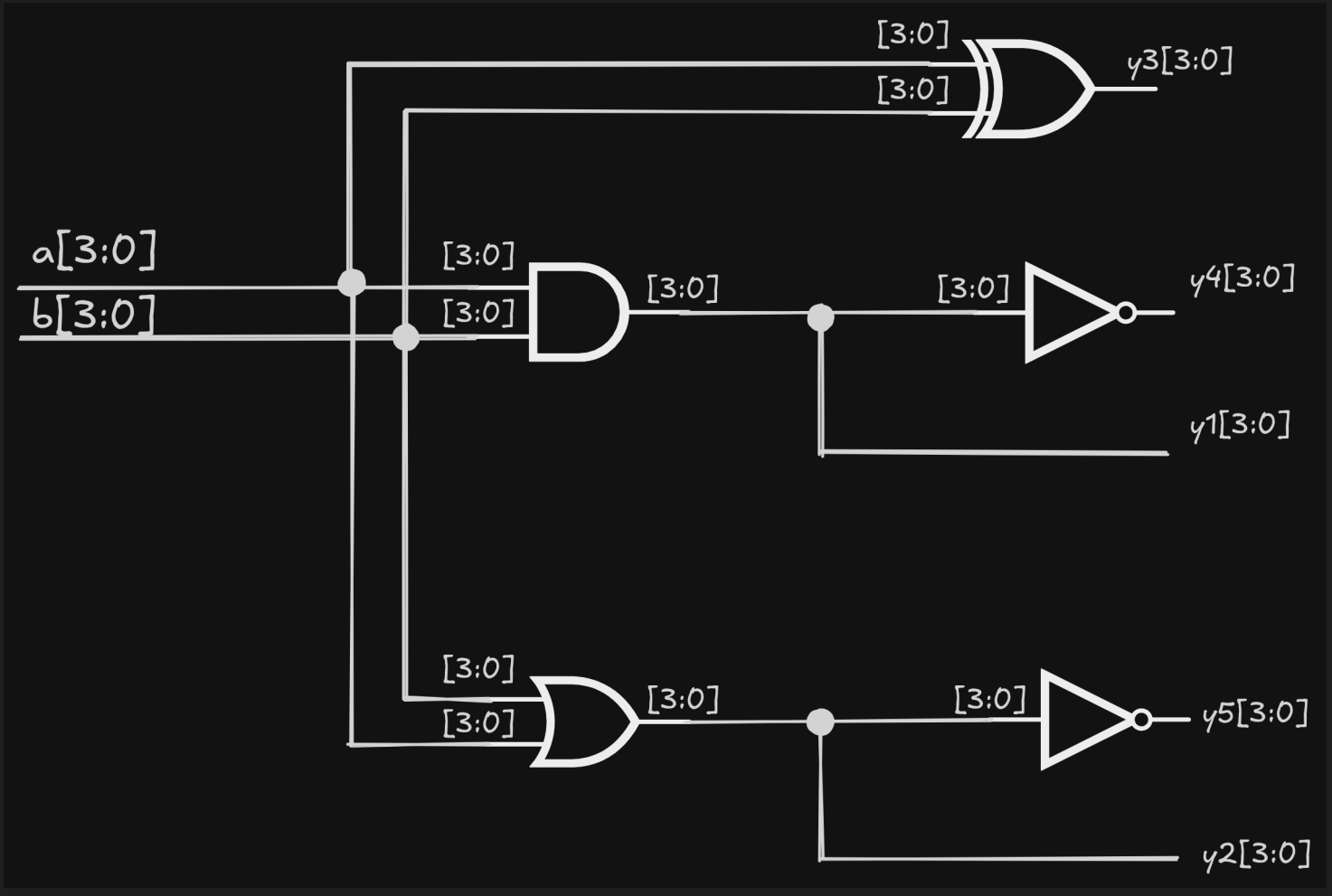
Con `a[3:0]` stiamo indicando un bus a 4 bit denominati dal più significativo al meno significativo, indicandoli con i propri indici: considerando per esempio `a[3]=1`, `a[2]=0`, `a[1]=1`, `a[0]=1`, allora `a` rappresenta `1011`.

Il modulo in questione prende in input due bus da 4 bit, ed in output restituisce il risultato delle operazioni tra i due bus effettuate tramite gli operatori bitwise, che agiscono sia su segnali a singolo bit che su bus multi-bit. Considerando per esempio `a=1011` e `b=1100`, il risultato delle operazioni sarà:

- `y1 = 1000`
- `y2 = 1111`
- `y3 = 0111`
- `y4 = 0111`
- `y5 = 0000`

Inoltre, i numeri usati nella notazione per rappresentare un bus sono gli indici, quindi gli stessi bus usati nell'esempio potevano anche essere scritti come `a[4:1]`, usando i relativi indici, oppure `a[0:3]` per specificare i bit dal meno significativo al più significativo.

Il circuito in forma minimale sintetizzato da questo modulo è il seguente:

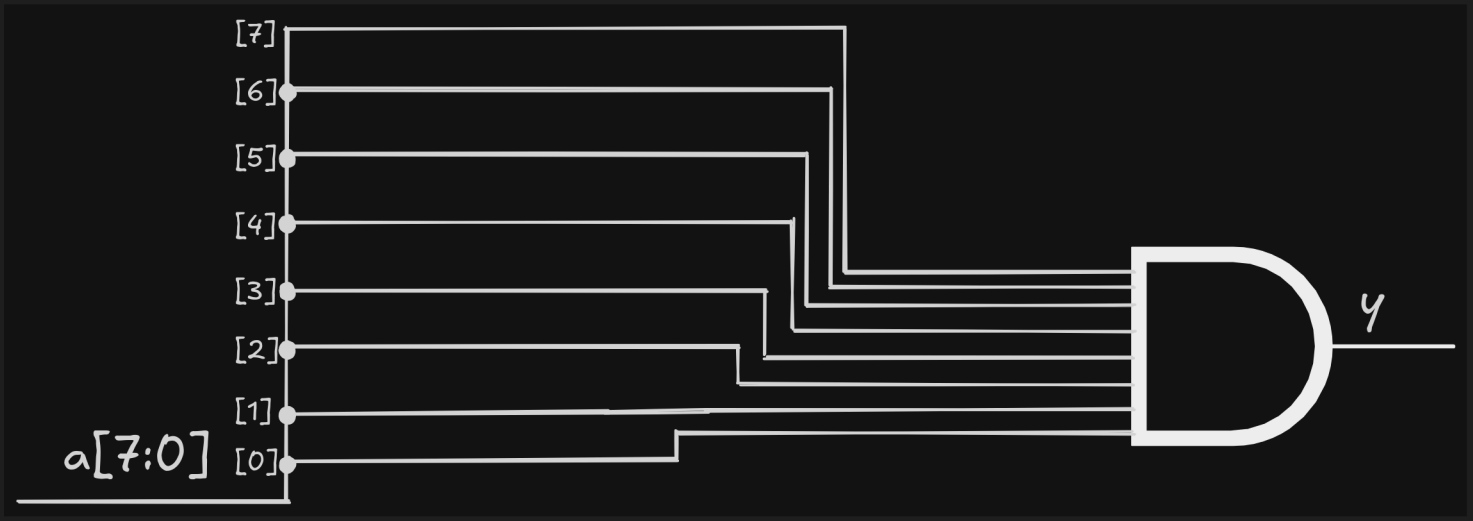


Quindi, quando scriviamo `[3:0]`, ci stiamo riferendo ad un bus di 4 bit, dal più significativo al meno significativo, del segnale. Per esempio, con un bus `a` ad 8 bit `10011100`, con `a[3:0]` ci stiamo riferendo a `1100`.

Infine, si possono applicare gli operatori di riduzione sui bus, che implicano l'azione di una porta logica ad input multipli su un singolo bus.

```
module and8(input logic [7:0] a,
                    output logic      y);
        assign y = &a;
        /* Equivale a scrivere
           assign y = a[7] & a[6] & a[5] & a[4] & a[3] & a[2] & a[1] &
a[0]*/
endmodule
```

Che corrisponde a:



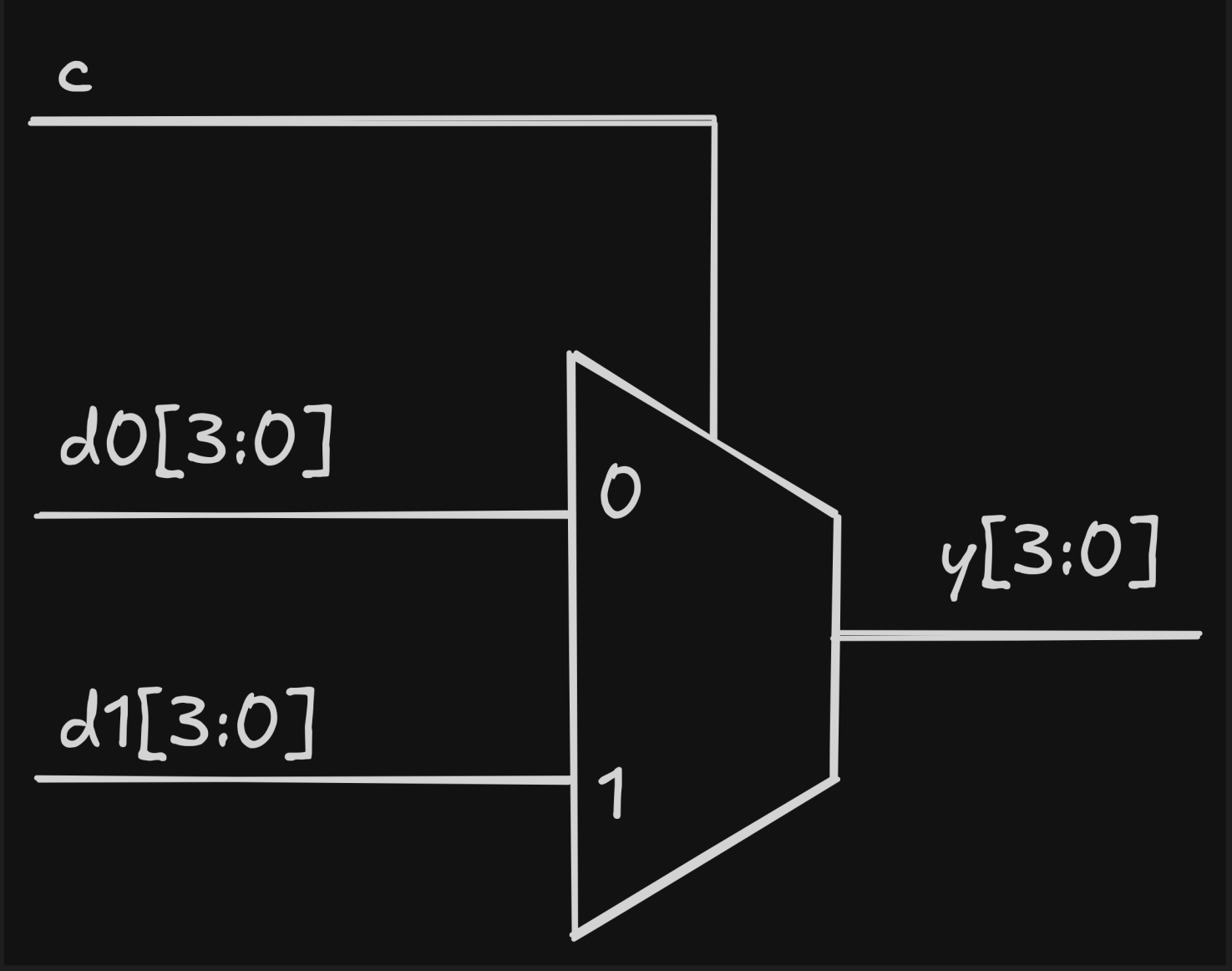
Gli operatori di riduzione si applicano anche per OR, XOR, NAND, NOR e XNOR, ricordando che l'XOR multi-input esegue il controllo di parità: restituisce `true` se una quantità dispari di input sono `true`.

Assegnamento condizionale

L'assegnamento condizionale del verilog permette di scegliere il valore di un segnale in base ad un input di condizione

```
module mux2(input logic [3:0] d0, d1,
                    input logic c,
                    output logic [3:0] y);
        assign y = c ? d1 : d0;
endmodule
```

Praticamente, si tratta dell'utilizzo di quello che in molti linguaggi di programmazione è detto operatore ternario. Si può leggere come "*Se c è vero, allora ritorna d1, altrimenti ritorna d0*".

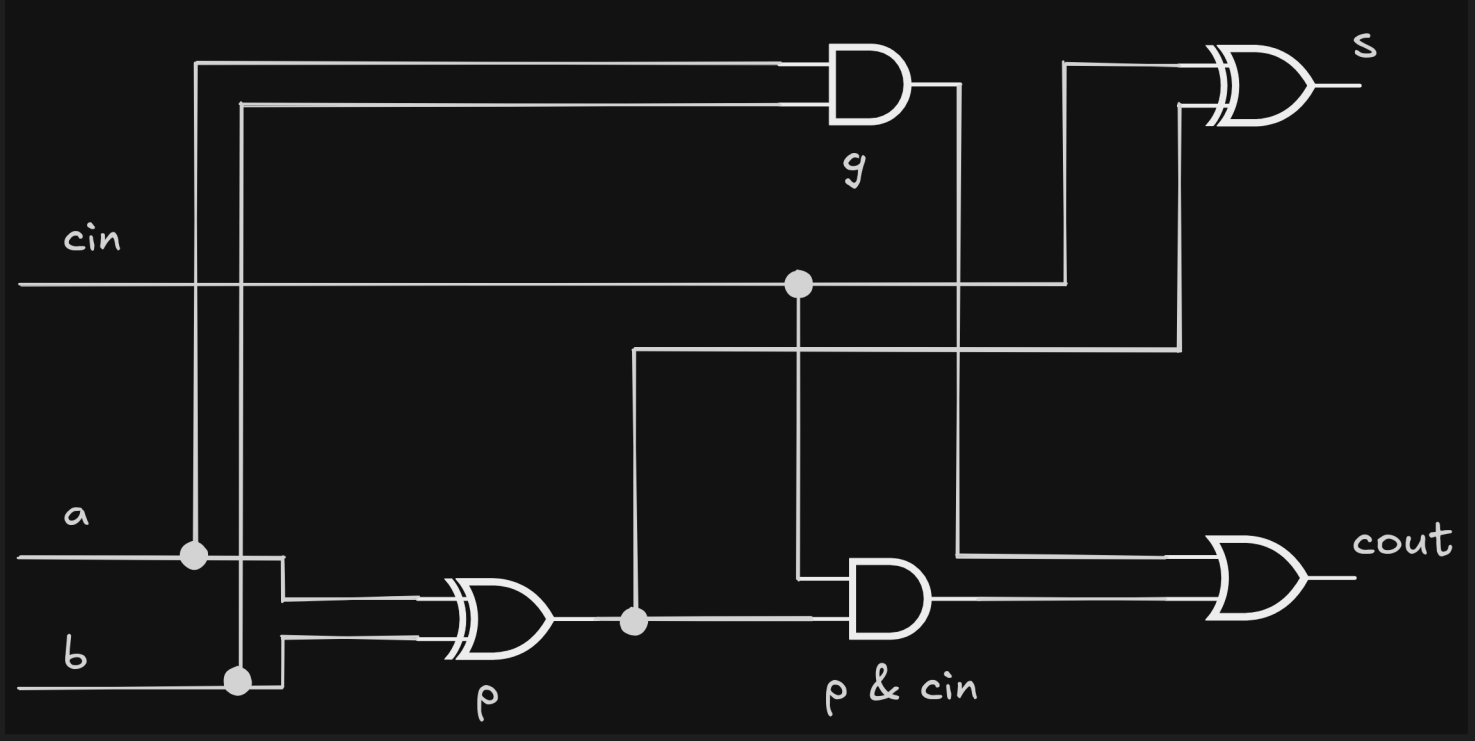


Il costrutto `if-else` esiste, ma può essere usato solo dentro il corpo di `initial` (per descrivere un processo di inizializzazione eseguito una volta sola) ed `always` (per descrivere logica sequenziale), che vedremo più avanti.

Segnali interni

Come abbiamo accennato in precedenza, si possono usare segnali interni che agiscono come variabili locali.

```
module fulladder(input logic a, b, cin,
                                 output logic s, cout);
        logic p, g; // Segnali interni

        assign p = a ^ b;
        assign g = a & b;

        assign s = p ^ cin;
        assign cout = g | (p & cin);
endmodule
```



Precedenza degli operatori

Nella valutazione delle espressioni, viene data priorità maggiore alle operazioni in alto, e minore alle operazioni in basso.

Simbolo	Significato
`~`	NOT
`*`, `/`, `%`	mult, div, mod
`+`, `-`	add, sub
`<<`, `>>`	shift
`<<<`, `>>>`	arithmetic shift
`<`, `<=`, `>`, `>=`	comparison
`==`, `!=`	equal, not equal
`&`, `~&`	AND, NAND
`^`, `~^`	XOR, XNOR
|, ~|	OR, NOR
`?:`	Ternary operator

Numeri

I numeri possono essere specificati in binario, ottale, decimale o esadecimale.
Di default viene usata la rappresentazione decimale, ma è consigliato specificare i numeri con la notazione *N'Bvalue*, dove N è il numero di bit, B è la base e value è il valore.

Numero	Numero di Bits	Base	Valore decimale	In memoria
3'b101	3	Binario	5	101
'b11	Non specificato	Binario	3	00...0011
8'b11	8	Binario	3	00000011
8b'1010_1011	8	Binario	171	10101011
3'd6	3	Decimale	6	110
6'o42	6	Ottale	34	100010
8'hAB	8	Esadecimale	171	10101011
42	Non specificato	Decimale	42	00...0101010

Il tipo logic

Il tipo logic in verilog può assumere come valori `0`, `1`, `z` e `x`.

- `x` indica un livello logico non valido: se un bus è simultaneamente portato a `0` ed `1`, il risultato è `x` che indica conflitto. All'inizio di una simulazione, nodi a stati come gli output dei flip-flop sono inizializzati su uno stato indefinito, che in verilog è `x`. Viene assegnato il tipo `x` anche a segnali non inizializzati.
- `z` indica un valore flottante. Viene utilizzato in modo particolare nel *tristate buffer*, il quale quando è abilitato ha come output lo stesso valore dell'input, e quando è disabilitato l'output è `z`.

Esempi di manipolazione dei bit

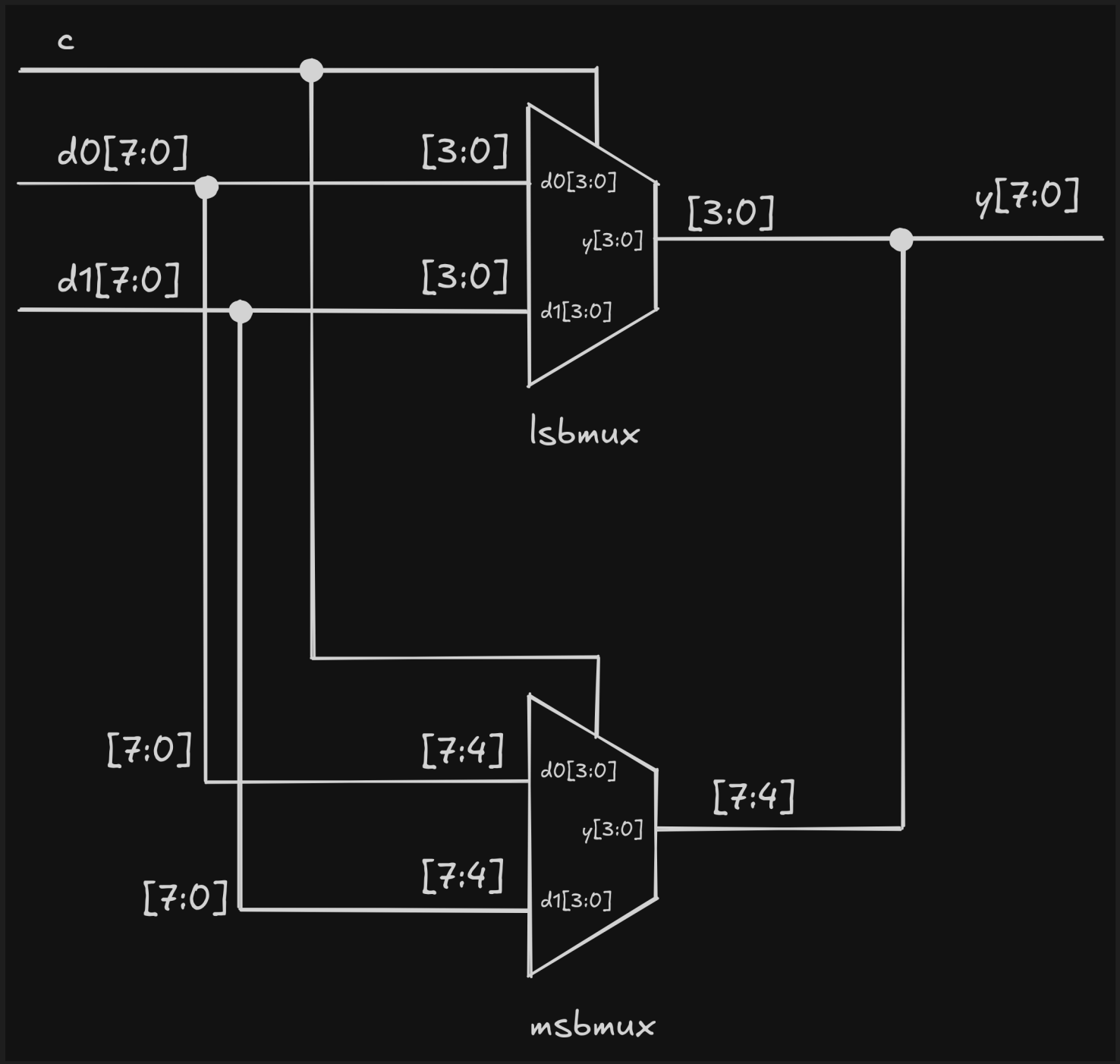
Assegnazione manuale dei bit

```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};

// Considerando y come un bus a 12 bit, y corrisponde allora a
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0
```

- `a[2:1]` prende il terzo ed il secondo bit di `a`
- `{3{b[0]}}` prende 3 volte il primo bit di `b`
- `a[0]` prende il primo bit di `a`
- `6'b100_010` prende `100010` in binario usando 6 bit. Ricordiamo che gli `_` nella rappresentazione dei numeri servono solo per rendere il codice più leggibile, verilog li ignora.

Uso di due mux per generare le due parti di un segnale

```
module mux2(input logic [3:0] d0, d1,
                    input logic c,
                    output logic [3:0] y);
        assign y = c ? d1 : d0;
endmodule
--------

module mux2_8(input logic [7:0] d0, d1,
                          input logic c,
                          output logic [7:0] y);
        /* uso due mux2: con il primo calcolo i bit meno
          significativi, con l'altro i più significativi. */
        mux2 lsbmux(d0[3:0], d1[3:0], c, y[3:0]); // Least significant
        mux2 msbmux(d0[7:4], d1[7:4], c, y[7:4]);
endmodule
```

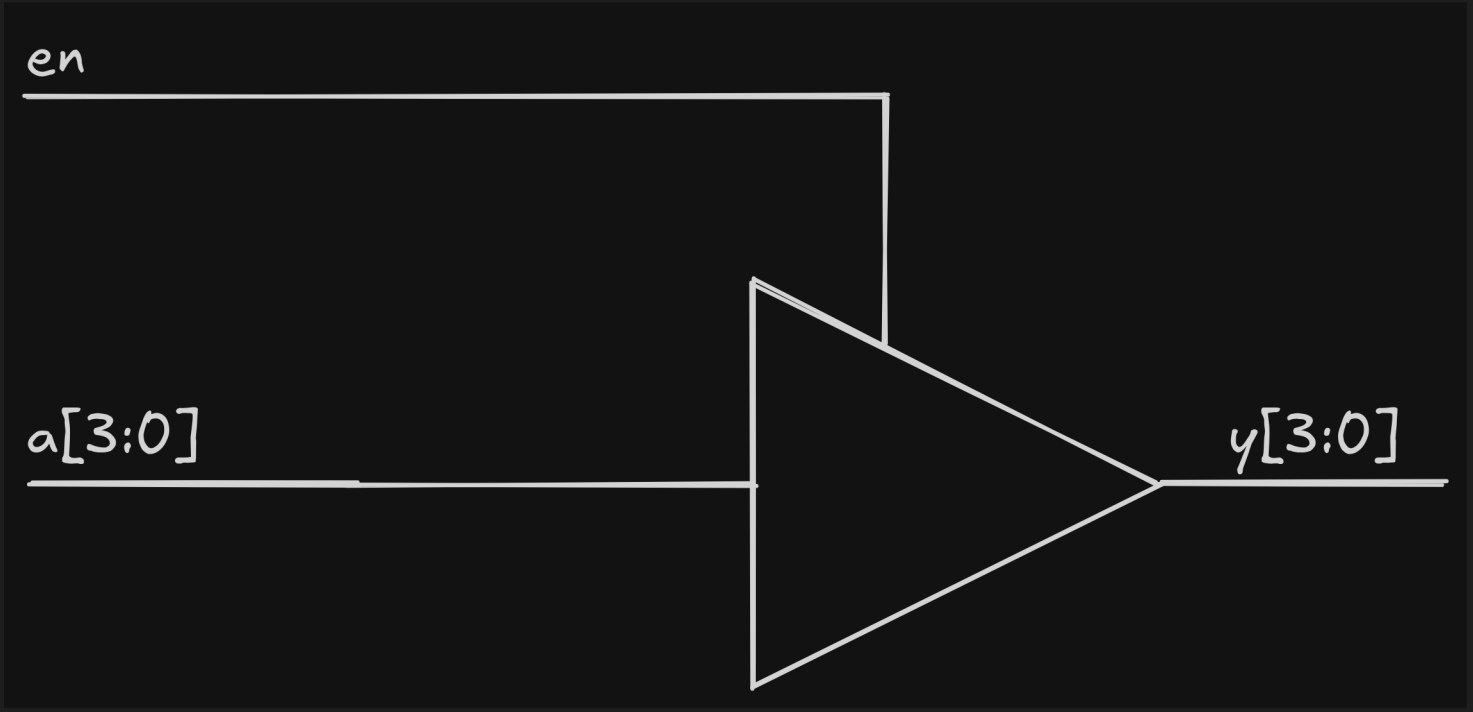


Tristate buffer

```
module tristate(input logic [3:0]a,
                            input logic en, // enable
                            output tri [3:0]y);
        assign y = en ? a : 4'bz;
endmodule
```

Il tristate buffer ha 3 possibili stati: `HIGH` (1), `LOW` (0) e `floating` (z).

- Quando `en` è vero, il tristate buffer trasferisce il valore di input in output
- Quando `en` è falso, il tristate buffer restituisce il valore `z` in output.



Logica sequenziale

SystemVerilog usa idiomi per descrivere latch, Flip Flip ed automi a stati finiti.

La maggior parte dei sistemi commerciali moderni usa Flip Flop di tipo D sensibili al fronte positivo di clock. In verilog, la parola chiave `always` rappresenta segnali che mantengono il loro vecchio valore finché un evento specificato nella loro sensitivity list non avviene, il che li fa cambiare eseguendo delle istruzioni specificate dall'utente. Dunque con una sensitivity list adatta, possiamo usare `always` per implementare circuiti sequenziali.

Inoltre, poiché definiamo logica sequenziale dentro i blocchi `always`, è possibile utilizzare il costrutto `if-else`.

Flip Flop

```
module FFD(input logic clock,
                   input logic [3:0] d,
                   output logic [3:0] q);
        always_ff@(posedge clock)
                q<=d; // Cioè "q è attraversato da d"
endmodule
```

🖊 Variazioni di `always`

`always_comb` è una versione di `always` usata per modellare circuiti combinatori e produce warning se viene usato per rappresentare logica non combinatoria.

`always_ff` è una versione di `always` usata appositamente per modellare la logica dei Flip Flop, e produce warning se viene usato per rappresentare una logica diversa da quella tipica dei Flip Flop.

`always_latch` è una versione di `always` usata appositamente per modellare la logica dei latch, e produce warning se viene usato per rappresentare una logica diversa da quella tipica dei latch.

Vedremo l'utilizzo di tutti e tre.

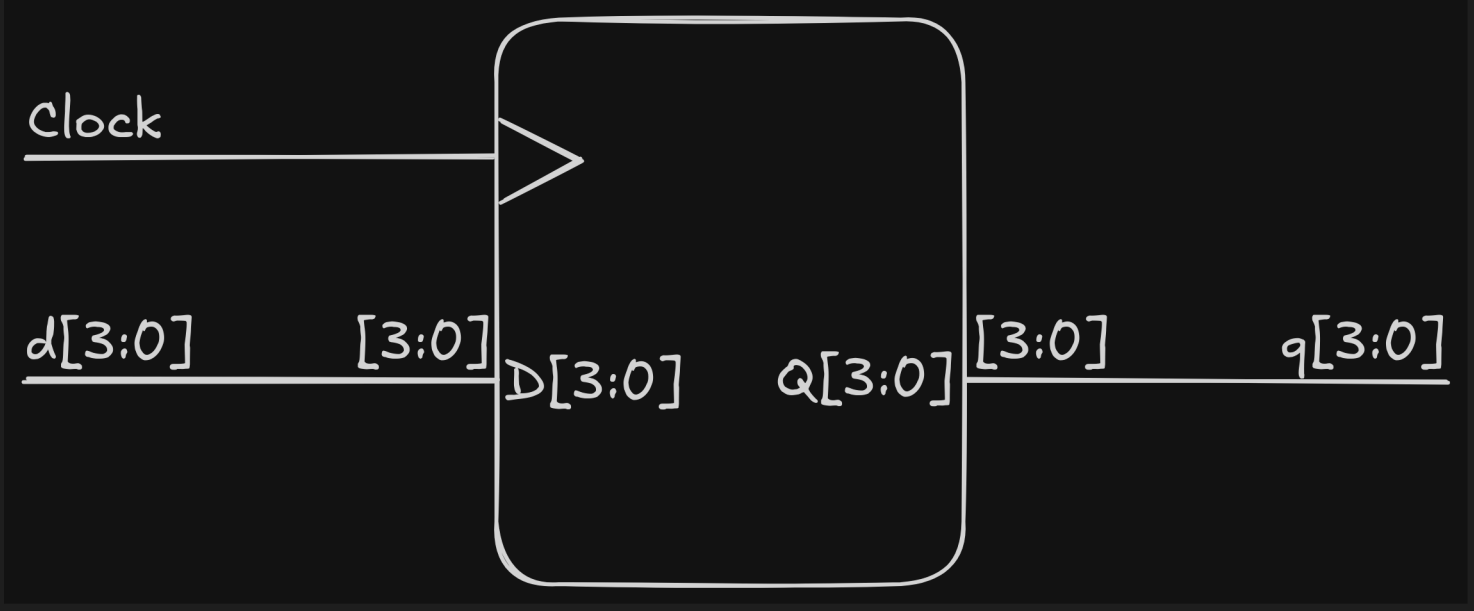
🖊 Assegnamenti bloccanti e non bloccanti

`assign` e `=` rappresentano assegnamenti bloccanti, ovvero i valori dei segnali sono assegnati uno alla volta nell'ordine in cui compaiono nel codice, come ci aspetta nei normali linguaggi di programmazione.

`<=` invece rappresenta un assegnamento non bloccante, ovvero i valori dei segnali sono assegnati tutti insieme contemporaneamente.

Specifichiamo la sensitivity list inserendo gli eventi da osservare tra parentesi, dopo il simbolo @. In questo caso, stiamo dicendo di eseguire il corpo dell' `always_ff` ogni volta che il clock raggiunge il fronte positivo.

Il simbolo `<=` rappresenta un' assegnamento non bloccante, ed è usato al posto di `assign` dentro gli `always` .

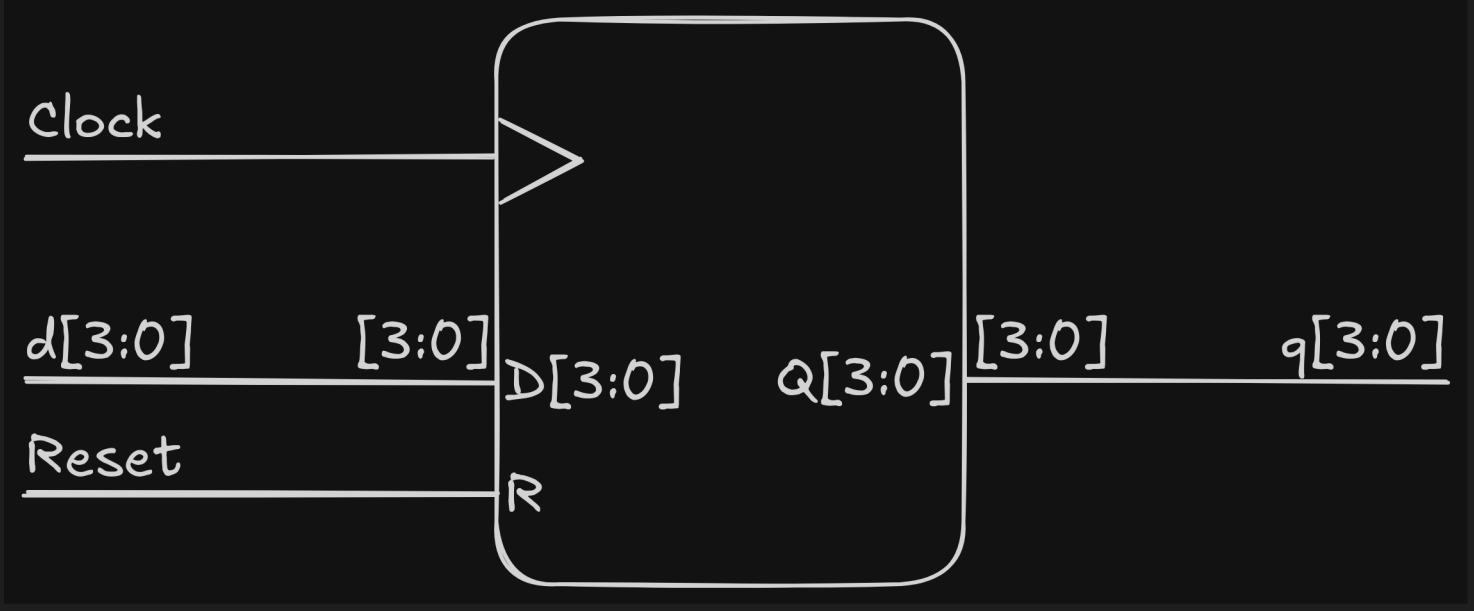


All' inizio della simulazione o quando un circuito viene alimentato per la prima volta, l'output di un Flip Flop o registro è sconosciuto (che come abbiamo detto in precedenza, è rappresentato da `x`). È dunque buona pratica usare registri resettabili per far partire il sistema in uno stato ben definito e conosciuto.
I reset possono essere

- asincroni, cioè avvengono immediatamente
- sincroni, cioè avvengono al prossimo fronte di clock positivo

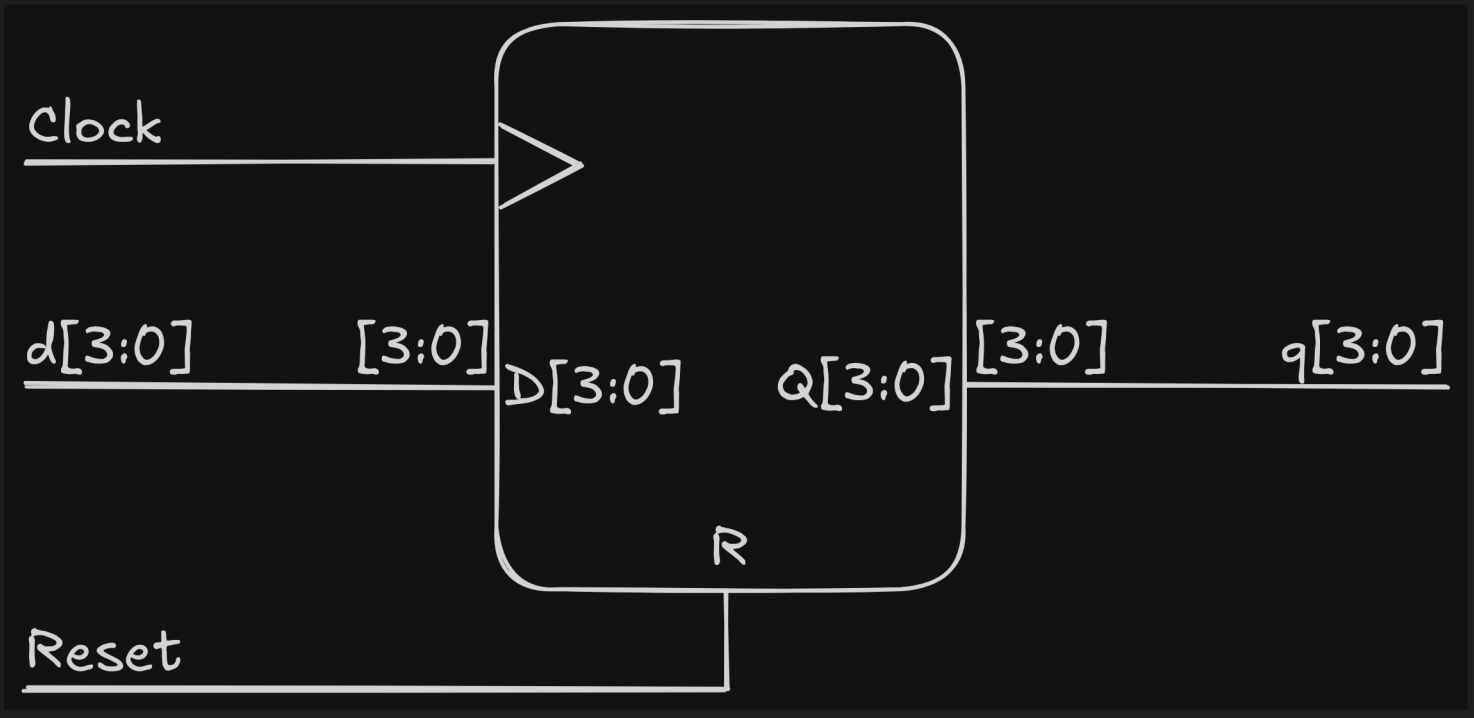
```
// Flip Flop di tipo D con reset sincrono
module FFD_ResetSync(input logic clock,
                                        input logic reset,
                                        input logic [3:0] d,
                                        output logic [3:0] q);
        always_ff @(posedge clock)
                if (reset) q <= 4'b0;
                else       q <= d;
```

Qui il reset è sincrono perché la sensitivity list aspetta il fronte di clock positivo per eseguire il corpo dell' `always_ff`, anche se il segnale di reset è stato impostato prima.




```
// Flip Flop di tipo D con reset asincrono
module FFD_ResetSync(input logic clock,
                                        input logic reset,
                                        input logic [3:0] d,
                                        output logic [3:0] q);
        always_ff @(posedge clock, posedge reset)
                if (reset) q <= 4'b0;
                else       q <= d;
```

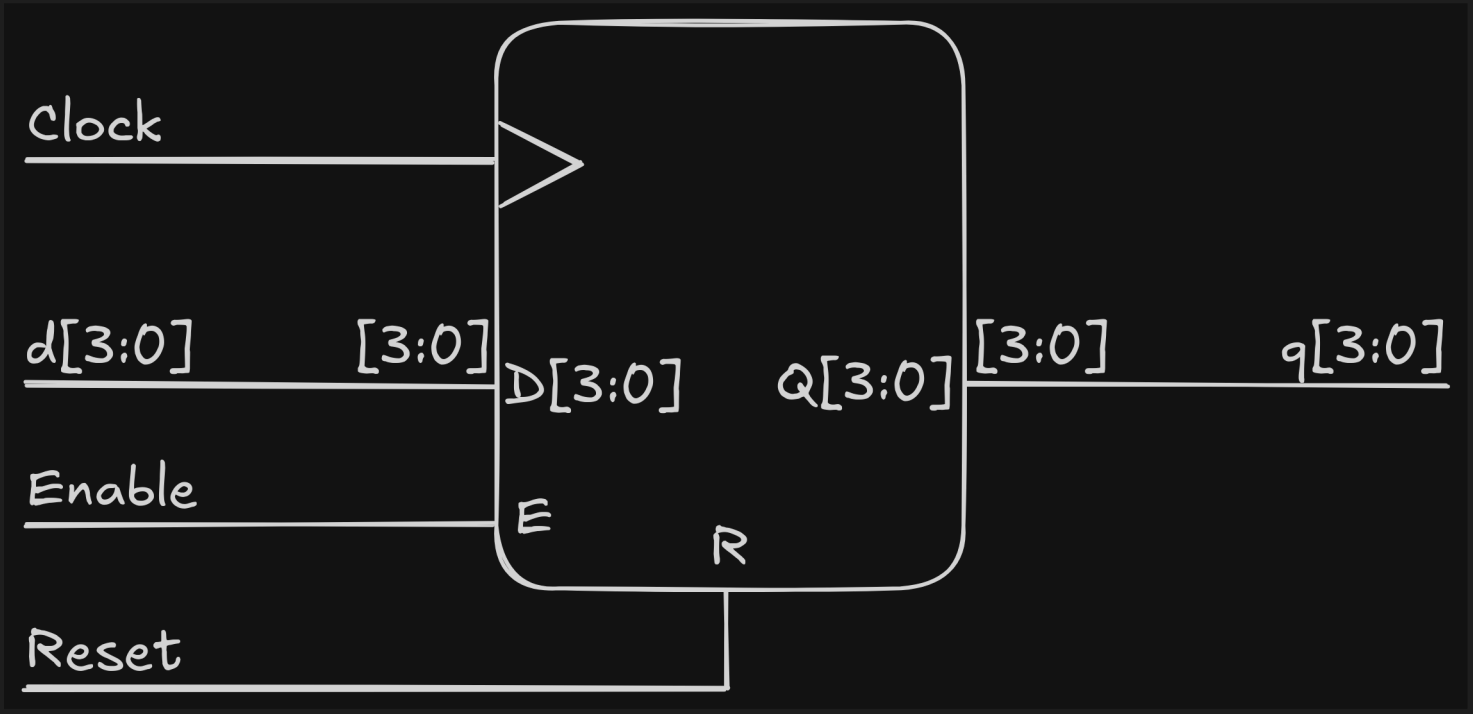
Il reset è asincrono perché la sensitivity list osserva anche il fronte positivo del reset per eseguire il corpo dell' `always_ff` , che dunque avviene appena reset è portato ad `1` .



Si potrebbe anche implementare un segnale di enable che abilita i cambiamenti del Flip Flop, semplicemente eseguendo l'assegnazione del valore di input solo quando tale segnale è attivo.

```
// Flip Flop di tipo D con reset asincrono e linea di enable
module FFD_ResetSync(input logic clock,
                                        input logic reset,
                                        input logic en,
                                        input logic [3:0] d,
                                        output logic [3:0] q);
        always_ff @(posedge clock, posedge reset)
                if (reset)   q <= 4'b0;
                else if (en) q <= d;
```

In questo modo il reset è istantaneo e l'ingresso del Flip Flop viene considerato solamente se anche il segnale di enable è attivo, altrimenti non viene eseguito nulla e viene mantenuto il vecchio valore.



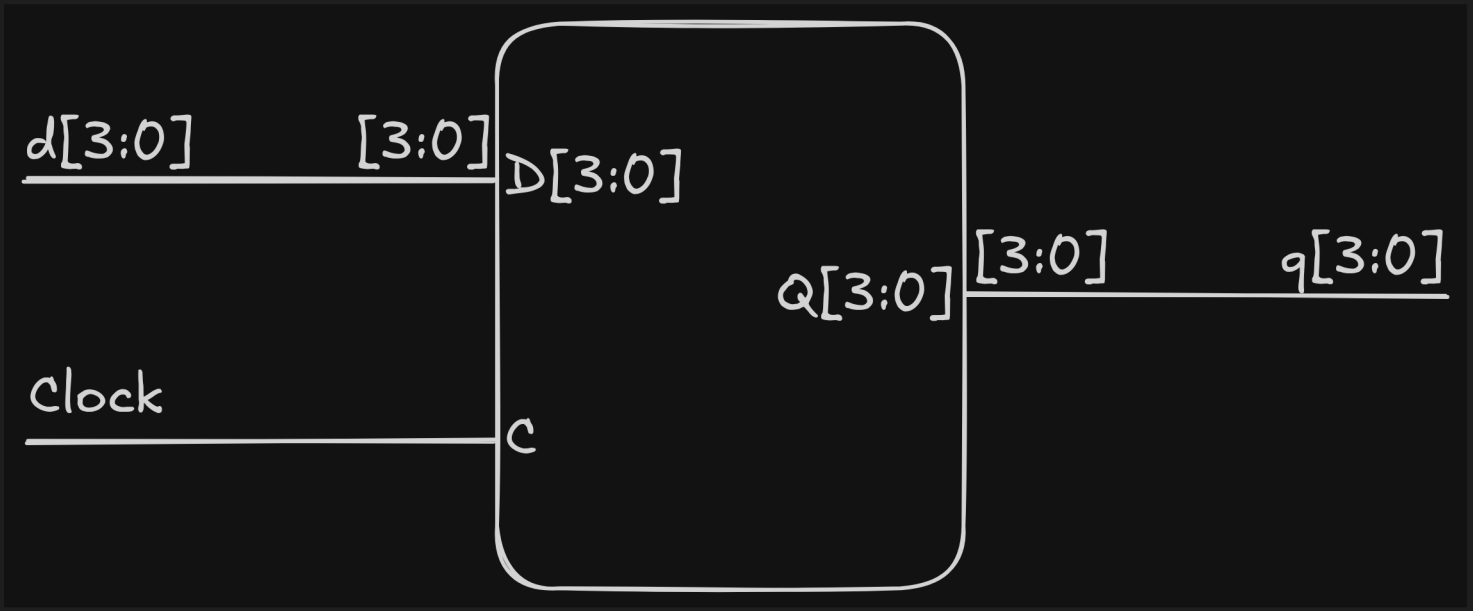
Latch

Come per i Flip Flop, possiamo usare `always` per modellare un latch.

```
// Latch di tipo D
module latchD(input logic clock, // come segnale di controllo
                        input logic [3:0] d,
                        output logic [3:0] q);
        always_latch
                if (clock) q <= d
```

`always_latch` da solo corrisponde all'uso di una sensitivity list con tutti i segnali di ingresso del modulo `always_latch`, quindi nel nostro caso corrisponde a scrivere `always @(clock, d)`, perché sono dichiarati come `input`.

Dunque, per come abbiamo definito il modulo, se il clock o `d` sono `HIGH` allora viene eseguito il corpo dell' `always_latch` , il quale indica che quando il clock è `HIGH` , `d` passa attraverso `q` , altrimenti `q` mantiene il suo vecchio valore.



Always nella logica combinatoria

`always` viene usato per descrivere circuiti sequenziali, perché ricordano lo stato precedente se non ne viene esplicitamente assegnato un'altro.

Tuttavia, è possibile descrivere anche logica combinatoria, ma:

- La sensitivity list *DEVE* rispondere ai cambiamenti di tutti gli input
- Tutti gli output *DEVONO* avere valori di default o comunque avere un valore indipendentemente dalla combinazione di ingresso, altrimenti verrebbe generato un latch che mantiene il valore corrente.

Per fare ciò, si usa `always_comb` .

```
module inverter(input logic [3:0] a,
                            output logic [3:0] y);
        always_comb
                y = ~a;
endmodule
```

`always_comb` non ha bisogno di una sensitivity list esplicita, perché il suo corpo viene eseguito ogni volta che i segnali sulla destra di `=` o `<=` vengono modificati.
Quindi, in questo caso si comporta come `always@(a)` , ma è più comodo da usare perché non mi devo preoccupare di aggiungere manualmente i segnali alla sensitivity list, o eventualmente rinominare i segnali se decido di cambiargli nome.

If-else, case e casez

Il vantaggio nell'uso di `always_comb` è la possibilità di usare `if-else`, `case` e `casez` per modellare logica combinatoria più complessa.

Case

`case` funziona come il costrutto "*switch*" nei linguaggi di programmazione normale, ovvero in base al valore di una variabile viene eseguita una certa operazione.

Tuttavia, per essere interpretato come un circuito combinatorio devono essere specificate tutte le combinazioni di input, oppure deve essere specificato un caso di default. In assenza di questi requisiti, il circuito sarà invece interpretato come sequenziale, e nei casi non specificati verrà mantenuto il vecchio valore.

```
module sevenseg(input logic [3:0] data,
                            output logic [6:0] segments);
        always_comb
                case(data)
                        4'b0000: segments = 7'b111_1110;
                        4'b0001: segments = 7'b011_0000;
                        4'b0010: segments = 7'b110_1101;
                        4'b0011: segments = 7'b111_1001;
                        4'b0100: segments = 7'b011_0011;
                        4'b0101: segments = 7'b101_1011;
                        4'b0110: segments = 7'b101_1111;
                        4'b0111: segments = 7'b111_0000;
                        4'b1000: segments = 7'b111_1111;
                        4'b1001: segments = 7'b111_0011;
                        // NECESSARIO, non ho specificato tutte le
                        // combinazioni di input
                        default: segments = 7'b000_0000;
                endcase
endmodule
```

If-Else

Si possono usare anche i costrutti `if` eventualmente seguiti da `else` ma, anche in questo caso, devono essere specificate tutte le combinazioni di input, altrimenti verrà prodotto un circuito sequenziale. Nel seguente codice viene definito un circuito con priorità, cioè si valuta l'espressione con il bit più singnificativo impostato a `True`.

```
module priorityckt(input logic [3:0] a
                                    output logic [3:0] y);
        always_comb
                // La priorità deriva dal fatto che valutiamo i bit a
                // partire dal più significativo
```

```
                if (a[3]) y = 4'b1000;
                else if (a[2]) y = 4'b0100;
                else if (a[1]) y = 4'b0010;
                else if (a[0]) y = 4'b0001;
                else y = 4'b0000;
endmodule
```

Casez

Il costrutto `casez` si comporta come `case`, ma nella definizione delle combinazioni è possibile usare il carattere `?` per definire un don't care. Per essere interpretato come circuito combinatorio, valgono le stesse regole del `case`.

Reimplementiamo il precedente circuito con priorità usando il `casez`:

```
module priority_casez(input logic [3:0] a,
                                        output logic [3:0] y);
        always_comb
                casez(a)
                        4'b1???: y = 4'b1000;
                        4'b01??: y = 4'b0100;
                        4'b001?: y = 4'b0010;
                        4'b0001: y = 4'b0001;
                        default: y = 4'b0000;
                endcase
endmodule
```

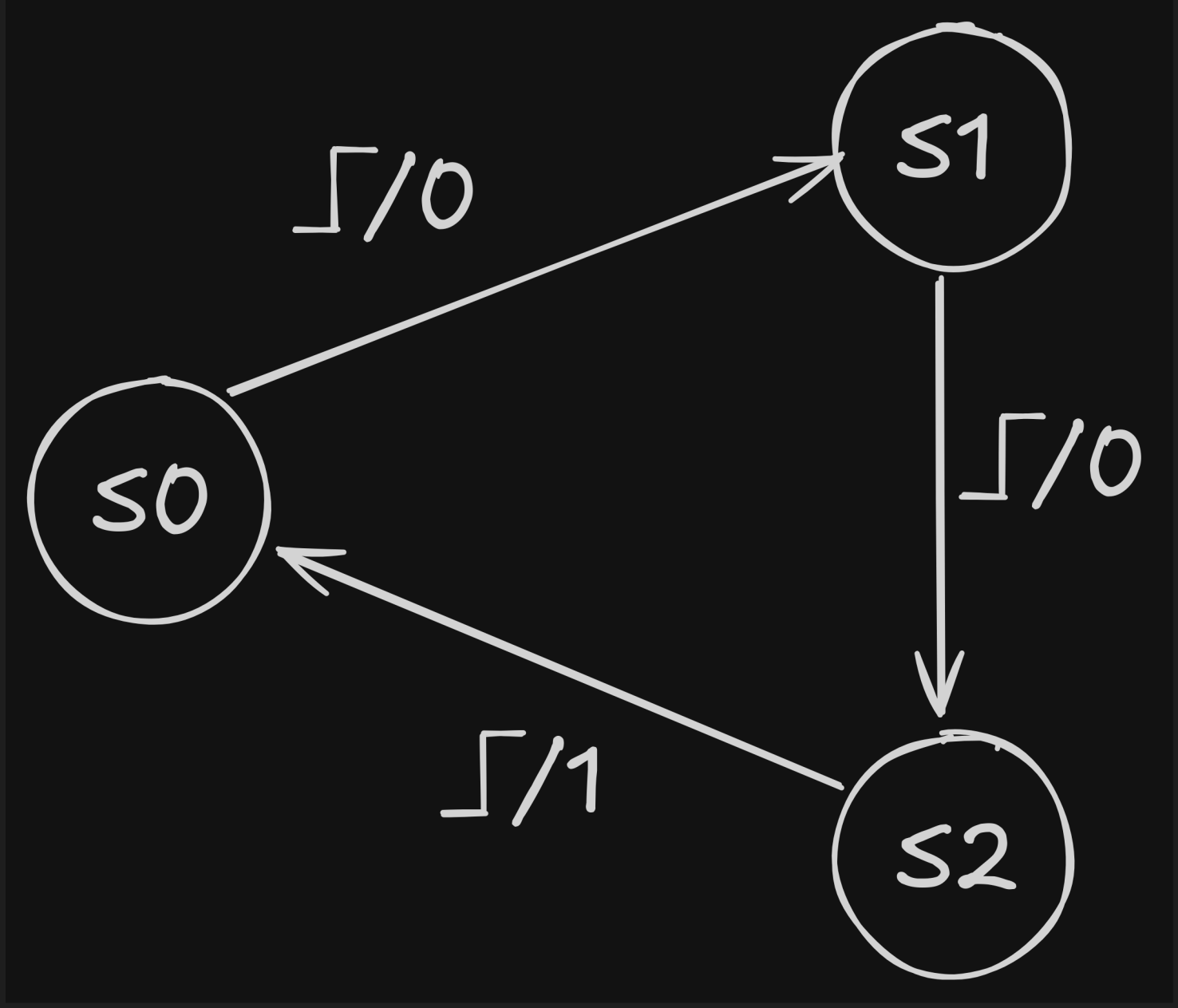
Automi a stati finiti

Per definire un automa a stati finiti in un HDL, bisogna modellare le 3 parti che lo formano:

- Un Flip Flop registratore di stato, che tiene traccia dello stato attuale dell'automa, detto *state register*
- Un circuito combinatorio che produce lo stato futuro dell' automa in base all'input ed allo stato attuale, detto *next state logic*
- Un circuito combinatorio che produce l'output, detto *output logic*

Definiamo dunque un'automa a stati finiti di un circuito che usa un contatore per contare i cicli di clock, e restituisce `1` sui i cicli di clock divisibili per 3.

Il passaggio di stato avviene sul fronte d'onda positivo, e restituisce un bit di output che è `1` quando torniamo allo stato $S0$, cioè ogni 3 colpi di clock.




```
module divideBy3FSM(input logic clock,
                                input logic reset,
                                output logic q);
        // Creiamo un tipo che definisce gli stati
        typedef enum logic [1:0] {S0, S1, S2} statetype;
        statetype [1:0] state, nextState;

        // state register
        always_ff @(posedge clock, posedge reset)
        if (reset) state <= S0;
        else state <= nextState;

        // next state logic
        always_comb
                case(state)
                        S0: nextState = S1;
                        S1: nextState = S2;
                        S2: nextState = S0;
                        default: nextState = S0;
```

```
            endcase

        // output logic
        assign q = (state == S0);
endmodule
```

- Con `typedef enum logic [1:0] {S0, S1, S2} statetype;` stiamo:
  - Definendo un nuovo tipo (`typedef`)...
  - ...che è una enumerazione (`enum`), cioè un insieme di valori costanti a cui ci possiamo riferire con un nome...
  - ... il cui valore è rappresentato da due bit di tipo logic (`logic [1:0]`)...
  - ...dove i nomi usati per riferirci a questi valori costanti (cioè la loro codifica) sono $S0$, $S1$ ed $S2$ (`{S0, S1, S2}`)...
  - ... ed infine per riferirmi a questo tipo uso il nome *statetype*.
- Il valore dell'enumerazione di default è incrementale, quindi i valori assegnati ad i nomi saranno $S0$= `00` , cioè 0, $S1$= `01` , cioè 1, e $S2$= `10` , cioè 2.
- Lo *state register* usa il segnale asincrono di clear per riportare in qualsiasi momento l'automa nello stato $S0$, altrimenti ad ogni colpo di clock l'automa passerebbe allo stato successivo. E'utile per impostare lo stato iniziale del circuito.
- Il *next state logic* controlla il valore dello stato (che ricordiamo essere uno degli enum $S0$= `00` , $S1$= `01` o $S2$= `10`) ed imposta lo stato futuro di conseguenza. L'aver usato l'enum ci ha permesso di usare il nome dello stato nel case, invece del valore binario che rappresenta lo stato, rendendo il codice più leggibile.
- Infine, l'*output logic* assegna all'output il risultato del confronto tra lo stato corrente ed $S0$: se il confronto risulta vero (quindi lo stato attuale è $S0$) l'output è `1` , altrimenti è `0` .

Testbenches

I testbench sono moduli usati per testare il corretto funzionamento di un altro modulo, che prende il nome di Device Under Test (DUT).

I testbench contengono istruzioni per applicare gli input al DUT e per verificare che gli output prodotti siano corretti. Gli input ed il rispettivo output desiderati sono chiamati test vectors (o stimoli).

I testbench possono essere:

- Semplici: Viene applicato lo stimolo e generato l'output, ma non ci sono controlli espliciti che verifichino l'equivalenza tra l'output generato e quello atteso
- Self-checking: Viene applicato lo stimolo, ed i valori di output sono verificati esplicitamente tramite l'uso di `assert` o altri costrutti di verifica. Il valore atteso è recuperato da una funzione.

- Self-checking con testvectors: Viene applicato lo stimolo, ed i valori di output sono verificati esplicitamente tramite l'uso di `assert` o altri costrutti di verifica. Il valore atteso è dichiarato come parte del test vector, dunque posso dichiarare una procedura di controllo ed applicarla sequenzialmente ad ogni test vector.

Definire un testbench

Testbench semplice

Consideriamo il seguente modulo:

```
module sillyfunction(input logic a, b, c,
                                        output logic y);
        assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
endmodule
```

Procediamo a creare un testbench semplice per verificarne il corretto funzionamento:

```
module testbench1();
        logic a, b, c;
        logic y;

        // Istanzia il device under test
        sillyfunction dut(a, b, c, y);
        // applica gli input uno alla volta
        initial begin
                $monitor("%4d a=%b b=%b c=%b y=%b", $time,
                                a, b, c, y);
                a = 0; b = 0; c = 0;
                #10;
                b = 1;
                #10;
                $finish;
        end
endmodule
```

- `initial begin` ed `end` specificano dove inizia un blocco di codice che deve essere eseguito una volta sola, all'inizio della simulazione. Dentro questo blocco è ammesso l'uso del costrutto `if-else`.
- `$monitor` è una funzione built-in che stampa sulla console (stdout) il valore delle variabili ogni volta che cambiano valore. `"%4d a=%b b=%b c=%b y=%b"` indica come formattare l'output, e dopo vengono inseriti i valori che vanno messi al posto dei codici con %, un po'come il printf in C.

- `$time` è una variabile interna di verilog che tiene il tempo in unità di simulazione (per esempio, cicli).
- `#10` Introduce un delay di 10 unità di simulazione prima di proseguire con il codice.

Questi delay permettono di applicare gli input nell'ordine desiderato.

- `$finish` fa terminare la simulazione.

Dunque, compilando con iverilog ed avviando la simulazione su questo testbench andiamo ad "osservare" le variabili `a`, `b`, `c` ed `y`, poi assegniamo a tutte il valore 0 (e questo cambiamento verrà notato da `$monitor`, che stamperà i valori), aspettiamo un delay di 10 unità, poi assegniamo il valore 1 a `b` (e questo cambiamento verrà notato da `$monitor`, che stamperà i valori nuovamente), aspettiamo un altro delay di 10 unità e poi la simulazione finisce.

Il problema di questo metodo di testbench è che l'utente deve manualmente controllare i risultati e verificare la correttezza degli output, dunque è un processo tedioso ed è possibile fare errori di distrazione.

Testbench self-checking

Ridefiniamo `sillyfunction` per essere un po' più semplice, in modo da rendere meno verboso il codice dei test self-checking e self-checking con test vectors.

```
module sillyfunction(input logic a, b, c,
                                        output logic y);
        assign y = ~b & ~c | a & ~b;
endmodule
```

I testbench self-checking consistono nel controllare in modo esplicito tutti i valori di output generati mano a mano che cambiamo i valori di input.

```
module testbench2();
        logic a, b, c;
        logic y;
        // instanzia il device under test
        sillyfunction dut(a, b, c, y);
        initial begin
                /* Applico gli input e controllo i risultati uno
                alla volta */
                a = 0, b = 0, c = 0; #10;
                if (y != 1) $display("000 failed.");
                c = 1; #10;
                if (y != 0) $display("001 failed.");
                b = 1, c = 0; #10;
                if (y != 0) $display("010 failed.");
```

```
                c = 1; #10;
                if (y != 0) $display("011 failed.");
                a = 1, b = 0, c = 0; #10;
                if (y != 1) $display("100 failed.");
                c = 1; #10;
                if (y != 1) $display("101 failed.");
                b = 1, c = 0; #10;
                if (y != 0) $display("110 failed.");
                c = 1; #10;
                if (y != 0) $display("111 failed.");
        end
endmodule
```

`$display()` è una funzione built-in che stampa sulla console (stdout) la stringa specificata.

Il problema di questo approccio è che va scritto codice di controllo per ogni test vector, che diventa scomodo sopratutto con moduli più grandi e complessi.

Testbench self-checking con testvectors

Il metodo più modulare per eseguire testbench è quello dei testbench self-checking con test vectors. Consiste nel definire il segnale di clock e delle variabili in cui mettere gli input ed il relativo output atteso, e poi:

- Metto tutti i test vectors in un file e li carico in un array
- Sul fronte di salita di clock, prendo il test vector, assegno i suoi valori alle variabili apposite create prima e genero l'output dalle combinazioni di input prese dal test vector
- Sul fronte di discesa di clock, confronto l'output generato e quello atteso, segnalando gli errori.
- Ripeto con il test vector successivo.

Inizio creando il file contenente i test vectors, con estensione `.tv`. I test vector vanno scritti nel formato `abc_y`, dove `abc` sono gli input ed `y` è l'output.

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

Adesso inizio a scrivere il modulo del testbench pezzo per pezzo. Innanzitutto definiamo le variabili, instanziamo il dut e definiamo come si comporta il clock.

```
module testbench3();
        logic clock, reset;
        // I valori letti dal file vanno qui dentro
        logic a, b, c, yExpected;
        // Qui dentro mettiamo l'output generato dal dut
        logic y;
        /* Vectornum indica il numero del test vector attuale, ed errors
indica i test vectors falliti */
        logic [31:0] vectornum, errors;
        // Qui dentro mettiamo gli effettivi test vector letti dal file
        logic [3:0] testvectors[10000:0];

        // instanzia il dut
        sillyfunction dut(a,b,c,y);

        /* Definisco il comportamento del clock, cioè cambia
        valore ogni 5 unità di tempo */
        always // No sensitivity list = è eseguito sempre
                begin
                        clock = 1; #5; clock = 0; #5;
                end
```

Adesso carichiamo i testvectors dal file "*myTests.tv*" dentro l'array. Abbiamo usato la funzione built-in `$readmemb` per leggere valori binari, ma avremmo potuto usare invece `$readmemh` per leggere valori esadecimali. Impostiamo inoltre il valore iniziale di `vectornum` ed `errors` a `0` ed eseguiamo un reset, in modo da poter iniziare da zero i test.

```
initial
                begin
                        $readmemb("myTests.tv", testvectors);
                        reset = 1; #27; reset = 0;
                        vectornum = 0; errors = 0;
                end
```

Ora assegno il valore dei test vectors sul fronte di salita del clock. Con `{a, b, c, yExpected} = testvectors[vectornum];` sto dicendo di prendere il `vectornum`-esimo elemento dell'array, ed assegnare i bit dell'elemento alle variabili `a`, `b`, `c` ed `yExpected`.

```
// Applica i test sul fronte positivo di clock
        always @(posedge clock)
```

```
                begin
                        #1; {a, b, c, yExpected} =
                        testvectors[vectornum];
                end
```

Poi sul fronte di discesa confronto l'output generato dal dut con quello atteso, e se il test fallisce stampo sulla console le informazioni del test in questione, l'output generato dal dut e l'output che mi aspettavo. Per stampare valori esadecimali invece che binari nel `$display`, usa `%h` invece di `%b`.

```
        // Controllo i risultati sul fronte di discesa
        always @(negedge clock)
                if (~reset) begin // salta durante i reset
                        if (y != yExpected) begin
                                $display("Error: inputs = %b", {a, b,
c});
                                $display(" outputs = %b (%b
expected)",y,yexpected);
                                errors = errors + 1;
                        end
```

Infine, incremento l'indice dell'array per caricare il nuovo test vector al prossimo fronte positivo, e se li ho finiti, stampo un resoconto e faccio finire il testbench. Gli operatori `===` e `!==` servono per eseguire comparazioni tra valori logici che possono essere `0`, `1`, `z` o `x`, e quando si cerca di accedere all'elemento di un array out of bounds viene restituito un valore composto solo da bit impostati a valore `x`, che posso dunque usare per identificare l'aver finito i testvector.

```
// Incremento l'indice dell'array per leggere il nuovo test vector
                        vectornum = vectornum + 1;
                        // Finiti i testvectors, stampa un resoconto
                        if (testvectors[vectornum]===4'bx) begin
                                $display("%d tests completed with %d
errors",
                                                        vectornum, errors);
                                $finish;
                        end
                end
endmodule
```

Questo è il codice per intero:

```
module testbench3();
        logic clock, reset;
```

```
        // I valori letti dal file vanno qui dentro
        logic a, b, c, yExpected;
        // Qui dentro mettiamo l'output generato dal dut
        logic y;
        /* vectornum indica il numero del test vector attuale, ed errors
indica i test vectors falliti */
        logic [31:0] vectornum, errors;
        // Qui dentro mettiamo gli effettivi test vector letti dal file
        logic [3:0] testvectors[10000:0];

        // instanzia il dut
        sillyfunction dut(a,b,c,y);

        /* Definisco il comportamento del clock, cioè cambia
        valore ogni 5 unità di tempo */
        always // No sensitivity list = è eseguito sempre
                begin
                        clock = 1; #5; clock = 0; #5;
                end
        /* All'inizio del test, carica i test vector dal file,
        fai partire da 0 vectornum ed errors, ed esegui un
        reset*/
        initial
                begin
                        $readmemb("myTests.tv", testvectors);
                        vectornum = 0; errors = 0;
                        reset = 1; #27; reset = 0;
                end

        // Applica i test sul fronte positivo di clock
        always @(posedge clock)
                begin
                        #1; {a, b, c, yExpected} =
                        testvectors[vectornum];
                end

        // Controllo i risultati sul fronte di discesa
        always @(negedge clock)
                if (~reset) begin // salta durante i reset
                        if (y != yExpected) begin
                                $display("Error: inputs = %b", {a, b,
c});
                                $display(" outputs = %b (%b
expected)",y,yExpected);
                                errors = errors + 1;
                        end
```

```
                        // Incremento l'indice dell'array per leggere il
nuovo test vector
                        vectornum = vectornum + 1;
                        // Finiti i testvectors, stampa un resoconto
                        if (testvectors[vectornum]===4'bx) begin
                                $display("%d tests completed with %d
errors",
                                                        vectornum, errors);
                                $finish;
                        end
                end
endmodule
```

Monitoraggio con forme d'onda

Per visualizzare il comportamento del circuito come forma d'onda attraverso GTKwave, diciamo ad iverilog di generare anche il file Value Change Dump (*VCD*), aggiungendo sotto ad `initial begin` le istruzioni per creare il file e modificando i valori per controllare l'andamento del circuito con le varie combinazioni di input.

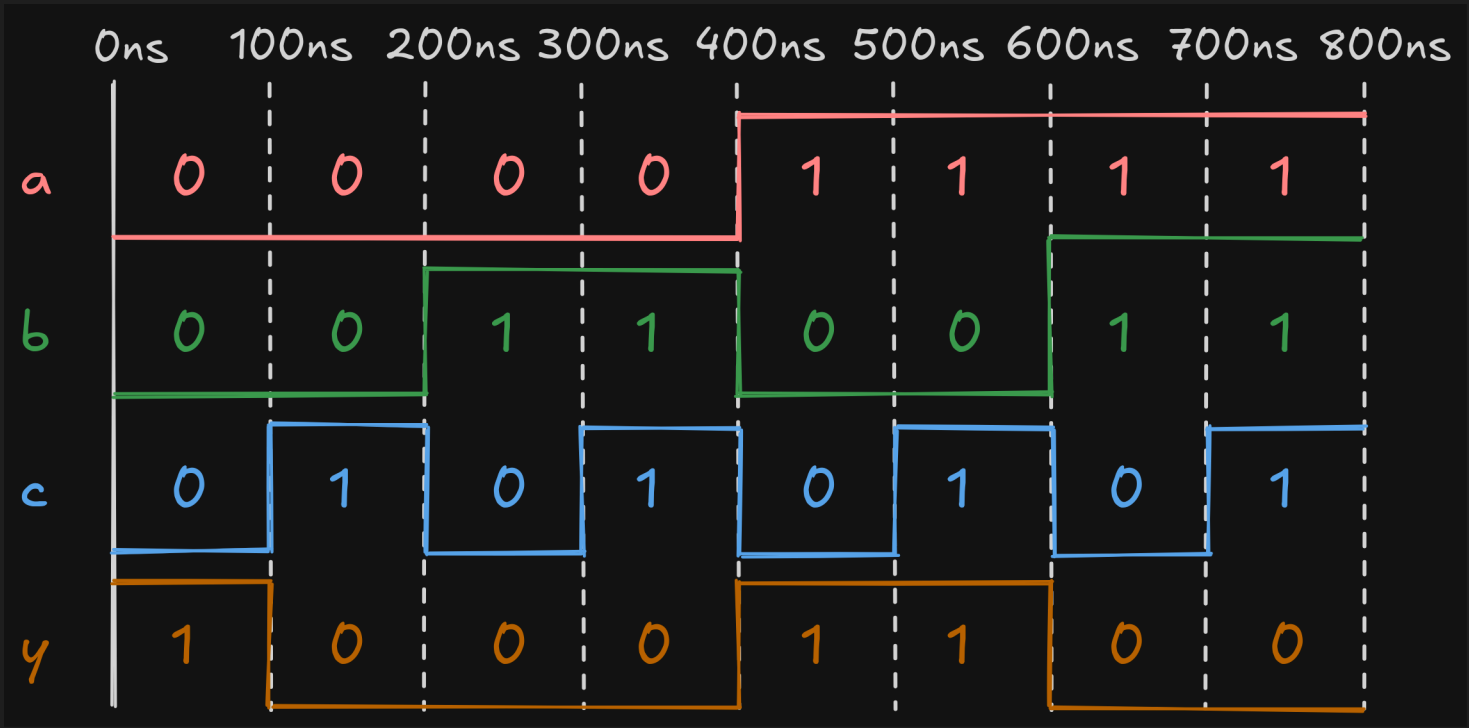
Proviamo ad applicare tali modifiche al testbench semplice che abbiamo definito prima:

```
module sillyfunction(input logic a, b, c,
                                        output logic y);
        assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
endmodule
```

```
module testbench1();
        logic a, b, c;
        logic y;

        // Istanzia il device under test
        sillyfunction dut(a, b, c, y);
        // applica gli input uno alla volta
        initial begin
    $dumpfile("dump.vcd"); // memorizza la variazione dei
                                                // valori come file
vcd
    $dumpvars; // Considera tutte le variabili nel file
        $monitor("%4d a=%b b=%b c=%b y=%b", $time,
                         a, b, c, y);
        a = 0; b = 0; c = 0;
        #10;
    // 001
        c = 1;
```

```
        #10;
    // 010
    b = 1; c = 0;
    #10
    // 011
    c = 1;
    #10
    // 100
    a = 1; b = 0; c = 0;
    #10
    // 101
    c = 1;
    #10
    // 110
    b = 1; c = 0;
    #10
    // 111
    c = 1;
    #10
        $finish;
        end
endmodule
```

Le forme d'onda visualizzabili in GTKwave sono simili alle seguenti:



In questo caso i delay `#10` sono considerati da 100ns, che è il tempo che passa tra le variazioni dei valori `a b c`.

Lanciare un testbench (Linux/WSL)

- Scarica Icaus Verilog

- Per visualizzare le forme d'onda, scarica GTKwave
- Compila il testbench:
`$ iverilog -g2005-sv -s nomeModuloTestbench NomeFile.sv` (genera nella stessa cartella un file `a.out`)
- Esegui il testbench:
`$ ./a.out`
- Visualizza la forma d'onda:
`$ gtkwave nomeFile.vcd` , dal menu a sinistra scegli il DUT da analizzare e fai doppio click sulle variabili che vuoi visualizzare per aggiungerle al grafico.