IL LINGUAGGIO HDL VERILOG PER LA SINTESI E LA SIMULAZIONE DI CIRCUITI DIGITALI

1. Introduzione

I linguaggi per la descrizione dell'hardware (HDL - Hardware Description Language) sono una realtà molto importante per la progettazione digitale. Essi inquadrano la descrizione di un sistema hardware in un insieme di regole formali che semplificano notevolmente la realizzazione dei circuiti.
Gli HDL nascono negli anni '80 ed avevano principalmente lo scopo di documentare i sistemi hardware. L'idea è quella di poter descrivere in maniera semplice le azioni svolte da un determinato sistema. Servono quindi una chiara identificazione degli ingressi e delle uscite del sistema hardware ed un linguaggio che descriva le operazioni effettuate sugli ingressi, per ottenere le uscite.
La principale differenza con i linguaggi pensati per il software è la presenza di costrutti per la descrizione di azioni che avvengono in parallelo, nonché la definizione di tipi di variabili atti a definire i livelli logici dei segnali interni ad un circuito.
La successiva estensione fu quella di utilizzare gli HDL per simulare il comportamento dei sistemi hardware. In questo modo è possibile verificare che il sistema hardware, con riferimento alle specifiche, abbia un comportamento corretto. Le prestazioni e l'effettiva realizzazione circuitale non sono un aspetto che viene affrontato in questa fase. Lo scopo della simulazione è capire se l'algoritmo utilizzato per realizzare il sistema è corretto o meno.
In questa fase gli HDL, una volta utilizzati per descrivere e simulare il sistema elettronico, fornivano al progettista una corretta impalcatura, composta da numerosi blocchi, relativamente semplici e dal comportamento ben caratterizzato, che andava comunque completata. La realizzazione finale veniva effettuata utilizzando componenti standard quali porte logiche, contatori, registri etc., che venivano connessi per realizzare i singoli blocchi da connettere per completare il progetto.
Con l'aumento della complessità dei sistemi divenne importante poter disporre di metodi automatici che passassero dalla descrizione HDL ad una descrizione circuitale. Questa operazione di traduzione è denominata fase di sintesi e permette di ottenere un circuito a partire dalla descrizione HDL. Per quanto la fase di sintesi non sia assolutamente banale e negli anni siano stati sviluppati algoritmi sempre più complessi e precisi che ne migliorano i risultati, è importante comprendere che la disponibilità di linguaggi HDL opportuni, è il punto fondamentale per la realizzazione di un'efficace fase di sintesi.
A tal proposito gli HDL devono essere utilizzati adeguatamente adoperando costrutti orientati alla sintesi (detti sintetizzabili), ed i circuiti descritti devono essere sincroni.

I linguaggi HDL attualmente più utilizzati sono il **VHDL** (VHSIC, Very High Speed Integrated Circuits HDL) e il **Verilog**. I concetti alla base dei due linguaggi sono simili. Da un punto di vista sintattico il VHDL offre un maggiore controllo sulla correttezza del codice (controllo stretto sui tipi di segnali) se confrontato con il Verilog. Dal punto di vista del codice, il Verilog è più sintetico del VHDL e spesso gli viene preferito proprio per questo motivo.

1.1 Storia del Verilog

Il Verilog HDL, Verilog da ora in avanti, fu sviluppato dall'azienda Gateway Design Automation, in seguito acquisita da Cadence Design Systems nel 1983. Inizialmente il linguaggio era orientato alla simulazione. Nel 1990 il linguaggio fu reso disponibile liberamente per incrementarne la popolarità e fu istituita l'organizzazione Open Verilog International (OVI) per promuovere il linguaggio. Nel 1992 si decise di definire uno standard IEEE per il linguaggio Verilog. La stesura fu completata nel 1995 e lo standard fu denominato IEEE Std 1364-1995. Ulteriori revisioni dello standard sono state effettuate nel 2001 e nel 2005. Una descrizione completa del linguaggio è disponibile nel Verilog Hardware Description Language Reference Manual. Gli esempi di codice presentati in questo capitolo sono congruenti con la sintassi dello standard Verilog del 2001.

2. Flusso di progetto con HDL

Il flusso di progetto di un sistema elettronico digitale implementato su FPGA o CPLD, è simile a quanto mostrato in Figura 2.1. L'utilizzo di un linguaggio HDL come il Verilog permette di avere un unico linguaggio per completare molte delle fasi del flusso di progetto. In Figura 2.1 le fasi del flusso di progetto indicate utilizzano simbologie differenti. Le fasi indicate con dei rettangoli indicano azioni nelle quali è richiesta un'attività progettuale intensa e che richiede profonde conoscenze circuitali. Le azioni mostrate in un ovale rappresentano azioni che non dipendono direttamente dal progettista o che, anche se indirizzate dalle specifiche indicate dal progettista, vengono eseguite in maniera quasi totalmente automatica. Le linee piene collegano i punti del flusso di progetto da seguire per ottenere il circuito finale. Le linee tratteggiate indicano invece delle relazioni logiche o un flusso di informazioni.
Il flusso di progetto è diviso verticalmente in un insieme di fasi che realizzano il circuito (dalla descrizione funzionale alla programmazione del FPGA) ed in un insieme di azioni di simulazione e verifica del circuito stesso. Volutamente le due sezioni sono poste al centro dell'immagine per evidenziare che le fasi di simulazione hanno la stessa importanza di quelle di realizzazione del circuito. Infatti, solo mediante accurate simulazioni del circuito che si sta realizzando è possibile scoprire e correggere precocemente gli errori di progettazione.
Di seguito una descrizione esemplificativa del flusso di progetto mostrato.

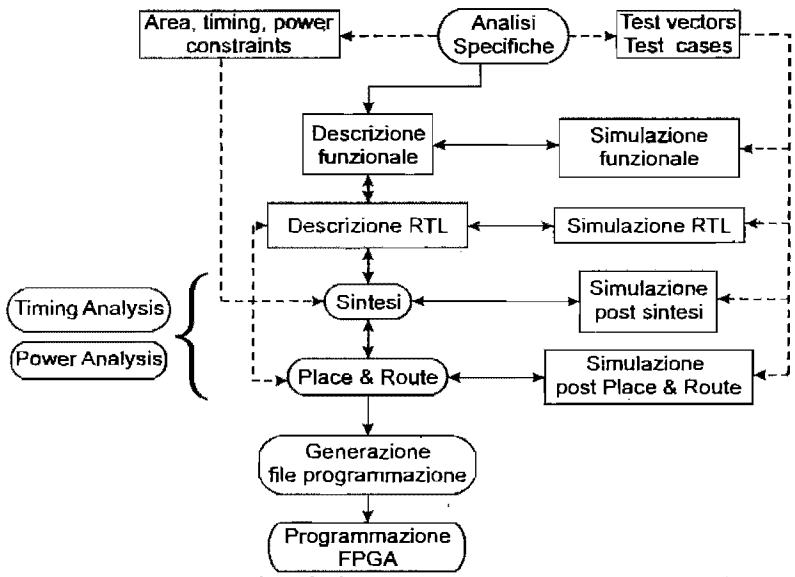


Figura 2.1: Flusso di progetto per circuiti digitali di tipo programmabile (FPGA e CPLD). I rettangoli indicano le azioni che richiedono un intervento di tipo progettuale. Gli ovali indicano fasi semi-automatiche che non richiedono interventi progettuali. Le linee piene indicano il flusso di progetto seguito dal progettista. Le linee tratteggiate relazioni logiche o flusso di informazioni. Le fasi di timing analysis e power analysis pongono dei vincoli e vengono utilizzate come verifiche dalle fasi di sintesi e di place & route.

Analisi delle specifiche

In questa fase si definiscono le caratteristiche del circuito che si vuole realizzare. Inizialmente la descrizione può essere realizzata utilizzando il linguaggio naturale o dello pesudo-codice per delineare le funzionalità del circuito. Il risultato dell'analisi delle specifiche riguarda non solo una comprensione precisa delle funzionalità del circuito, ma anche il formato degli ingressi e delle uscite. Le specifiche possono, una volta comprese ed identificate senza ambiguità, essere definite nuovamente utilizzando una descrizione con un linguaggio procedurale (non di tipo HDL).

Esempio: Si desidera realizzare un circuito che effettui la FFT (Fast Fourier Transform) di una sequenza di dati. Dall'analisi delle specifiche sapremo: su quanti bit sono rappresentati i dati di ingresso; la frequenza di arrivo dei dati di ingresso; su quanti dati deve essere eseguita la FFT; il tipo di FPGA che desideriamo utilizzare; il tipo di package da utilizzare; in che formato si desidera avere l'uscita.

La fase di analisi delle specifiche può essere completata realizzando un codice in un linguaggio procedurale (C, Matlab etc.) che effettui le operazioni richieste sui dati.

Area, timing, power constraints.

Una volta compreso cosa deve fare il circuito si dovranno determinare le prestazioni richieste al circuito.
Analizzando le specifiche e, se necessario, effettuando delle analisi con il software sviluppato, si potranno determinare le prestazioni richieste.
Si avranno ad esempio indicazioni sulla velocità e sulla potenza dissipata richieste dal circuito. Se sono state date specifiche sul tipo di FPGA da utilizzare si avrà anche un limite sul numero di elementi logici programmabili che possono essere utilizzati dal circuito.

L'analisi delle specifiche richiederà inoltre la determinazione di un insieme di casi tipici da utilizzare come test del funzionamento del circuito.

Esempio: Del circuito per la realizzazione della FFT conosciamo, dall'analisi delle specifiche, le caratteristiche dei segnali di ingresso, dei segnali di uscita e l'algoritmo che il circuito eseguirà. Analizzando questi dati avremo già un limite sull'occupazione di area (determinata dal FPGA scelto), del numero di pin di ingresso e di uscita (verifica di compatibilità con il package scelto), della potenza dissipata (verifica di compatibilità con il package scelto e con eventuali specifiche di durata della batteria) e della minima frequenza di clock che il circuito dovrà gestire (funzione della quantità di dati da elaborare nell'unità di tempo). Questi dati rappresentano i vincoli sulle prestazioni del circuito da realizzare.

Test vectors, test cases

Aver determinato le funzionalità da implementare nel circuito e le prestazioni richieste ci permette di definire, in modo non ambiguo, un insieme di casi tipici da utilizzare come test del funzionamento del circuito. Tra questi troveremo degli ingressi pensati per testare separatamente le varie funzionalità del circuito e quindi adatti ad evidenziare e correggere eventuali errori di progettazione. Troveremo inoltre dei casi tipici di funzionamento ed anche dei casi limite adatti a testare il comportamento del circuito in condizioni critiche.

Esempio: Comprese la funzionalità e le prestazioni richieste al circuito passiamo a definire un insieme di casi tipici da utilizzare per testare l'applicazione di calcolo della FFT.

I test di tipo generale, adatti ad evidenziare errori di progettazione, potrebbero essere i casi di ingresso costante (con varie ampiezze) e di ingressi sinusoidali. Infatti per questi ingressi possiamo calcolare facilmente la FFT e siamo quindi facilmente in grado di capire dove avvengono gli errori se il risultato non dovesse essere concorde con le specifiche.

I casi di test tipici possono essere dei segnali reali forniti dal committente dell'applicazione o rilevati durante campagne sperimentali. Potremo inoltre determinare dei test adatti a verificare il comportamento del circuito in condizioni estreme. Ad esempio, se abbiamo una specifica sulla potenza dissipata potremmo determinare un segnale di test che causi molte transizioni e che quindi sia in grado di valutare la massima dissipazione di potenza del circuito.

Descrizione funzionale

La prima descrizione del circuito può già essere effettuata utilizzando un HDL, come ad esempio il Verilog o il VHDL. Infatti sia il Verilog sia il VHDL, sono linguaggi per la descrizione dell'hardware che permettono la descrizione di un sistema digitale in un ampio spettro di livelli di astrazione, dal comportamentale allo strutturale. Essi includono inoltre costrutti gerarchici che permettono al progettista di gestire anche sistemi molto complessi.

In questa fase la descrizione del circuito è effettuata ad un livello molto alto, siamo ancora lontani dal circuito ma possiamo già avere una divisione del circuito in blocchi funzionali. E' fondamentale comprendere che tale descrizione potrebbe essere anche di tipo "non sintetizzabile". Vale a dire che potrebbero essere utilizzati costrutti che non si possono tradurre direttamente in un circuito.

Esempio: Il circuito per la FFT viene diviso in blocchi che si occupano di sincronizzare ed ordinare gli ingressi; di ordinare ed allineare le uscite; di effettuare i calcoli per la FFT. Alcuni (o tutti) i blocchi possono essere "non sintetizzabili". Ad esempio alcuni punti necessari al calcolo della FFT possono utilizzare dei calcoli effettuati su numeri reali. Tali calcoli sono possibili in Verilog ma non hanno una diretta corrispondenza con la realizzazione hardware. In pratica un software per la sintesi, non saprebbe come tradurre in termini circuitali un numero reale.

Simulazione funzionale.

Utilizzando i vettori di test sviluppati precedentemente realizziamo uno o più test bench. Il test bench è una descrizione HDL che applica dei segnali ad un circuito e ne analizza le uscite. Il test bench è anch'esso scritto in Verilog.

Il test bench può, se necessario, prelevare gli ingressi per il circuito da un file e memorizzare le uscite del circuito su di un file. Il test bench può anche verificare la congruenza delle uscite con un insieme di uscite attese e segnalare eventuali errori.

La simulazione funzionale ci dice se il circuito si comporta da un punto di vista comportamentale nel modo che ci attendiamo. Se il risultato della simulazione funzionale non dovesse essere coerente con le attese, si ritorna alla fase di descrizione funzionale correggendo gli errori fatti.

In questa fase non possiamo avere nessuna informazione sul rispetto delle specifiche relative alla frequenza di funzionamento, all'occupazione di logica programmabile o alla potenza dissipata.

Esempio: il test bench per il circuito FFT applica al circuito, descritto per il momento in Verilog con costrutti non sintetizzabili, i segnali di test definiti in precedenza. Poiché conosciamo le uscite desiderate dal circuito nei casi considerati, il test bench controlla automaticamente la correttezza della simulazione ed evidenzia errori di progettazione. Eventuali errori, sulla suddivisione e sull'interazione dei moduli elementari, o sulle operazioni attribuite ad ogni modulo, possono essere identificati e corretti in questa fase.

Le uscite desiderate possono: essere state calcolate manualmente; essere state fornite dal committente; essere state generate utilizzando la descrizione software del circuito che abbiamo ottenuto a valle della fase di analisi delle specifiche.

Descrizione RTL (Register Transfer Level).

Utilizzando un HDL come il Verilog, i moduli, precedentemente descritti utilizzando costrutti non sintetizzabili, o poco

efficienti da un punto di vista circuitale, vengono elaborati ottenendo dei componenti sintetizzabili.
In questa fase ogni parte del circuito deve trovare una sua composizione in termini di componenti digitali, anche complessi, la cui realizzazione pratica non sia in discussione.
Gli ingressi e le uscite dovranno essere caratterizzati in termini di numero di bit e di trasferimento dati. Lo stesso varrà per i segnali che connettono i vari moduli e che sono interni al circuito.
La descrizione di ogni modulo dovrà essere curata tenendo bene in conto il fatto che ciò che dobbiamo realizzare è un circuito. Si utilizzeranno quindi componenti digitali ben definiti come ad esempio porte logiche, multiplexer, encoder/decoder, addizionatori/sottrattori, moltiplicatori, registri, registri a scorrimento, memorie etc. per arrivare ad una descrizione circuitale dei blocchi logici che sia digitale e di tipo sincrona.
La descrizione RTL non deve solo tener conto della funzionalità del circuito. La Figura 2.1 mostra come siano da utilizzare come ingressi per questa fase anche i vincoli di prestazione ottenuti dall'analisi delle specifiche. Infatti, la descrizione RTL è il momento in cui si effettuano scelte progettuali che risultano in un compromesso tra velocità, dissipazione di potenza ed area occupata. In questa fase, ad esempio, si decidono il numero di colpi di clock che saranno necessari ad ottenere un determinato risultato intermedio. Questo è il momento in cui si decide se utilizzare o meno tecniche per la riduzione della potenza dissipata, si decide se riutilizzare dei blocchi per più fasi di elaborazione o se replicare pezzi di circuito.
Alla fine di questa fase si ottiene una descrizione HDL del circuito che utilizza costrutti sintetizzabili e che quindi è molto prossima al circuito finale.

Esempio: Per ognuno dei blocchi che compongono il circuito FFT si decide il formato degli ingressi e delle uscite. Nel far ciò si effettuano dei compromessi tra precisione del circuito, area occupata, potenza dissipata e velocità. Infatti, se per i segnali intermedi utilizzo un numero maggiore di bit, posso attendermi minori errori di troncamento e quindi migliore precisione. Ciononostante, devo aspettarmi una frequenza di clock massima ridotta, una maggiore occupazione di logica programmabile ed una maggiore energia dissipata per ogni elaborazione effettuata.

I singoli blocchi vengono quindi descritti in termini di componenti digitali elementari come registri multiplexer ed addizionatori. La descrizione terrà anche conto dei vincoli sulle prestazioni richiesti al circuito. Ad esempio, se ho vincoli stringenti sull'occupazione di area, ma il circuito non ha specifiche stringenti di velocità, posso pensare di utilizzare degli addizionatori seriali. Al contrario, se il limite più stringente è sulla velocità del circuito, sarà forse necessario replicare alcuni blocchi funzionali per effettuare le operazioni in parallelo. Tecniche di riduzione della potenza dissipata, come ad esempio il clock gating, possono essere implementate in questa fase.

Ribadiamo che la descrizione HDL dovrà utilizzare costrutti sintetizzabili. Particolare attenzione verrà riservata alla realizzazione di un circuito che sia di tipo sincrono.

Simulazione RTL

La simulazione della descrizione a livello RTL, non fornisce informazioni sulle prestazioni finali del circuito.
E' invece in grado di verificare che la descrizione di tipo sintetizzabile effettuata corrisponda ancora al circuito che era richiesto dalle specifiche. Risulta quindi importante per verificare di non aver fatto errori nel passaggio dalla descrizione funzionale alla descrizione RTL.
La simulazione, riguardando ormai un circuito sincrono completo, permette di aggiungere alcune informazioni al progetto che si sta conducendo. Pur non avendo idea di quale sarà la frequenza di lavoro del circuito finale, sarà possibile capire quanti colpi di clock sono necessari per ottenere il risultato atteso nei vari punti del circuito. In questa fase si potranno quindi curare gli aspetti relativi alla sincronizzazione dei dati di ingresso con i risultati intermedi e con le uscite.

Esempio: Il circuito per la realizzazione della FFT riceve gli ingressi necessari per il calcolo di uno dei risultati in vari blocchi in diversi colpi di clock. La simulazione RTL ci permette di capire se il circuito è in grado di cominciare ad elaborare correttamente i primi dati che arrivano ed utilizzare i successivi dati per completare il calcolo. Ricordiamo che potremmo aver deciso di utilizzare circuiti aritmetici di tipo seriale che utilizzano più colpi di clock per effettuare le elaborazioni. In questo caso dovremmo verificare di aver previsto degli opportuni registri a scorrimento in grado di memorizzare gli ingressi a mano a mano che arrivano.

Se invece avessimo scelto di replicare alcune sezioni del circuito per aumentare la velocità di elaborazione, potremmo dover verificare che il circuito che si occupa di smistare i dati in ingresso ai vari blocchi funzionali funzioni correttamente.

Sintesi (Translate e Mapping)

La successiva fase prevede la sintesi della descrizione RTL. Durante la fase di sintesi, sulla base dei componenti disponibili sul FPGA considerato, i vari blocchi funzionali descritti in HDL vengono tradotti in componenti digitali. A volte la corrispondenza può essere completa (ad esempio un flip flop si traduce in un flip flop) altre volte lo strumento di sintesi adatterà la descrizione RTL ai componenti disponibili nel FPGA (utilizzo di una memoria per implementare una logica combinatoria complessa o un registro a scorrimento particolarmente grande). La fase di sintesi utilizza i vincoli sulla temporizzazione per indirizzare la realizzazione circuitale.
In alcuni sistemi di sviluppo la procedura di sintesi viene divisa in due fasi distinte. La prima fase viene detta di *translate*. La fase di translate effettua una sintesi alla fine della quale il circuito è descritto utilizzando componenti elementari che sono indipendenti dal FPGA o dal CPLD che si desidera utilizzare. In linea di principio a valle della fase

di translate il circuito potrebbe anche essere realizzato utilizzando componenti standard o librerie di celle standard per circuiti VLSI di tipo ASIC.
La seconda fase è detta di Map o Mapping. E' la fase in cui gli elementi atomici del circuito dopo la fase di Translate vengono mappati sui componenti effettivamente disponibili nel circuito di destinazione. Se ad esempio realizziamo il circuito per le più comuni FPGA presenti sul mercato, la fase di Map terrà conto del fatto che tali FPGA contengono dei componenti programmabili in grado di implementare qualsiasi funzione con 4 ingressi ed un solo bit di uscita (LUT4, Look Up Table a 4 ingressi). Conseguentemente le sezioni combinatorie del circuito ottenuto dopo la fase di translate saranno suddivise in funzioni con 4 ingressi per adattarsi alle LUT (Es: y=AB+CD+E si implementerà come y=E+x con x=AB+CD, utilizzando due LUT a 4 ingressi).
Il risultato finale è, anche in questo caso, una netlist che utilizza i componenti disponibili nel circuito di destinazione (FPGA o CPLD) opportunamente configurati e collegati per realizzare la funzione indicata nella descrizione RTL.

Esempio: I blocchi logici che implementano la FFT, descritti in Verilog, vengono tradotti in circuiti. Addizionatori, registri, logica combinatoria e macchine a stati finiti descritti in Verilog sono dapprima ottimizzati per eliminare ridondanze e componenti inattivi e poi adattate ai componenti presenti nel FPGA scelta. Se sono disponibili diverse realizzazioni circuitali per lo stesso blocco, i vincoli sulla temporizzazione, sull'occupazione di logica programmabile e sulla potenza dissipata possono essere utilizzati per indirizzare la fase di sintesi. Ad esempio per un addizionatore si può decidere di utilizzare i blocchi programmabili oppure uno dei blocchi pensati per i DSP. Per un registro a scorrimento si possono utilizzare i blocchi programmabili o una memoria. Per un multiplexer si possono utilizzare i blocchi logici programmabili o i buffer tristate etc.
Al termine della fase di sintesi si ottiene una netlist del circuito sintetizzato.

Simulazione post sintesi

Dopo la sintesi il circuito è quasi completo. La necessaria fase di simulazione che si associa alla sintesi è denominata simulazione post sintesi e serve in primo luogo per verificare che la fase di sintesi non abbia introdotto errori (bug del software di sintesi).
La simulazione post sintesi, fornisce inoltre delle indicazioni approssimate sulle transizioni dei segnali per effetto dei ritardi di propagazione. La maggioranza delle transizioni sui segnali sono previste e servono a modificare l'uscita dei circuiti. Altre transizioni sono transitorie e sono dovute ai ritardi di propagazione dei segnali lungo il circuito (si pensi alla propagazione del riporto in un addizionatore). Alcune volte si possono avere delle transizioni multiple su di uno stesso segnale, sempre dovute ai ritardi di propagazione, che lasciano il segnale nello stato iniziale. Tali transizioni, non significative per la funzionalità del circuito sono dette glitch.
I glitch e le transizioni sui segnali presenti nel circuito sono significativi per la determinazione della potenza dissipata. Se sono state utilizzate tecniche che agiscono sul segnale di clock, come il clock gating, sono anche determinanti per la funzionalità del circuito.
Eventuali problemi evidenziati durante la simulazione post sintesi richiedono, a seconda dei casi, una nuova fase di sintesi (con nuovi vincoli tesi ad eliminare i problemi riscontrati) o una nuova realizzazione della descrizione RTL.

Place & Route

Quando il circuito è composto da blocchi elementari disponibili nel FPGA, al fine di ottenere il circuito finale è solo necessario indicare in quale posizione sono i componenti del FPGA che utilizzeremo (Piazzamento o Placing) ed infine collegare i vari blocchi (Collegamento o Routing). Le fasi di Place & Route non sono complesse da un punto di vista concettuale ma sono spesso dispendiose in termini di risorse di calcolo. Infatti una porzione considerevole delle prestazioni finali dipendono dall'aver piazzato i vari blocchi in modo tale da minimizzare la lunghezza dei collegamenti (pur riuscendo a collegare tutti i blocchi necessari). Poiché non vi sono algoritmi in grado di prevedere a priori un piazzamento ed un collegamento ottimali, la soluzione generalmente utilizzata è quella di effettuare un piazzamento iniziale basato su regole euristiche, seguito da una serie di tentativi di miglioramento sostanzialmente casuali.
Se il circuito è complesso, e tende a riempire l'intero FPGA, questo significa dover effettuare milioni di tentativi e dover utilizzare numerose risorse di calcolo. La fase di place & route è praticamente automatica. Il progettista in genere indica l'impegno di risorse di calcolo, e quindi il tempo, che intende impiegare nella fase di place & route. Tale impegno di risorse è denominato in inglese "effort" (sforzo). Un maggiore effort produce un circuito con migliori prestazioni ma richiede molto più tempo. E' consigliabile riservare i piazzamenti con effort massimo alle fasi finali del progetto, quando si è ragionevolmente certi di non dover ripetere la fase di place & route.

Timing analysis

La fase di sintesi fornisce una descrizione precisa degli elementi del FPGA che comporranno il circuito finale e del loro collegamento. Possiamo quindi avere una stima approssimata del ritardo del circuito. Infatti, di ogni blocco presente nel FPGA conosciamo il ritardo in funzione del fan-out. Le uniche informazioni che mancano per una caratterizzazione completa sono le capacità parassite (e quindi i ritardi dovuti ai collegamenti).
A valle della fase di place & route si ha, infine, anche la conoscenza esatta della lunghezza e del tipo dei collegamenti tra i componenti del circuito. La stima della frequenza di funzionamento e dei ritardi può quindi essere molto precisa.
Per stimare la frequenza di funzionamento del circuito si utilizza un software denominato Timing Analyzer.
Il software che effettua la timing analysis calcola i ritardi del circuito, definendo il ritardo del cammino critico (percorso con il maggiore ritardo tra ingresso ed uscita) nonché degli altri cammini. Sulla base dei ritardi e dei vincoli sui tempi di setup e di hold tale software è in grado di stimare la frequenza di funzionamento del circuito e la presenza di problemi riguardanti il tempo di hold.

La timing analysis si effettua a vari livelli di realizzazione del circuito. Dopo la sintesi l'analisi sarà condotta senza conoscere i ritardi causati dalle interconnessioni.
Se le analisi sulle prestazioni ci dicono, già in questa fase, che le prestazioni finali non soddisferanno le specifiche, è necessario tornare indietro alla fase di descrizione RTL per esplorare nuove possibilità progettuali. Poiché il software indica il percorso del segnale che minaccia il soddisfacimento delle specifiche, la nuova fase di descrizione RTL potrà essere condotta più efficacemente ed avendo ben chiaro in quale punto del circuito agire.
Dopo la fase di place & route anche i ritardi delle interconnessioni sono noti e quindi si avrà una stima molto più precisa della frequenza di funzionamento. Una nuova timing analysis viene utilizzata per ricavare le prestazioni finali del circuito e per indirizzare nuove fasi di place & route, se necessarie.
E' importante evidenziare che la timing analysis non è solo utilizzata dopo le fasi di sintesi e place & route, per verificare le prestazioni del circuito. Essa è anche utilizzata durante le fasi di sintesi e di place & route, per indirizzare il software verso la realizzazione di un circuito ottimizzato sulla base dei vincoli richiesti dal progettista.

Power analysis

La dissipazione di potenza di un circuito programmabile di tipo FPGA o CPLD si divide in una componente statica ed una dinamica.
A valle della fase di sintesi si ha una netlist che indica precisamente gli elementi del FPGA che comporranno il circuito finale. Questo permette di avere una stima abbastanza precisa della potenza dissipata di tipo statico dissipata dal circuito.
Per aver una stima della potenza dissipata è necessario conoscere i ritardi di propagazione dei segnali, che incidono sul tipo e sul numero di transizioni dei segnali interni al circuito. Tale stima è possibile già a valle della fase di sintesi, ma è decisamente più precisa a valle della fase di place & route.
La stima della potenza dissipata può anche essere effettuata sulla base delle simulazioni del circuito che indicheranno le condizioni di funzionamento tipiche del circuito stesso.
Anche se meno diffusamente utilizzata della timing analysis, anche la power analysis può essere utilizzata sia durante le fasi di sintesi e place & route per indirizzarne il risultato finale, sia per valutare a posteriori il risultato ottenuto.

Simulazione post place & route

E' la simulazione più precisa che si possa effettuare, ma anche la più lenta. La simulazione tiene conto di tutti i blocchi logici e degli elementi parassiti presenti del circuito.
E' la simulazione finale che certifica il funzionamento del circuito nelle condizioni di funzionamento scelte come test.

Generazione file di programmazione e programmazione del FPGA.

Sono le fasi in cui viene creato il file binario che si usa per programmare la FPGA e si programma il circuito finale.
In queste fasi il progettista non interviene.

Il flusso di progetto mostrato non è l'unico possibile. E' possibile utilizzare flussi di progetto semplificati per progetti molto semplici così come è possibile utilizzare flussi di progetto più complessi se il sistema da progettare lo richiede.
In molti casi si utilizzano ulteriori strumenti software dedicati alla semplificazione o alla simulazione di alcune fasi progettuali. Ad esempio, poiché molti circuiti contengono una o più macchine a stati finiti, molti flussi di progetto mettono a disposizione degli strumenti software che semplificano la descrizione RTL delle stesse.
I concetti di base sono comunque gli stessi per tutti i flussi di progetto.

3. Il linguaggio HDL Verilog.

Il linguaggio Verilog permette di descrivere un circuito in diversi modi. Per le applicazioni che tratteremo il Verilog sarà utilizzato per descrivere i circuiti digitali con costrutti sintetizzabili, e per definire i test bench che saranno utilizzati per verificare la funzionalità dei circuiti realizzati.
L'approccio seguito per la presentazione del linguaggio è orientato all'applicazione. I costrutti sono presentati insieme ad esempi pratici. Non tutti i costrutti del Verilog sono inclusi in queste note. In alcuni casi in quanto si tratta di costrutti avanzati non necessari in questo primo corso. In altri casi in quanto si tratta di costrutti che l'autore non ritiene utili per la progettazione di circuiti digitali efficienti.

3.1 Esempio di circuito descritto in Verilog

Esaminiamo un primo esempio di circuito descritto in Verilog. Realizziamo un semplice decoder con 2 ingressi e 4 uscite. Il codice Verilog è mostrato di seguito.

```
module decoder2to4 (A1,A0,Y3,Y2,Y1,Y0);
input  A1,A0;
output Y3,Y2,Y1,Y0;

wire A1,A0,Y3,Y2,Y1,Y0;

assign Y0 = (~A1)&(~A0);
assign Y1 = (~A1)&( A0);
assign Y2 = ( A1)&(~A0);
assign Y3 = ( A1)&( A0);

endmodule
```

Il codice mostrato è molto semplice ed è possibile anche per coloro che non conoscono il linguaggio Verilog convincersi del fatto che stia implementando un decoder.
L'unità base di descrizione nel Verilog è il **module**. Esso descrive la funzione o la struttura di un circuito, oltre alle porte attraverso il quale il progetto comunica con l'esterno o con altri moduli. Inoltre, è possibile instanziare (neologismo che deriva dall'inglese instance) un modulo all'interno di un altro creando circuiti gerarchici. Di seguito riportiamo la sintassi base di un modulo.

```
module nome_modulo (lista_porte);

 Dichiarazioni:
reg, wire, parameter, input, output, inout, function, tasks,...

 Costrutti:
 initial, always, istanziazione modulo,  assegnazione continua...

endmodule
```

Un modulo comincia con la direttiva **module** e si conclude con la direttiva **endmodule**.
La direttiva module è seguita dal nome del modulo, che dovrebbe essere indicativo della funzionalità del circuito, e dalla lista delle porte di collegamento con l'esterno.
Un progetto ben strutturato deve prima di tutto aver ben chiara la struttura di questa prima riga. Nel caso mostrato in precedenza la prima riga è:

```
module decoder2to4 (A1,A0,Y3,Y2,Y1,Y0);
```

L'analisi del modulo chiarisce il nome del circuito (decoder2to4) e le porte di collegamento (A1, A0, Y3, Y2, Y1, Y0).
Il modulo è quindi diviso in due parti. La prima parte contiene le dichiarazioni nella quale si indica se le porte sono di ingresso o di uscita, si definiscono ulteriori segnali, si definisce il tipo dei segnali e delle porte del circuito e vengono fissati eventuali parametri necessari all'implementazione del circuito. Il metodo migliore per completare la sezione delle dichiarazioni è avere uno schema del circuito che vogliamo realizzare. Da tale schema risultano immediatamente evidenti tutti i dati che servono per completare la fase di dichiarazione.
La seconda parte contiene i costrutti che determinano le funzionalità del circuito. I costrutti servono a implementare porte logiche, componenti digitali, ad utilizzare gerarchicamente dei moduli precedentemente descritti.
La descrizione delle funzionalità sembra essere la più complessa ma si semplifica notevolmente se l'analisi delle specifiche e la sezione delle dichiarazioni sono state gestite accuratamente.

La sezione delle dichiarazioni per il decoder indica quali tra le porte di ingresso sono ingressi (input) o uscite (output). Si definisce inoltre che tutte le porte, sia di ingresso, sia di uscita sono di tipo wire (filo). Il tipo wire corrisponde ad un nodo del circuito o, equivalentemente, ad un filo di collegamento. Di seguito la fase dichiarativa del decoder.

```
input  A1,A0;
output Y3,Y2,Y1,Y0;

wire A1,A0,Y3,Y2,Y1,Y0;
```

La descrizione della funzionalità, a questo punto è molto semplice ed utilizza dei costrutti che assegnano alle uscite una funzione booleana degli ingressi (la AND logica si indica con '&', la negazione logica si indica con '~').

```
assign Y0 = (~A1)&(~A0);
assign Y1 = (~A1)&( A0);
assign Y2 = ( A1)&(~A0);
assign Y3 = ( A1)&( A0);
```

Ricordate di far corrispondere ad ogni sezione della descrizione Verilog un componente circuitale. Se non riuscite ad effettuare tale associazione probabilmente significa che avete utilizzato del codice non sintetizzabile o scarsamente leggibile.
Ad esempio il codice Verilog mostrato si presenta come un circuito con due segnali di ingresso e quattro uscite ad un bit. Le uscite sono determinate a partire dagli ingressi mediante 4 funzioni booleane. Ognuna delle righe che cominciano con *assign* corrispondono ad un circuito combinatorio con due ingressi ed una uscita (una porta AND con alcuni degli ingressi negati).

3.2 Commenti

Risulta utile usare spazi bianchi (blanks, tab e linee vuote) e commenti per migliorare la leggibilità del codice. In Verilog HDL esistono due tipi di commenti:

```
/* Prima forma: il commento può
estendersi su più righe */

// Seconda forma: il commento può estendersi su una sola riga
```

Sia i commenti sia gli spazi bianchi vengono ignorati dal compilatore.

3.3 Maiuscole minuscole (Case sensitive)

Il linguaggio Verilog è 'case sensitive', vale a dire che distingue tra lettere maiuscole e minuscole.
Tutte le parole riservate per la sintassi del Verilog sono minuscole. Ad esempio:

```
input          // a Verilog Keyword
wire           // a Verilog Keyword
WIRE           // a unique name ( not a keyword)
Wire           // a unique name (not a keyword)
```

3.4 Segnali vettoriali

La precedente descrizione del decoder con 2 ingressi è 4 uscite diventerebbe molto scomoda se gli ingressi dovessero essere, ad esempio, quattro e quindi dovessimo avere 16 uscite.
Il Verilog permette di definire, per casi come quello indicato, dei segnali vettoriali. Di seguito il codice del decoder.

```
module decoder2to4 (A,Y);
input  [1:0] A;        // Ingresso vettoriale a 2 bit
output [3:0] Y;        // Uscita vettoriale a 4 bit

wire [1:0] A;          // Il segnale A è a 2 bit
wire [3:0] Y;          // Il segnale Y è a 4 bit

assign Y[0] = (~A[1])&(~A[0]);
assign Y[1] = (~A[1])&( A[0]);
assign Y[2] = ( A[1])&(~A[0]);
assign Y[3] = ( A[1])&( A[0]);

endmodule
```

Il codice è facilmente leggibile. I segnali di ingresso e di uscita del *module* sono definiti come bus rispettivamente di due e quattro bit.
In definitiva in Verilog le variabili possono essere scalari o vettoriali, ad esempio:

```
wire [1:0] select;     // bus di 2 bit
input [7:0] address;   // bus di 8 bit
output [8:1] out;      // bus di 8 bit
wire enabled;          // segnale scalare
```

La dimensione dei vettori è generalmente definita con:

[<start>:<end>]

dove indichiamo con start il bit più significativo (MSB) e con end il bit meno significativo (LSB). La scelta di assegnare

un indice numerico più alto al MSB è una convenzione che si consiglia caldamente di seguire. Per quanto sia possibile dichiarare un vettore anche come:

```
wire [0:3] sel;        // bus di 4 bit
input [1:8] add;       // bus di 8 bit
```

tale utilizzo è molto raro e prono ad errori ed incomprensioni molto difficili da correggere.

Come ulteriore esempio presentiamo la descrizione circuitale di un codificatore con otto ingressi e tre uscite. La descrizione richiede poche modifiche rispetto al circuito precedentemente mostrato.

```
module encoder8to3(A,Y);
input  [7:0] A;    // declaration section
output [2:0] Y;

wire [7:0] A;
wire [2:0] Y;

// description section
assign Y[2] = A[7] | A[6] | A[5] | A[4];
assign Y[1] = A[7] | A[6] | A[3] | A[2];
assign Y[0] = A[7] | A[5] | A[3] | A[1];

endmodule
```

3.5 Nodi interni

Non sempre è agevole determinare le uscite del circuito direttamente dagli ingressi. In questi casi sono necessari dei segnali interni per facilitare la descrizione del circuito. Prendiamo ad esempio un encoder con priorità. Le equazioni booleane di un encoder di questo tipo sono state descritte in precedenza e sono mostrate di seguito.

Idle = (~A7) & (~A6) & (~A5) & (~A4) & (~A3) & (~A2) & (~A1) & (~A0).
H7 = A7
H6 = A6 & (~A7)
H5 = A5 & (~A6) & (~A7)
...
H1 = A1 & (~A2) & (~A3) & (~A4) & (~A5) & (~A6) & (~A7).

I segnali *Hi* sono al più alti uno alla volta e sono utilizzati come ingressi ad un codificatore senza priorità:

Y2 = H4 | H5 | H6 | H7
Y1 = H2 | H3 | H6 | H7
Y0 = H1 | H3 | H5 | H7

Il circuito da realizzare, che nonostante la sua semplicità si mostra già sufficientemente complesso, è mostrato in Figura 3.1. I segnali *H1...H7* sono nodi interni che non sono collegati né all'ingresso né all'uscita.

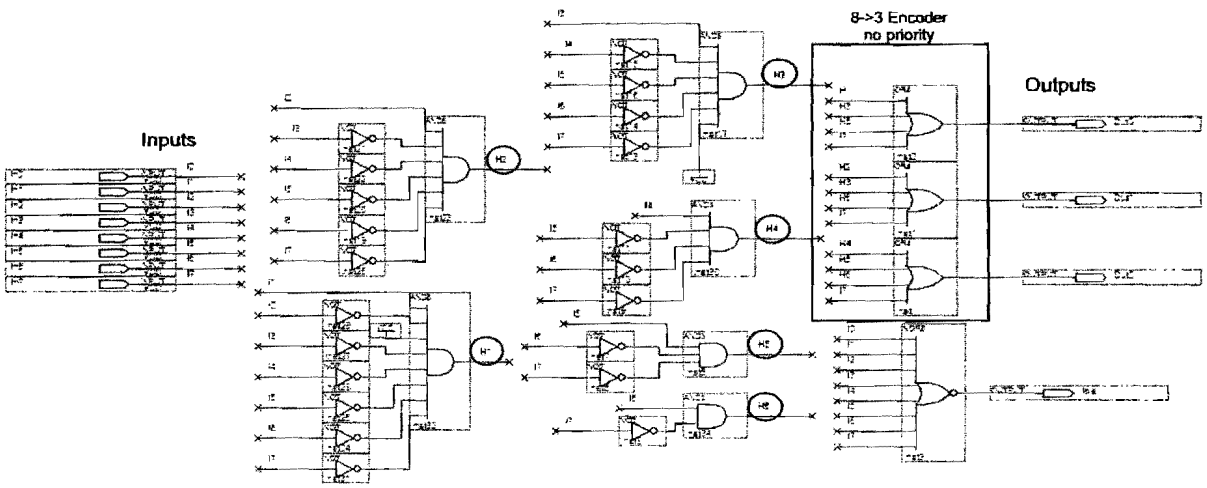


Figura 3.1: Schema di un encoder 8 ingressi 3 uscite con priorità. Si evidenziano i nodi interni H1...H7. La sezione nel riquadro è il circuito corrispondente ad un encoder senza priorità tra gli ingressi.

La descrizione in Verilog del circuito di Figura 3.1 è invece molto semplice. Una possibile descrizione è mostrata di seguito.

```
module prenc83(A,Y,idle);
input  [7:0] A;   //declaration section
output [2:0] Y;
output idle;

wire [7:0] A;
wire [2:0] Y;
wire [7:1] H;      // declare 7 internal nodes

// description section
assign Y[2] = H[7] | H[6] | H[5] | H[4];   // encoder with no priority
assign Y[1] = H[7] | H[6] | H[3] | H[2];
assign Y[0] = H[7] | H[5] | H[3] | H[1];

assign idle = (~A[7]) & (~A[6]) & (~A[5]) & (~A[4]) & (~A[3]) & (~A[2]) & (~A[1]) & (~A[0]);
assign H[7] = A[7];
assign H[6] = A[6] & (~A[7]);
assign H[5] = A[5] & (~A[6]) & (~A[7]);
assign H[4] = A[4] & (~A[5]) & (~A[6]) & (~A[7]);
assign H[3] = A[3] & (~A[4]) & (~A[5]) & (~A[6]) & (~A[7]);
assign H[2] = A[2] & (~A[3]) & (~A[4]) & (~A[5]) & (~A[6]) & (~A[7]);
assign H[1] = A[1] & (~A[2]) & (~A[3]) & (~A[4]) & (~A[5]) & (~A[6]) & (~A[7]);

endmodule
```

La descrizione dell'encoder con priorità comincia a mostrare la grande semplicità e flessibilità di un HDL. Il linguaggio è semplice e diretto. Permette di descrivere circuiti complessi minimizzando gli errori banali, come gli errati collegamenti ed è evidente come sia preferibile utilizzare il codice HDL piuttosto che realizzare il circuito di Figura 3.1.
Si noti la dichiarazione:

```
wire [7:1] H;      // declare 7 internal nodes
```

che definisce i nodi interni, non collegati, all'ingresso o all'uscita, ma necessari alla descrizione del circuito.

Parallelismo negli HDL

Un aspetto importante del Verilog, e degli HDL in generale, è esemplificato nella descrizione HDL del codificatore con priorità mostrata in precedenza.
Come potete notare i primi costrutti *assign* utilizzano i segnali *Hi* che nel codice HDL sono determinati solo successivamente. Se la stessa cosa dovesse essere tentata in linguaggio C o in un altro linguaggio procedurale, si avrebbe un errore.
Se invece utilizziamo un HDL lo stile di descrizione mostrato è lecito in quanto il linguaggio HDL è intrinsecamente parallelo. Una descrizione HDL corrisponde ad un circuito. Ciò che definisce un circuito è l'insieme delle connessioni tra i nodi ed i sottocircuiti che lo compongono. L'ordine con cui questi elementi vengono definiti nel codice HDL non influenza il circuito finale. Per convincersene è sufficiente cambiare a piacimento l'ordine delle istruzioni nel codice HDL e ripetere la sintesi e la simulazione del circuito notando che il risultato finale rimane inalterato.
E' fondamentale ricordare questo concetto per evitare errori progettuali. Se infatti, nel descrivere un circuito, vi trovate a dover utilizzare un segnale che 'deve' essere calcolato precedentemente, pena il malfunzionamento del circuito, è probabile che la descrizione HDL che state realizzando non sia sintetizzabile.

Ottimizzazione

Il codificatore con priorità mostrato utilizza 11 costrutti *assign*. Da un punto di vista della descrizione HDL dobbiamo associare ad ognuno degli *assign* una porta logica. Il progettista potrebbe tentare di minimizzare le funzioni logiche o di accorpare alcuni segnali, per ridurre il numero di porte logiche utilizzate e quindi ridurre l'occupazione di logica del circuito finale.
Fortunatamente tale operazione non è necessaria. Infatti, la descrizione HDL, durante la fase di sintesi, viene anche ottimizzata eliminando eventuali ridondanze e minimizzando le funzioni booleane. Il risultato circuitale, quindi, potrebbe non utilizzare 11 porte logiche. Ad esempio il codificatore con priorità, a valle della fase di sintesi, viene implementato con 7 Logic Elements di un FPGA CycloneII. La vista Technology Map del circuito finale (ottenibile effettuando un click su *Technology Map Viewer* nella finestra *Tasks* del software *QuartusII*) è mostrata in Figura 3.2,

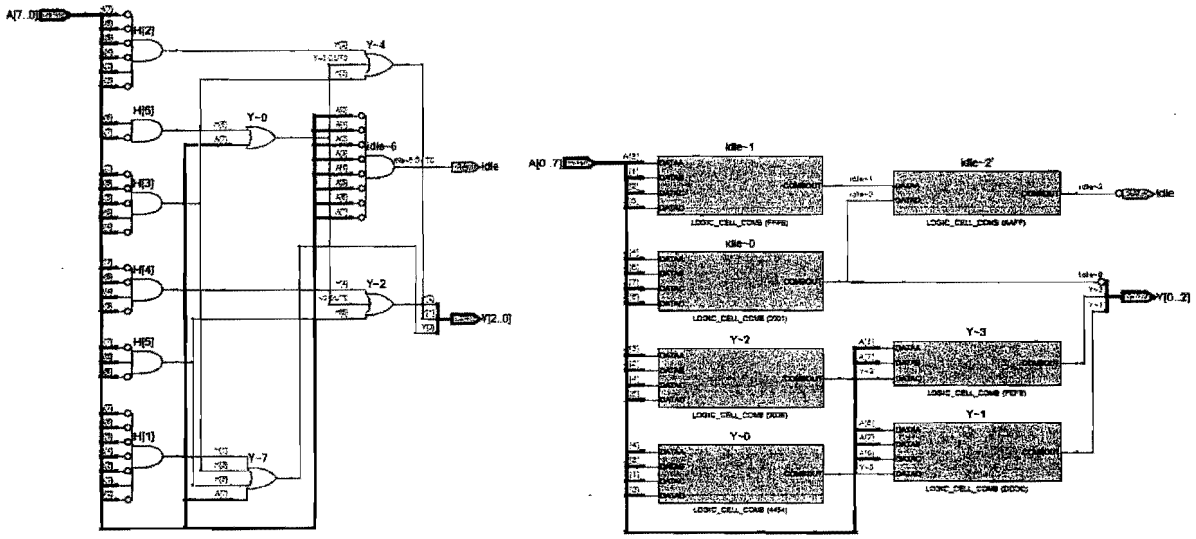


Figura 3.2: Codificatore con priorità. A sinistra vista RTL del circuito. A destra vista a valle della fase di sintesi e di Map. Il circuito è realizzato mediante 7 Logic Elements.

Per convincersi della scarsa utilità progettuale di una fase manuale di ottimizzazione delle funzioni booleane si proceda a modificare la descrizione Verilog del circuito, ad esempio, eliminando i segnali *Hi* ed accorpando la loro definizione nel calcolo di Y, ed aggiungere dei mintermini ridondanti come nell'esempio seguente.

```
assign idle = (~A[7]&A[7])|(~A[7])&(~A[6])&(~A[5])&(~A[4])&(~A[3])&(~A[2])&(~A[1])&(~A[0]);
assign Y[2] = A[7]|(A[6]&(~A[7]))|(A[5]&(~A[6])&(~A[7]))|(A[4]&(~A[5])&(~A[6])&(~A[7]));
```

Nei costrutti esemplificati il termine `(~A[7]&A[7])` è sempre nullo, mentre, ad esempio, il termine `(A[6]&(~A[7]))` sostituisce `H[6]`. Dopo aver sintetizzato il circuito si esaminino nuovamente le viste RTL e Technology Map. Il risultato è mostrato in Figura 3.3 ed è assolutamente indistinguibile dal precedente, ottenuto senza ottimizzare le funzioni, mostrato in Figura 3.2.

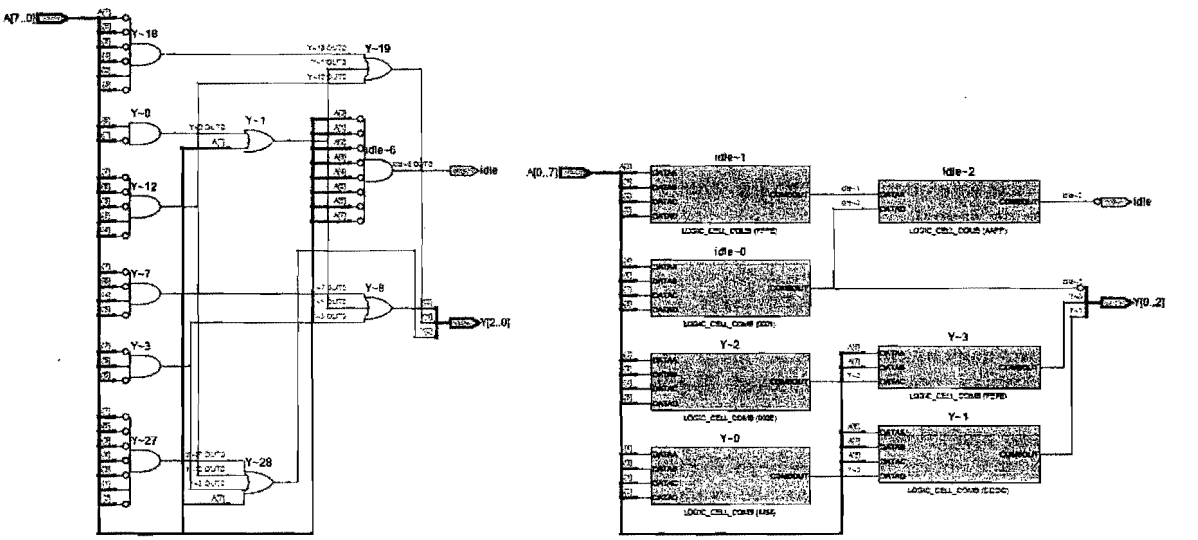


Figura 3.3: Codificatore con priorità. A sinistra la vista RTL ed a destra la vista post Map. Entrambe sono identiche a quanto mostrato in Figura 3.2.

3.6 Concatenazione

Un operatore spesso utile nelle descrizioni di circuiti in Verilog è quello di concatenazione, che si utilizza mediante le parentesi graffe {,}. Definiamo un vettore A di 3 bit, un vettore B di 4 bit, entrambi di ingresso, ed un vettore d'uscita C di 7 bit:

```
input  [2:0] A;
input  [3:0] B;
output [6:0] C;
```

Tramite l'operatore di concatenazione (e l'assegnazione continua) possiamo scrivere:

assign C = {A,B};

in cui il bit più significativo di C sarà l'MSB di A, mentre quello meno significativo sarà l'LSB di B. La restante parte di C è costituita dagli altri bit di A e B.

L'operatore di concatenazione è utilizzabile parimenti per definire il segnale di destinazione di un'assegnazione. Anche la seguente assegnazione è quindi lecita.

assign {A,B}=C;

3.7 Valori logici

Una variabile booleana assume i valori logici '1' o '0' (vero o falso) e ci si attende un comportamento simile dai circuiti elettronici digitali.

Purtroppo la rigida suddivisione dei segnali in '1' e '0' non rappresenta adeguatamente la realtà. Un esempio tipico sono i circuiti tristate, per i quali l'uscite può assumere il valore di 'alta impedenza', ed i registri, che all'accensione possono assumere sia lo stato '1' sia lo stato '0' ed è quindi corretto indicare il loro stato come 'sconosciuto' (unknown). Il Verilog, al fine di catturare adeguatamente il comportamento dei circuiti digitali suppone che l'insieme degli stati logici che un segnale può assumere sia costituito da:

- *0* - rappresenta uno zero logico, o una condizione falsa;
- *1* – rappresenta un uno logico, o una condizione vera;
- *x* – rappresenta un valore logico sconosciuto. Indica anche il valore "don't care" in alcune condizioni.
- *z* – rappresenta uno stato di alta impedenza.

3.8 Operatori Booleani (bitwise operators)

Eseguono le operazioni logiche sui segnali. Gli operatori sono mostrati in Tab. 3.1. Se per gli operatori binari gli ingressi sono di tipo vettoriale, l'ingresso con meno bit viene esteso a sinistra con degli zero. L'operazione è quindi eseguita su ognuno dei bit e ritorna un risultato rappresentato sul numero di bit corrispondente al più lungo degli ingressi. Di seguito alcuni esempi:

```
~1'b0 = 1
~5'b10001 = 01110
 4'b0011 & 4'b0101  = 0001
 4'b0011 | 4'b0101  = 0111
 4'b0011 ^ 4'b0101  = 0110
 4'b0011 ~^ 4'b0101 = 1001
 6'b110011 & 4'b0101 =   6'b110011 & 6'b000101 = 000001
 4'b0101 & 6'b110011 =   6'b000101 & 6'b110011 = 110001
```

Operatore	Descrizione
~	NOT
&	AND
|	OR
^	XOR
~^ oppure ^~	XNOR o equivalenza

Tabella 3.1: Funzioni Booleane che operano sui singoli bit degli operandi (bitwise) equivalgono alle usuali funzioni Booleane sui segnali.

3.9 Gerarchia

Qualsiasi sistema che raggiunga una complessità non elementare deve essere progettato dopo averlo suddiviso in moduli più semplici da testare individualmente. Tradizionalmente la progettazione di un sistema complesso è di tipo top-down (si suddivide il sistema completo in elementi via via più semplici sino a renderlo trattabile); bottom-up (si parte da componenti elementari che vengono connessi in vista del risultato finale, sino a giungere al sistema completo); oppure mista (il sistema iniziale viene suddiviso per semplificarlo e quando i sottosistemi sono di complessità intermedia vengono realizzati utilizzando componenti elementari già disponibili).

In tutti i casi si rende necessario connettere dei componenti per realizzare componenti più complessi.

Si supponga di aver progettato un componente che si vuole utilizzare nel sistema che si sta descrivendo. Il componente, a sua volta definito da un *module* in linguaggio Verilog, può essere presente nello stesso file nel quale è utilizzato o in un file esterno. Nel secondo caso è necessario che il sistema di sviluppo che realizza il circuito finale abbia incluso il file tra quelli appartenenti al progetto.

Per descrivere l'azione con cui si utilizza un componente descritto precedentemente esternamente si eseguono le azioni di istanziazione e collegamento. Con l'istanziazione si indica quale componente si utilizza e gli si attribuisce un nome. Ad esempio, se si utilizza un decoder per decodificare i segnali di ingresso, dalla libreria di componenti si istanzia un componente decoder e, ad esempio, lo si indica con il nome input_decoder. Con la fase di collegamento si indicano i segnali che vanno collegati agli ingressi ed alle uscite del componente (in inglese 'interface port' o semplicemente

'port') che si istanzia.
La sintassi per l'utilizzo di un componente è la seguente.

<module name> <instance name>(<Interface ports>)

La sintassi prevede che si indichi dapprima il module che si intende istanziare, seguito dal nome attribuito al componente istanziato nel circuito che si descrive e, tra parentesi, l'elenco dei segnali di interfaccia del modulo e la corrispondenza con i segnali del circuito che si descrive.
Un discorso a parte va fatto per la realizzazione dei collegamenti con le porte di interfaccia del modulo che in Verilog può essere effettuato in vari modi.
E' possibile effettuare i collegamenti per posizione. Si supponga di aver definito un modulo per un full adder definito dal seguente **module**.

```
module fulladd (sum, c_out, a, b, c_in);
output sum, c_out;
input a, b, c_in;
      .
      .
      .
endmodule
```

Se si intende descrivere un sistema che utilizzi dei full adder si può decidere di utilizzare il componente fulladd precedentemente descritto. Si supponga di aver definito delle net dai nomi `s, co, add1, add2, ci`, un possibile costrutto per l'istanziazione della UUT è:

```
fulladd UUT(s, co add1, add2, ci)
```

che connette, in funzione della posizione il segnale *s* al primo port del module fulladd (*sum*), il secondo segnale, *co*, al secondo port (*c_out*) etc.
La sintassi mostrata è molto compatta, ma poco leggibile e pericolosa. Infatti richiede un controllo preciso dell'ordine dei segnali di interfaccia del modulo istanziato, e causa errori che sono molto difficili da scovare.
Molto più sicura e consigliata è la connessione per nome dei segnali alle porte di interfaccia del modulo. Nel caso del full adder considerato il costrutto che effettua tale collegamento è, ad esempio:

```
fulladd UUT(.sum(s),.a(add1), .b(add2), .c_out(co), .c_in(ci))
```

La connessione per nome richiede che si indichi il nome del port di interfaccia del module preceduto da un punto con, tra parentesi l'indicazione del segnale da collegare.

.<port_name>(<signal_name>)

Il collegamento per nome non dipende dall'ordine dei segnali, è più leggibile, riduce gli errori dovuti ai collegamenti ed è fortemente consigliato.

Addizionatore carry ripple a 4 bit realizzato mediante descrizione gerarchica.

Realizziamo l'addizionatore unendo gerarchicamente dei full adder. Il module relativo al full adder è il seguente.

```
module fulladd (sum, c_out, a, b, c_in);
output sum, c_out;
input  a, b, c_in;

assign sum  =  a ^ b ^ c_in;
assign c_out= ( (a^b) & c_in ) | ( a & b );

endmodule
```

L'addizionatore può quindi essere descritto istanziando 4 addizionatori ad un bit e connettendo opportunamente gli ingressi e le uscite. Sarà necessario un segnale interno per la connessione dei riporti uscenti degli addizionatori di ordine minore ai riporti entranti degli addizionatori di ordine maggiore. Il circuito risultante, che non utilizza un riporto entrante né calcola il riporto uscente, è il seguente.

```
module add_ger (A,B,S);
input [3:0] A,B;
output [3:0] S;
wire [3:0] A,B,S;
wire [2:0] C;

fulladd add_0(.sum(S[0]), .c_out(C[0]), .a(A[0]), .b(B[0]), .c_in(1'b0));
fulladd add_1(.sum(S[1]), .c_out(C[1]), .a(A[1]), .b(B[1]), .c_in(C[0]));
fulladd add_2(.sum(S[2]), .c_out(C[2]), .a(A[2]), .b(B[2]), .c_in(C[1]));
fulladd add_3(.sum(S[3]),               .a(A[3]), .b(B[3]), .c_in(C[2]));

endmodule
```

Si notino le tre istanze dell'addizionatore completo. Interessante notare che il primo addizionatore ha il carry in (c_in) collegato a massa mediante la notazione `.c_in(1'b0)`. L'ultimo addizionatore, invece, ha il carry out non collegato in quanto il collegamento per nome non è effettuato. E' questo il metodo utilizzato per lasciare un'uscita flottante. Durante la sintesi tale operazione causa un warning che può essere tollerato. La fase di ottimizzazione circuitale si occupa poi di

rimuovere la logica per il calcolo dell'uscita che è rimasta flottante e che è quindi non più necessaria.

3.10 Blocchi procedurali

Come precedentemente evidenziato i costrutti Verilog sono concorrenti. Vengono eseguiti in parallelo e l'ordine in cui sono presentati nella descrizione non ha effetto sul circuito risultante.
In alcuni casi, la descrizione di un circuito include componenti che non è agevole descrivere mediante semplici funzioni Booleane o connettendo moduli descritti da funzioni Booleane perché, ad esempio, risulterebbero in funzioni troppo complesse. In altri casi si desidera descrivere dei circuiti per i quali l'effettiva realizzazione circuitale è meno importante della precisa descrizione del compito eseguito dal circuito.
In questi casi è conveniente, per descrivere il circuito, utilizzare dei blocchi di codice Verilog, denominati blocchi procedurali, che sono composti da differenti istruzioni che vengono eseguite sequenzialmente. Per tali blocchi il circuito risultante è quindi dipendente dall'ordine delle istruzioni.
I punti fondamentali da comprendere per quanto riguarda i blocchi procedurali sono:

- Ogni blocco procedurale descrive un circuito digitale, combinatorio o sequenziale. Differenti blocchi procedurali corrispondono a diversi circuiti il cui funzionamento è quindi intrinsecamente parallelo. Se, analizzando il codice di un blocco procedurale, il circuito di cui descrive il comportamento non è chiaro, è altamente probabile che il blocco sia non sintetizzabile. Il risultato progettuale di un blocco procedurale di questo tipo è imprevedibile.
- Durante la simulazione del codice Verilog che contiene i blocchi procedurali, le singole istruzioni sono eseguite sequenzialmente. E' questa l'unica fase durante la quale è conveniente immaginare il comportamento del blocco come somma dell'esecuzione ordinata delle singole istruzioni.
- Durante la sintesi, il sintetizzatore analizza nel loro ordine le istruzioni di un blocco procedurale, le elabora, e cerca di ottenere un circuito digitale che si comporti come descritto nel blocco procedurale. Durante la simulazione post sintesi l'insieme delle istruzioni non esiste più. Ciò che viene simulato è il circuito ottenuto dalla descrizione.

I blocchi, procedurali sono definiti dalla seguente sintassi

```
always @(<event control>)
begin
<procedural statements>
end
```

La parola chiave *always* indica l'inizio di un blocco procedurale. Le istruzioni che vengono eseguite sequenzialmente sono racchiuse tra le parole chiave *begin* e *end*. Il simbolo @ è seguito, tra parentesi, da una condizione logica che, se vera, abilita l'esecuzione del blocco procedurale. Altre caratteristiche dei blocchi procedurali sono:

- tutti i segnali il cui valore è assegnato all'interno di un blocco procedurale devono essere dichiarati di tipo *reg*.
- quando si descrive un circuito l'assegnazione di un valore ad un segnale si effettua utilizzando la sintassi A <= B; L'assegnazione con il simbolo <= è denominata assegnazione *non blocking*.

Un esempio di circuito di tipo combinatorio che abbiamo già descritto mediante le equazioni Booleane e che può risultare complesso da descrivere quando il numero di bit in ingresso e in uscita aumenta, è il codificatore con priorità. La descrizione in basso mostra un esempio di come sia possibile descrivere un codificatore con priorità che abbia sedici ingressi e quattro uscite.

```
module prencl6to4 (A,Y,Idle);
 output [3:0] Y;
 output Idle;
 input [15:0] A;
 reg [3:0] Y;

 assign Idle = ~(A[15] | A[14] | A[13] | A[12] | A[11] | A[10] | A[9] | A[8] |
 A[7] | A[6] | A[5] | A[4] | A[3] | A[2] | A[1] | A[0] );

 always @(A) // Procedural block. Activated whenever A experiences an event
 begin
    if (A[15] == 1'b1) Y = 4'd15;          // If statement. Provides priority to the inputs
    else if (A[14] == 1'b1) Y <= 4'd14;
    else if (A[13] == 1'b1) Y <= 4'd13;
    else if (A[12] == 1'b1) Y <= 4'd12;
    else if (A[11] == 1'b1) Y <= 4'd11;
    else if (A[10] == 1'b1) Y <= 4'd10;
    else if (A[9] == 1'b1)  Y <= 4'd9;
    else if (A[8] == 1'b1)  Y <= 4'd8;
    else if (A[7] == 1'b1)  Y <= 4'd7;
    else if (A[6] == 1'b1)  Y <= 4'd6;
    else if (A[5] == 1'b1)  Y <= 4'd5;
    else if (A[4] == 1'b1)  Y <= 4'd4;
    else if (A[3] == 1'b1)  Y <= 4'd3;
    else if (A[2] == 1'b1)  Y <= 4'd2;
    else if (A[1] == 1'b1)  Y <= 4'd1;
    else Y = 4'd0;                         // Manages all the remaining cases
 end
 endmodule
```

La descrizione precedente utilizza un blocco procedurale per il calcolo dell'uscita. La prima riga del blocco definisce gli eventi che causano una possibile variazione dell'uscita del circuito. Per un circuito combinatorio gli eventi in questione saranno i segnali di ingresso al circuito (il segnale A nel caso mostrato). Per un circuito sequenziale gli eventi possono essere anche delle transizioni sui segnali (fronte di salita o fronte di discesa).
Il corpo del blocco procedurale, mediante una serie di istruzioni *if else* innestate, controlla tutti i bit di ingresso a partire dal bit A[15] e assegna l'uscita corrispondente. L'utilizzo delle *if else* innestate garantisce che il circuito si comporti come un'encoder con priorità in quanto non appena una delle condizioni viene verificata, si assegna l'uscita del circuito e si esce dal blocco procedurale.
E' molto importante definire la lista degli eventi che attivano il blocco procedurale in quanto:

- per alcuni circuiti la lista degli eventi (sensitivity list), determina in gran parte il comportamento finale del circuito. Si pensi ad esempio ai flip-flop nei quali l'evento di fronte di salita sul clock è quello che determina la variazione dell' uscita;
- durante la simulazione funzionale la lista degli eventi determina l'attivazione del blocco. In assenza di lista degli eventi il blocco procedurale viene eseguito di continuo e blocca la simulazione;
- un circuito digitale è intrinsecamente dipendente da modifiche dei segnali di ingresso. Se nessuno dei segnali di ingresso si modifica le uscite non dovrebbero cambiare. Indicare al sintetizzatore quali sono gli eventi che abilitano una possibile modifica dell'uscita significa definire il circuito senza alcuna ambiguità;
- un caso tipico di errore di descrizione è quello di dimenticare alcuni segnali nella sensitivity list di un circuito combinatorio. Questo tipo di errore si evidenzia tipicamente con un comportamento differente tra la simulazione funzionale e la simulazione post-sintesi. Una sensitivity list incompleta non è mai desiderata. Genera un warning durante l'elaborazione e la sintesi del circuito finale ed introduce errori molto difficili da scoprire.

Si prenda ad esempio la seguente descrizione di una porta logica AND, che utilizza un blocco procedurale con una sensitivity list incompleta.

```
module and_sensitivity(A,B,Y);
input A,B;
output Y;
wire A,B;
reg Y;

always @(A)
begin
Y<=A & B;
end
endmodule
```

Il circuito descritto, stando alla sintassi del Verilog, assegna un nuovo valore al segnale *Y* solo se si ha una variazione sul segnale *A*. Volendo essere precisi non si sta quindi descrivendo una porta logica AND.
Infatti, la simulazione funzionale del circuito, che esegue il codice Verilog senza prima sintetizzarlo, riporta il risultato mostrato in Figura 3.4. Si noti coma al quarto vettore di test, con i segnali *A* e *B* entrambi alti, l'uscita *Y* rimanga bassa.

Ciò è dovuto al fatto che si è giunti alla condizione A='1' e B='1' con una transizione su B. Al quinto vettore di test, al contrario, poiché si è giunti alla condizione A='1' e B='1' con una transizione su A, l'uscita si porta alta. E' evidente come questo non sia il comportamento desiderato dal progettista.
Durante la sintesi il sintetizzatore notifica la sensitivity list incompleta con un warning e sintetizza una semplice porta logica AND. La simulazione post-sintesi è mostrata in Figura 3.5 e mostra il corretto funzionamento della porta logica. Evidenziamo quindi due problemi. In primo luogo la simulazione funzionale e quella post-sintesi risultano incongruenti. Ciò rende impossibile la fase di test in quanto non sappiamo, a priori, quale dei due comportamenti sia quello corretto. In secondo luogo, il risultato della sintesi dipende da come il sintetizzatore decide di completare la sensitivity list. Per circuiti complessi il risultato finale è dipendente dal sintetizzatore, con ovvi problemi di portabilità ed affidabilità.

Figura 3.4: Simulazione funzionale di una porta logica AND con sensitivity list che comprende solo il segnale di ingresso A. Si evidenzia un comportamento anomalo, per una porta logica AND, al quarto vettore di test.

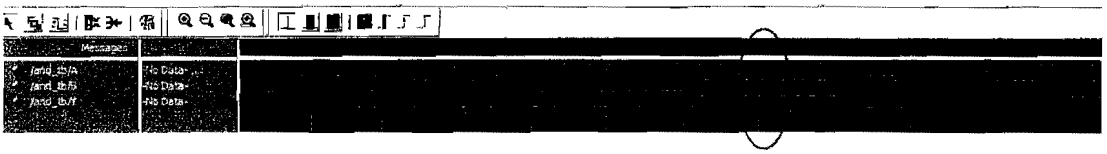


Figura 3.5: Simulazione post-sintesi di una porta logica AND con sensitivity list che comprende solo il segnale di ingresso A. Durante la sintesi la sensitività list è stata artificialmente completata. Il risultato è una porta logica AND convenzionale.

Esempio tipico di circuito che si descrive egregiamente mediante un blocco procedurale è il registro. E' questo il tipico caso in cui è molto importante descrivere precisamente il comportamento del circuito, lasciando al sintetizzatore la scelta sull'elemento sequenziale da utilizzare per riprodurre il comportamento descritto.
Un flip flop, senza segnale di reset, preset o abilitazione è descritto in Verilog come mostrato di seguito. Si noti la funzione 'posedge' che torna il risultato vero quando il segnale considerato ha una transizione basso-alto.

```
module df (d,q,clk);          // Module definition
input d,clk;
output q;
wire d,clk;
reg q;                        // Assigned in a procedural block -> reg type

always @(posedge clk)         // Procedural block. Activated by the positive edge of the clock
begin
q <=d;                        // D flip flop
end
endmodule
```

Il blocco procedurale descrive un circuito il cui comportamento è tale da assegnare a q il valore di d ogni volta che il segnale *clk* ha una transizione basso-alto. Il circuito è quindi un flip-flop di tipo D. Dopo la sintesi il blocco procedurale è sostituito da un flip flop. Durante la fase di simulazione funzionale, il blocco di istruzioni compreso tra *begin* e *end*, viene eseguito sequenzialmente, ad ogni transizione basso-alto del segnale di clock.

4. Test bench

Il test bench è un file in linguaggio HDL, Verilog nel caso considerato, che ha lo scopo di gestire la simulazione di un altro modulo. Il test bench non è la descrizione di un circuito e, nella maggioranza dei casi, utilizza costrutti non sintetizzabili. Da un punto di vista gerarchico bisogna pensare al test bench come un sistema che ingloba il circuito che si desidera testare (spesso indicato con l'acronimo UUT, Unit Under Test), decide quali ingressi applicare e rileva le uscite del circuito. Un possibile schema di un test bench è mostrato in Figura 4.1. Il quadrato esterno rappresenta il *module* del test bench, si noti come il test bench non abbia né ingressi né uscite.
I componenti interni del test bench possono essere vari. In Figura 4.1 con il tratto pieno si indicano i componenti indispensabili alla realizzazione di un test bench. In primo luogo è necessario che il test bench contenga il componente da testare, denominato UUT. E' necessario inoltre il generatore dei segnali di ingresso (*stimuli generation*) da applicare alla UUT. Spesso tale generatore è suddiviso in una sezione che genera il segnale di clock e una sezione che genera gli ingressi (*test vectors*). A valle della simulazione si avrà la risposta della UUT, rappresentata dai segnali denominati *Out* in Figura 4.1, in funzione dei segnali applicati in ingresso, che in Figura 4.1 sono denominati *In*.
Opzionalmente il test bench può contenere altri moduli che gli permettono di eseguire compiti più complessi. Ad esempio:

- gli ingressi da applicare alla UUT possono essere letti da un file (*test vector file*)
- un modulo, denominato *Output analysis* in Figura 4.1, può analizzare il risultato della simulazione per evidenziare eventuali malfunzionamenti. Opzionalmente tale modulo può leggere da un file le uscite attese dalla UUT (*Expected output file*), oppure le uscite possono essere confrontate con una differente descrizione dal circuito da testare (*Reference UUT*).
- i risultati dell'analisi possono essere salvati su di un file (*Simulation results file*) oppure riportate a schermo mediante opportune chiamate di sistema (*Results given as system calls*).

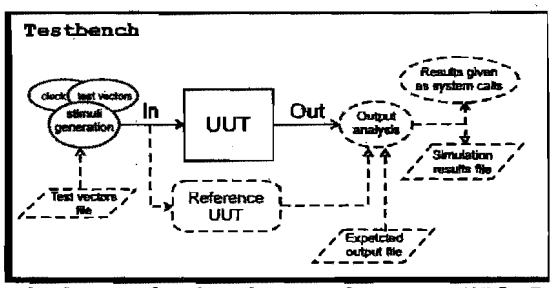


Figura 4.1: Configurazioni tipiche di un test bench realizzato in linguaggio HDL. Con tratto pieno gli elementi che sono indispensabili. Le sezioni tratteggiate indicano elementi e relazioni che non sono indispensabili.

4.1 Test bench per circuiti combinatori

Un test bench che deve testare il comportamento di un circuito combinatorio si differenzia dallo schema di Figura 4.1 per l'assenza del modulo che genera il segnale di clock. Considerando il codificatore con priorità mostrato precedentemente, un possibile test bench è mostrato di seguito.

```
1       `timescale 1ns/1ps              // define simulation time units
2       module prenc83_tb();            // module with no inputs or outputs
3       reg  [7:0] Atb;                 // reg net type. To be assigned in the initial block
4       wire [2:0] Ytb;
5       wire idle;

6       prenc83 d1(.A(Atb),.Y(Ytb)); // UUT named d1

7       initial                         // Just once at the beginning of the simulation
8       begin
9       Atb = 8'h01;                    // Eight bits hexadecimal value
10      #100;                           // 100 time units delay -> 100 ns
11      Atb = 8'h02; #100;
12      Atb = 8'h04; #100;
13      Atb = 8'h08; #100;
14      Atb = 8'h10; #100;
15      Atb = 8'h20; #100;
16      Atb = 8'h40; #100;
17      Atb = 8'h80; #100;
18      Atb = 8'h44; #100;
19      Atb = 8'hF0; #100;
20      Atb = 8'h00; #100;
21      Atb = 8'hCF; #100;
22      $stop;                  // System call that stops the simulation
23      end
24      endmodule
```

Analizziamo le singole istruzioni del test bench mostrato. Si rammenta come il test bench non intende rappresentare un circuito e quindi le istruzioni in esso contenute non sono sintetizzabili e non rappresentano dei componenti circuitali.
La riga 1 definisce le unità temporali da utilizzare durante la fase di test. La direttiva da utilizzare è *timescale*.

`timescale <unit delay>/<time resolution>

Il primo carattere non è un normale apice ma va inserito mediante il corrispettivo codice ASCII (ALT+96). Il primo tempo inserito dopo timescale rappresenta l'entità di un ritardo unitario e viene espresso in secondi o nei suoi sottomultipli. Nel caso mostrato il ritardo unitario è di 1 ns. Il secondo tempo rappresenta la risoluzione temporale da utilizzare durante la simulazione. Nel caso mostrato la risoluzione è fissata ad 1 ps. Ad esempio se il periodo di variazione degli ingressi è 1 ns ed il segnale A viene fatto variare dopo un terzo di periodo, il segnale A varierà dopo 333 ps se la risoluzione è 1 ps. Se la risoluzione fosse 10 ps A varierebbe dopo 330 ps.
La seconda riga definisce il module del test bench con il suo nome. Questo nome è quello da specificare nella finestra *Edit Test Bench Settings* nel campo *Top level module in test bench*, quando, utilizzando il software QuartusII, si specificano le caratteristiche di un test bench (finestra *Settings* | *Simulation* | *Test Benches...* | *Edit...*) come mostrato in Figura 4.2 . Si noti come il test bench sia caratterizzato dal non avere né ingressi né uscite. Non sono dunque neanche presenti i costrutti input ed output che definiscono quali dei segnali di interfaccia del test bench siano di ingresso o di

uscita.
Le righe 3,4 e 5 definiscono i segnali interni utilizzati dal test bench. I segnali saranno i collegamenti con la UUT alla quale applicheranno i vettori di test e dalla quale preleveranno i risultati delle elaborazioni. I segnali di tipo wire definiti alle righe 4 e 5 sono già stati presentati ed è stato chiarito che rappresentano dei nodi elettrici.
Un'attenzione maggiore richiede la riga 3 nella quale il segnale (net) denominata Atb è definita di tipo *reg* (register).
I collegamenti di tipo *reg* in Verilog rappresentano nodi del circuito che memorizzano un valore.
Il tipo *reg* memorizza il valore dell'ultimo assegnamento in un blocco procedurale (blocchi nei quali l'ordine delle istruzioni determina il comportamento). Visto che il tipo *reg* immagazzina un valore mediante assegnamento, esso, come vedremo successivamente, può essere utilizzato per modellare registri hardware, ad esempio flip-flop e latch trasparenti; può anche essere utilizzato per la logica combinatoriale.
Il tipo *reg* è inoltre necessario ogni volta che si desidera un segnale al quale è possibile assegnare un valore logico senza che vi sia un circuito a generarlo, una funzionalità che è tipica dei test bench ed è non sintetizzabile.
Nei test bench descritti in Verilog è necessario definire di tipo reg tutti i segnali di ingresso alla UUT.
La linea 6 istanzia il componente che desideriamo testare, anche detto UUT (Unit Under Test). Si segue la sintassi tipica dell'istanziazione, con il collegamento dei segnali per nome.

<module name> <instance name>(<Interface ports>)

Nell'istruzione si indica per primo il module che si intende testare (UUT), istanziandolo come un componente nel test bench. Di seguito si attribuisce al componente un nome che lo identificherà all'interno del test bench (instance name). Questo nome è quello da specificare nella finestra *Edit Test Bench Settings* nel campo *Design instance name in test bench*, come mostrato in Figura 4.2. Gli altri campi della finestra *Edit Test Bench Settings* sono *Test bench name*, nel quale si identifica il test con un nome, e *File name*, nel quale si indica il nome del file che contiene il testbench.
In seguito, tra parentesi, l'elenco dei segnali di interfaccia del modulo e la corrispondenza con i segnali del circuito che si descrive. La corrispondenza con i segnali di interfaccia è consigliabile effettuarla per nome. Volendo testare il componente encoder con priorità caratterizzato dalla seguente interfaccia:

```
prenc83 (A,Y);
```

una volta definiti i segnali interni al test bench Atb e Ytb, con Atb di tipo *reg* e Ytb di tipo *wire*, il componente da testare è istanziato come mostrato alla riga 6.

```
prenc83 d1(.A(Atb),.Y(Ytb));          // UUT named d1
```

Si rimarca che la connessione per nome richiede che si indichi il nome del port di interfaccia del module preceduto da un punto con, tra parentesi, l'indicazione del segnale da collegare.

.<port_name>(<signal_name>)

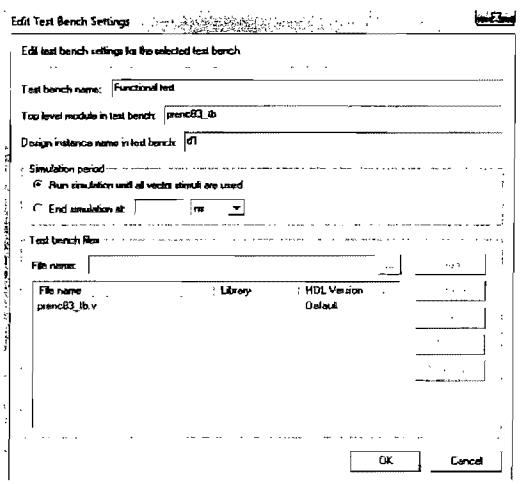


Figura 4.2: Impostazione del testbench in QuartusII. I campi rappresentano nell'ordine: un nome identificativo del test bench; il nome del module di livello gerarchico più alto nel test bench; il nome del UUT istanziato nel test bench; il nome del file che contiene il testbench.

La linea 7 contiene il costrutto *initial*. Si tratta di un costrutto procedurale e non sintetizzabile riservato ai test bench. Tutte le istruzioni contenute tra l'istruzione *begin* della linea 8 e l'istruzione *end* della linea 23 vengono eseguite in sequenza e l'ordine delle stesse influenza il risultato finale. Il costrutto *initial* indica un blocco procedurale che deve essere eseguito una sola volta all'inizio della simulazione del module. E' quindi un costrutto molto utile per i test bench che hanno il compito di applicare, una sola volta, i vettori di test alla UUT.
La linea 9 assegna il valore '0' al segnale vettoriale Atb. E' importante notare che in questo caso l'assegnazione avviene mediante l'operatore '='. Tale assegnazione è detta di tipo 'blocking' e si contrappone all'assegnazione di tipo 'non-blocking' che utilizza l'operatore '<='. Non si entra nei dettagli che differenziano i due tipi di assegnazione. Basti sapere che l'assegnazione 'non blocking' (<=) si utilizza nella descrizione dei circuiti mentre l' assegnazione 'blocking' (=) si utilizza per la definizione dei vettori di test nel test bench.

La sintassi per l'indicazione di un numero è:

<num_bit>'<b | h | d | o><value>

Il primo numero indica il numero di bit che viene seguito da un apice, da una lettera che indica la base ('b' per binaria, 'h' per esadecimale, 'd' per decimale, 'o' per ottale) e dal valore che si vuole assegnare.
La linea 10 mostra un cancelletto, che è il costrutto che indica il ritardo temporale, seguito dal numero di unità di ritardo. Nel caso mostrato, poiché l'istruzione *timescale* ha definito l'unità di tempo pari a 1 ns, l'istruzione causa un ritardo di 100 ns.
Le restanti righe, dalla 11 alla 21, ripetono l'assegnazione di un nuovo valore all'ingresso Atb seguito da un ritardo di 100 ns. Le due istruzioni sono sulla stessa riga. La scelta è dovuta unicamente a criteri di leggibilità in quanto in Verilog le andate a capo e gli spazi sono ininfluenti sulla descrizione circuitale.
La linea 22 contiene l'istruzione *$stop;* che ferma la simulazione. Le istruzioni che iniziano con il '$' sono delle funzioni di sistema utilizzate per visualizzare un messaggio a schermo, aprire chiudere file, o, come in questo caso, per terminare la simulazione. Alcune funzioni di sistema sono utili anche per la descrizione dei circuiti sintetizzabili e verranno presentate in seguito.

Come utile esercizio progettuale si realizzi un progetto per FPGA e si descriva in Verilog un codificatore con priorità ad 8 ingressi e 3 uscite. Lo si simuli utilizzando il test bench mostrato. Si modifichi il test bench inserendo, ad esempio, un ritardo di 50 ns subito dopo l'istruzione *begin* e si verifichino i cambiamenti nella simulazione del circuito. Si analizzi inoltre cosa accade se agli ingressi del circuito sono assegnati valori 'z' o 'x'.

4.2 Test bench per circuiti sequenziali

La simulazione di un circuito sequenziale mediante l'utilizzo di un test bench si distingue dal caso combinatorio per la presenza di segnali dei segnali di clock e reset.
Il metodo più efficiente per gestire il segnale di clock è quello di definire un blocco procedurale che generi la forma d'onda richiesta. Un possibile esempio è mostrato di seguito

```
// Generates the clock waveform
parameter PERIOD=20;
always
begin
CLKtb=0;
#(PERIOD/2);   // Delays by PERIOD/2 time units
CLKtb=1;
#(PERIOD/2);
end
```

Il blocco procedurale mostrato non è abilitato da alcun controllo sugli eventi poiché non presenta l'istruzione @(...). Nel linguaggio Verilog ciò significa che il blocco è attivato di continuo. La prima istruzione assegna '0' al clock, poi si attende un numero di time units pari a PERIOD/2, si assegna '1' al clock e si attende nuovamente mezzo periodo di clock.
Per la restante parte del file di test bench si utilizza nuovamente il costrutto initial, che rappresenta un blocco procedurale eseguito una sola volta all'inizio della simulazione, per applicare i vettori di test al circuito.
La simulazione di un blocco, sequenziale, come dettagliato al capitolo 4, prevede prima di tutto una fase di reset del circuito.
E' buona norma che anche durante la fase di reset tutti i segnali di ingresso abbiano un valore definito e quindi la parte iniziale dei vettori di test deve contenere assegnazioni per i segnali di reset e per tutti i segnali di ingresso.
In seguito si procede a disattivare il segnale di reset, ad ogni modifica di un segnale è conveniente ridefinire tutti i segnali di ingresso.
Dal momento in cui il segnale di reset si disabilita i segnali di ingresso dovrebbero essere gli unici che variano. Conseguentemente si può evitare di riassegnare il segnale di reset quando si assegnano i nuovi valori agli ingressi.
Un esempio di corretta assegnazione dei segnali di ingresso e reset è mostrata di seguito.

```
initial
begin
CLR=1'b0;              // Start with a reset
PSET=1'b1;             // Start with a reset
D=1'b1;                // Assign a value to every input. Do not simulate with unspecified signals.
#115;                  // Delay
PSET=1'b1;             // disable the reset signal
CLR=1'b1;              // disable the reset signal
D=1'b1;                // Also not changing data are assigned, for readability
#PERIOD;               // Wait one clock period
D=1'b0;                // Apply inputs. Reset inputs are now constant. No need to repeat them anymore
#PERIOD; D=1'b1;
#PERIOD; D=1'b1;
end
```

5. Circuiti elementari descritti in Verilog

Un circuito digitale, anche complesso, è sempre composto connettendo un insieme di componenti digitali che sono ben noti e, trascurando le possibili varianti, raggruppabili in poche topologie. E' fondamentale quindi apprendere le modalità di descrizione dei componenti di base che risultino in circuiti non dipendenti dal sintetizzatore, e senza differenze di comportamento tra la simulazione funzionale e quella post sintesi.

Tra i componenti combinatori, ad esempio, troviamo: il multiplexer (MUX), il codificatore ed il decodificatore, la Look Up Table (tabella di verità).
Tra i componenti sequenziali, ad esempio, annoveriamo: il flip flop (con o senza abilitazione e reset), il latch, i registri, il contatore, il registro a scorrimento.

5.1 Multiplexer

Il multiplexer, spesso indicato con la sigla MUX, è un circuito generico che instrada diversi ingressi su di un'unica uscita. Il multiplexer si caratterizza per il numero di ingressi (indicato con n) e per il numero di bit degli ingressi (indicato con p). Il numero di segnali di uscita è generalmente pari ad 1. I bit di selezione sono m. Per poter selezionare tutti gli ingressi è necessario che si abbia $2^m \geq n$. Un multiplexer è completo se $2^m = n$. Se i segnali di ingresso sono composti da p bit anche il segnale di uscita è composto da p bit. In Figura 5.1 si mostra il caso più generale di multiplexer con n ingressi da p bit ed m bit di selezione. E' facile verificare che un multiplexer con n ingressi da p bit ed m bit di selezione si realizza ponendo in parallelo p MUX con n ingressi da 1 bit ed m bit di selezione.

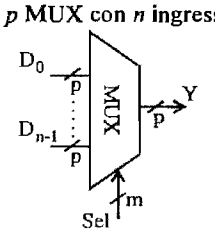
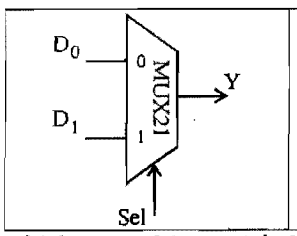


Figura 5.1: Multiplexer con n ingressi da p bit ed m bit di selezione.

MUX2

Il più semplice multiplexer ha due ingressi ad un bit. Lo schema e la tabella di verità del circuito sono mostrati in Figura 5.2. Se l'ingresso di selezione è '0' l'uscita è uguale a D_0, l'ingresso di selezione è '1' l'uscita è uguale a D_1.



		Y	Sel	
			0	1
D1	D0	00	0	0
		01	1	0
		11	1	1
		10	0	1

Figura 5.2: Multiplexer con 2 ingressi da 1 bit. A destra la tabella di verità del circuito.

La descrizione in Verilog riguarderà quindi un circuito con due ingressi ed un'uscita.

```
module mux21(D,Sel,Y)
input  [1:0] D;
input  Sel;
output Y;
```

La descrizione del comportamento interno del circuito si realizza agevolmente in vari modi.
E' possibile utilizzare le equazioni Booleane derivate dalla tabella di verità e l'assegnazione continua:

```
assign Y = Sel & D[1] | (~Sel) & D[0];
```

E' possibile utilizzare l'operatore condizionale, identificato dal punto interrogativo, e l'assegnazione continua. L'operatore condizionale è l'unico operatore ternario del Verilog. Il costrutto da utilizzare è:

```
assign Y = Sel ? D[1] : D[0];
```

L'assegnazione con l'operatore condizionale valuta in primo luogo il segnale *Sel*. Se *Sel* è vero (='1'), ad *Y* si assegna D[1], altrimenti ad *Y* si assegna D[0].
Una terza opzione, probabilmente troppo prolissa per un circuito di tale semplicità, è l'utilizzo di un blocco procedurale. L'uscita del blocco procedurale, *Y*, in quanto assegnata in un blocco procedurale dovrà essere dichiarata di tipo *reg*. La sensitivity list del blocco conterrà sia il segnale *D* sia il segnale *Sel*. Il corpo del blocco procedurale è preferibile definirlo mediante il costrutto *case* ma è accettabile anche il costrutto *if else*.

```
reg Y;
always @(D or Sel)
begin
case (Sel)
       1'b0 : Y<=D[0];
       1'b1 : Y<=D[1];
default:  Y<=1'bx;
endcase
end
```

Il costrutto case è molto utile quando si devono effettuare differenti operazioni basate sul valore di un segnale. L'operazione effettuata dal costrutto case è quella di valutare l'espressione tra parentesi dopo la parola chiave *case* (il segnale *Sel*). Poi si confronta il valore dell'espressione con il valore che contraddistingue la prima condizione (1'b0). Se le due espressioni sono uguali (su tutti i bit) viene eseguita la prima operazione indicata (Y<=D[0]) altrimenti si procede con l'espressione seguente.
L'ordine delle espressioni è importante per la realizzazione finale. Il costrutto case esprime quindi una implicita priorità tra le espressioni in quanto anche se più di una condizione è verificata, si esegue la prima.
Di particolare importanza è l'ultimo ramo del *case* denominato *default*. Esso rappresenta l'azione da eseguire quando nessuno degli altri casi è stato verificato. L'assenza del caso *default* porta a comportamenti inattesi sia durante la simulazione funzionale del circuito sia a valle della fase di sintesi. Tali comportamenti sono dovuti alla presenza di latch che vengono creati nel circuito.

MUXn con ingressi a p bit

Consideriamo ora il caso generico. Supponiamo di voler realizzare un MUX con 9 ingressi da 5 bit. Per questo MUX servono 4 bit di selezione.
Trattandosi di un circuito abbastanza complesso è preferibile evitare le equazioni Booleane e descrivere il circuito mediante un blocco procedurale. La descrizione è la seguente.

```
module mux_9_5(D8, D7, D6, D5, D4, D3, D2, D1, D0, Sel, Y);
input [4:0] D8, D7, D6, D5, D4, D3, D2, D1, D0;
input [3:0] Sel;
output [4:0] Y;

wire [4:0] D8, D7, D6, D5, D4, D3, D2, D1, D0;
wire [3:0] Sel;
reg [4:0] Y;

always @(Sel or D8 or D7 or D6 or D5 or D4 or D3 or D2 or D1 or D0)
begin
case (Sel)
 4'h0 : Y<=D0;
 4'h1 : Y<=D1;
 4'h2 : Y<=D2;
 4'h3 : Y<=D3;
 4'h4 : Y<=D4;
 4'h5 : Y<=D5;
 4'h6 : Y<=D6;
 4'h7 : Y<=D7;
 4'h8 : Y<=D8;
 default: Y<=5'b00000;
endcase
end
endmodule
```

Da evidenziare è la compattezza della descrizione. Il numero di bit dei segnali può essere variato molto semplicemente e la descrizione è compatta e facilmente leggibile. Nella sensitivity list sono inseriti tutti i segnali di ingresso combinati mediante l'operatore OR. In maniera equivalente sarebbe possibile separare i segnali mediante una virgola.
`always @(Sel, D8, D7, D6, D5, D4, D3, D2, D1, D0)`
Il multiplexer mostrato è incompleto in quanto con 4 bit di selezione sarebbe possibile selezionare 16 segnali di ingresso. Ne consegue che il costrutto case elenca solo 9 dei 16 valori possibili (supponendo che i bit possano assumere solo i valori '0' e '1'). E' questo il caso in cui la parola chiave *default* risulta determinante in quanto definisce il comportamento in tutti gli altri casi non elencati.
La simulazione deve essere eseguita utilizzando un testbench. Di seguito un possibile listato.

```
`timescale 1ns/1ps
module mux_95_tb();
wire [4:0] Y;
reg [3:0] Sel;
reg [4:0] D8, D7, D6, D5, D4, D3, D2, D1, D0;

mux_95 UUT(.D8(D8),.D7(D7),.D6(D6),.D5(D5),.D4(D4),.D3(D3),.D2(D2),.D1(D1),.D0(D0),.Sel(Sel),.Y(Y));
initial
```

```
begin
D8=5'b10000;D7=5'b01000;D6=5'b00100;                    // Assign the D[i] inputs
D5=5'b00010;D4=5'b00001;D3=5'b10001;
D2=5'b01001;D1=5'b00000;D0=5'b10101;
Sel=4'b0000; #100;                                      // Assign and vary the Sel signal
Sel=4'b0001; #100;
Sel=4'b0010; #100;
Sel=4'b0011; #100;
Sel=4'b0100; #100;
Sel=4'b0101; #100;
Sel=4'b0110; #100;
Sel=4'b0111; #100;
Sel=4'b1000; #100;
Sel=4'b1001; #100;
Sel=4'b000x; #100;
Sel=4'bzzx0; #100;
$stop;
end
endmodule
```

Il risultato della simulazione funzionale è mostrato in Figura 5.3 mentre la simulazione post sintesi è mostrata in Figura 5.4.

Il comportamento del mux è congruente con le aspettative. Si nota una differenza per gli ultimi due ingressi del segnale *Sel* che sono rispettivamente '000x' e 'zzx0'. Se ci riferiamo alla simulazione funzionale, che esegue direttamente il file Verilog utilizzato come descrizione, notiamo che gli ultimi due casi ricadono nella sezione default e quindi devono fornire uscita '00000', coerentemente a quanto si osserva nella simulazione funzionale.

La simulazione post sintesi simula invece un circuito digitale realizzato con porte logiche ed elementi sequenziali. In tale caso il comportamento del circuito mostra un risultato differente in quanto una porta logica NOT, ad esempio, non può essere forzata a dare un uscita '0' quando in ingresso ha un segnale sconosciuto 'x' o di alta impedenza 'z'. E' probabile che la simulazione di una porta logica NOT con gli ingressi indicati fornisca in uscita il segnale 'x'. Notiamo quindi che in presenza di segnali 'x' o 'z' su Sel, in uscita avremo una combinazione di segnali 'x' o 'z'.

Da questo esempio impariamo che il comportamento dei circuiti sintetizzati non può essere forzato a valori noti quando gli ingressi sono unknown (sconosciuti, 'x').

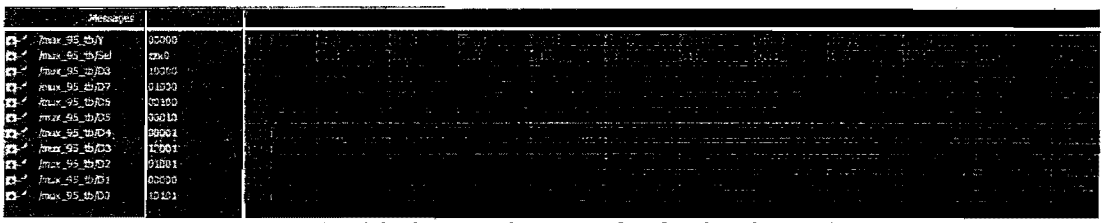


Figura 5.3: Multiplexer con 9 ingressi da 5 bit. Simulazione RTL.

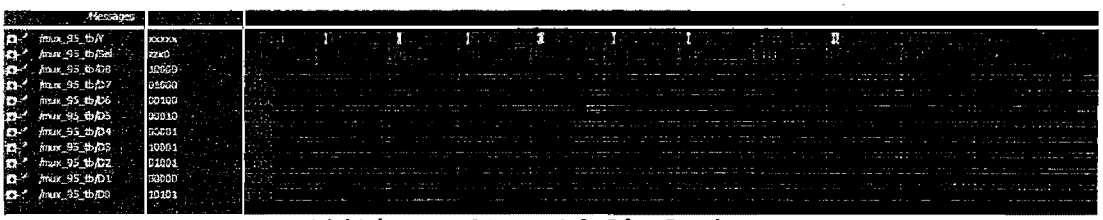


Figura 5.4: Multiplexer con 9 ingressi da 5 bit. Simulazione post-sintesi.

5.2 Decodificatore BCD – don't care.

Un decodificatore di comune utilizzo è il decodificatore per display a sette segmenti. Il circuito a 4 bit di ingresso e 7 uscite che sono utilizzate per accendere i 7 segmenti del relativo display. Nel prosieguo si considerino i segmenti del display numerati come mostrato in Figura 5.5 e attivi bassi (si accendono quando il livello logico applicato è '0').

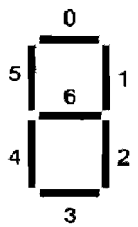


Figura 5.5: Numerazione dei led del display a sette segmenti utilizzato nelle esercitazioni.

Un display a 7 segmenti può visualizzare le sedici cifre della numerazione esadecimale ma è spesso utilizzato per visualizzare le sole 10 cifre della numerazione decimale (BCD – Binary Coded Decimal). Il circuito è di tipo combinatorio ed è quindi rappresentabile mediante la tabella di verità con 4 ingressi e 7 uscite mostrata in Tab. 5.1.

	A3	A2	A1	A0	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	1							
2	0	0	1	0							
3	0	0	1	1							
4	0	1	0	0							
5	0	1	0	1							
6	0	1	1	0							
7	0	1	1	1							
8	1	0	0	0							
9	1	0	0	1							

Tabella 5.1: Numerazione dei led del display a sette segmenti utilizzato nelle esercitazioni.

La funzione Booleana è anche rappresentabile mediante 7 tabelle di verità con 4 ingressi ed un'uscita. In Tab. 5.2 e Tab. 5.3 si mostrano le tabelle di verità relative alle uscite Y0 ed Y4.

Y0	A1 A0 / A2 A3	00	01	11	10
	00	0	1	0	0
	01	1	0	0	0
	11				
	10	0	0		

Tabella 5.2: Tabella di verità per la funzione Booleana dell'uscita Y0.

Y4	A1 A0 / A2 A3	00	01	11	10
	00	0	1	1	0
	01	1	1	1	0
	11				
	10	0	1		

Tabella 5.3: Tabella di verità per la funzione Booleana dell'uscita Y4.

Le tabelle di verità evidenziano che oltre ad avere delle uscite definite in corrispondenza della dieci combinazioni di ingresso considerate, esistono altre sei combinazioni di ingresso che non sono specificate.
Se si definisce il circuito mediante le funzioni booleane, si definiscono implicitamente le uscite anche nei casi non specificati. Se invece si descrive il circuito mediante un blocco procedurale ed il costrutto *case* si richiede una particolare attenzione nel definire il comportamento atteso nei casi non specificati.

Se non si esplicita il comportamento del circuito nei casi non specificati, si ottiene un circuito con dei latch che incrementa inutilmente le risorse utilizzate ed ha un comportamento scorretto (mantiene l'uscita all'ultimo valore valido in ingresso).

E' quindi imperativo utilizzare il costrutto *default* per definire l'uscita in tutti i casi non specificati. Una possibile scelta è quella di fissare l'uscita ad un valore noto (es: `default : 7'b0000000;`).
Il circuito risultante è funzionante, non contiene latch ed ha anche un'uscita definita per ogni possibile ingresso. E' la scelta più sicura ed affidabile sicuramente da utilizzare se le prestazioni del circuito non necessitano di ottimizzazioni spinte.
Se invece desideriamo ottimizzare al massimo il circuito finale, è possibile considerare gli ingressi che non dovrebbero mai avvenire e che non sono utili per la nostra applicazione come ingressi 'non rilevanti' (termine ottenuto traducendo liberamente il termine inglese, e largamente utilizzato, don't care).

Comunemente la notazione utilizzata per indicare il valore logico 'dont'care' è il simbolo '-' che però non è disponibile nel linguaggio Verilog. Se si utilizza il linguaggio Verilog il valore don't care si indica con il valore logico 'x'.
Quando l'uscita della tabella di verità è don't care, il sintetizzatore è spinto a scegliere l'uscita che ottimizza le prestazioni del circuito finale. Consideriamo ad esempio la tabella di verità per l'uscita Y4 con le condizioni don't care mostrata in Tab. 5.4. Volendo implementare la funzione Booleana per i mintermini che portano l'uscita ad '1' (ON set), la tabella viene realizzata mediante 3 termini prodotto che utilizzano 3 o 2 variabili di ingresso.
Se le uscite don't care sono impostate ai valori mostrati in Tab. 5.5, si realizza la stessa funzione utilizzando due soli mintermini, uno composto dal solo ingresso A0 ed il secondo composto da (~A1)&A3.

Y4dc	A1 A2 \ A0 A3	00	01	11	10
	00	0	1	1	0
	01	1	1	1	0
	11	-	-	-	-
	10	0	1	-	-

Tabella 5.4: Tabella di verità e copertura minima per l'uscita Y4. Si trascurano i don't care.

Y4opt	A1 A2 \ A0 A3	00	01	11	10
	00	0	1	1	0
	01	1	1	1	0
	11	1	1	1	0
	10	0	1	1	0

Tabella 5.5: Tabella di verità e copertura minima per l'uscita Y4 tenendo conto dei don't care.

La descrizione del decoder per display BCD in linguaggio Verilog che definisce come condizioni di don't care tutti gli ingressi non specificati è la seguente.

```
module BCD_decoder (A,Y);
input [3:0] A;          // 4 input bits
output [6:0] Y;         // Active low 7 segment display
wire [3:0] A;
reg [6:0] Y;            //Y0 is the top segment. Y1..Y5 proceed clockwise. Y6 is the middle segment.

always @(A)
begin
case (A)
 4'b0000 : Y<=7'b1000000;     // 0
 4'b0001 : Y<=7'b1111001;     // 1
 4'b0010 : Y<=7'b0100100;     // 2
 4'b0011 : Y<=7'b0110000;     // 3
 4'b0100 : Y<=7'b0011001;     // 4
 4'b0101 : Y<=7'b0010010;     // 5
 4'b0110 : Y<=7'b0000010;     // 6
 4'b0111 : Y<=7'b1011000;     // 7
 4'b1000 : Y<=7'b0000000;     // 8
 4'b1001 : Y<=7'b0010000;     // 9
 default : Y<=7'bxxxxxxx;     // don't care
endcase
end
endmodule
```

Il decodificatore, se implementato su CPLD MAX7000AE utilizza 7 macrocelle, una per ogni uscita, e 16 termini prodotto. Se si eliminano i don't care e si imposta l'uscita a 0 in tutti i casi non specificati, il numero di termini prodotto utilizzati aumenta sino a 19.
Se l'implementazione è effettuata per FPGA CycloneII non vi sono differenze di occupazione di area tra le due descrizioni che sono realizzate mediante 7 Logic Elements. Ciò non deve meravigliare se si ricorda la struttura di un LE delle FPGA CycloneII. I LE sono composti da una LUT a 4 ingressi e quindi le 7 tabelle di verità con 4 ingressi ed una uscita sono realizzate con 7 LE e non è possibile ridurre l'utilizzo di hardware programmabile al di sotto di questo limite.
La simulazione funzionale e post sintesi del decoder BCD, nei tre casi considerati in precedenza è mostrata in Fig.5.6.
La simulazione indica: con Y l'uscita del BCD decoder con l'istruzione `default:7'bxxxxxxx;` con Y_zero l'uscita del BCD decoder con l'istruzione `default:7'b0000000;` con Y_latch l'uscita del BCD decoder realizzato, erroneamente, omettendo l'istruzione default.
La simulazione funzionale dei tre circuiti considerati, in alto in Figura 5.6, mostra che per gli ingressi A=0,1,2,...,9 le uscite sono identiche. Per gli ingressi A=10,11,...,15 l'uscita Y è unknown, l'uscita Y_zero è '0' mentre l'uscita Y_latch mantiene l'ultimo valore valido di uscita (infatti per A=15 fornisce una volta Y_latch = 7'h10 e la seguente Y_latch = 7'h02). La simulazione evidenzia quindi la presenza dei latch nel caso in cui si ometta l'istruzione default.
La simulazione post sintesi non mostra differenze nei confronti della simulazione funzionale se si considerano le uscite Y_zero e Y_latch. La differenza è visibile per l'uscita Y che assume valori definiti. Avendo analizzato i risultati della sintesi è invece logico affermare che tali uscite sono state scelte dal sintetizzatore al fine di minimizzare l'utilizzo di logica nel circuito sintetizzato.

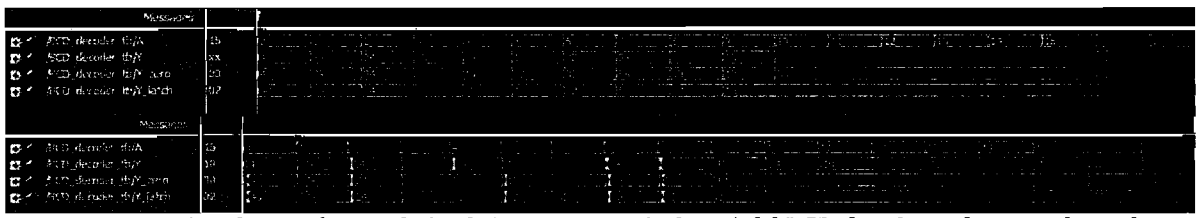


Figura 5.6: Simulazione funzionale (in alto) e post sintesi (in basso) del BCD decoder realizzato utilizzando: l'istruzione `default:7'bxxxxxxx` *per l'uscita Y; l'istruzione* `default:7'b0000000` *per l'uscita Y_zero;e nessuna istruzione default per l'uscita Y_latch.*

5.3 Codificatore (encoder)

Il codificatore ha n bit di ingresso in uscita ha $\lceil \log_2(n) \rceil$ bit. I dettagli dell'encoder sono stati spiegati al capitolo 2.

I codificatori possono essere del tipo con priorità tra gli ingressi. Nel codificatore con priorità i bit di ingresso vengono valutati in un determinato ordine. L'uscita rappresenta il numero binario corrispondente al primo bit che è pari ad '1'.
Nel codificatore senza priorità si suppone che di tutti gli ingressi solo uno sia pari ad '1'. L'uscita rappresenta il numero binario corrispondente al bit che è pari ad '1'.
Un codificatore senza priorità si realizza egregiamente con un costrutto case ed un blocco procedurale. Si consideri ad esempio un codificatore con 8 bit di ingresso e 3 bit di uscita. Le descrizione Verilog è data di seguito.

```
module enc83(A,Y,idle);
input  [7:0] A;        // declaration section
output [2:0] Y;
output idle;

wire [7:0] A;
wire idle;
reg [2:0] Y;

always @(A)            // description section
begin
case (A)
 8'b10000000 : Y<=7;
 8'b01000000 : Y<=6;
 8'b00100000 : Y<=5;
 8'b00010000 : Y<=4;
 8'b00001000 : Y<=3;
 8'b00000100 : Y<=2;
 8'b00000010 : Y<=1;
 8'b00000001 : Y<=0;
default: Y<=0;
endcase
end
assign idle =  (A==8'b00000000) ? 1'b1 : 1'b0;            // Conditional assignement
endmodule
```

L'ordine tra le assegnazioni per il costrutto *case* non è importante in questo caso poiché le singole condizioni sono mutuamente esclusive. Il costrutto default indica il valore dell'uscita per tutti gli ingressi non ammessi. Si noti l'assegnazione del segnale *idle* che utilizza l'assegnazione continua e l'operatore condizionale. In Figura 5.7 si mostra la simulazione funzionale e post sintesi del codificatore. Il comportamento è corretto. Il codificatore senza priorità è implementato utilizzando 10 Logic Elements di una FPGA di tipo CycloneII.

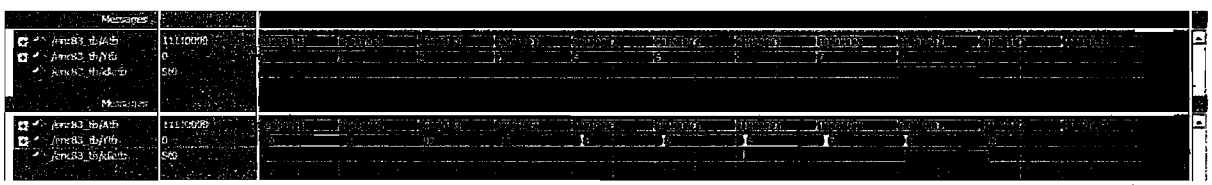


Figura 5.7: Encoder senza priorità descritto mediante il costrutto case. In alto la simulazione funzionale ed in basso la simulazione post sintesi.

Un codificatore con priorità si realizza, ad esempio, mediante l'assegnazione continua e le funzioni booleane, come mostrato al paragrafo 3.5 oppure utilizzando dei costrutti *if else* innestati ed un blocco procedurale, come mostrato al paragrafo 3.10.
La descrizione può essere realizzata anche con un costrutto *casex* ed un blocco procedurale.
In un costrutto *casex* le espressioni da valutare possono contenere anche i valori 'x' e 'z'. Tali valori sono considerati come 'don't care' indicando che al posto di 'x' o 'z' si può sostituire sia '1' sia '0'.

In un costrutto *casex* l'espressione

```
casex (A)
4'b1xxx : <expression1>
```

indica che l'istruzione *<expression1>* deve essere eseguita se il bit più significativo di A è '1'. Gli atri bit, indicati come 'x', non interessano alla valutazione (e sono quindi considerati 'don't care'). Se non avessimo utilizzato il costrutto *casex*, il circuito descritto darebbe in uscita *<expression1>* solo quando l'ingresso è esattamente '1xxx'. Ciò può avvenire solo in una simulazione funzionale e il circuito descritto sarebbe non sintetizzabile.
Si consideri ad esempio un codificatore con 8 bit di ingresso e 3 bit di uscita. La descrizione è la seguente.

```
module encpri83(A,Y,Idle);
input  [7:0] A;        //declaration section
output [2:0] Y;
output idle;
wire [7:0] A;
wire idle;
reg  [2:0] Y;

always @(A)            // description section
begin
casex (A)
 8'b1xxxxxxx : Y<=7;
 8'b01xxxxxx : Y<=6;
 8'b001xxxxx : Y<=5;
 8'b0001xxxx : Y<=4;
 8'b00001xxx : Y<=3;
 8'b000001xx : Y<=2;
 8'b0000001x : Y<=1;
 8'b00000001 : Y<=0;
default: Y<=0;
endcase
end
assign idle = ~(A[7]|A[6]|A[5]|A[4]|A[3]|A[2]|A[1]|A[0]);
endmodule
```

L'istruzione *default* indica il comportamento atteso quando *A* assume uno dei valori non considerati a monte dell'istruzione default. Analizzando il listato ricaviamo che i valori non considerati sono A='00000000', A='0000000x' e A='0000000z'. Il primo valore non considerato indica l'uscita assegnata nella condizione di ingesso nullo, mentre gli ultimi due valori sono significativi solo per la simulazione funzionale. L'uscita *idle* è assegnata mediante il costrutto di assegnazione continua. La Figura 5.8 mostra le simulazioni del circuito, congruenti con il comportamento di un codificatore con priorità.

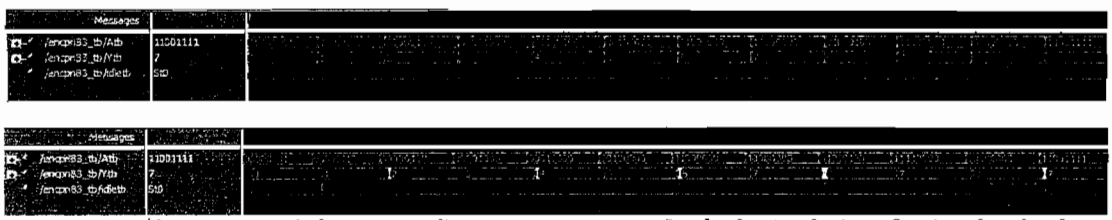


Figura 5.8: Encoder con priorità descritto mediante costrutto casex. In alto la simulazione funzionale ed in basso la simulazione post sintesi.

5.4 La tabella di verità o Look Up Table.

Se la funzione Booleana da implementare non è semplice da descrivere, è spesso preferibile definirla mediante la tabella di verità. Lo stesso accade se si vogliono implementare funzioni non lineari, con pochi bit di risoluzione. Invece di ricorrere alla realizzazione di complessi circuiti aritmetici che approssimino la funzione non lineare, è possibile calcolare in anticipo le uscite attese per ogni ingresso e memorizzare in una Look Up Table (LUT), o una ROM il risultato.
Il circuito risultante è un circuito combinatorio che preleva il valore memorizzato dalla LUT e lo presenta in uscita. Le tabelle di verità, se molto complesse, possono essere più efficacemente realizzate utilizzando una memoria. Questa possibilità non va scartata ed è molto utile se si progettano circuiti da implementare su FPGA moderne che contengono blocchi di memoria molto utili esattamente a questo scopo.

Tabella di verità con uscita a singolo bit.
La tabella di verità per una funzione booleana con una singola uscita si rappresenta facilmente in forma tabellare ma può essere presentata in modo più compatto se si numerano i mintermini e si indica il cosiddetto ON set.
Consideriamo ad esempio una funzione con 5 bit di ingresso $A_4,...,A_0$ ed uscita Y. Possiamo numerare i mintermini

indicando con m0 il mintermine '00000', m1 il mintermine '00001', m16 il mintermine '10000' e con m31 il mintermine '11111'. L'insieme dei mintermini per i quali l'uscita è '1' e definito ON set. La AND tra i 5 ingressi è definita da ON set pari a {m31}. La XOR tra i 5 ingressi è definita da ON set pari a {m1,m2,m4,m7,m8,m11,m13,m14,m16,m19,m21,m22,m25,m26,m28,m31}.
In alternativa è possibile definire, lo OFF set come l'insieme dei mintermini in cui la funzione assume valore '0'. Ad esempio, per una funzione OR a 5 ingressi lo OFF set è {m0} mentre lo ON set è {m1,m2,...,m31}.

Una tabella di verità così descritta si realizza mediante un blocco procedurale ed il costrutto *case*. Di seguito un esempio che mostra la funzione XOR tra 5 ingressi considerata in precedenza.

```
module truth_table (A,Y);
input [4:0] A;
output Y;
wire [4:0] A;
reg Y;

always @(A)
begin
case (A)
1,2,4,7,8,11,13,14,16,19,21,22,25,26,28,31 : Y<=1'b1;
 default : Y<=1'b0;
endcase
end
endmodule
```

Se la funzione da implementare ha più di una uscita ($Y_{n-1},...Y_0$) si può scegliere se descrivere *n* tabelle di verità a singola uscita od utilizzare un costrutto case che definisce tutti i casi possibili in ingresso.
Di seguito un esempio di tabella di verità con 5 bit di ingresso e tre di uscita.

```
module truth_table (A,Y);
input [4:0] A;
output [2:0] Y;
wire [4:0] A;
reg [2:0] Y;

always @(A)
begin
case (A)
1: Y<=3'b010;
2: Y<=3'b100;
        .
        .
        .
31: Y<=3'b110;
 default : Y<=3'b000;
endcase
end
endmodule
```

5.5 Flip flop

I flip flop sono elementi indispensabili per la realizzazione di circuiti digitali. Per quanto semplici nel loro comportamento è molto importante che la descrizione HDL segua precise regole stilistiche che aiutano il sintetizzatore a individuare i flip-flop senza ambiguità.
Un esempio di descrizione Verilog per un flip flop di tipo D senza reset, preset ed enable è stata presentata al paragrafo 3.10 e viene ripetuta per facilità di lettura in basso.

```
module df (d,q,clk);          // Module definition
input d,clk;
output q;
wire d,clk;
reg q;                        // Assigned in a procedural block -> reg type

always @(posedge clk)         // Procedural block. Activated by the positive edge of the clock
begin
q <=d;                        // D flip flop
end
endmodule
```

In primo luogo il flip-flop è descritto mediante un blocco procedurale. Altre modalità di descrizione, per quanto valide in linea di principio sono sconsigliate in quanto potrebbero indurre in errore il software per la sintesi circuitale.
La sensitivity list è incompleta in quanto non è incluso il segnale *d* il cui valore è assegnato al segnale *q*. Tale scelta riflette la caratteristica del flip flop di agire sull'uscita (q) solo in presenza di un fronte di salita sul segnale di clock

(posedge clk).
E' da evidenziare il tipo di assegnazione utilizzata per modificare il segnale di uscita. L'assegnazione è di tipo 'non blocking' che corrisponde all'operatore <=.
Per quanto in queste note sia già stato consigliato di utilizzare l'assegnazione 'non blocking' per descrivere circuiti sintetizzabili mediante costrutti procedurali, è il caso di rimarcare il concetto in questo paragrafo.

Circuiti sequenziali descritti mediante l'assegnazione 'blocking' (=) risultano in comportamenti inattesi sia in fase di simulazione, sia in fase di sintesi e sono uno degli errori più comuni per i principianti.

La sintesi del circuito mostrato è esattamente ciò che ci si attende, vale a dire un flip flop di tipo D come verificabile dalla vista RTL del circuito sintetizzato visibile in Figura 5.9 A).
Un flip flop senza reset è un circuito che dovrebbe essere utilizzato solo per particolari applicazioni nelle quali si occupa di memorizzare dati temporanei generati da altri circuiti (i flip flop di una pipeline sono il caso tipico).
Nella maggioranza dei casi il flip flop necessita di un segnale di reset, o di preset che permetta all'utilizzatore di portare il circuito in uno stato noto prima di utilizzarlo.
I flip flop con reset si descrivono differentemente se sono con reset sincrono o con reset asincrono.
Se il reset è sincrono, il flip flop si azzera se il segnale di reset è attivo (attivo basso o attivo alto) ma solo in corrispondenza del fronte del clock. Se il reset è asincrono, il flip flop si azzera ogni volta che il segnale di reset si attiva, indipendentemente dal fronte del clock.
La descrizione di un flip flop con reset sincrono attivo alto è la seguente.

```
// positive edge triggered FF with synchronous reset
always @(posedge clk)
begin
        if (reset)
                q<=1'b0;
        else
                q<=d;
end
```

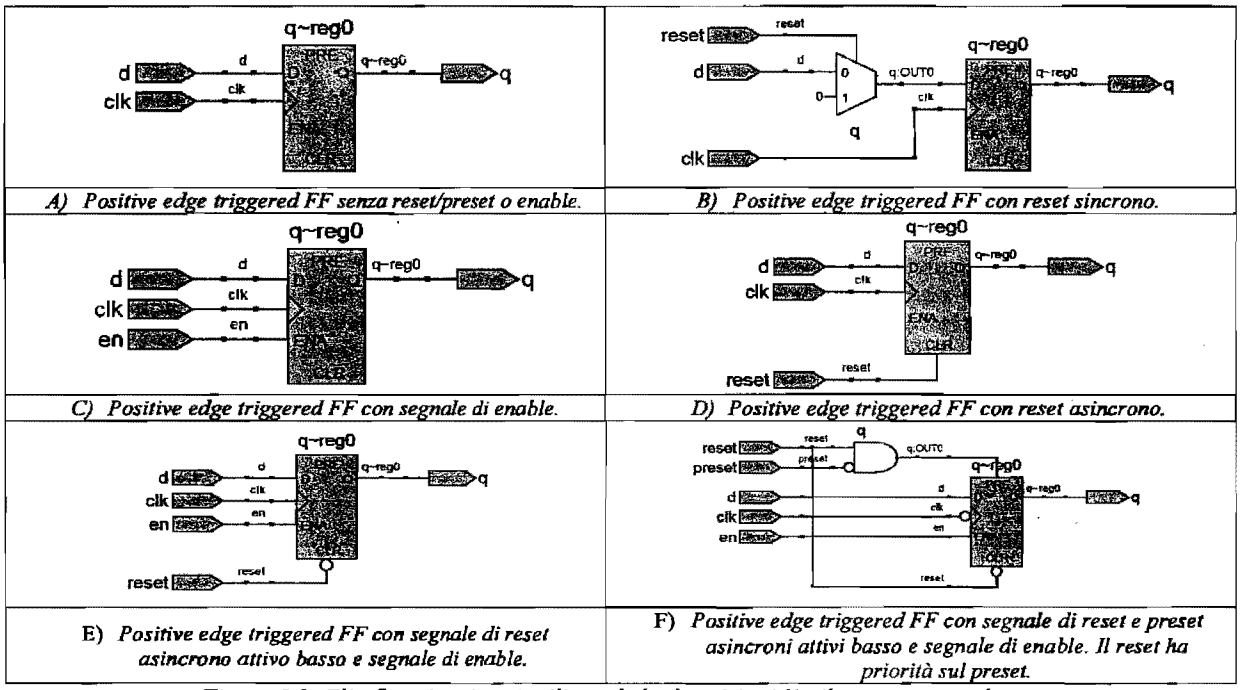
Ancora una volta la sensitivity list è composta dal solo fronte di salita sul segnale di clock. Nella descrizione del flip flop il costrutto *if else* è utilizzato per azzerare l'uscita se il segnale di reset è alto.
Descrizioni di flip flop attivi sul fronte di discesa del clock sono realizzabili utilizzando la funzione *negedge* nella sensitivity list. La sensitivity list risulta la seguente:

```
always @(negedge clk)
```

Con modifiche ovvie si ottiene un flip flop con preset sincrono o flip flop con reset o preset attivi bassi.
Un caso particolare è quello del flip flop con abilitazione (enable).
Un flip flop abilitato presenta un ingresso aggiuntivo, spesso indicato con EN che se attivo abilita il funzionamento normale del flip flop, se disattivato il flip flop memorizza l'ultimo stato e non è più sensibile ai segnali di clock e, se presente, al reset. Un flip flop abilitato può essere descritto nel seguente modo.

```
// positive edge triggered FF with enable
always @(posedge clk)
begin
        if (en)
                q<=d;
end
```

Il flip flop è sensibile al fronte di salita del clock ma, solo se il segnale di enable (en) è alto memorizza sull'uscita il segnale di ingresso. Da evidenziare come nel blocco procedurale non sia specificato cosa avviene nel caso in cui il segnale di abilitazione sia basso. Da un punto di vista della simulazione la descrizione mostrata, in presenza di un fronte di salita del clock e segnale di abilitazione basso, mantiene tutti i segnali invariati e quindi il flip flop è in modalità di memorizzazione. Il comportamento potrebbe anche essere esplicitato specificando `q<=q;` come azione corrispondente al segnale di abilitazione basso. Il circuito risultante non cambia.
La Figura 5.9, nelle sue sezioni A), B) e C), mostra le viste RTL corrispondenti alle descrizioni mostrate di flip flop di tipo D (il cui acronimo è DFF), DFF con reset sincroni, DFF con abilitazione.



A) Positive edge triggered FF senza reset/preset o enable.

B) Positive edge triggered FF con reset sincrono.

C) Positive edge triggered FF con segnale di enable.

D) Positive edge triggered FF con reset asincrono.

E) *Positive edge triggered FF con segnale di reset asincrono attivo basso e segnale di enable.*

F) *Positive edge triggered FF con segnale di reset e preset asincroni attivi basso e segnale di enable. Il reset ha priorità sul preset.*

Figura 5.9: Flip flop sintetizzati utilizzando le descrizioni Verilog mostrate nel testo.

La realizzazione di flip flop che rispondono asincronamente al segnale di reset o di preset deve seguire alcune regole precise. Di seguito si mostra la descrizione Verilog per un flip flop D con reset asincrono attivo alto.

```
always @( posedge reset or posedge clk)
begin
        if (reset==1'b1)
                q<=1'b0;
        else
                q<=d;
end
```

Notate come la sensitivity list includa il fronte di salita del segnale di reset. Per quanto ciò possa sembrare incongruente con la definizione di una risposta asincrona, tentare di descrivere il flip flop differentemente causa errori o malfunzionamenti. Se, ad esempio, si modifica la descrizione di evidenziando la differenza tra il segnale di clock, il cui fronte causa il campionamento del segnale di ingresso, ed il segnale di reset, del quale ci interessa solo il valore e non il fronte si ottiene la seguente, errata, descrizione del flip flop con reset asincrono.

```
// WRONG DESCRIPTION for FF positive edge async reset
always @(reset or posedge clk)
begin
        if (reset)
                q<=1'b0;
        else
                q<=d;
end
```

La descrizione è funzionalmente corretta in presenza di un fronte basso alto del segnale di clock e quando il reset si porta al valore logico '1'. Se invece il reset ha una transizione che lo porta al valore logico '0' il blocco procedurale si attiva e valuta la condizione sul reset relativa al costrutto *if*. Poiché il reset è basso l'uscita del flip flop non viene azzerata, si esegue l'istruzione relativa al ramo else, e si pone $q<=d$.
La descrizione risultante è quella di un flip flop con reset asincrono, sincronizzato sia sul fronte di salita del clock sia sul fronte di discesa del reset.
L'esempio mostrato è indicativo di cosa accade se la descrizione dei circuiti non segue precise regole stilistiche e se non si ha un'idea ben precisa del circuito che stiamo descrivendo.
Si noti inoltre che, poiché il sintetizzatore non dispone di circuiti sequenziali sincronizzati sui fronti di due diversi segnali, il tentativo di sintetizzare la descrizione mostrata riporta l'errore:

```
mixed single- and double-edge expressions are not supported
```

Lo stesso errore si presenta se si tenta di realizzare un flip flop sincronizzato su entrambi i fronti del clock (double edge triggered) con sensitivity list *always @(negedge clk or posedge clk)*. Il sintetizzatore, in presenza di tale sensitivity list riporta l'errore:

```
Event control cannot test for both positive and negative edges of variable "clk".
```

La descrizione di un flip flop sincronizzato sul fronte di salita del clock, con segnale di abilitazione e reset asincrono attivo basso, permette di approfondire le descrizioni Verilog di circuiti sequenziali.
La descrizione è mostrata di seguito.

```
module FF (d,clk,reset,en,q);
input d,clk,reset,en;
output q;
wire d,clk,reset,en;
reg q;

// FF positive edge triggered D FF with active low asynchronous reset and active high enable
always @( negedge reset or posedge clk)
begin
        if (reset==1'b0)
                q<=1'b0;
        else
                if (en)
                        q<=d;
end
endmodule
```

La sensitivity list rileva il segnale di reset ed il segnale di clock poiché il flip flop deve modificare l'uscita solo in presenza del fronte del clock od in presenza di un segnale di reset che si porta basso. Il primo costrutto *if* del blocco procedurale discrimina in quale delle due condizioni ci si trova. Se il reset è '0' allora si azzera l'uscita del flip flop.
Il costrutto else descrive cosa deve accedere se il segnale di reset non vale '0'. Siamo quindi nel secondo caso corrispondente al fronte di salita del clock. In questo caso, se il segnale di abilitazione (en) è alto, si esegue la memorizzazione dell'ingresso (q<=d;) altrimenti l'uscita del flip flop non cambia.

Una versione ancora più complessa di flip flop, che probabilmente è di scarsa utilità nelle applicazioni pratiche, potrebbe utilizzare i segnali di reset, preset e di abilitazione. Per il segnale di abilitazione si è scelto che sia attivo basso. Una possibile descrizione è la seguente.

```
module FF (d,clk,reset,preset,en,q);
input d,clk,reset,preset,en;
output q;
wire d,clk,reset,preset,en;
reg q;

// FF negative edge triggered D FF with active low asynchronous reset/preset and active low enable
always @(negedge reset or negedge preset or negedge clk)
begin
        if (reset==1'b0)
                q<=1'b0;
        else
        if (preset==1'b0)
                q<=1'b1;
        else
                if (en)
                        q<=d;
end
endmodule
```

La sensitivity list rileva il segnale di reset, il segnale di preset, ed il segnale di clock. La descrizione nel blocco procedurale privilegia il reset nei confronti del preset ed un doppio costrutto if innestato viene utilizzato per discriminare le condizioni di reset, preset e di fronte attivo del clock.
La Figura 5.9, nelle sue sezioni D), E) e F), mostra le viste RTL corrispondenti alle descrizioni mostrate di: DFF con reset asincrono; DFF con abilitazione e reset asincrono attivo basso; DFF con abilitazione, reset e preset asincroni attivi basso.

5.6 Latch

Il latch è un elemento di memorizzazione sensibile al livello del segnale di clock. Molto studiato da un punto di vista circuitale in quanto componente base dei flip flop, è relativamente poco utilizzato per la descrizione di circuiti digitali sincroni.
La descrizione del latch è simile a quella di un circuito combinatorio. Utilizza un blocco procedurale con una sensitivity list che include sia i dati di ingresso sia il segnale di clock.
La caratteristica che permette la realizzazione di un latch è un costrutto *case*, o un costrutto *if*, nei quali non è specificato il valore da assegnare all'uscita per tutti i possibili ingressi. Nei casi non specificati l'uscita rimane quindi al valore precedente, e tale descrizione funzionale risulta, a valle della sintesi, nell'implementazione del latch.
Si consideri ad esempio un latch di tipo D sensibile al livello alto del clock. Una possibile descrizione è la seguente.

```
module ltch (d,q,clk);
input d,clk;
output q;
wire d,clk;
reg q;

always @(clk or d)
begin
if (clk)
        q <=d;
end
endmodule
```

Da evidenziare il costrutto *if* senza il corrispettivo *else* che caratterizza la definizione di un latch. Una descrizione equivalente, che esplicita il comportamento in fase di memorizzazione è la seguente.

```
module ltch (d,q,clk);
input d,clk;
output q;
wire d,clk;
reg q;

always @(clk or d)
begin
if (clk)
        q <=d;
else
        q <=q;
end
endmodule
```

La simulazione RTL del latch è mostrata in Figura 5.10 mentre in Figura 5.11 si mostra il risultato della sintesi che, come atteso, consiste in un latch.

Figura 5.10: Simulazione funzionale di un latch. L'uscita segue l'ingresso quando il segnale di clock è alto.

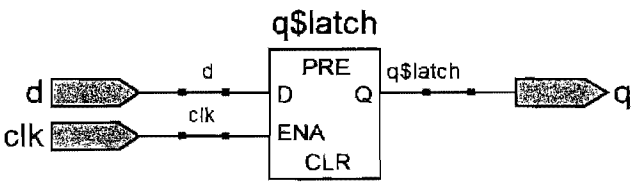


Figura 5.11: Latch sintetizzato mediante una descrizione Verilog nella quale si utilizza un costrutto if senza specificare, il comportamento nei casi non inclusi nel costrutto if.

5.7 Registri

Un registro si descrive molto semplicemente come un flip flop con ingressi vettoriali.
Si consideri ad esempio un registro ad 8 bit sincronizzato sul fronte di discesa del clock e con reset sincrono attivo alto.
La descrizione è mostrata di seguito mentre il circuito sintetizzato è visibile in Figura 5.12.

```
module reg8 (d,clk,reset,q);
input clk,reset;
input [7:0] d;
output [7:0] q;
wire clk,reset;
wire [7:0] d;
reg [7:0] q;

always @(posedge clk)
begin
        if (reset)
                q<=8'b00000000;
        else
                q<=d;
end
endmodule
```

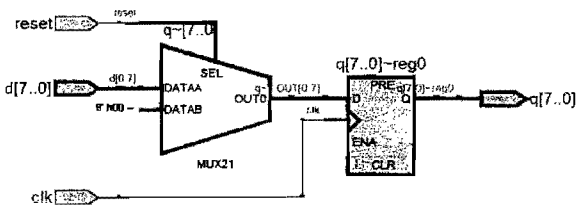


Figura 5.12: Vista RTL del circuito sintetizzato a partire dalla descrizione del registro ad 8 bit con reset sincrono.

Un insieme di registri rappresenta una memoria.
Per questo tipo di struttura dati il Verilog dispone dei vettori bidimensionali di segnali di tipo reg. Un cenno a questo tipo di struttura è presentato al paragrafo 7.4.
L'istruzione che definisce un vettore bidimensionale di registri è:

```
reg [Width-1:0] mem [Num_reg-1:0];   // use a bidimensional array named mem
                                     //with Num_reg registers width bit wide
```

5.8 Registro a scorrimento

Un registro a scorrimento può essere descritto in Verilog modificando leggermente la descrizione di un registro convenzionale. Di seguito si mostra la descrizione di un registro a scorrimento da 8 bit con reset, in cui i bit sono traslati a destra in concomitanza con il fronte di salita del clock.

```
module shift_reg(din,a,clk,reset);
input clk,din,reset;
output [7:0] a;
wire clk,din,reset;
reg [7:0] a;


always@(posedge clk)
begin
if (reset)
        a<=8'h00;
else
        begin
        a[6:0]<=a[7:1];
        a[7]<=din;
        end
end
endmodule
```

La descrizione del registro a scorrimento permette di evidenziare una caratteristica fondamentale delle assegnazioni di tipo 'non blocking' (contraddistinte dall'operatore '<=') utilizzate nei blocchi procedurali sintetizzabili. Il termine 'non blocking' è infatti usato per indicare una assegnazione che non ha effetto immediato sul segnale considerato il cui aggiornamento viene effettuato solo all'uscita dal blocco procedurale. In pratica, supponendo che il segnale a[7:0] abbia

il valore '10111111' prima del fronte del clock, l'assegnazione a[6:0]<=a[7:1] equivale a a[6:0]<='1011111' indipendentemente dal fatto che altre assegnazioni 'non blocking' possano aver modificato i bit a[7:1] all'interno del blocco procedurale.
Ad esempio, se le due assegnazioni del blocco procedurale venissero invertite, e fossero quindi:

```
a[7]<=din;
a[6:0]<=a[7:1];
```

il comportamento del circuito non cambierebbe in quanto la prima assegnazione indica come verrà modificato a[7] all'uscita dal blocco procedurale. La seconda assegnazione indica che, sempre all'uscita del blocco procedurale, i bit a[6:0] dovranno avere il valore che avevano i bit a[7:1] prima che il blocco procedurale venisse eseguito.
Per fissare al meglio il discorso mostrato analizziamo il blocco procedurale mostrato di seguito.

```
always @(a)
begin
        b<=0;
        y<=a & b;
end
```

Se il segnale *b* ha valore logico '1' ed il blocco procedurale viene eseguito in conseguenza di un evento sul segnale *a* che lo rende '1', i valori dei segnali *b* e *y*, all'uscita del blocco procedurale saranno '0' e '1', rispettivamente. Questo a causa del fatto che l'istruzione `y<=a & b` è eseguita considerando il valore che b ha all'ingresso del blocco procedurale. Si ribadisce quindi che l'assegnazione 'non blocking' programma una modifica del segnale che avviene solo all'uscita del blocco procedurale.
Un possibile test bench per la simulazione del registro a scorrimento è il seguente.

```
`timescale 1ns/1ps
module shift_reg_tb();
parameter period=30;
reg ck,dtb,r;
wire [7:0] atb;

shift_reg UUT(.din(dtb),.a(atb),.clk(ck),.reset(r));

always
begin
ck=0;
#(period/2);
ck=1;
#(period/2);
end

initial
begin
r=1;
dtb=1'b1; #period;
r=0;dtb=1'b1; #(period*8);
dtb=1'b0; #(period*8);
$stop;
end
endmodule
```

La simulazione funzionale del registro a scorrimento è mostrata in Figura 5.13.

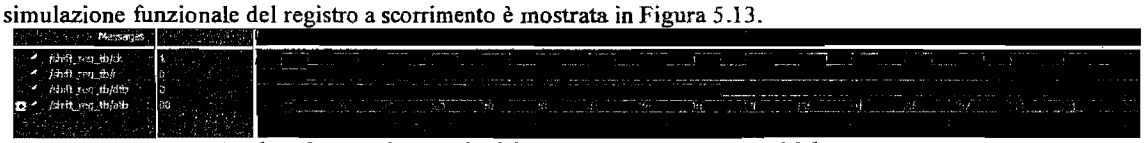


Figura 5.13: Simulazione funzionale del registro a scorrimento ad 8 bit con reset sincrono.

5.9 Contatori

Molto utilizzati nei circuiti digitali i contatori tengono traccia degli eventi e risultano indispensabili. La maggioranza dei blocchi logici programmabili delle FPGA moderne sono ottimizzati per la realizzazione di contatori. Il contatore è un componente sequenziale composto da un banco di registri da n bit e un addizionatore. Uno schema circuitale, nel quale non è indicato l'indispensabile senale di reset, è mostrato in Figura 5.14.
Il contatore può essere anche studiato come una macchina a stati finiti o come un circuito aritmetico. Per i dettagli relativi ai circuiti aritmetici ed alle macchine a stati finiti ci si riferisca ai paragrafi corrispondenti. In questo ci si limita a riportare che il linguaggio Verilog dispone dell'operatore addizione, '+' (anche dell'operatore sottrazione) che permette un'immediata descrizione del contatore.

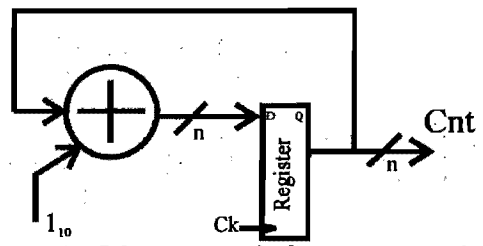


Figura 5.14: Schema circuitale di un contatore ad n bit.

Lo schema di principio mostrato in Figura 5.14 non è molto utile nelle applicazioni reali in quanto non dispone di un segnale di reset ed incrementa il conteggio ad ogni fronte del clock.
Un circuito maggiormente utile deve prevedere almeno un segnale di abilitazione che indica se incrementare il conteggio o meno. La descrizione seguente si riferisce ad un contatore ad 8 bit sincronizzato sul fronte di salita del clock e con segnale di reset sincrono. Il contatore è inoltre dotato di un segnale, *en*, che se alto abilita il conteggio.

```
module counter(ck,reset,en,cnt);
input ck,reset,en;
output [7:0] cnt;
wire  ck,reset,en;
reg [7:0] cnt;

always @(posedge ck)
begin
if (reset)
        cnt<=8'b00000000;
        else
        if (en)
        cnt<=cnt + 1'b1;
end
endmodule
```

La descrizione è molto compatta. Utilizza un blocco procedurale attivato dal fronte di salita del clock. Il blocco procedurale azzera il conteggio se il segnale di reset è alto, in caso contrario effettua un controllo sul segnale di abilitazione, *en*, e se è al livello logico alto somma uno al valore del conteggio.
Il caso in cui il segnale di abilitazione è basso non è esplicitato. In tal caso il valore del conteggio rimane inalterato.
La vista RTL del circuito sintetizzato è mostrata in Figura 5.15. Come previsto il circuito è realizzato con un registro ed un addizionatore ad 8 bit con uno degli operandi fissato al valore uno. Sono anche presenti due multiplexer, gestiti rispettivamente dal segnale di abilitazione e da quello di reset. Il multiplexer gestito dal segnale di abilitazione seleziona come ingresso al registro il valore precedente del conteggio o il valore incrementato. Il secondo multiplexer seleziona il risultato del primo oppure il valore '00000000'. L'ordine dei multiplexer ne definisce la priorità che nel caso mostrato è data al segnale di reset. La simulazione funzionale e la simulazione post sintesi del contatore sono mostrate in Figura 5.16. Il comportamento è quello atteso. Il contatore viene inizialmente azzerato e poi conta sino a cinque. Viene quindi disabilitato e blocca il suo valore di conteggio. Un successivo reset fa ripartire il conteggio da zero una volta che anche il segnale di abilitazione è al livello logico '1'.
Il circuito di Figura 5.15 permette una riflessione più generica sulla realizzazione moderna ed efficiente dei circuiti digitali di tipo sincrono. Il segnale di clock funge unicamente da sincronizzatore ed arriva direttamente al registro che dovrà memorizzare il risultato. Un circuito combinatorio, anche molto complesso, utilizza i segnali di ingresso, quelli di controllo, nonché il valore memorizzato nel banco di registri per calcolare il nuovo valore da memorizzare nel banco di registri. Il fronte del clock è l'evento che permette la memorizzazione del nuovo valore nel banco di registri ed avvia l'elaborazione del nuovo dato.

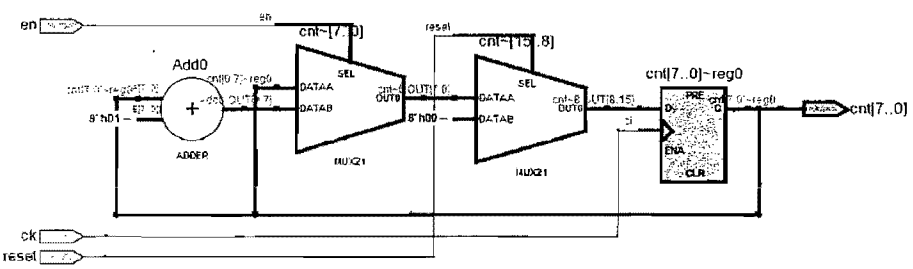


Figura 5.15: Vista RTL del risultato della sintesi della descrizione Verilog di un contatore ad 8 bit con reset sincrono ed abilitazione. Da rilevare la presenza dell'addizionatore e dei multiplexer che sono gestiti dai segnali di reset ed abilitazione.

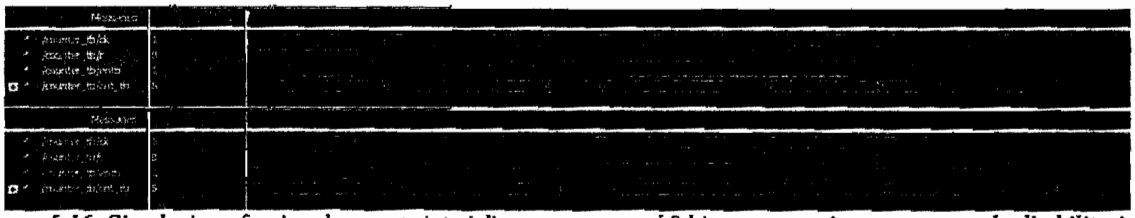


Figura 5.16: Simulazione funzionale e post sintesi di un contatore ad 8 bit con reset sincrono e segnale di abilitazione entrambi attivi al livello logico alto.

Una versione più completa di contatore potrebbe includere un segnale di caricamento (load) di un valore iniziale di conteggio diverso da zero. E' inoltre molto diffuso anche il contatore bidirezionale che, in funzione di un segnale di controllo, conta sia in maniera crescente, sia in maniera decrescente.
Anche questo tipo di circuito viene descritto alquanto semplicemente utilizzando il linguaggio Verilog. Si richiedono ulteriori segnali di ingresso ed un insieme di costrutti *if else* più elaborato. In Figura 5.17 si mostra il risultato della simulazione funzionale del circuito descritto in basso. Si evidenzia, verso la fine della simulazione, l'inizializzazione a 15 del valore di conteggio. A valle della sintesi e del place & route il circuito, nonostante la sua complessità, occupa 13 Logic Elements di una FPGA CycloneII. Il risultato evidenzia quanto le FPGA moderne siano efficienti nell'implementare i circuiti più comunemente utilizzati dai progettisti.

```
module count_udl(ck,reset,en,ud,load,d,cnt);
input ck,reset,en,ud,load;
input [7:0] d;
output [7:0] cnt;
wire  ck,reset,en,ud,load;
wire [7:0] d;
reg [7:0] cnt;

always @(posedge ck)
begin
if (reset)                                      // reset has the highest priority
        cnt<=8'b00000000;
        else
        if (en)                                 // check enable
        begin
                if (load)                       // eventually load count value
                cnt<=d;
                else
                        if (ud)                 // up/down counter
                        cnt<=cnt + 1'b1;
                        else
                        cnt<=cnt - 1'b1;
        end
end
endmodule
```

Figura 5.17: Simulazione funzionale e post sintesi di un contatore bidirezionale ad 8 bit con reset, abilitazione e segnale di load.

x6. Circuiti aritmetici

Il linguaggio HDL Verilog dispone di operatori per l'esecuzione di varie operazioni aritmetiche su segnali binari. Di interesse per la sintesi di circuiti digitali sono solo gli operatori che risultano sintetizzabili.
Tra gli operatori sintetizzabili si annoverano la somma, la sottrazione e la moltiplicazione ('+','-','*'). Sintetizzabili, e particolarmente utili sono anche gli operatori di traslazione (shift) a destra e a sinistra ('<<','>>').
Non sono sintetizzabili gli operatori di divisione tra segnali binari e l'operatore di divisione per una costante. Un caso particolare è la divisione per potenze di due che si effettua mediante l'operatore di shift a destra.
Non sono infine disponibili per la sintesi le funzioni non lineari come esponenziale, logaritmo e le funzioni trigonometriche.

x6.1 Rappresentazione dei numeri

Il metodo più comune per rappresentare i numeri che sono utilizzati nei circuiti digitali è la rappresentazione in complementi alla base con virgola fissa.
Si consideri un numero intero rappresentato su n bit in complementi alla base. La sua rappresentazione sia:
$\{A_{n-1},A_{n-2},...,A_1,A_0\}$ i pesi dei singoli bit varranno $\{-2^{n-1},2^{n-2},...,2^1,2^0\}$. Si noti che il MSB ha peso negativo.
Il massimo numero rappresentabile si ha per $\{A_{n-1}=0\ A_{n-2}=1\ A_{n-3}=1\ ...\ A_1=1\ A_0=1\}$ e vale $2^{n-1}-1$.
Il minimo numero rappresentabile è $\{A_{n-1}=1\ A_{n-2}=0\ A_{n-3}=0\ ...\ A_1=0\ A_0=0\}$ e vale -2^{n-1}.
Nel caso di numeri interi ad 8 bit il peso del bit più significativo (MSB) è -128, il peso del bit meno significativo (LSB) è 1 e l'ambito dei numeri rappresentabili è {127,-128}.
Se consideriamo numeri frazionari la posizione della virgola sarà interna al numero rappresentato. E' conveniente in questo caso specificare la rappresentazione dei numeri indicando il peso del bit più significativo ed il peso del bit meno significativo. Si consideri un numero con tre bit a destra della virgola e rappresentato su n bit in complementi alla base. La sua rappresentazione è: $\{A_{n-4},A_{n-5},...\ A_1\ A_0\ ,\ A_{-1},\ A_{-2},A_{-3}\}$ i pesi dei singoli bit varranno
$\{-2^{n-4}\ 2^{n-5}\ ...\ 2^1\ 2^0\ 2^{-1}\ 2^{-2}\ 2^{-3}\}$. Anche in questo caso il MSB ha peso negativo. I bit a destra della virgola hanno pesi che corrispondono alle potenze negative di due ($2^{-1}=1/2\ 2^{-2}=1/4$ etc.).
Il massimo numero rappresentabile si ha per $\{A_{n-4}=0\ A_{n-5}=1\ A_{n-6}=1\ ...\ A_{-2}=1\ A_{-3}=1\}$ e vale $2^{n-4}-2^{-3}$.
Il minimo numero rappresentabile è $\{A_{n-4}=1\ A_{n-5}=0\ A_{n-6}=0\ ...\ A_{-2}=0\ A_{-3}=0\}$ e vale -2^{n-4}.
Se consideriamo numeri ad 8 bit con tre bit a destra della virgola il peso del bit più significativo (MSB) è -16, il peso del bit meno significativo (LSB) è 1/8 e l'ambito dei numeri rappresentabili è {16-1/8=15.875,-16}.
In generale si indicano le rappresentazioni di numeri con segno e virgola fissa con la notazione $Q_{n,m}$. Dove Q indica i numeri con segno, *n* indica il peso del MSB ed *m* indica il peso del LSB. In generale sia *n* sia *m* possono essere negativi. Numeri interi ad 8 bit vengono indicati con la notazione $Q_{7,0}$. Il caso precedentemente mostrato con 8 bit e tre bit a destra della virgola si indica con $Q_{4,3}$. Un numero frazionario senza alcun bit a sinistra della virgola si indica con $Q_{-1,m}$.

In linguaggio Verilog il trattamento dei numeri e delle operazioni aritmetiche è stato migliorato nello standard presentato nel 2001.
Nello standard del 1995 i segnali di tipo *wire* e *reg* erano utilizzati come numeri senza segno (unsigned). Gli unici dati con segno erano gli interi (ed i numeri a virgola mobile) di difficile utilizzo per la realizzazione di sistemi sintetizzabili.
A partire dallo standard del 2001 anche i segnali di tipo *reg* e *wire* possono essere dichiarati come numeri con segno (signed). L'istruzione da utilizzare utilizzano la parola chiave *signed*. Di seguito alcuni esempi.
wire signed [7:0] A;
reg signed [15:0] B;

Le operazioni su segnali che rappresentano numeri con segno possono diventare molto complesse da gestire. In generale un'operazione che coinvolge numeri unsigned ha risultato unsigned. Un'operazione che coinvolge numeri signed ed unsigned ha risultato unsigned. Si ha un risultato signed solo se tutti gli operandi sono signed.
Per gestire più facilmente le conversioni si possono utilizzare, anche per descrivere codice sintetizzabile, le funzioni di sistema **$signed()** e **$unsigned()** per le conversioni di tipo.

x6.2 Troncamento ed arrotondamento (rounding)

Tra le operazioni più comuni quando si opera con numeri in virgola fissa troviamo il troncamento. Si utilizza quando del risultato di un'operazione si vogliono trascurare i bit meno significativi. Ad esempio il risultato di un'operazione potrebbe essere su 16 bit mentre il registro nel quale memorizziamo il risultato è da 8 bit. Il modo più semplice per effettuare il troncamento è eliminare i bit che non si utilizzano. Un minimo di riflessione mostra che in questo caso commettiamo un errore di troncamento che è sempre dello stesso segno in quanto annulliamo dei bit con peso positivo.
Se eliminiamo 4 bit il cui LSB ha peso 2^{-5} l'errore che si commette varia tra 0 (quando i 4 bit eliminati sono nulli) a $0.4688 = 2^{-2}+2^{-3}+2^{-4}+2^{-5}$ (quando i 4 bit eliminati valgono '1').
Un metodo più preciso per ridurre il numero di bit per una rappresentazione è utilizzare un troncamento con

arrotondamento (rounding). Il rounding prevede che prima di eliminare i bit in eccesso si sommi il peso del primo bit trascurato. L'operazione permette di ottenere un errore a media che tende a zero se il numero di bit trascurati cresce.
Ritornando all'esempio in cui si eliminano 4 bit il cui LSB ha peso 2^{-5} l'operazione di rounding e troncamento prevede che prima di eliminare gli ultimi 4 bit si sommi al numero da troncare il valore 2^{-2}. L'errore che si commette varia tra -0.25 (quando i 4 bit eliminati sono '1000') a $0.2188 = 2^{-3}+2^{-4}+2^{-5}$ (quando i 4 bit eliminati valgono '0111').
La Tab. x6.1 mostra gli effetti e l'errore commesso in caso di operazione di troncamento e di troncamento con rounding.
L'operazione di troncamento in linguaggio HDL Verilog può essere effettuata in due passi. Dapprima si somma la costante per il rounding e quindi si procede a selezionare i bit più significativi del segnale risultante. Di seguito un estratto del codice che tronca un segnale a 16 bit per ottenere un segnale a 8 bit.

```
wire [7:0] A;                          // Truncated signal
wire [15:0] B, tmp;                    // Original signal and temporary signal

assign tmp = B + 16'b0000_0000_1000_0000;   // The _ symbol is ignored in the representation of the number
                                            // is included for readability
                                            // 16'b0000_0000_1000_0000 is the rounding constant
assign A = tmp[15:8];                  // select the most significant part
```

Il risultato della sintesi di un circuito che effettua il troncamento di un segnale da 16 ad 8 bit utilizza un addizionatore ad 8 bit ed è mostrato in Figura . La sintesi per FPGA CycloneII utilizza 8 Logic Elements.

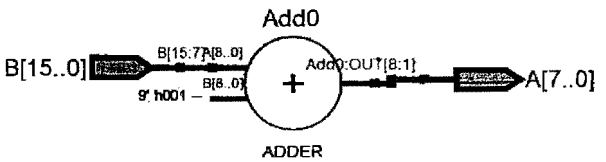


Figura x6.1: Circuito sintetizzato a partire dalla descrizione di un'operazione di troncamento da 16 ad 8 con arrotondamento.

Number to be truncated								Truncation		Truncation & rounding	
2^n	2^{n-1}	..	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	Result	Error	Result	Error
A_n	A_{n-1}	...	A_m	0	0	0	0	$A_n A_{n-1} \dots A_m$	0	$A_n A_{n-1} \dots A_m$	0
A_n	A_{n-1}	...	A_m	0	0	0	1	$A_n A_{n-1} \dots A_m$	0.0313	$A_n A_{n-1} \dots A_m$	0.0313
A_n	A_{n-1}	...	A_m	0	0	1	0	$A_n A_{n-1} \dots A_m$	0.0625	$A_n A_{n-1} \dots A_m$	0.0625
A_n	A_{n-1}	...	A_m	0	0	1	1	$A_n A_{n-1} \dots A_m$	0.0938	$A_n A_{n-1} \dots A_m$	0.0938
A_n	A_{n-1}	...	A_m	0	1	0	0	$A_n A_{n-1} \dots A_m$	0.1250	$A_n A_{n-1} \dots A_m$	0.1250
A_n	A_{n-1}	...	A_m	0	1	0	1	$A_n A_{n-1} \dots A_m$	0.1563	$A_n A_{n-1} \dots A_m$	0.1563
A_n	A_{n-1}	...	A_m	0	1	1	0	$A_n A_{n-1} \dots A_m$	0.1875	$A_n A_{n-1} \dots A_m$	0.1875
A_n	A_{n-1}	...	A_m	0	1	1	1	$A_n A_{n-1} \dots A_m$	0.2188	$A_n A_{n-1} \dots A_m$	0.2188
A_n	A_{n-1}	...	A_m	1	0	0	0	$A_n A_{n-1} \dots A_m$	0.2500	$A_n A_{n-1} \dots (A_m+1)$	-0.2500
A_n	A_{n-1}	...	A_m	1	0	0	1	$A_n A_{n-1} \dots A_m$	0.2813	$A_n A_{n-1} \dots (A_m+1)$	-0.2188
A_n	A_{n-1}	...	A_m	1	0	1	0	$A_n A_{n-1} \dots A_m$	0.3125	$A_n A_{n-1} \dots (A_m+1)$	-0.1875
A_n	A_{n-1}	...	A_m	1	0	1	1	$A_n A_{n-1} \dots A_m$	0.3438	$A_n A_{n-1} \dots (A_m+1)$	-0.1563
A_n	A_{n-1}	...	A_m	1	1	0	0	$A_n A_{n-1} \dots A_m$	0.3750	$A_n A_{n-1} \dots (A_m+1)$	-0.1250
A_n	A_{n-1}	...	A_m	1	1	0	1	$A_n A_{n-1} \dots A_m$	0.4063	$A_n A_{n-1} \dots (A_m+1)$	-0.0938
A_n	A_{n-1}	...	A_m	1	1	1	0	$A_n A_{n-1} \dots A_m$	0.4375	$A_n A_{n-1} \dots (A_m+1)$	-0.0625
A_n	A_{n-1}	...	A_m	1	1	1	1	$A_n A_{n-1} \dots A_m$	0.4688	$A_n A_{n-1} \dots (A_m+1)$	-0.0313
								Average	0.2344	**Average**	-0.0156

Tabella x6.1: Operazione di troncamento con e senza arrotondamento (rounding). Risultato finale ed errore commesso.

x6.3 Estensione del segno

Se si opera su numeri rappresentati in virgola fissa con differenti numeri di bit prima di effettuare delle operazioni aritmetiche è necessario allineare i bit degli operandi al fine di rappresentare i numeri con la virgola nella stessa posizione. Si procede poi con delle operazioni che portano gli operandi ad essere rappresentati sullo stesso numero di bit.
Per quanto riguarda i bit meno significativi si agisce, se necessario, con un'operazione di troncamento nei confronti dell'operando che ha il maggior numero di bit a destra della virgola o aggiungendo degli zeri in coda all'operando ha il minor numero di bit a destra della virgola.
Per quanto riguarda i bit a sinistra della virgola, non potendo troncare i bit più significativi, è necessario agire

sull'operando che ha il minor numero di bit a sinistra della virgola.
Per numeri senza segno la rappresentazione può essere estesa aggiungendo banalmente degli zeri in testa alla rappresentazione. Non altrettanto immediata è l'operazione da eseguire nel caso di numeri rappresentati in complementi alla base per i quali il MSB rappresenta anche il segno. Si opera in questo caso con l'operazione di estensione del segno che, nel caso in cui si desideri aggiungere n bit in testa alla rappresentazione di un numero positivo (MSB=0) aggiunge n bit pari a '0' in testa alla rappresentazione; nel caso in cui si desideri aggiungere n bit in testa alla rappresentazione di un numero negativo (MSB=1) aggiunge n bit pari a '1' in testa alla rappresentazione.
E' facile dimostrare che tale operazione non modifica il numero rappresentato.
Ad esempio il numero -7 nella rappresentazione $Q_{4,0}$ è '11001'. Volendo passare alla rappresentazione $Q_{7,0}$ il numero diventa '11111001'.

La descrizione in linguaggio Verilog dell'operazione di estensione del segno può essere eseguita utilizzando costrutti standard che controllano il bit più significativo e lo replicano per ottenere una rappresentazione su di un maggior numero di bit. Modernamente è invece preferibile utilizzare la parola chiave *signed* per dichiarare segnali che rappresentano numeri con segno in quanto per tali segnali il Verilog effettua automaticamente l'operazione di estensione del segno, se necessaria. Di seguito un esempio.

```
wire          [7:0] Au;          // Au is unsigned  8 bit number
wire          [15:0] Bu;         // Bu is unsigned  16 bit number
assign Bu = Au;                  // Adds 8 '0' bit on the left of Au to get a 16 bit unsigned result

wire signed   [7:0] As;          // As is signed  8 bit number
wire signed   [15:0] Bs;         // Bs is signed  16 bit number
assign Bs = As;                  // Sign extends As to get a 16 bit signed result
```

x6.4 Somme e sottrazioni

Somma e sottrazione si effettuano mediante gli operatori '+' e '-'.
In generale il risultato dell'operazione ha le dimensioni del segnale di destinazione. Il risultato dell'operazione ha il tipo (signed/unsigned) del segnale di destinazione.
La gestione dell'overflow deve essere attentamente curata dal progettista in quanto, ad esempio, la somma tra due numeri ad n bit è in generale un numero ad n+1 bit. Se il risultato è assegnato ad un numero ad n bit si rischia di perdere il bit più significativo e quindi le informazioni sull'overflow. Di seguito alcuni semplici esempi.

Somme tra segnali unsigned.

```
module sum(A,B,Y);
input  [7:0] A,B;
output [7:0] Y;
wire [7:0]  A,B,Y;     // 8 bit unsigned operands and result

assign Y = A + B;      // The sum loses the overflow information
endmodule
```

La somma di A e B, che potrebbe essere su 9 bit, è troncata ad 8 bit di rappresentazione. Non si hanno informazioni sull'overflow.

```
module sum(A,B,Y);
input  [7:0] A,B;
output [8:0] Y;
wire [7:0]  A,B;       // 8 bit unsigned operands
wire [8:0]  Y;         // 9 bit unsigned result

assign Y = A + B;      // The sum includes the overflow information in the 9th bit
endmodule
```

La somma di A e B viene assegnata ad un segnale su 9 bit. Le informazioni sull'overflow sono nel bit più significativo di Y.

```
module sum(A,B,Y,ov);
input  [7:0] A,B;
output [7:0] Y;
output ov;
wire [7:0]  A,B,Y;     // 8 bit unsigned operands and result
wire ov;               // overflow bit

assign {ov,Y} = A + B;  // The overflow information is assigned to the ov signal
endmodule
```

Gli otto bit meno significativi della somma tra A e B sono assegnati al risultato. Il nono bit, che indica la presenza di

overflow è assegnato all'apposito segnale, *ov*.

Somma con saturazione

In molte applicazioni digitali, al fine di ridurre l'occupazione di logica programmabile, velocizzare il circuito e ridurre la dissipazione di potenza, si utilizza un numero di bit ridotto per la rappresentazione dei numeri.
La conseguenza è che la dinamica dei numeri può essere non sufficiente a rappresentare tutte le possibili condizioni e si possono avere problemi di overflow.
In molti casi l'overflow può rappresentare un problema grave. Si supponga di voler controllare la velocità di un motore utilizzando un controllo in retroazione. Ad ogni colpo di clock si decide quanta coppia imprimere al motore incrementando o decrementando un registro. Se ad un certo punto l'incremento causa un overflow del registro che passa da '11111111' a '00000000', il segnale di coppia invece di incrementare si azzera. Il risultato è almeno una instabilità del sistema di controllo.
Il problema può essere risolto utilizzando un sommatore con saturazione che, nel caso in cui la somma dovesse superare il massimo valore rappresentabile, fissa il valore di uscita al massimo. Uno schema di principio del circuito sommatore con saturazione è visibile in Figura x6.2. Un possibile descrizione Verilog utilizza due assegnazioni continue

```
assign {ov,Y} = A + B;                    // Calculate 8bit sum and overflow bit
assign Ysat = (ov) ? 8'b1111_1111 : Y;    // Saturate the 8bit sum with a mux on the ov bit.
```

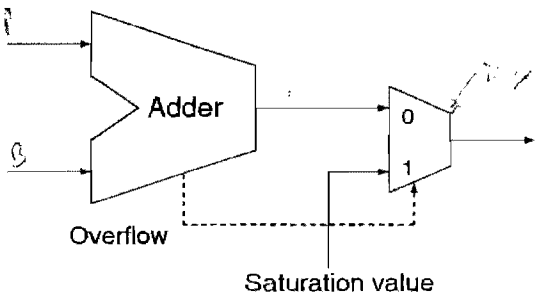


Figura x6.2: Schema di principio di un addizionatore con saturazione dell'uscita.

x6.5 Moltiplicazione

L'operazione di moltiplicazione tra segnali su n bit fornisce un risultato su 2n bit.
Il Verilog dispone dell'operatore '*' che permette la realizzazione efficiente di una moltiplicazione. Se si effettua una moltiplicazione tra segnali unsigned e lo si assegna ad un segnale con un numero sufficiente di bit il comportamento e la descrizione del circuito sono molto semplici. Un esempio di moltiplicatore ad 8 bit è il seguente.

```
module mult(A,B,Y);
input  [7:0] A,B;
output [15:0] Y;

wire [7:0]  A,B;
wire [15:0]  Y;

assign Y = A * B;
endmodule
```

Sintetizzando il circuito per una FPGA di tipo CycloneII la moltiplicazione viene implementata utilizzando una dei DSP block interni al FPGA. Se il numero di bit della moltiplicazione supera il massimo implementabile con un blocco DSP l'operazione è effettuata mettendo in cascata due blocchi DSP.

Un caso interessante è la moltiplicazione di un segnale per una costante. Anche tale circuito si realizza utilizzando l'operatore '*'. La costante può essere esplicitata nel codice o definita mediante un'istruzione *parameter*.

```
module multcost(A,Y);
input  [7:0] A;
output [15:0] Y;
parameter [7:0] C=8'b1001_0110;   // 8'd150
wire [7:0]  A;
wire [15:0] Y;

assign Y = A * C;
endmodule
```

x6.6 Operazione di moltiplicazione e somma (Multiply and Accumulate, MAC)

In molti casi pratici l'operazione da effettuare sui dati corrisponde a moltiplicare il dato per una costante e sommare al risultato un ulteriore ingresso. L'operazione viene definita di Multiply And Accumulate (MAC) ed è molto utile, ad esempio, nella realizzazione di filtri numerici.
Per l'operazione di MAC esistono strutture circuitali molto efficienti e per le FPGA sono stati realizzati i DSP block che sono particolarmente adatti alle operazioni di MAC.
L'effettivo utilizzo di tali blocchi è però dipendente dalle impostazioni del sintetizzatore e dalle prestazioni richieste al circuito. Di seguito un esempio di modulo Verilog che implementa una funzione di MAC.

```
module mac(A,B,Y);
input  [7:0] A,B;
output [15:0] Y;
wire [7:0]  A,B;
wire [15:0]  Y;
parameter [7:0] C=8'b1001_0110;   // 8'd150

assign Y = (A * C) + B;
endmodule
```

7. Macchine a stati finiti

Le macchine a stati finiti sono i componenti che permettono la realizzazione di circuiti digitali in grado di reagire agli eventi in funzione dello stato dell'elaborazione. Sono il metodo migliore per realizzare un circuito che implementi un algoritmo e sono indispensabili in ogni circuito che debba avere un'utilità pratica. Controllo di flusso, gestione degli eventi e degli errori, attesa e controllo degli ingressi e delle uscite sono alcune delle applicazioni tipiche.

La descrizione di una macchina a stati può essere di tipo Moore (uscite dipendenti solo dallo stato) o di tipo Mealy (uscite dipendenti solo dallo stato e dagli ingressi). I due tipi di descrizione sono funzionalmente equivalenti. Le macchine di Moore tendono ad avere un maggior numero di stati, non presentano glitch sulle uscite e forniscono l'uscita sincrona con il segnale di clock. Le macchine di Mealy hanno generalmente un minor numero di stati, e l'uscita, essendo dipendente dall'ingresso, si modifica non appena l'ingresso cambia (senza attendere il segnale di clock), non è sincrona con il segnale di clock e può presentare glitch.

La realizzazione efficace di una macchina a stati dipende in gran parte dal tipo di codifica dello stato che si sceglie. La codifica è un inevitabile compromesso tra i parametri di compattezza, velocità, affidabilità e resistenza ai guasti per la macchina a stati.

In Verilog la codifica degli stati deve essere esplicitata dal progettista. I nomi degli stati possono essere enumerati, utilizzando la parola chiave *parameter*, rendendo il codice molto flessibile e leggibile. L'istruzione che si utilizza per definire gli stati è la seguente:

```
parameter [n-1:0]
          <name_state_1> = n'b<cod_state_1>,
          <name_state_2> = n'b<cod_state_2>,
          .
          .
          <name_state_S> = n'b<cod_state_S>;
```

La definizione dei parametri definisce il numero di bit utilizzati per la codifica, indicati con il simbolo *n*. Per una codifica binaria si utilizzeranno $\lceil \log_2(S) \rceil$ bit per S stati. Per altri tipi di codifica il numero può aumentare. Vengono poi definiti i nomi dei singoli stati e la rispettiva codifica. I nomi sono rappresentati da stringhe alfanumeriche che dovrebbero rispecchiare lo stato del sistema per aumentare la leggibilità del codice. L'esempio mostrato usa una rappresentazione binaria per la codifica, ma è possibile utilizzare ogni altra rappresentazione se questo dovesse semplificare la lettura del codice.

E' da evidenziare che il codice mostrato non effettua alcun controllo sulla codifica scelta. Se il numero di bit utilizzati dovesse essere inadeguato o se due stati dovessero avere la stessa codifica, si avrebbe un errore nel comportamento della macchina a stati senza che il sintetizzatore fornisca alcun errore.

Come menzionato in precedenza lo standard del Verilog non permette l'utilizzo di tipi di dati enumerati per i quali non si definisca la codifica. Tali tipi di dato potrebbero infatti essere un modo per lasciare il sintetizzatore libero di ottimizzare la realizzazione della macchina a stati in funzione delle prestazioni richieste. Nonostante questo la maggioranza dei sistemi di sviluppo è in grado di riconoscere il codice HDL relativo alla descrizione di macchine a stati finiti ed ottimizza la codifica indipendentemente da quanto indicato dal progettista. Ad esempio, il software QuartusII è in grado di estrarre ed ottimizzare le macchine a stati finiti autonomamente. Se si desidera che il sintetizzatore non modifichi le scelte sulla codifica effettuate dal progettista è sufficiente attivare la finestra relativa alle opzioni del sintetizzatore (*Assignments* | *Settings...* | *Analysys and Synthesys Settings* | *More Settings*) ed indicare *User-Encoded* in corrispondenza delle opzioni di *State Machine Processing*.

Da un punto di vista circuitale le macchine a stati finiti (FSM), sono realizzate con un banco di registri e con due funzioni combinatorie che calcolano il prossimo stato e l'uscita.

Per migliorare la leggibilità del codice è conveniente che la descrizione Verilog rispecchi la realizzazione circuitale.

Le descrizioni più comuni utilizzano da uno a tre blocchi procedurali. Probabilmente le descrizioni più diffuse utilizzano due blocchi procedurali.

Supponendo di voler utilizzare tre blocchi procedurali il primo implementa il registro di stato e sarà attivato dal segnale di reset e dal fronte del clock. Il secondo elabora il prossimo stato ed è attivato dalla variazione dello stato presente e dalla variazione dell'ingresso. Il terzo implementa la logica combinatoria per l'elaborazione dell'uscita. Se la macchina a stati è di tipo Moore, il terzo blocco procedurale è attivato unicamente da una variazione dello stato presente. Se la macchina a stati è di tipo Mealy, il terzo blocco procedurale è attivato da una variazione dello stato presente o degli ingressi.

Una descrizione, mediante due blocchi procedurali utilizza un blocco procedurale per la descrizione del registro di stato e della logica per il calcolo del prossimo stato. Il secondo blocco procedurale calcola l'uscita della FSM.

7.1 Moore finite state machine

Si consideri un riconoscitore di sequenza con un ingresso ed un uscita. L'uscita diventa alta quando sono stati ricevuti 5 consecutivi ingressi validi (ingresso con valore logico '1'). Utilizziamo una descrizione di Moore ed una codifica dello stato di tipo binario.

Supponiamo che si ritorni allo stato iniziale se la sequenza di ingressi validi viene interrotta e sono giunti meno di 4 oppure 5 ingressi validi consecutivi. Se invece sono giunti 4 ingressi validi consecutivi si torna allo stato corrispondente a 3 ingressi validi consecutivi. La tabella di transizione è mostrata in Tab. 7.1 e contiene 6 stati che denominiamo S0...S5. Per una codifica binaria saranno necessari 3 bit. La descrizione Verilog della macchina a stati finiti è la seguente.

	Stato / Uscita	S0 / 0	S1 / 0	S2 / 0	S3 / 0	S4 / 0	S5 / 1
Ingresso	**0**	S0	S0	S0	S0	S3	S0
	1	S1	S2	S3	S4	S5	S0

Tabella 7.1: FSM di Moore per un riconoscitore di sequenza.

```
module FSM_moore (clk,reset,din,dout);
input clk,reset,din;
output dout;
wire clk,reset,din;
reg dout;

reg [2:0] ac_state, nx_state;   // declare present and next state

parameter [2:0]   // Binary state coding
S0=3'b000,
S1=3'b001,
S2=3'b010,
S3=3'b011,
S4=3'b100,
S5=3'b101;

always @(posedge reset or posedge clk) // state register
begin
        if (reset)
                ac_state <= S0;
        else
                ac_state <= nx_state;
end

always @(ac_state,din)  // Comb. logic for next state
begin
case (ac_state)
S0: nx_state <= (din) ? S1 : S0;
S1: nx_state <= (din) ? S2 : S0;
S2: nx_state <= (din) ? S3 : S0;
S3: nx_state <= (din) ? S4 : S0;
S4: nx_state <= (din) ? S5 : S3;
S5: nx_state <= (din) ? S0 : S0;
default: nx_state <= S0;
endcase
end

always @(ac_state)  // Comb. Moore logic for output
begin
case (ac_state)
S0: dout <= 1'b0;
S1: dout <= 1'b0;
S2: dout <= 1'b0;
S3: dout <= 1'b0;
S4: dout <= 1'b0;
S5: dout <= 1'b1;
default: dout <= 1'b0;
endcase
end
endmodule
```

A valle della sintesi utilizzando lo strumento Netlist Viewers | State Machine Viewer nella finestra Tasks del sistema di sviluppo QuartusII si verifica che il sintetizzatore ha rilevato la presenza di una FSM.
Lo strumento State Machine Viewer mostra un diagramma a bolle della FSM, la sua tabella di transizione e la codifica dello stato utilizzata. Se è impostata l'opzione indicare *User-Encoded* tra le opzioni di *State Machine Processing* del sintetizzatore la codifica utilizzata è quella definita nel file Verilog.
In Figura 7.1 si mostra il diagramma dagli stati ricavato dallo strumento State Machine Viewer. Una simulazione del circuito, eseguita utilizzando il test bench riportato di seguito, è mostrata in Figura 7.2. Per analizzare al meglio il comportamento della FSM è preferibile visualizzare durante la simulazione anche lo stato della stessa. Poiché lo stato non è un ingresso o un'uscita della FSM esso non viene visualizzato automaticamente. Per visualizzare lo stato si selezioni *UUT* (il nome dato al sistema da testare nel test bench) nella finestra *Workspace* del software *ModelSim*. Nella finestra *Objects* si selezioni il segnale da visualizzare che in questo caso è *ac_state*. Si faccia click con il tasto destro del

mouse e si esegua l'azione *Add* | *To Wave* | *Selected items*. E' necessario eseguire nuovamente la simulazione facendo click sulla finestra Wave di ModelSim e eseguendo il comando di menu *Simulate* | *Run* | *Restart* e quindi *Simulate* | *Run* | *Run –all.*

```
`timescale 1ns/1ps
module FSM_moore_tb();

parameter period=30;
reg ck,r,din_tb;
wire dout_tb;

FSM_moore UUT(.clk(ck),.reset(r),.din(din_tb),.dout(dout_tb));

always
begin
ck=0;
#(period/2);
ck=1;
#(period/2);
end

initial
begin
r=1;din_tb=1'b0; #period;
r=0;din_tb=1'b0; #period;
r=0;din_tb=1'b1; #(period*5);
r=0;din_tb=1'b1; #(period*5);
r=0;din_tb=1'b0; #(period*1);
r=0;din_tb=1'b1; #(period*2);
$stop;
end
endmodule
```

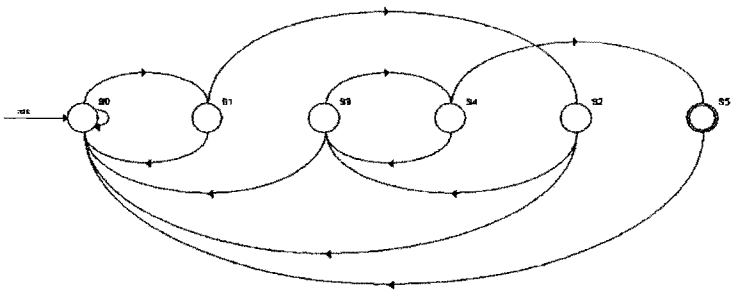


Figura 7.1: FSM di Moore. Diagramma a bolle del riconoscitore di sequenza.

Figura 7.2: FSM di Moore. Simulazione funzionale del riconoscitore di sequenza.

La stessa FSM può essere descritta utilizzando due soli blocchi procedurali. Per questo tipo di descrizione il blocco che descrive i registri effettua l'elaborazione dello stato prossimo nonché l'aggiornamento dello stato.

```
module FSM_moore (clk,reset,din,dout);
input clk,reset,din;
output dout;
wire clk,reset,din;
reg dout;
reg [2:0] ac_state;                          // declare present state
parameter [2:0]                              // Binary state coding
S0=3'b000,S1=3'b001,S2=3'b010,
S3=3'b011,S4=3'b100,S5=3'b101;

always @(posedge reset or posedge clk)       // state register and logic for next state
begin
        if (reset)
                ac_state <= S0;
        else
                case (ac_state)
                        S0: ac_state <= (din) ? S1 : S0;
                        S1: ac_state <= (din) ? S2 : S0;
```

```
                        S2: ac_state <= (din) ? S3 : S0;
                        S3: ac_state <= (din) ? S4 : S0;
                        S4: ac_state <= (din) ? S5 : S3;
                        S5: ac_state <= (din) ? S0 : S0;
                        default: ac_state <= S0;
                endcase
end

always @(ac_state)                              // Comb. Moore logic for output
begin
case (ac_state)
S0: dout <= 1'b0;
S1: dout <= 1'b0;
S2: dout <= 1'b0;
S3: dout <= 1'b0;
S4: dout <= 1'b0;
S5: dout <= 1'b1;
default: dout <= 1'b0;
endcase
end
endmodule
```

7.2 Mealy finite state machine

La macchina a stati descritta precedentemente può essere anche realizzata mediante una descrizione di tipo Mealy. La tabella di transizione per questo circuito è mostrata in Tab. 7.2. La descrizione in Verilog mediante due blocchi procedurali è mostrata di seguito. La codifica dello stato utilizzata è di tipo Gray.

	Stato / Uscita	S0	S1	S2	S3	S4
Ingresso	0	S0 / 0	S0 / 0	S0 / 0	S2 / 0	S0 / 0
	1	S1 / 0	S2 / 0	S3 / 0	S4 / 0	S0 / 1

Tabella 7.2: FSM di Mealy per un riconoscitore di sequenza.

```
module FSM_mealy (clk,reset,din,dout);
input clk,reset,din;input clk,reset,din;
output dout;
wire clk,reset,din;
reg dout;

reg [2:0] ac_state;                             // declare present state

parameter [2:0]                                 // Gray state coding
S0=3'b000,
S1=3'b001,
S2=3'b011,
S3=3'b010,
S4=3'b110;

always @(posedge reset or posedge clk)          // state register and logic for next state
begin
        if (reset)
                ac_state <= S0;
        else
                case (ac_state)
                        S0: ac_state <= (din) ? S1 : S0;
                        S1: ac_state <= (din) ? S2 : S0;
                        S2: ac_state <= (din) ? S3 : S0;
                        S3: ac_state <= (din) ? S4 : S2;
                        S4: ac_state <= (din) ? S0 : S0;
                        default: ac_state <= S0;
                endcase
end

always @(ac_state,din)                          // Comb. Mealy logic for the output
begin
case (ac_state)
S0: dout <= 1'b0;
S1: dout <= 1'b0;
S2: dout <= 1'b0;
S3: dout <= 1'b0;
S4: dout <= (din) ? 1'b1 : 1'b0;
default: dout <= 1'b0;
endcase
end
endmodule
```

Lo strumento State Machine Viewer consente di visualizzare la FSM risultante dalla sintesi. Il risultato è visibile in Figura 7.3. Come anticipato la FSM di Mealy utilizza uno stato in meno della corrispondente descrizione di tipo Moore. Il funzionamento, invece è simile.
A patto di anticipare gli ingressi di un colpo di clock il comportamento è lo stesso se si considerano i valori assunti dall'uscita in corrispondenza del fronte del clock. In alcuni casi la FSM di Mealy causa delle transizioni spurie sul'uscita che non si protraggono sino al fronte del clock successivo e quindi non risultano un problema per l'implementazione. La Figura 7.4 mostra la simulazione funzionale e quella post sintesi della FSM. La simulazione di una FSM di Mealy risulta, ad un occhio non abituato, più leggibile se effettuata post sintesi. Infatti, i ritardi del circuito mostrano il segnale di uscita che varia dopo il fronte del clock ed evidenziano chiaramente il valore dell'uscita sul fronte del clock. Al contrario, in una simulazione funzionale l'uscita varia in corrispondenza del fronte del clock rendendo leggermente più difficile la lettura del comportamento del circuito.
La simulazione post sintesi mostra anche lo stato della FSM. Se si esegue la sintesi del codice Verilog mostrato in precedenza ricavare il nome dei segnali che rappresentano lo stato può essere molto complicato. Se si desidera visualizzare tali stati è preferibile indicare esplicitamente al sintetizzatore che si vogliono preservare i nomi di alcuni segnali a valle della sintesi. Ciò è possibile utilizzando delle apposite direttive di sintesi che vengono aggiunte al codice Verilog come dei commenti, ma prima del punto e virgola che conclude l'istruzione considerata. Ad esempio per preservare il nome dello stato nella FSM di Mealy si utilizza:

```
reg [2:0] ac_state  /* synthesis preserve */ ;
```

Il commento comincia con la parola chiave 'synthesis' che segnala una direttiva di sintesi e continua con 'preserve' che è l'azione richiesta. La direttiva di sintesi è precedente al punto e virgola che termina la dichiarazione dello stato.
Direttive simili possono essere utilizzate per segnali di tipo wire. La direttiva in questo caso è 'keep'.

```
wire [2:0] <signal_name>  /* synthesis keep */ ;
```

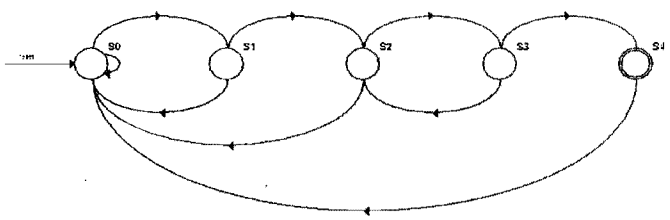


Figura 7.3: FSM di Mealy. Diagramma a bolle del riconoscitore di sequenza.

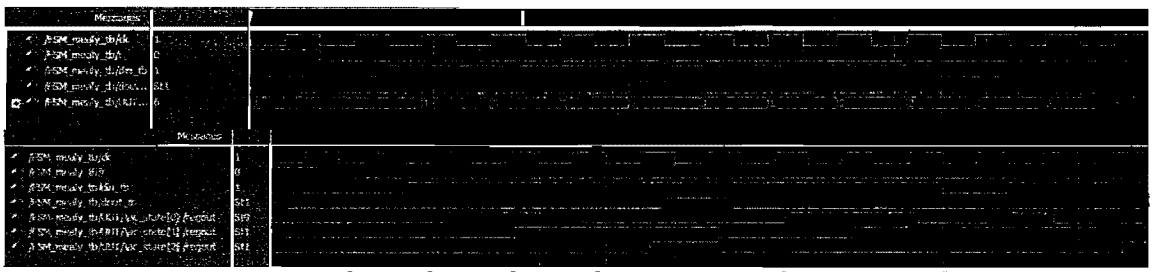


Figura 7.4: FSM di Mealy. Simulazione funzionale e post sintesi del riconoscitore di sequenza.

7.3 Macchine a stati con uscite sincronizzate

7.4 Macchine a memoria finita

Le macchine a memoria finita sono una descrizione particolarmente semplice che può essere utilizzata per macchine a stati le cui uscite dipendono solo dall'ingresso e da un numero preciso di ingressi passati ed uscite passate.
Lo schema di principio di una FSM descritta come macchina a memoria finita è in Figura 7.5. Il circuito è composto da due registri a scorrimento che memorizzano gli ultimi ingressi e le ultime uscite. L'uscita del circuito è una funzione combinatoria dei valori di ingresso e dei valori memorizzati nei registri a scorrimento.
Un circuito che si realizza agevolmente mediante una macchina a memoria finita è il riconoscitore di sequenza.
Supponiamo di avere un ingresso a due bit e di voler riconoscere la sequenza di ingressi {'11','00','11','00','01',10','10','10'}. La descrizione Verilog è mostrata di seguito ed utilizza alcuni costrutti avanzati.

In primo luogo si utilizza un vettore bidimensionale di segnali di tipo reg. L'istruzione che definisce 8 segnali a due bit, denominandoli *mem*, è:

```
reg [1:0] mem [7:0];
```

La definizione di vettore bidimensionale è molto compatta ma ha numerose limitazioni che devono essere attentamente considerate. Ad esempio l'istruzione mostrata definisce un vettore bidimensionale denominato *mem*, ma non è lecito accedere a tutto il vettore. L'unica operazione lecita è l'accesso ad un singolo elemento del vettore. In altre parole non è lecito un costrutto del tipo *if (mem==0)*, ma è lecito *if (mem[1]==0)*.
Quando si descrive il reset asincrono di *mem*, sarebbe necessario azzerare separatamente gli 8 segnali vettoriali (*mem[0],mem[1],....,mem[7]*). Per evitare la prolissità di tale descrizione il reset è stato descritto utilizzando il costrutto *for*, utilizzabile nei blocchi procedurali, ed un segnale di tipo intero. Le definizione di intero è:

```
integer k;                              // define an integer value for the loop
```

mentre il costrutto for è utilizzato come:

```
for (k=0; k<8 ; k=k+1)
       mem[k] <= 2'b00;                 // reset the whole shift register
```

Un ulteriore costrutto for è utilizzato per descrivere lo shift register.

Come mostrato il Verilog dispone del tipo di dato intero e tale tipo di dato può essere anche utilizzato per descrivere circuiti sintetizzabili. Se il tipo intero dovesse essere sintetizzato in un circuito, esso corrisponderebbe ad un segnale su 32 bit. E' quindi molto pericoloso utilizzare gli interi per descrivere i circuiti sintetizzabili. E' sempre preferibile utilizzare dei segnali di tipo *wire* o *reg*, dei quali si possa agevolmente controllare il numero di bit. Il caso mostrato è una delle poche eccezioni in cui l'utilizzo del tipo intero è sicuro poiché è utilizzato solo nella definizione di un ciclo.

```
module FSM_memfin (clk,reset,din,dout);
input clk,reset;
input [1:0] din;
output dout;
wire clk,reset;
wire [1:0] din;
reg dout;
integer k;                              // define an integer value for the loop

// state memory for shift register
reg [1:0] mem [7:0];          // use a bidimensional reg array

parameter [1:0] code0=2'b10, code1=2'b10, code2=2'b10, code3=2'b01,
code4=2'b00, code5=2'b11, code6=2'b00, code7=2'b11;

always @(posedge reset or posedge clk) // shift register
begin
       if (reset)
              for (k=0; k<8 ; k=k+1)
                     mem[k] <= 2'b00;            // resets the whole shift register
       else
              begin
              for (k=7; k>0 ; k=k-1)             // Use for statement to shift
              mem[k]<=mem[k-1];
              mem[0]<=din;
              end
end

always @(mem[7],mem[6],mem[5],mem[4],mem[3],mem[2],mem[1],mem[0])  // Comb. Check the last inputs
begin
if ({mem[0],mem[1],mem[2],mem[3],mem[4],mem[5],mem[6],mem[7]}==
  {code0,code1,code2,code3,code4,code5,code6,code7}) //concatenate operation on mem and parameter
       dout<=1'b1;
 else
       dout<=1'b0;
end
endmodule
```

Il circuito sintetizzato per FPGA CycloneII utilizza 16 registri ed un totale di 17 Logic Elements, di questi 5 sono in modalità combinatoria. La descrizione è quindi risultata in un circuito estremamente compatto. La vista RTL del circuito sintetizzato è mostrata in Figura 7.6. Come atteso il circuito non è altro che un registro a scorrimento con un comparatore tra i dati memorizzati nel registro ed una costante. La simulazione funzionale del circuito realizzato è mostrata in Figura 7.7.

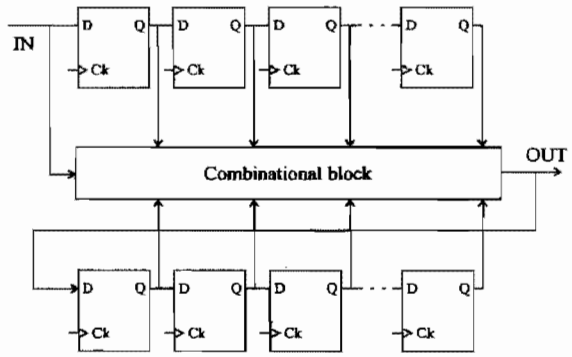


Figura 7.5: Schema di FSM implementata come macchina a memoria finita.

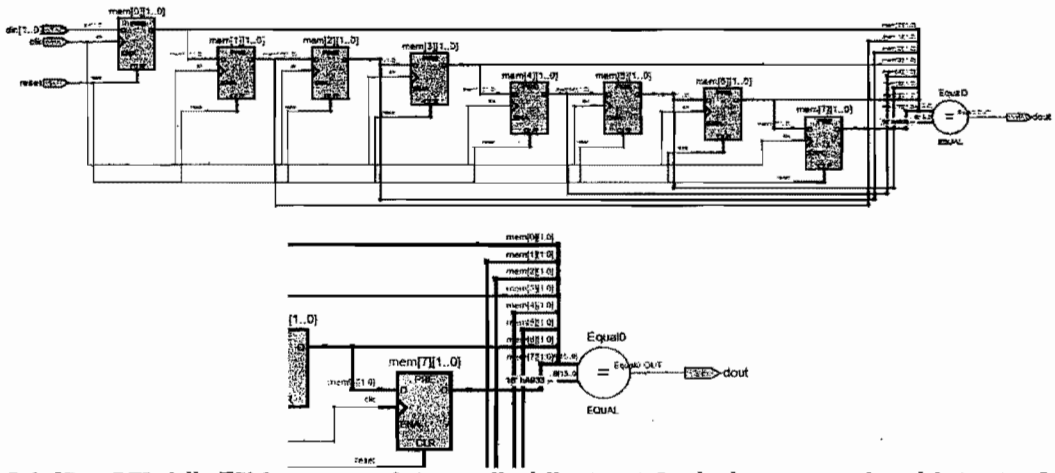


Figura 7.6: Vista RTL della FSM a memoria finita a valle della sintesi. In alto la vista completa del circuito. In basso una vista espansa del comparatore in uscita.

Figura 7.7: Simulazione di un riconoscitore di sequenza implementato come macchina a memoria finita.