Sistemi Elettronici Digitali

Alberto Tibaldi

25 giugno 2009

Indice

Capitolo 1

Introduzione all'elettronica digitale

L'elettronica digitale basa tutta la propria forza sulla semplicità derivata dall'uso di un'algebra binaria: l'algebra di Boole; al fine di studiare l'elettronica digitale, quindi sarà fondamentale un'introduzione ai principi dell'algebra di Boole, che sarà utilizzata per tutta la trattazione.

In termini elettronici, l'algebra di Boole si può pensare come un linguaggio formale in grado di gestire, in sostanza, gli stati di un interruttore.

Gli interruttori possono essere collegati tra loro (ed eventualmente ad altri dispositivi) mediante diverse topologie: in serie, in parallelo, o in altri modi. Tutti questi altri modi vengono rappresentati, mediante un particolare tipo di simboli: le porte logiche.

1.1 Algebra di Boole

L'algebra booleana è basata sulla trattazione logica di due valori attribuibili ad una generica variabile x: $x = 0$, $x = 1$. Lo studio dell'algebra booleana si può riassumere nei seguenti otto assiomi:

$$\boxed{0 \cdot 0 = 0} \quad \boxed{1 + 1 = 1} \quad \boxed{1 \cdot 1 = 1} \quad \boxed{0 + 0 = 0}$$

$$\boxed{0 \cdot 1 = 1 \cdot 0 = 0} \quad \boxed{1 + 0 = 0 + 1 = 1}$$

$$\boxed{\text{Se } x = 0 \Longrightarrow \overline{x} = 1}$$

$$\boxed{\text{Se } x = 1 \Longrightarrow \overline{x} = 0}$$

A partire da questi assiomi, è possibile ricavare le seguenti proprietà:

$$\boxed{x \cdot 0 = 0} \quad \boxed{x + 1 = 1} \quad \boxed{x \cdot 1 = x} \quad \boxed{x + 0 = x}$$

$$\boxed{x \cdot x = x} \quad \boxed{x + x = x} \quad \boxed{x \cdot \overline{x} = 0} \quad \boxed{x + \overline{x} = 1} \quad \boxed{\overline{\overline{x}} = x}$$

Valgono inoltre le seguenti proprietà, per quanto riguarda l'algebra booleana:

- Proprietà commutativa:

$$\begin{cases} x \cdot y = y \cdot x \\ x + y = y + x \end{cases}$$

- Proprietà associativa:

$$\begin{cases} x \cdot (y \cdot z) = (x \cdot y) \cdot z \\ x + (y + z) = (x + y) + z \end{cases}$$

- Proprietà distributiva:

$$\begin{cases} x \cdot (y + z) = x \cdot y + x \cdot z \\ x + y \cdot z = (x + y) \cdot (x + z) \end{cases}$$

- Proprietà combinativa:

$$\begin{cases} x \cdot y + x \cdot \overline{y} = x \\ (x + y) \cdot (x + \overline{y}) = x \end{cases}$$

- Proprietà "Absopition":

$$\begin{cases} x + x \cdot y = x \\ x \cdot (x + y) = x \end{cases}$$

- Teoremi di De Morgan:

$$\boxed{\overline{x \cdot y} = \overline{x} + \overline{y}} \quad \boxed{\overline{x + y} = \overline{x} \cdot \overline{y}}$$

$$\boxed{x + \overline{x} \cdot y = x + y} \quad \boxed{x \cdot (\overline{x} + y) = x \cdot y}$$

- Proprietà "Consensus":

$$\boxed{x \cdot y + y \cdot z + \overline{x} \cdot z = x \cdot y + \overline{x} \cdot z}$$

$$\boxed{(x + y) \cdot (y + z) \cdot (\overline{x} + z) = (x + y) \cdot (\overline{x} + z)}$$

1.2 Introduzione al progetto di circuiti combinatori

In elettronica digitale esistono due filoni: i circuiti di tipo combinatorio, ossia con n ingressi, m uscite, in cui le uscite hanno valori dipendenti esclusivamente dal valore degli ingressi, e circuiti sequenziali, in cui le uscite hanno dipendenza anche dagli stati in cui si trova il circuito; questi ultimi vengono realizzati mediante un sistema di reazione, che riesce a far evolvere la macchina introducendo nel sistema una memoria della storia passata degli stati.

Come si lavora, in pratica, per sintetizzare circuiti elettronici digitali? Innanzitutto, la prima, fondamentale cosa da fare è capire qual è il problema, quindi esprimerlo mediante un formalismo in grado di fornire in maniera semplice una strada per la sintesi: la tavola di verità.

Una volta scritta la tavola di verità, è necessario scrivere la funzione logica in una forma facile da sintetizzare; a tal scopo, esistono sostanzialmente due strategie, finalizzate a esprimere la funzione canonica:

1. Dalla colonna delle uscite si studia quando la funzione ha uscita unitaria, asserita, pari a "1"; in tal caso, la forma canonica si esprime in termini di somme di prodotti, o "somme di minterm";

2. Dalla colonna delle uscite si studiano tutti i casi in cui la funzione ha uscita pari a "0"; in tal caso, la forma canonica si esprime in termini di prodotti di somme, o "prodotti di maxterm"; si noti un fatto: se prima si introducevano i termini come si "trovavano nella riga" corrispondente all'uscita unitaria, ora ciascun termine, sommato, va invertito.

Dalle funzioni in queste maniere ricavabili è possibile, con semplicità, sintetizzare circuiti logici semplicemente sostituendo ai simboli logici le porte logiche, e collegando dei fili a seconda delle indicazioni della formula.

Si noti un fatto: spesso le funzioni espresse in forma canonica non sono ottimali, osisa richiedono molti componenti in più rispetto a circuiti in grado

di realizzare la stessa funzione logica; spesso quindi si utilizzano, a partire dalla tavola di verità, tecniche in grado di semplificare notevolmente le espressioni rispetto a quelle canoniche, riducendo di conseguenza il numero di componenti da utilizzare; il più famoso di questi metodi consiste nell'uso delle mappe di Karnaugh.

1.2.1 VHDL

Al fine di progettare un sistema digitale, è possibile usare, come si dirà frequentemente nel resto della trattazione, metodi fondamentalmente manuali, basati sull'uso della teoria che verrà almeno in buona parte presentata, o metodi informatici, basati sull'uso di linguaggi di descrizione dell'hardware. I due linguaggi di questo genere più utilizzati ai giorni nostri sono VHDL e Verilog.

Si noti che si parla di "linguaggi di descrizione" e non di "linguaggi di programmazione": come si vedrà meglio in seguito, VHDL (il linguaggio che verrà utilizzato per il resto della trattazione) è un linguaggio nato (per quanto poi si sia evoluto anche in altre direzioni) per descrivere sistemi elettronici digitali. Assieme al linguaggio di descrizione, in seguito, sono stati introdotti altri strumenti informatici ed elettronici i quali, basandosi sulla descrizione proposta dal linguaggio, permettono la simulazione (a componenti ideali e pure reali, come si vedrà) del sistema, e addirittura la realizzazione pratica mediante tecnologie di vario tipo (processo di "sintesi").

Si noti che questo tipo di linguaggio non è una soluzione definitiva, una soluzione in grado di eliminare il mestiere del progettista: è sempre e comunque necessario avere le idee chiare riguardo il sistema che si intende progettare, prima di realizzarlo (o farlo realizzare dal sintetizzatore); nella fattispecie, i grandi passi da seguire sono i seguenti:

- Descrivere su carta o comunque mediante diagrammi a blocchi il sistema, mediante tecniche che verranno esposte in seguito;
- Ragionare su come si intende realizzare l'hardware in questione, ad esempio se mediante porte logiche o blocchi di tipo differente quali flip flop o altro;
- Descrivere nella maniera più prossima possibile, per quello che VHDL permette, il circuito ideato, in modo da effettuare in seguito i processi di simulazione e sintesi.

1.3 Progetto di circuiti combinatori

Sia per quanto riguarda i circuiti combinatori sia quelli sequenziali, esistono tecniche informatiche in grado di descrivere e sintetizzare, "in pratica", i circuiti logici di qualsiasi tipo. Al fine di poter tuttavia implementare metodi migliori per la sintesi, e comunque di poter correggere facilmente eventuali errori o mancate minimizzazioni, un buon ingegnere deve essere in grado di sintetizzare manualmente (nei limiti della complessità) un circuito; le motivazioni dietro questo fatto sono sostanzialmente due dunque:

1. Un sintetizzatore di circuiti logici non è mai un software "completo"; bisogna sempre essere in grado di poterlo migliorare, in modo da ottenere circuiti sintetizzati migliori rispetto a quelli precedenti;

2. Un sintetizzatore di circuiti logici di cattiva qualità non è in grado di raggiungere una soluzione ottimale "da solo": è necessario introdurre alcuni "suggerimenti", descrivendo il circuito in modo da renderlo "simile" a quello finale; essendo "pigro", il sintetizzatore cercherà di migliorare il progetto, ma senza cambiarne lo scheletro; supponendo ad esempio di voler realizzare un circuito logico, inserendo esclusivamente porte logiche nella descrizione ci si può aspettare di trovare un circuito dotato di sole porte logiche, anche se magari ottimizzato rispetto a quello descritto, in uscita; introducendo elementi di tipo differente, quali ad esempio multiplexer o simili, ci si può aspettare di aver a che fare con circuiti, ottimizzati, contenenti elementi di questo tipo.

Al fine di studiare circuiti di tipo combinatorio e metodi per sintetizzarli, ricordiamo ancora una volta la definizione: un circuito combinatorio è un circuito con m ingressi, n uscite, in cui esistono n funzioni che lavorano al variare delle 2^m combinazioni degli ingressi.

I passi per il progetto di un circuito sono i seguenti:

1. Definire le specifiche del progetto, ossia capire esattamente "cosa devo fare";

2. Formalizzare le specifiche, ossia calcolare la tavola di verità, eventualmente organizzando una gerarchia tra i vari blocchi costituenti il progetto;

3. Ottimizzare il progetto, cercando di ridurre il numero di porte logiche necessarie;

4. Si mappa il progetto finora realizzato in una tecnologia disponibile (porte logiche del tipo a disposizione);

5. Si verifica il corretto funzionamento del sistema.

I punti 3 e 4 sono piuttosto critici: al fine di ottimizzare e selezionare una certa tecnologia, sarà introdotta, come vedremo, una serie di tecniche di progett, atte a realizzare in differenti modi gli stessi sistemi. L'ottimizzazione si effettua innanzitutto manualmente con il metodo degli implicanti principali della mappa di Karnaugh, per poi usare semplificazioni legate ai teoremi precedentemente proposti, in termini di "teoremi fondamentali dell'algebra di Boole".

Definite le funzioni e semplificate, mediante i vari teoremi di De Morgan o simili, è possibile scegliere uno di due approcci:

- Bottom-up: si costruisce l'intero sistema partendo da piccoli blocchi collegati direttamente tra di loro, facendo crescere passo-passo il sistema;

- Top-down: si parte da un'idea completa, che fornisce immediatamente l'idea del progetto intero, per poi sviluppare solo in seguito i blocchi contenenti il sistema. Spesso, per sistemi digitali complessi, questo tipo di sistema risulta essere il migliore.

1.3.1 Mappatura tecnologica

Supponendo di aver basato il progetto sulla sintesi mediante mappa di Karnaugh, si è ottenuta un'espressione della funzione logica da sintetizzare; quello che a questo punto può capitare è il fatto di avere a disposizione solo porte logiche di un certo tipo: NOR, NAND, o simili; per questo motivo, esistono tecniche semplici in grado di "mappare" la propria funzione logica in altre, equivalenti, ma costituite da sole porte di tipo NOR o NAND ad esempio, o in altri. Si parla soprattutto di logiche negate poichè, come si può studiare dalle implementazioni reali delle tecnologie CMOS, è possibile realizzare solo funzioni di tipo AOI (And Or Invert), quindi aver a che fare con famiglie logiche di questo tipo è molto comune, nelle tecnologie moderne.

Vediamo quali sono i procedimenti per mappare una generica funzione, mediante questi due tipi di tecnologie:

- Mappatura a porte NAND:

  - Una porta AND viene mappata in una NAND semplicemente introducendo un inverter in serie all'uscita;

  - Una orta OR viene mappata in una porta NAND semplicemente introducendo un inverter prima di ciascun ingresso;

- Mappatura a porte NOR: la mappatura è del tutto uguale a quella per le porte NAND, come ora vedremo:
  - Una porta OR viene mappata in una NOR semplicemente introducendo un inverter in serie all'uscita di quest'ultima;
  - Una porta AND viene mappata in una porta NAND introducendo, prima di ciascun ingresso, un inverter.

1.3.2 Implementazione a multiplexer

Un multiplexer (noto anche come MUX) è un blocco combinatorio in grado di selezionare uno solo di m ingressi, ossia, di m ingressi, crea a una singola uscita un collegamento; un segnale di controllo seleziona l'ingresso che si intende collegare all'uscita, e lo collega.

Tipicamente, un multiplexer ha n bit dedicati al segnale di controllo, e $m = 2^n$ segnali di dato, ossia m possibili ingressi, uno dei quali sarà collegato all'uscita Y.

Un'implementazione di un multiplexer 2 to 1 può ad esempio essere la seguente:

$$Y = \overline{S} \cdot I_0 + S \cdot I_1$$

Come funziona questo circuito? si ha un meccanismo di decodifica del segnale di controllo, regolato in questo caso da un inverter, e una serie di porte AND, le cui uscite si collegano a una OR. Sostanzialmente, dati i segnali di selezione, al fine di poter selezionare solo uno dei segnali di ingresso, è necessario effettuare un'operazione di "splitting", di "suddivisione", cosa effettuabile con un decoder, ossia un circuito in grado di "decodificare" un certo segnale. Il fatto di mandare ciascun segnale di selezione decodificato in porte AND, il cui altro ingresso è un segnale di ingresso, permette di selezionare, mediante tutti questi segnali "splittati", un singolo segnale, ossia l'ingresso che si intende selezionare. Questa tecnica si può generalizzare: introducendo in ingresso un decoder per i segnali di controllo, collegando le uscite di questo a un'AND e le uscite delle AND a degli OR, è possibile generalizzare la costruzione logica del multiplexer. Come implementare un decoder sarà visto in seguito.

Si noti che, a partire da multiplexer a "pochi" ingressi, è possibile realizzarne di più "grossi":

Ogni segnale di controllo, tendenzialmente, viene usato per comandare i segnali di controllo di un "livello" di multiplexer, in modo da selezionare il segnale "a livelli", fino a ottenere in uscita, di tutti gli ingressi, quello interessato[1].

Una volta descritto e implementato un multiplexer, studiamo un modo di realizzare, a partire da essi, generici circuiti combinatori. Una volta scritta la tavola di verità, la si ordina in modo da avere gli ingressi in ordine crescente; generalmente, a questo punto, è possibile suddividere in maniera facile la funzione logica in due parti: una con gli ingressi che "cominciano per zero", una con gli ingressi che "cominciano con uno"; il fatto di aver detto "dividere" non è casuale: un multiplexer può servire per "dividere" e "scegliere" quale delle due metà della tavola di verità realizzare, con funzioni logiche di tipo diverso, a seconda della relazioni.

Per chiarezza, riassumiamo come si deve procedere:

1. Si ordina la tavola di verità per ingressi;
2. Si divide la tavola di verità fino a ottenere sotto-tavole di verità a due ingressi; quale delle sotto-tavole di verità si utilizzerà, lo sceglieranno i segnali di controllo dei multiplexer;
3. Si studiano gli ingressi delle sotto-tavole di verità, in modo da riconoscere in essi funzioni logiche semplici: AND, OR; NAND, NOR, XOR, EXNOR, e così via;
4. Si realizza il circuito sintetizzato a partire da queste idee, unendo i fili nella maniera suggerita dalla tavola di verità così suddivisa e organizzata.

1.3.3 Teorema di espansione di Shannon

Un teorema fondamentale per quanto concerne l'algebra di Boole, utilissimo soprattutto in ambito di sintesi di circuiti logici, è la cosiddetta "espansione di Shannon", affermante il fatto che:

$$f(w_1; w_2; ...; w_n) = \overline{w}_1 \cdot f(0; w_2; ...; w_n) + w_1 \cdot f(w_1; w_2; ...; w_n)$$

Questo procedimento, naturalmente, è re-iterabile:

[1] Un altro modo di realizzare un multiplexer è basato sull'uso di buffer tri-state; trattandosi di tecniche difficili da gestire sotto il punto di vista del sincronismo, tuttavia, queste sono solite essere ignorate.

$$\overline{w}_1 \cdot f(0; w_2; ...; w_n) + w_1 \cdot f(w_1; w_2; ...; w_n) = \overline{w}_1 \cdot \overline{w}_2 \cdot f(0; 0; w_3; ...; w_n) +$$

$$+\overline{w}_1 \cdot w_2 \cdot f(0; 1; w_3; ...; w_n) + w_1 \cdot \overline{w}_2 \cdot f(1; 0; w_3; ...; w_n) + w_1 \cdot w_2 \cdot f(1; 1; w_3; ...; w_n)$$

Iterando n volte questo metodo su quest'espressione, si otterrebbe, semplicemente, l'espressione in somma di prodotti della funzione f.

Questo teorema trova applicazione nei metodi di sintesi di circuiti combinatori, nel seguente senso: si sceglie una variabile di riferimento (a tentativi, cercando di fatto la soluzione "meno costosa" tra tutte), calcolando i cofattori relativi ad essa, ossia i termini $f_{w1,w2}$ e così via, relativi a ciascuna variabile w_i.

Sostanzialmente, questo metodo di espansione permette di utilizzare una variabile w_i come segnale di controllo per un multiplexer, i cui ingressi di dato saranno i cofattori relativi.

I cofattori si calcolano considerando la funzione f, sostituendo "0" e "1" alla variabile di riferimento, semplificando, ove possibile, l'espressione: una funzione "AND" con uno "0" si semplificherà in "0", una funzione "OR" con un "1" si semplificherà in "1", e così via.

1.3.4 Look-Up Tables

Una tattica molto particolare, in ambito di sintesi di sistemi combinatori, èl'uso di particolari blocchi in grado di realizzare funzioni logiche qualunque, detti "look-up table". Una look-up table in pratica è una memoria, in grado di contenere tutte le configurazioni di zeri e uni di una certa tavola di verità, riproducendo in uscita la combinazione associata ad un certo ingresso. Eccitando dunque questo blocco con un certo ingresso, è possibile avere in uscita ciò che è contenuto all'indirizzo corrispondente alla configurazione di ingresso, ottenendo di fatto un blocco combinatorio estremamente versatile.

Ci sono alcune "indicazioni" per l'uso in un circuito di una LUT:

1. Il numero di LUT nel circuito è da minimzzare: si tratta di blocchi molto costosi da realizzare, quindi, di fatto, meno se ne usano e meglio è, per quanto siano semplici da usare!

2. Spesso, capita di dover sintetizzare circuiti in cui gli ingressi della funzione logica sono in numero maggiore rispetto a quelli della LUT tecnologicamente realizzabile. In questi casi, può tornare molto utile il

teorema di espansione di Shannon, in grado di "separare" dalle funzioni una variabile (che diventerà la variabile di riferimento), in modo da realizzare con una o più LUT i cofattori, quindi usare una o più LUT per "agganciare" tra loro i vari "pezzi di funzione". Trucco spesso conveniente è quello di cercare una variabile "sempre presente" nelle varie sottofunzioni, e, se possibile, fare in modo che i cofattori siano uno l'opposto (il NOT) dell'altro, in modo da semplificare ulteriormente il circuito e ridurre ulteriormente il numero di LUT da utilizzare.

1.3.5 Decoders

Un altro metodo si sintesi di circuiti combinatori è basato sull'uso di decoders.

Un decoder è un blocco combinatorio in grado di accettare un certo insieme di ingressi, codificati in qualche maniera, e di abilitare una sola linea in uscita; spesso il circuito è dotato di un segnale di "enable", atto ad attivare o meno il circuito, e stabilire se usare o meno una delle uscite, ossia se abilitarne una o se non abilitarne nessuna.

L'idea alla base del decoder è la seguente:

Sostanzialmente, collegando delle porte AND a $n + 1$ ingressi, dove n è il numero di ingressi del decoder, lasciando l'ultimo per il segnale di enable, e a tutte le combinazioni degli ingressi e dei loro complementari, si riesce ad ottenere questo tipo di funzione. Ciascun ingresso viene complementato; nelle porte AND vengono introdotti i segnali che, se entrambi affermati, faranno uscire "1". Solo un segnale in uscita di tutte le AND può essere affermato: le AND sono introdotte in modo da essere attivate se e solo se tutti i segnali al loro interno sono unitari; usando combinazioni diverse, solo una combinazione sarà interamente affermata. A seconda degli ingressi, dunque, solo una AND sarà attiva. Si aggiunge per ciascuna AND un segnale aggiuntivo, ossia quello di enable: se il segnale di enable è attivo allora una delle AND potrà avere uscita affermata, altrimenti nessuna.

Trattandosi di blocchi abbastanza complicati, esistono metodi semplici per effettuare la sintesi di decoder complessi a partire da blocchi semplici, a più stadi, proprio come nel caso dei multiplexer: volendo realizzare, a partire dal decoder 2 to 4, un decoder 4 to 16, è sufficiente usare un primo decoder, dotato di segnale di enable, che gestirà i successivi stadi; questi verranno "guidati" da due segnali w_0 e w_1, e i loro segnali di enable deriveranno, semplicemente, dalle uscite del primo stadio!

I decoder sono fondamentali, per molti scopi, ad esempio per la realizzazione di memorie: per gestire le interfacce delle memorie vi sono sistemi di decodifica in grado di selezionare quale degli ingressi va aperto, quindi quale delle parole di memoria va usata. In una memoria da un megabyte, ad

esempio, serviranno diverse parole da un bytte, decoder a 20 linee di ingresso e quindi 2^{20} linee in uscita.

Un altro uso particolare dei decoder è la realizzazione di multiplexer: usando un decoder associato (in uscita) a una serie di buffer tri-state (uno per uscita), si ottiene un dispositivo di tipo selettivo:

Si noti che se, usando invece che dei buffer tri-state delle porte AND, le cui uscite confluiscono in un OR, si può ottenere comunque un multiplexer, senza ricorrere ai tri-state.

Un modo generale di realizzare un decoder è il seguente: date 2^n uscite, ciascuna proverrà da una batteria di porte AND; partendo dall'uscita, si va a ritroso, poichè questa batteria "finale" di AND sarà a sua volta controllata da stadi di decodifica di ordine inferiore. Quando si arriva, a suon di batterie di AND, al dover introdurre lo stadio 1 to 2, esso sarà semplicemente un filo in parallelo ad un inverter: esso di fatto "sdoppia" il segnale, dividendolo in sè stesso e nel suo complementare, generando il più elementare degli stadi di decodifica.

Terminata la descrizione del decoder, passiamo ad una descrizione del metodo di sintesi già citato: il metodo di sintesi di circuiti mediante decoder è sostanzialmente il seguente: come si sa, il decoder fornisce, dato un certo numero di ingressi, tutte le possibili combinazioni in uscita; data questa osservazione, l'idea potrebbe essere quella di prendere la tavola di verità, e usare il fatto che un decoder produce un "1" in uscita quando si desidera. Collegando a delle porte OR ciascuna delle uscite interessate, si realizza il circuito.

Proponiamo un esempio banale: supponendo di voler realizzare la funzione logica f, definita come:

$$f(w_1; w_2; w_3) = \sum_{m=1}^{8} m(1; 2; 5; 8)$$

Una volta espressa la funzione in termini di somma di minterm, è sufficiente collegare agli ingressi di una porta OR le uscite 1, 2, 5, 8 del decoder.

1.3.6 Altri blocchi sequenziali

Presentiamo rapidamente alcuni blocchi meno importanti sotto il punto di vista della sintesi di circuiti combinatori, ma comunque molto frequentemente utili per varie ragioni, in modo da fornire quantomeno un'infarinatura del comportamento di tutti i blocchi combinatori.

Demultiplexer

Si tratta di un blocco combinatorio duale al multiplexer: dato un singolo ingresso, e n bit di selezione, il blocco combinatorio semplicemente collega l'ingresso ad una delle 2^n possibili uscite.

Encoder

Dati 2^n ingressi, il blocco produce n uscite, in maniera del tutto duale al decoder. Questo blocco usa una codifica nota come "one-hot encoding": uno solo degli ingressi deve valere "1", tutti gli altri "0"; in altri casi, l'uscita dell'encoder vale "don't care", e non è possibile dire altro.

Priority Encoder

Il priority encoder rappresenta una variante rispetto al precedentemente descritto encoder: si utilizza in sostanza un concetto di priorità, nel seguente senso: a partire dal primo bit a "1", stabilito un senso di priorità, l'uscita dell'encoder è definita; in seguito al primo bit (più significativo) a uno, l'uscita è definita, gli altri ingressi (meno significativi) sono don't care. Come già detto, si tratta di un'estensione del precedente blocco: in questo modo un encoder può solo accettare "combinazioni permesse", rendendo di fatto molto più flessibile il blocco, annullando in sistanza le uscite insensate.

Convertitori di Codici

Senza entrare nello specifico, molto spesso è necessario passare da codifiche di un tipo a codifiche di altri tipi; i convertitori di codici sono blocchi i quali, dato un certo ingresso, restituiscono un'uscita con un significato differente, in termini di codifica, da quella appena entrata, in modo da convertire di fatto la codifica di un numero da una di partenza a una di arrivo, come potrebbero essere binario e BCD, o 7-segment.

Capitolo 2

VHDL per circuiti combinatori

Come già precedentemente visto, una via per sintetizzare circuiti combinatori è quella di partire dalla tavola di verità, quindi mediante procedimenti semplificativi quali mappe di Karnaugh o uso di teoremi dell'algebra booleana si semplificano e si sintetizzano con una certa tecnologia a disposizione.

Quando bisogna progettare sistemi elettronici complicati, metodi "manuali" quali quelli appena citati sono assolutamente impossibili da utilizzare; un buon sistemista digitale deve essere in grado di sintetizzare circuiti con migliaia di transistori e moltissimi componenti. Al fine di fare progetti "seri", sfruttando la facilità di gestione di ciascuna singola porta logica, è possibile utilizzarle come blocchetti fondamentali al fine di costruire blocchi sempre più grandi, blocchi che, una volta collegati in maniera appropriata, costituiranno il sistema elettronico. Si utilizzano dei CAD in grado di sviluppare circuiti complicati; serve una metodologia di tipo "industriale": si deve prima di tutto, pensare al sistema, visualizzare il progetto e comprendere come si intende svilupparlo; dopo, esso va descritto in qualche maniera, mediante il linguaggio di descrizione.

Al fine di effettuare la descrizione del sistema si possono utilizzare mezzi "grafici", quali i vari "Schematic editors", o qualcosa di più fine: un linguaggio di descrizione dell'hardware, quale il già citato VHDL. VHDL è un acronimo per la seguente sigla: VHSIC Hardware Description Language, dove per VHSIC si intende il sotto-acronimo di Very High Speed Integrated Circuits. Come il nome suggerisce, in sostanza, esso è un linguaggio atto a rappresentare e descrivere circuiti che verranno poi integrati (o in realtà anche implementati su schede, come si vedrà in seguito). Nel resto della trattazione si considererà, come mezzo per la descrizione dell'hardware, proprio questo linguaggio, VHDL: mediante il compilatore / simulatore / sintetizzatore, una volta introdotto il codice, è possibile verificare che tutto funzioni, effettuare simulazioni, e occuparsi del design fisico del progetto.

Lavorare con il sintetizzatore, ossia con il software che, a partire dalla descrizione VHDL dell'oggetto fornisce un'implementazione reale (che verrà quindi caricata su di un hardware appropriato, ad esempio su di una FPGA), non è mestiere facile: il sintetizzatore implementa ma con "poca fantasia", nel senso che elabora, con l'hardware e/o la tecnologia a disposizione, un'implementazione in qualche modo "vicina", "simile" a quella descritta nel listato VHDL, per quanto gli sia possibile; se si descriverà un circuito con porte logiche, dunque, ci si potrà aspettare un'implementaziona basata sull'uso di porte logiche; se si descriverà un circuito basato su multiplexer, decoder, o altri blocchi di questo tipo, ci si può aspettare, nell'implementazione, la presenza di componenti di questo genere.

Il VHDL nasce negli anni '80, e solo nel 1987 viene adottato come standard per la IEEE; subisce diverse revisioni, tra cui la fondamentale, del 1993, nota come standard IEEE 1164; inizialmente, esso veniva utilizzato solo per documentazione (ossia per la descrizione di sistemi) e per la loro simulazione; a questo punto dell'evoluzione tecnologica esso è lo strumento fondamentale per il progetto di un sistema elettronico digitale.

2.1 Costrutti di base

Ciascun file VHDL, contenente la descrizione del progetto di un blocco o di un intero sistema digitale, è sostanzialmente suddiviso in tre parti:

- Chiamata di librerie (LIBRARY, USE): si richiamano tutti i pacchetti utilizzati nel progetto. Nel corso della trattazione sarà sufficiente utilizzare le seguenti due righe di codice:

  ```
  LIBRARY ieee;
  USE ieee.std_logic_1164.all;
  ```

  Si noti che VHDL non è case-sensitive, dunque le istruzioni possono essere liberamente inserite con caratteri minuscoli o maiuscoli; si sceglie di attribuire caratteri maiuscoli in modo da evidenziare quali sono le istruzioni e quali i parametri da utilizzare in merito ad esse.

- Definizione del blocco (ENTITY): si definiscono tutti gli ingressi e le uscite di un certo blocco, che si intende rappresentare mediante linguaggio VHDL. In sostanza, mediante ENTITY, è possibile caratterizzare

il blocco con approccio "blackbox": si definiscono tutti i parametri concernenti ingressi, uscite, eventualmente ritardi (per simulazioni, ovviamente: non è possibile "imporre" un ritardo al sintetizzatore, poichè esso utilizzerà hardware a disposizione dotato di un determinato ritardo intrinseco), e altri eventuali parametri riguardanti però esclusivamente l'aspetto "esteriore" del blocco o del sistema;

- Descrizione del blocco (ARCHITECTURE): se mediante ENTITY si definiscono tutti i parametri concernenti l'esterno del blocco, mediante il comando ARCHITECTURE si descrive, mediante tecniche di tipo differente, il comportamento del blocco al suo interno, imponendo determinate relazioni tra i segnali di ingresso, eventuali segnali interni al blocco, e i segnali di uscita. Questa è la sezione dove si definisce dunque il comportamento del dispositivo, sezione che influenzerà in maniera fondamentale il sintetizzatore: a seconda del tipo di istruzioni che verranno inserite (uso di porte logiche piuttosto che di dispositivi come multiplexer o decoder), il sintetizzatore si muoverà in una direzione o un'altra, cercando comunque di "seguire" il codice.

2.1.1 Libreria IEEE

Si consideri in maniera più dettagliata il primo dei tre aspetti finora presentati: l'uso di particolari librerie. Esse sono fondamentali dal momento che, nella versione "base" di VHDL, come tipo per la gestione e creazione di segnali di tipo digitale esiste il "BIT" (o la possibilità di creare array di BIT); questo tipo di segnali è estremamente limitato, al fine di realizzare sistemi reali, dal momento che prevede esclusivamente la presenza di due valori: "0" e "1". Nei sistemi reali, come si può vedere spesso, può esserci la necessità di introdurre dei segnali di tipo "don't care", i cui valori non devono per forza essere ben definiti al fine di ottenere un corretto funzionamento della macchina, permettendo dunque semplificazioni di vario tipo. Non si considera inoltre il fatto che il sistema, per un motivo o per l'altro, possa propagare segnali di tipo "alto" o "basso" in modo "forte", accentuato, ben definito, o in modo "debole", ad esempio a causa di fenomeni parassiti che tolgono tensione al segnale. Altro fenomeno che non si considera può essere quello di non avere definizione del segnale non nel senso "don't care", ma nel senso di avere lo "scontro", l'imposizione sullo stesso punto, di un segnale basso e di uno alto.

Lo standard IEEE 1164 prevede tutte queste possibilità in un nuovo tipo, creato ad hoc al fine di aumentare le potenzialità del VHDL: il tipo standard logic.

```
SIGNAL y : STD_LOGIC;
SIGNAL x : STD_LOGIC_VECTOR(7 DOWNTO 0);
```

Sono stati ora presentate due righe di codice da utilizzare molto frequentemente: le dichiarazioni di segnali. Nella fattispecie, una volta chiamati i pacchetti precedentemente illustrati, è possibile utilizzare questo tipo, contenente una maggior varietà di informazioni rispetto al tipo vecchio, nativamente implementato in VHDL.

Si introduce brevemente dunque il tipo standard logic, esponendo i principali valori che può assumere:

- 0 e 1, ossia i due valori logici "forti", ben definiti, con un significato sostanzialmente analogo a quello di "bit";
- Z (alta impedenza): nel caso il segnale, il "filo", sia collegato a un bus, può essere utile avere la possibilità di introdurre uno stato di alta impedenza; il sintetizzatore eventualmente potrebbe implementare questo tipo di segnale mediante l'uso di buffer tri-state;
- D (don't care): il segnale può assumere valore zero o uno, dal momento che non è fondamentale ai fini del funzionamento finale del sistema;
- U (indefinito): il segnale risulta essere indefinito, ossia non assume alcun valore particolare; ciò è diverso da don't care, dal momento che don't care assume un valore; in questo caso, il valore non è in alcuna maniera definito;
- L (livello logico basso debole): si tratta di uno "0" debole, non ben riconoscibile dall'hardware; il senso della frase "non ben riconoscibile" significa che, in presenza di uno scontro con un altro segnale "forte", esso prevale su quello debole; un esempio di questo tipo di segnali può essere un segnale collegato ad un resistore di pull-down;
- H (livello logico alto debole): analogamente a prima, però per quanto riguarda un "1" logico; dualmente a prima, un esempio di questo tipo di segnali può essere un segnale collegato a un resistore di pull-up;
- X (valore non-valido): il fatto che due valori forti si "scontrino", ad esempio a causa della presenza di un corto circuito che non dovrebbe essere presente tra due componenti del sistema, comporta una non-definizione del segnale di questo tipo;

- W (valore non-valido debole): il fatto che due segnali deboli si scontrino tra loro, provoca un'indeterminazione analoga alla precedente, ma che viene definita per valori esclusivamente deboli.

Questo tipo di nozione è utile soprattutto in termini di simulazione: attribuire a un segnale un determinato valore, in termini di sintesi, non è assolutamente ammissibile: la sintesi è un progetto che, a partire dalla descrizione dell'hardware, produce in qualche modo (su scheda FPGA piuttosto che in termini di maschera per l'integrazione) un supporto fisico implementante il sistema progettato. Dal momento che il sistema è fisico, imporre che un certo segnale abbia un certo valore non ha senso, poichè deve essere fatto dall'utente o dal tester del sistema; attribuire valori ai segnali può essere invece fondamentale in termini di simulazione su calcolatore, dal momento che si può in questo modo prevedere il comportamento almeno qualitativo del sistema al variare di diversi tipi di eccitazione.

Vettori e matrici

Precedentemente è stato proposto un costrutto in grado di realizzare array monodimensionali, ossia vettori costituiti da un certo tipo (standard logic, il tipo prevalentemente utilizzato nell'ambito dei sistemi elettronici digitali); dal momento che le memorie tuttavia, come si vedrà, hanno un aspetto "matriciale", può tornare utile il progetto di un array bidimensionale di elementi, ossia una matrice di standard logic. Non è difficile produrre qualcosa di questo tipo: è sufficiente utilizzare ricorsivamente il comando precedentemente presentato, ottenendo qualcosa del tipo:

```
TYPE RegArray IS ARRAY(3 DOWNTO 0) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL mat : RegArray;
```

Cosa è stato fatto? Beh, qua è stata buttata molta carne al fuoco, carne che andrebbe controllata passo per passo: il comando TYPE permette di definire un tipo nuovo, non presente nello standard IEEE 1164. RegArray è il nome che si attribuisce al tipo: si decide di chiamarlo così, ma un qualsiasi nome sarebbe stato valido, restando nei limiti della sintassi del VHDL; il comando ARRAY permette di definire un vettore (in questo caso di 4 elementi) di vettori di standard logic (di 8 elementi); in questo modo, al fine di accedere a ciascun elemento della matrice, si può utilizzare una sintassi del tipo:

```
mat(i)(j);
```

Dove i permette di scegliere quale degli array di standard logic utilizzare, dunque varia da 0 a 3, mentre j sceglie quale elemento dell'array selezionato considerare, dunque varia da 0 a 8. Per il sintetizzatore, un elemento di questo genere è sostanzialmente una memoria; esso dunque cercherà di ottenere in termini di hardware qualcosa del genere, producendo una memoria di uscita. Al fine di garantire l'accesso a ciascuna cella, sarà necessario eventualmente definire meccanismi di decodifica in modo da attribuire a ciascun elemento un determinato indirizzo, dunque poter realizzare una memoria vera e propria.

2.1.2 Concetti riguardo la ENTITY

Per quanto riguarda il secondo dei punti fondamentali, la cosiddetta ENTITY, ossia il costrutto in grado di descrivere mediante un approccio blackbox il blocco in questione, ci sono alcuni elementi da osservare meglio, con maggior attenzione.

Il primo punto riguarda il comando PORT, tipicamente presente in ciascuna entity: dal momento che in questa sezione è necessario definire la "scatola" del blocco, fondamentale è avere un listato contenente tutti gli ingressi e le uscite presenti nel sistema. PORT permette proprio di elencare questi elementi.

Fondamentalmente, le uscite e gli ingressi devono essere trattati come dei segnali; all'interno dell'ARCHITECTURE verranno definiti i vari collegamenti logici, in modo da completare la descrizione del progetto. Esistono quattro tipi di collegamenti, per quanto riguarda il PORT di una ENTITY:

- IN : si tratta della definizione di un segnale in ingresso al blocco;
- OUT: si tratta della definizione di un segnale in uscita dal blocco;
- INOUT: si tratta della definizione di un segnale che può essere sia in uscita sia in ingresso da un blocco; in realtà, si considera in questo ambito un segnale spesso utilizzato come ingresso e uscita per un altro blocco, dunque, per evitare incompatibilità, spesso quando si usa questo comando il sintetizzatore introduce buffer tri-state;
- BUFFER: si tratta di un'opzione analoga a INOUT, con la differenza che il sintetizzatore non tende a introdurre buffer di tipo tri-state, con-

siderando tutto "già a posto" in termini di interconnessioni con altri blocchi.

2.1.3 Concetti riguardo la ARCHITECTURE

La ARCHITECTURE è la parte che riguarda la descrizione del comportamento logico del circuito, precedentemente definito con approccio blackbox nell'ambito della ENTITY. Si può dire sotto certi punti di vista che in essa sia il cuore del progetto: la descrizione del comportamento reale del circuito è qui contenuta.

Si tratta di una sezione abbastanza importante: qui è possibile da un lato costruire blocchi, o mediante i comandi basilari del VHDL o mediante il riutilizzo di blocchi precedentemente ideati. Ciò può essere utile, dal momento che definire l'intero progetto in un singolo file può essere molto scomodo: il debug nonchè eventuali migliorie risulterebbero essere molto più difficili, utilizzando un approccio di questo tipo. Fondamentale dunque è l'uso di COMPONENTS, ossia di elementi già precedentemente presentati, e che ora vengono esclusivamente istanziati in modo da costituire un macroblocco.

Sostanzialmente, al fine di istanziare un COMPONENT, è sufficiente utilizzare un costrutto molto simile a quello di una ENTITY; esso permette di definire tutti i parametri che esso contiene, anche se esso è effettivamente descritto in un altro file del progetto, dunque è possibile istanziarlo effettivamente mediante il seguente comando (si propone un esempio pratico):

```
Multiplexer1 : mux2to1 PORT MAP ( A, B, C );
```

"Multiplexer1" è il nome che si attribuisce al componente istanziato, di tipo "mux2to1"; PORT MAP permette, come il nome del comando suggerisce, di "istanziare", di "mappare" i segnali alle porte di ingresso del componente.

Al termine del capitolo verrà descritto un esempio pratico completo, che permetterà di comprendere meglio tutto ciò che è stato finora detto.

2.2 Sintesi e simulazione

In seguito alla descrizione dell'hardware mediante VHDL, come già detto, è possibile descrivere un sistema hardware, in modo da poterlo simulare e sintetizzare. La funzione di simulazione è stata introdotta quasi nativamente al linguaggio, e verrà descritta in seguito maggiormente nel dettaglio. La

funzione di sintesi è stata introdotta al fine di utilizzare la descrizione proposta per il sistema per una realizzazione materiale del progetto. Una volta nati i processi di integrazione su semiconduttore o le logiche FPGA, disporre di un mezzo semplice e al contempo efficace di descrivere un sistema poteva essere importante per la produzione su scala industriale di un certo prodotto: una FPGA come si vedrà in seguito è un dispositivo logico programmabile mediante un calcolatore elettronico, dispositivo sul quale è possibile caricare le caratteristiche di un sistema realizzandolo di fatto fisicamente su di esso. Molte aziende produttrici di integrati, inoltre, distribuiscono ai progettisti che lavorano con loro modelli PSpice o VHDL dei loro dispositivi, in modo da poter produrre un listato in grado di essere immediatamente realizzato su maschera, per quanto riguarda i progetti più costosi, da realizzare su scala industriale. Il sintetizzatore, dunque, è in grado di realizzare su di un qualche supporto un'implementazione fisica dell'hardware, analogamente a come un utente fa con una breadboard e componenti discreti.

Il simulatore lavora a livello puramente logico: a partire dalla descrizione contenuta nel listato VHDL, è possibile effettuare una simulazione del comportamento del circuito, senza introdurne alcuna implementazione reale. Esistono costrutti in grado di introdurre eventuali ritardi, in modo da poter rendere più realistica la simulazione, rendendola dunque non solo "funzionale", bensì "comportamentale", prossima a quella del circuito reale.

Il processo di sintesi è piuttosto complicato e non riguarda questo tipo di trattazione; si sceglie tuttavia di introdurre alcuni concetti ulteriori riguardo la simulazione, in modo da comprenderla maggiormente ed evidenziare alcune differenze con software quali PSpice.

2.2.1 Simulazione

Studiando l'elettronica analogica, molto spesso risulta fondamentale l'uso di un simulatore in grado di fornire un andamento delle uscite di un generico circuito. L'elettronica digitale risulta essere una semplificazione dell'elettronica analogica: anzichè poter assumere qualsiasi valore, un segnale digitale può assumere sostanzialmente due valori. Il fatto di introdurre questa notevole semplificazione (senza considerare gli aspetti elettrici contenuti nell'elettronica digitale, considerando dunque che tutte le porte logiche siano della stessa famiglia e quindi si "parlino" correttamente) permette diverse semplificazioni anche per quanto riguarda il modello complessivo, e la tecnica di simulazione: se da un lato, nell'elettronica analogica, è necessario continuamente presentare nuovi valori, dal momento che essi sono reali, dunque effettuare una simulazione a tempo continuo (in prima approssimazione continuo, dal momento che comunque si discretizza il tempo reale per poter

ricondurre le equazioni differenziali dei componenti a equazioni alle differenze finite), dall'altro è sufficiente accorgersi di un fatto: la variazione di un determinato segnale. Di fatto, non è necessario effettuare continuamente la simulazione, ossia simulare tutti gli istanti di tempo, bensì può essere sufficiente effettuare una simulazione del comportamento del sistema solo in determinati istanti, dove "avviene qualcosa", ossia dove, in seguito a determinate variazioni di un certo segnale, si abbiano condizioni in grado di far nascere fenomeni importanti.

Questo tipo di simulazione è detta "simulazione ad eventi": ogni qual volta si abbia un cambio di stato o di uscita per quanto riguarda il sistema, il simulatore introduce in una sorta di "coda ordinata", dove il parametro ordinante è il tempo in cui deve avvenire un certo nuovo evento, le caratteristiche dell'evento. Una volta eseguito e terminato il primo evento, il simulatore entra nella coda, verifica le caratteristiche e gli istanti in cui il nuovo evento deve arrivare, quindi lo esegue; questo nuovo evento a sua volta potrà introdurre uno o più nuovi eventi nella cosiddetta "event list", eventi che potranno essere eseguiti, e così via.

Questo tipo di simulazione è, come si vede, molto differente da quella classica di PSpice, a tempo continuo: in seguito a una certa eccitazione, fornita per esempio da un ingresso, si ha un inserimento di un evento in una lista, contenente esclusivamente informazioni riguardo i suddetti; questi fanno variare le caratteristiche del segnale, la risposta del sistema. Non si considera punto per punto la soluzione numerica di una certa equazione, bensì una tecnica in grado di modificare le caratteristiche di un segnale, appuntare eventuali future caratteristiche da presentare e con quale priorità, dove la priorità è dipendente dall'istante temporale in cui l'evento deve apparire. Il tempo di simulazione non "scorre effettivamente" per il calcolatore, nel senso che serve esclusivamente ad attribuire una determinata priorità ai diversi elementi della coda, in modo da fornire un ordine.

Come anticipato, è possibile introdurre un ritardo per ciascuna porta logica: un'analisi di tipo funzionale fornisce un andamento puramente comportamentale del codice, senza introdurre in alcun modo ritardi; è possibile introdurre ritardi mediante un costrutto di questo tipo:

```
x <= (a XOR b) AFTER 5 ns;
```

Il comando "¡=" rappresenta l'assegnazione dell'uscita di un'elaborazione di un certo numero di segnali al segnale x. Senza il comando "after" quest'assegnazione sarebbe pressochè immediata (come vedremo tra breve in realtà

si introduce un piccolo ritardo simbolico, per un certo motivo); specificando mediante AFTER un determinato tempo, si introduce un ritardo nella simulazione, in modo da simulare l'andamento di una porta reale.

Per quanto riguarda la simulazione, a seconda di ciò che si intende fare e del software da utilizzare, si possono progettare segnali di prova. Al fine di effettuare una simulazione è necessario simulare un determinato ingresso, in modo da simulare l'eccitazione del sistema. I software spesso presentano metodi per la realizzazione grafica di segnali di questo tipo; una tecnica per realizzare questi segnali è basata sull'uso di after:

```
sig <= '0', '1' AFTER 10 ns, '0' AFTER 20 ns, '1' AFTER 40 ns;
```

Si notino due fatti:

- Ciascun ritardo parte dal tempo nullo; ciascun AFTER si riferisce dunque allo stesso istante, quindi, nell'esempio proposto, si parte da $t = 0$, si va a $t = 10$, passan 10 ns e si arriva a $t = 20$, e così via. Ciascun "after" si riferisce al tempo nullo.
- Questo tipo di istruzioni non è particolarmente gradita dal sintetizzatore: imporre un ritardo alle porte, per un programma che deve selezionare da un elenco un certo insieme di dispositivi, non è assolutamente possibile. Per questo motivo si sceglie di suddividere il progetto in molti files: dedicando un file esclusivamente al segnale di simulazione, si permette di evitare il fatto che il sintetizzatore provi a sintetizzarlo, bloccando dunque il processo in malo modo.

Per quanto riguarda l'argomento della modellizzazione dei ritardi, ne esistono sostanzialmente di tre categorie:

- Ritardo inerziale: "after" è un ritardo inerziale, dal momento che si tratta di un'inerzia introdotta dal progettista. In sostanza, il ritardo inerziale identifica il ritardo di un circuito logico fisico, pur rimanendo nella simulazione; quel che può capitare è il fatto che il ritardo con inerzia sia tale da non far considerare eventuali variazioni di determinati segnali al simulatore, dal momento che le condizioni per riconoscerle non sono presenti a causa proprio di questi ritardi. Ciò può portare dunque alla cancellazione di eventi;

- Ritardo di trasporto: se il ritardo inerziale è molto realistico, nel senso che potrebbe portare alla cancellazione di eventi a causa di mancata presenza di alcune condizioni sui segnali causate dai delay, il ritardo di trasporto semplicemente "trasla" tutti i segnali di un certo ritardo, senza però avere l'effetto "reale" che aveva il ritardo precedentemente descritto;

- Delta delay: un ritardo nullo. In una simulazione di tipo funzionale dove non si specificano ritardi, ci si può aspettare che, contemporaneamente alla variazione di un segnale, si abbia l'esecuzione di un evento. Ciò non è vero: deve sempre e comunque essere mantenuto il rapporto di causa-effetto, anche per quanto riguarda una simulazione puramente comportamentale del sistema elettronico in questione. Non è possibile che vi sia una variazione istantanea, dal momento che la event list non potrebbe funzionare: un evento deve essere inserito nella event list, al fine di essere poi catturato ed eseguito. Quello che si fa in pratica è introdurre, nel caso di eventi teoricamente "istantanei", un cosiddetto delta delay, ossia un ritardo minimo, impercettibile, puramente simbolico, atto a mantenere intere le relazioni di causa-effetto.

Ciascun evento può contenere uno o più comandi; la cosa sempre vera, comunque, è che ciascuno di essi viene eseguito in maniera concorrente rispetto agli altri, ossia "contemporaneamente": ogni evento è un insieme di azioni che vengono eseguite allo stesso istante di tempo. Si consideri ad esempio il fatto che in un evento, in qualche modo, vengano eseguite queste espressioni:

```
A <= B;
C <= A;
```

Cosa capita? Si supponga che $A = 0$, $B = 1$; al termine del processo, si avrà $C = 0$, $A = 1$. L'ordine potrebbe suggerire il fatto che prima A acquisisca il valore "1", quindi solo dopo C acquisisca il valore di A, ossia "1". In questo caso, dal momento che le istruzioni sono concorrenti, il loro ordine non conta: al termine dell'evento tutti i collegamenti tra segnali (si ricorda il fatto che il simbolo <= indica il collegamento tra due segnali, come un utente potrebbe fare su di una breadboard) saranno eseguiti al contempo e a partire da tutti gli istanti di tempo. Si presenterà e spiegherà eventualmente meglio in seguito questo concetto, che comunque va sempre tenuto a mente.

Capitolo 3

Programmable Logic Devices (PLD)

Finora abbiamo studiato diverse tecniche di sintesi per funzioni combinatorie di vario genere. Fino a metà degli anni '80 l'elettronica lavorava con componenti prettamente discreti; la nascita e l'enorme sviluppo delle tecnologie di integrazione ha creato la possibilità di realizzare circuiti programmabili dall'utente.

Integrando, e utilizzando software di vario genere, è possibile, dato dal produttore un blocco contenente funzioni di qualsiasi tipo in numero elevato, realizzare facilmente progetti di circuiti di qualsiasi genere; ciò presenta vantaggi e svantaggi:

- I circuiti progettati si possono provare facilmente, e correggere dunque altrettanto facilmente;
- Generalmente, in una logica programmabile, si hanno prestazioni inferiori rispetto a quelle che si potrebbero ottenere con circuiti integrati realizzanti direttamente il progetto; molti dei transistori delle porte logiche programmabili inoltre spesso non sono utilizzati: la maggior parte di essi viene usata per la programmabilità della scheda, non per la realizzazione del progetto.

Qual è dunque l'idea? Qualcuno (i produttori di PLD) ci fornisce la possibilità di avere, all'interno di una fettina di silicio, tutte le porte logiche, sequenziali o combinatorie, in grado di realizzare un sistema elettronico digitale completo. L'utente ha la possibilità di creare le interconnessioni tra i vari componenti già esistenti, in modo da scegliere cosa si deve utilizzare e cosa no.

Esistono diverse tecniche per la gestione delle varie interconnessioni:

- Utilizzare dei fusibili, ossia dei dispositivi in grado di far passare la corrente entro un certo limite di intensità, per poi "fondersi" in modo irreversibile, di fatto "programmando" un'interconnessione mancante, un circuito aperto;
- Utilizzare degli antifusibili, ossia dei dispositivi duali ai fusibili: fino a quando non si introduce un'opportuna differenza di potenziale ai capi degli antifusibili, non si ha un contatto; da quando si è introdotta questa tensione, il dispositivo si chiuderà irreversibilmente, permettendo al segnale di passare;
- Utilizzare transistori associati a memorie di tipo EEPROM o Flash (meglio descritti in seguito), programmando la memoria mediante zeri o uni;
- Collegando un transistore a una memoria RAM, funzionante in modo simile a prima.

Si hanno collegamenti di questo tipo:

Come funziona di fatto un sistema logico? Data una linea collegata ad una resistenza di pull-up, si vuol far passare un segnale su di essa; i segnali derivano dalle linee verticali, ossia quelle non collegate alla "linea principale"; i collegamenti, per essere gestiti facilmente, vengono realizzati mediante transistori di tipo MOS.

L'idea è quella di utilizzare una connessione di tipo wired-or, realizzando una generica funzione f del tipo:

$$f = \overline{I_1 + \overline{I_2}}$$

Data una matrice di questo tipo, costituita da linee come queste, in un circuito integrato dunque si devono semplicemente posizionare i transistori nella maschera in modo da definire la funzione logica che si intende progettare; volendo realizzare circuiti programmabili, in ogni incrocio vi sarà il transistore che, anzichè al potenziale di riferimento del sistema, sarà collegato a un elemento programmabile (in modo permanente o meno), a sua volta collegato allo 0 V.

la logica appena presentata permette di realizzare una logica di tipo NOR; una volta realizzata, mediatne i teoremi di De Morgan, è possibile definire piani (ossia rappresentazioni bi-dimensionali) di tipo AND, OR, NAND, e simili, ossia insiemi di possibili funzioni logiche selezionabili e utilizzabili in seguito a un processo di programmazione.

Presensiamo a questo punto le principali strutture programmabili, accennando ai metodi utilizzati per programmarle, e ai loro limiti e vantaggi. In

tutte queste, l'idea fondamentale è quella di avere due "piani", due insiemi di funzioni logiche di un certo tipo, collegato a piani di funzioni logiche dello stesso o di un altro tipo, ad esempio AND e OR; le varie tecnologie ora descritte si differenziano dal fatto che solo alcuni elementi, o piani, possono essere programmati, o entrambi.

PROM

Per PROM si intendono le Programmable Read Only Memory: il primo piano è non-programmabile, e spesso costituito da funzioni logiche di tipo AND (fixed AND plane); le interconnessioni tra questo piano e il piano OR in cascata a esso sono programmabili, dunque solo la "seconda parte" risulta essere programmabile.

Dal momento che il piano AND è "fixed", in uscita da questo si avranno, in sostanza, dei minterm già "scelti" dal progettista della PROM; sostanzialmente, selezionando nel piano OR quali di questi minterm vanno sommati, si ottiene la funzione di uscita. Il piano AND è praticamente un decoder: in uscita da esso si hanno minterm decodificati, che poi verranno "filtrati" dal piano OR programmabile. Una PROM, quindi, si può in sostanza pensare come una memoria i cui ingressi sono gli indirizzi nei quali sono dislocati i dati, e le cui uscite sono i dati stessi.

PAL

Le PAL, o Programmable Array Logic, sono dispositivi duali alle PROM: il piano AND è infatti interamente programmabile, ma il piano OR è fixed. Questo fatto presenta vantaggi e svantaggi:

- Uno svantaggio è il fatto che, pur essendovi una buona flessibilità, potrebbe non essere possibile mappare qualsiasi funzione, dal momento che si è vincolati dal numero di ingressi, finito;
- Un vantaggio è il fatto che, a parità di complessità, si possono realizzare più minterm, ossia si possono usare più ingressi per realizzare la medesima funzione;
- Un grosso vantaggio è il fatto che si possano realizzare funzioni "su più livelli": in una ROM non si ha la possibilità di avere tra gli ingressi una delle uscite, cosa fattibile invece in ambito PAL; si noti tuttavia che ciò che è stato detto non consiste nell'introdurre, nel vero senso della parola, una reazione: una reazione introduce infatti nel circuito un elemento di memoria degli stati precedenti, ciò semplicemente "introduce nuovi

ingressi" nel sistema, in modo da aumentare la flessibilità del sistema, senza introdurvi per forza una memoria o un controllo. Le uscite del piano OR possono dunque essere re-introdotte, in modo da essere usate come ingressi aggiuntivi, ovviamente al prezzo di aumentare i tempi di latenza (dal momento che, per re-introdurre il segnale, è necessario che esso venga elaborato dalla logica!);

- Nelle PAL moderne vengono introdotti talvolta elementi aggiuntivi con memoria, ossia uno o più stadi sequenziali, in modo da, eventualmente, permettere la memorizzazione di stati, ottenendo dunque sistemi di tipo sequenziale.

PLA

Le PLA, o Programmable Logic Array, sono le strutture più flessibili tra quelle finora analizzate, e dotate delle maggior risorse, che permettono di sintetizzare circuiti combinatori anche piuttosto complessi. Nelle PLA, sia il piano AND che il piano OR sono programmabili, da qui la flessibilità dei blocchi. Le PLA presentano vantaggi, ma anche svantaggi, come ora esporremo:

- Sono in grado di implementare funzioni/equazioni dei tipi più svariati, risultando dunque essere ben superiori alle ROM; tutti i minterm sono realizzabili e sommabili, dunque superando anche i limiti delle PAL;
- Introducendo sul'uscita uno XOR programmabile è possibile prelevare l'uscita e complementarla, se si intende farlo;
- Uno svantaggio è il fatto che vi è un numero limitato di product terms realizzabili, limitato dal numero di righe dell'array;
- Spesso non vi è la possibilità di "portare indietro" le uscite per fare strutture multilivello, poichè la struttura è già intrinsicamente flessibile.

Una PLA può essere implementata sia con piani AND-OR sia con piani NOR-NOR: usando funzioni di tipo NOR e complementando il tutto con degli inverter, è possibile infatti ottenere funzioni logiche di tipo qualunque.

Le PLA si possono implementare in diversi modi: usando transistori programmabili mediante fusibili (o, più comunemente negli ultimi anni, antifusibili), o mediante transistori di tipo EEPROM (in serie ai transistori si introduce una memoria EEPROM/Flash/RAM).

CPLD

I CPLD, o Complex Programmable Logic Devices, sfruttano dei blocchi PAL-like (detti "macrocelle") per realizzare strutture complesse basate sull'interconnessione di blocchi più semplici; il grosso limite legato a queste strutture è la lunghezza delle linee: grandi lunghezze implicano grandi capacità parassite, quindi grandi latenze.

Negli anni '90 è stato introdotto, al fine di migliorare le logiche, un circuito noto come "sense amplifier": si tratta di un amplificatore analogico differenziale atto a studiare le differenze tra un certo segnale e uno di riferimento.

Il problema di questo circuito, divenuto fondamentale, è il grosso consumo di corrente che provoca: esso consuma sempre e molto, cosa che rende i circuiti poco appetibili per applicazioni di tipo portatile.

LUT

Le LUT o Look-Up Tables, sono strutture già descritte che sfruttano la seguente idea (almeno, in una delle possibili realizzazioni che si possono usare): si può utilizzare un multiplexer a due vie (anche di più, per realizzare strutture più complesse), con gli ingressi collegati a due segnali esterni, x e y; si può fare una cosa di questo tipo:

A seconda di come si collegano x e y al multiplexer, si possono ottenere funzioni di qualsiasi tipo. Usando un multiplexer a più vie, si possono realizzare funzioni logiche ancora più complesse, fino a ottenere un dispositivo in grado di realizzare funzioni di fatto qualsiasi: una look-up table!

FPGA

Le FPGA, o Field Programmable Gate Array, si basano sull'unione di due idee: date da un lato le LUT, in grado di creare funzioni generiche, e dall'altro le strutture delle CPLD, dall'unione delle due nascono quelli che al giorno sono gli strumenti più potenti e versatili per il progetto di sistemi elettronici digitali: le FPGA.

Come nelle vecchie CPLD, è possibile utilizzare in uscita sia un blocco puramente combinatorio che uno sequenziale, in modo da introdurre quindi elementi in grado di mantenere la memoria degli stati precedenti del sistema. Molto spesso si usa una struttura tipo "Manhattan", ossia composta da linee "tra di loro ortogonali": Manhattan, come d'altra parte Torino, ha una struttura "a scacchiera" come organizzazione stradale, che ha ispirato molti progettisti di FPGA. Questo tipo di tecnologia presenta un grosso problema: spesso, per circuiti complicati, non è possibile effettuare l'operazione di

sintesi, dal momento che, una volta che si crea un'interconnessione tra due blocchi, essa non può essere poi riutilizzata. Per questo motivo molti dei gate di una FPGA non possono essere usati in un progetto: un buon progetto riesce a sfruttare tendenzialmente il 60 % dei gate presenti in una scheda.

Duranete il processo di sintesi un algoritmo stabilisce quali e quante interconnessioni creare; questo è detto "algoritmo di routing", e deve essere particolarmente ottimizzato, se si vuole sfruttare correttamente il potenziale della scheda.

Solitamente si fa in modo da non realizzare "vie" molto lunghe: una volta che si "blocca una via" si preclude di fatto la possibilità, a molti gate, di essere utilizzati; una soluzione a questo tipo di inconvenienti è quella di adottare una struttura fortemente gerarchizzata, cercando di privilegiare l'elaborazione "locale", ossia cercando di usare interconnessioni tra blocchi "vicini" disponendoli in modo intelligente, facendo in modo da usare meno possibile le "vie principali", ossia quelle che collegano blocchi "distanti" tra loro. Si introduce dunque una struttura multilivello, atta a massimizzare la localizzazione dell'elaborazione.

Come si programmano questi circuiti? Beh, vediamo quali sono le due possibilità:

- OTP: One Time Programmable; mediante la tecnologia ad antifusibile, si programma, facendo cadere sui punti "giusti" della scheda, una tensione sufficientemente elevata.

- Quando il progetto è in fase embrionale, conviene usare schede programmabili basate su memorie RAM, in modo da poter correggere in modo semplice e indolore la realizzazione circuitale del progeto. Esistono tecniche, quali la JTAG, che permettono, mediante software, di programmare schede di questo tipo.

Non volendo usare software, si usa programmare, specialmente in ambito OTP le schede, in ambito industriale, mediante una memoria flash collegata a macchinari che, con interfacce, programmano serialmente diversi circuiti allo stesso modo.

Alcune osservazioni sui circuiti programmabili

Le FPGA rappresentano un'enorme innovazione nell'elettronica poichè, grazie al fatto che contengono milioni di gate in un singolo integrato, rappresentano un sistema elettronico completo, programmabile da un utente, a costi molto ridotti.

Se l'utente intende vendere un prodotto, tuttavia, non può sempre usare una FPGA come base del proprio sistema: si tratta di circuiti comunque abbastanza costosi, dunque spesso inaccessibili al mercato. Volendo vendere ai grandi mercati, integrare il processo può essere una buona idea; il processo di integrazione, tuttavia, richiede l'uso di maschere di integrazione, molto costose da realizzare (si parla dell'ordine dei 100.000 euro); volendo vendere milioni di circuiti l'integrazione è senza dubbio la via più conveniente, anche sotto il punto di vista delle prestazioni, ma può essere inaccessibile al piccolo produttore. Per integrare si usa spesso il metodo delle "standard cell": la sylicon foundary fornisce librerie VHDL/Spice con i modelli dei propri dispositivi realizzabili, e l'ingegnere deve solo usarli al fine di realizzare il progetto; l'algoritmo di routing gestirà le interconnessioni.

Esiste una via di mezzo tra l'uso di FPGA e l'integrazione: poichè i costi delle maschere sono molto elevati, le aziende forniscono maschere "già pronte" per i componenti più importanti quali porte logiche, flip-flop e simili; l'utente da qui deve solo fornire indicazioni riguardo le interconnessioni, pagando dunque solo i costi delle maschere per le interconnessioni, ptogettando una matrice di gate (gate array); i costi si abbattono, poichè solo parte delle maschere va realizzata, tagliando anche di cinque volte i costi non ricorrenti.

Questo tipo di tecnica è meno costosa dell'integrazione vera e propria, ma presenta anche uno svantaggio: essendo il gate array già progettato, spesso capita di avere in un chip molte più porte di quante ne servano. Dei gate array robusti e con un numero di livelli sufficiente alla realizzazione di circuiti abbastanza complessi può anche valere 20000 euro, quindi relativamente poco.

Capitolo 4

Circuiti aritmetici

Finora sono stati introdotti circuiti logici combinatori in grado di realizzare funzioni di vario tipo, senza considerare un particolare fatto: quello secondo cui gli ingressi finora considerati erano insignificanti, non dotati di un particolare significato aritmetico; ciò che si intende fare da ora è sostanzialmente considerare il fatto che gli ingressi sono dati organizzati, nella fattispecie sono numeri binari espressi con una particolare codifica. Verranno dunque analizzate diverse funzioni aritmetiche realizzabili mediante differenti circuiti, in diverse maniere.

4.1 Sommatori

Quella dei sommatori, ossia dei circuiti logici (combinatori) in grado di realizzare la più semplice delle funzioni aritmetiche, probabilmente è una delle classi più ampie di circuiti di questo genere. Il problema sostanzialmente è il seguente: dati due bit in due differenti ingressi, volendo ottenere un circuito in grado di realizzare la somma dei due bit in termini di "bit di somma" S e "bit di riporto" C, l'idea più semplice potrebbe essere la seguente:

Questa è la più semplice delle realizzazioni per un circuito di questo tipo. Per ottenerla, e per ottenere le altre possibili rappresentazioni, potrebbe essere ad esempio possibile calcolare la tavola di verità, utilizzare il metodo di sintesi mediante mappa di Karnaugh, e a seconda della scelta degli implicanti ottenere altre espressioni del circuito logico. Blocchi di questo tipo, che da due bit ricavano somma e carry, sono comunemente noti col nome di "half adder".

Un'estensione di questo tipo di blocchi potrebbe essere il cosiddetto "full-adder": un circuito in grado di prelevare parole di bit e sommarle tra loro non può essere basato su half adder, poichè essi non prevedono un fatto: che

il numero in ingresso sia già dotato di un carry. Ciò che si potrebbe dunque fare è costruire una tavola della verità in grado di prevedere anche l'introduzione di un bit di carry, dunque mediante mappe di Karnaugh trovare un'espressione in grado di fornire, in termini di porte logiche elementari, il carry per il bit successivo ($i+1$-esimo) e la somma per il blocco presente (i-esimo); mediante questo procedimento, si può dimostrare che:

$$s_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = x_i \cdot y_i + x_i \cdot c_i + y_i \cdot c_i$$

Date queste funzioni è possibile sintetizzare, riciclando eventualmente il blocco "half adder", un circuito per il full-adder. L'idea sostanzialmente è quella di considerare, in uscita dagli half adder, l'OR dei vari carry, ottenendo qualcosa di questo tipo:

Questo circuito è abbastanza complesso, ma la cosa positiva è che si può fare di meglio, studiando in maniera più idonea l'idea nascosta dietro al concetto di carry, di riporto. In un full adder, sostanzialmente, potrebbero capitare due cose, per quanto riguarda il carry, da quello in ingresso ad esso a quello che deve produrre: il carry potrebbe rimanere inalterato, a causa della somma nel blocco, o potrebbe essere modificato. Quello che si fa è una sorta di "cambio di base", per la funzione di generazione del carry: anzichè considerare l'attuale casistica, si scompongono le casistiche in "carry generato", ossia modifica rispetto al carry attualmente presente, o "carry propagato", ossia mantenimento dell'attuale carry. Si considerano dunque due segnali, G_i e P_i, che permettono, come si può dimostrare, di riscrivere l'espressione del carry in uscita da un blocco full-adder, C_o, nel seguente modo:

$$C_o = G + P \cdot C_i$$

Ossia, o si ha la generazione di un carry (OR), o si ha la propagazione del carry già presente nell'ingresso del blocco. In questo modo si può semplicemente estendere il concetto di half adder precedentemente prodotto, introducendo una logica di questo tipo:

Questa idea sarà molto interessante, e in seguito ripresa e sviluppata per presentare alcuni tipi di sommatori veloci. Per ora comunque ci ha aiutati a semplificare l'espressione di partenza, rendendola più compatta.

4.1.1 Ripple-carry adder

Si propone a questo punto la prima idea di sommatore su più bit: il ripple-carry adder:

Questo è il metodo più semplice per realizzare una somma, ossia un'operazione di addizione tra due vettori di bit. Ciò che si può intuire immediatamente, come si vedrà in seguito, è il fatto che circuiti di questo tipo sono estremamente lenti e inefficienti: per quanto la realizzazione richieda blocchi relativamente semplici, per generare ogni riporto è necessario attendere che tutti gli altri riporti vengano prima generati, cosa che di fatto potrebbe essere eccessivamente fastidiosa al fine di implementare una somma in sistemi elettronici veloci. Se non è richiesta una velocità particolare, tuttavia, questo sommatore è ottimo e poco costoso da realizzare.

4.2 Sottrazioni

Finora è stata realizzata una tecnica e una circuiteria in grado di effettuare addizioni binarie; ci si potrebbe a questo punto chiedere come realizzare l'operazione duale, ossia le sottrazioni. Finora si è attribuito ai vettori di bit trattati il significato di "numeri senza segno", "unsigned"; ai fini di fare la sottrazione con numeri di questo tipo, di fatto, servono circuiti estremamente complicati e mai utilizzati. Fondamentalmente, i passi da seguire per ottenere qualcosa di questo tipo sono i seguenti:

1. Sottrarre il sottraendo N dal minuendo M;
2. Se $M \geq N$, il risultato è positivo, dunque non si han problemi di alcun tipo;
3. Se $M < N$, il risultato sarà:

$$D = M - N + 2^n$$

   In tal caso, si prende il risultato dell'operazione, e lo si tratta sottraendo 2^n, ottenendo così il risultato desiderato.

Questo algoritmo funziona ma è molto complicato da realizzare, circuitalmente parlando: l'hardware necessario è molto oneroso.

Quando si vogliono trattare sottrazioni, è buona cosa utilizzare numeri con segno, basandosi su notazioni particolari, ideate a partire da considerazioni sui "complementi" dei numeri binari. Cosa significa ciò? Beh, la somma da fare è un'operazione facile, dunque un'idea potrebbe ad esempio

essere quella di definire un metodo di conversione dei numeri da positivi a negativi in modo che, per fare una differenza, sia possibile semplicemente complementare il sottraendo, dunque fare una normale somma di numeri apparentemente unsigned. Avendo a disposizione una circuiteria semplice per il processo di complementazione ed avendo già nozioni riguardo la somma di numeri unsigned, un processo di questo genere risulterebbe molto semplice.

Senza affrontare nei dettagli questi fatti, si considerano procedure semplici per effettuare i processi di complementazione (senza introdurre la teoria nascosta dietro di essi):

- Per fare il "CA1" (complemento a 1), è sufficiente complementare ogni bit: ciascuno zero diventa uno, e ciascun uno diventa zero. Un procedimento del genere non è di fatto molto utilizzato.
- Per fare il "CA2" (complemento a 2), si può usare uno dei seguenti procedimenti:
  - Da destra verso sinistra, ossia dal LSB al MSB, lasciare invariati i bit finchè non si incontra un primo "1"; questo viene ancora lasciato inalterato, mentre tutti i bit di lì in poi, fino al MSB, dovranno essere complementati; questo procedimento è preferibile quando si effettuano le operazioni "a mente", da un operatore umano.
  - Complementare ogni bit, dunque fare il CA1 del vettore di bit, quindi sommare 1. Questo metodo non è molto consigliabile per un operatore umano, ma è ideale quando si deve realizzare un complementatore mediante sistemi elettronici.

Il complemento a 2 è il metodo più utilizzato nella sistemistica per realizzare sottrazioni: per effettuarle, infatti, è sufficiente complementare a 2 il sottraendo, ed effettuare la solita somma. Questo tipo di rappresentazione, per quanto usata, ha un grosso vantaggio e uno svantaggio che tuttavia non è molto importante:

- Il complemento a 2 è l'unica delle rappresentazioni studiate che non ha una duplice rappresentazione dello zero, ossia il cosiddetto ± 0;
- Si tratta di una rappresentazione asimmetrica, "sbilanciata": si tende ad avere un valor medio dei valori rappresentabili, a parità di bit a disposizione, non nullo, cosa che alla lunga potrebbe dare problemi, quando si effettuano approssimazioni pesanti del numero da elaborare.

4.2.1 Overflow

Si è detto che il complemento a 2 è la miglior soluzione, quando si intende effettuare operazioni aritmetiche con segno. Riguardo le operazioni con numeri signed, esiste sostanzialmente un grosso limite che ci blocca: dati n bit a disposizione, potrebbe capitare che siano necessari, al fine di presentare correttamente il risultato, $n + 1$ bit. In questi casi, per fortuna abbastanza limitati e controllabili, accade che il risultato è privo di significato, avendo un fenomeno detto "overflow". Poichè quando si ha overflow il risultato è privo di significato, è opportuno introdurre un sistema in grado di stabilire la validità del risultato e segnalare eventuali problemi all'utente o al resto del sistema.

Una prima osservazione potrebbe essere la seguente: si può avere overflow sostanzialmente in due situazioni:

- Quando si effettua una somma tra due numeri con lo stesso segno;
- Quando si effettua una differenza tra due numeri di segno opposto.

Una seconda osservazione potrebbe essere la seguente: quando si ha un fenomeno di overflow, la somma di due numeri positivi è negativa, o viceversa la somma di due numeri negativi positiva. Osservando i carry si può trovare tuttavia un risultato molto interessante, in grado di realizzare con estrema semplicità un controllo sulla validità del risultato:

- Se gli ultimi due bit di carry sono uguali, non si ha mai presenza di overflow;
- Se i due bit sono diversi, si ha la certezza di avere overflow.

Ciò permette di introdurre con molta semplicità un segnale di overflow: collegando in uscita ad uno XOR i due ultimi bit (ossia i due MSBits), lo XOR può rilevare una differenza e attivarsi o meno, segnalando all'utente la presenza dell'errore, confermando o smentendo la veridicità del risultato.

Si noti che tutto ciò che è stato appena detto è assolutamente e sempre verificato, a patto di utilizzare, come rappresentazione "signed", quella basata sul complemento a 2. In altri casi queste osservazioni non sarebbero più vere, dunque sarebbe necessario studiare i casi specifici.

4.3 Sommatori veloci

Finora, per effettuare operazioni di somma, abbiamo analizzato una singola struttura, evidenziandone i pregi e i difetti: il ripple-carry adder. Come già

annunciato, il problema di questa struttura è il fatto che essa è estremamente lenta: supponendo di dover realizzare un sommatore a 64 bit da 1 GHz, le porte dovrebbero avere una latenza inferiore ai 4 ps (provare a fare i conti per credere!), quando le migliori tecnologie, per costi elevatissimi, possono arrivare circa a 25 ps. Quello che bisogna dunque fare non è cercare di migliorare la tecnologia delle porte logiche, bensì cercare modi per "rivoluzionare" il circuito, partendo sempre dal full adder, ma evitando di utilizzare metodi di questo tipo.

Un'idea è già stata proposta: l'introduzione di questi segnali G_i e P_i, per ciascun i-esimo full adder. Ciò che si può per ora fare è cercare di migliorare, concettualmente parlando, il ripple-carry adder, introducendo una notazione basata sull'uso di questo nuovo concetto.

Come si può dimostrare, osservando le tavole di verità, G e P possono essere ricavati utilizzando le seguenti espressioni:

$$G = A \cdot B$$

$$P = A \oplus B$$

A partire da queste informazioni, proponiamo alcune idee che permettano di velocizzare l'attuale sistema.

4.3.1 Carry-bypass adder

Una prima idea potrebbe essere la seguente: avendo a disposizione il già noto ripple-carry adder, avendo a disposizione i segnali di propagate, è possibile determinare a priori un fatto: se tutti i P_i per ciascun blocco sono a 1, allora si può dire che il circuito non dovrà influenzare in alcun modo i carry; si può dire per certo che $C_i = C_o$, per l'intero circuito. Per questo motivo, si può dire che far elaborare i carry sia tempo sprecato. Ciò che si può fare dunque è creare un percorso preferenziale, utilizzando un'idea di questo tipo:

Introducendo un multiplexer in cui per un ramo si introduce il classico ripple-carry adder, nell'altro un semplice corto circuito, e come segnale di pilotaggio si usa l'AND di tutti i segnali propagate P_i, se si ha $\prod_i P_i = 1$ si può direttamente "bypassare" il ripple-carry, ottenendo immediatamente il carry finale.

4.3.2 Carry-select adder

Si propone a questo punto la seguente idea: per ogni coppia di bit da sommare si ricavano i due bit detti "generate" (G) e "propagate" (P); il carry-select

adder calcola per ciascun bit da sommare i carry che dovrebbero esservi, mediante la semplice applicazione delle formule precedentemente introdotte, considerando due casistiche differenti, in parallelo: il fatto che il carry in ingresso al sommatore, C_{in}, valga "0" o "1". Mediante un multiplexer, ai cui ingressi sono collegati i carry prodotti con le due possibilità, si selezionano, usando come segnale di ingresso C_{in}, i carry "corretti", che verranno quindi utilizzati per produrre le somme parziali di bit, mediante i soli segnali P e C_i; sostituendo le espressioni precedentemente introdotte nell'espressione della somma di due bit, si può ottenere:

$$S_i = P \oplus C_i$$

Costruendo blocchi combinatori in grado di riprodurre questa funzione logica e unendoli (secondo lo schema a blocchi fornito nell'esercitazione), si può ottenere il sommatore basato sul meccanismo del carry-select.

L'idea sostanziale è quella di "selezionare": si producono dunque in parallelo due differenti uscite, che vengono poi unificate. Il blocco di produzione dei segnali di generate e propagate sono decisamente più veloci rispetto al tradizionale full adder, dunque si riesce a ottenere, rispetto ai sommatori precedentemente presentati, un notevole miglioramento.

Per architetture di questo genere esistono alcune sotto-varianti, che verranno ora presentate in modo quantomeno primordiale:

- Linear carry-select adder: utilizzando blocchetti "tutti uguali", dotati degli stessi ritardi, si ottiene un buon risultato, quello "classicamente" aspettato.

- Square-root-select adder: mettendo blocchi con ritardi atti a sincronizzare tutte le uscite, con "numeri di bit crescenti", si riesce ad ottenere un risultato ancora maggiore: se la velocità nella somma era prima lineare, ora diventa addirittura pari alla radice del numero di bit del sommatore.

4.3.3 Carry-lookahead adder

Ciò che è stato finora fatto sostanzialmente corrisponde in miglioramenti del ben noto ripple-carry adder. Non è stata fatta tuttavia alcuna rivoluzione concettuale nell'idea della somma: si genera qualcosa passo passo, e in qualche modo si cerca di migliorare le prestazioni. Il carry-lookahead adder rappresenta un'architettura molto diversa e molto più veloce rispetto alle precedenti, poichè sfrutta in modo molto più potente le definizioni dei segnali "generate" e "propagate".

L'idea fondamentale è quella di realizzare una logica combinatoria la quale, per quanto complessa, possa determinare "a priori" tutti i carry in partenza, appena introdotto il numero. Sfruttando le definizioni di generate e propagate, nella fattispecie, si potrebbe pensare a qualcosa del genere:

$$c_{i+1} = x_i \cdot y_i + x_i \cdot c_i + y_i \cdot c_i = g_i + p_i \cdot c_i$$

Partendo da $i = 0$:

$$c_1 = g_0 + p_0 \cdot c_0$$

Re-iterando, come si mostra in questo esempio, si può ottenere:

$$c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0)$$

Allo stesso modo si potrebbe ricavare c_3 e i seguenti allo stesso modo.

Implementando una logica combinatoria di questo tipo si potrebbe immediatamente generare tutti i carry, immediatamente. Il problema di questa architettura però risulta essere lampante: la complessità delle porte logiche da usare! Come si può intuire, per un sommatore a 256 ingressi sarebbe necessario usare degli AND a 256 ingressi, o cose del genere; utilizzare porte con un fan-in molto elevato non è positivo, dunque è necessario limitarsi.

Un'alternativa potrebbe essere quella di utilizzare qualcosa del genere, con una struttura di tipo gerarchica: introducendo blocchi complessi, ma in numero limitato, si potrebbe ovviare questo problema almeno in parte, rendendo il tutto più semplice e più facilmente realizzabile.

4.4 Altre funzioni aritmetiche

Abbiamo finora trattato tecniche atte alla realizzazione sostanzialmente di somme (e sottrazioni); analizziamo a questo punto altre operazioni, e le principali tecniche/metodologie atte a realizzarle, in modo da avere una più vasta gamma di operazioni realizzabili.

4.4.1 Incrementatore - Contrazione

Può essere utile, in molte occasioni, avere blocchi combinatori in grado, a partire da un numero in ingresso, di produrne in uscita uno incrementato di un'unità rispetto ad esso. L'idea potrebbe essere semplice: si considera un sommatore, il cui ingresso è sempre "1", mentre l'altro ingresso è occupato dal numero da sommare a "1", ossia da incrementare. Questa soluzione sicuramente funziona, ma non è ottimale: un sommatore richiedere l'uso di

un certo numero di porte logiche, che tuttavia non saranno sempre utilizzate: fissando infatti un ingresso, è possibile che l'uscita non sia più complicata come nel caso di due ingressi variabili, liberi. L'idea di partenza è sicuramente valida, ma si può far di meglio: si potrebbe osservare come funziona, data l'imposizione di un ingresso costante, che non verrà variato (o 0 o 1), la tavola di verità del sistema, dunque notare eventuali semplificazioni; se se ne trovano, dunque si potrà sostituire una porta logica con una meno complessa, come ad esempio un filo o un inverter. Si supponga ad esempio di avere uno XOR, con un ingresso B fissato a "0":

Se A = 1, $A \oplus B = 1$; se $A = 0$, $0 \oplus 0 = 0$; si può dunque dire che, dato B = 0, sull'uscita si avrà sostanzialmente l'ingresso A: lo XOR è trasparente ad A. Nella sintesi, in definitiva, si potrà dire che lo XOR si può rimpiazzare con un semplice filo che collega A all'uscita. Questo processo di semplificazione è noto con il nome di "contrazione". In modo del tutto simile si può realizzare, anzichè un incrementatore, un decrementatore, a partire dal circuito di somma, contratto.

4.4.2 Moltiplicatore / divisore per 2^n

Si vuol dedicare una piccola sottosezione a questo tipo di operazione per evitare di "sparare alla formica con il cannone": un'operazione di questo tipo è molto semplice da realizzare con circuiti di tipo sequenziale, quali il classico registro a scorrimento (shift register); non avendo a disposizione elementi con memoria, tuttavia, l'unica possibilità per effettuare questa operazione è collegare ad un sommatore (o al blocco che necessità l'uscita) in modo opportuno i cavetti, i pin, in modo da "moltiplicare" semplicemente shiftando le cifre, ottenendo moltiplicatori per una potenza di due "fissa"; per avere più versatilità, si usino gli shift register.

4.4.3 Moltiplicatore binario

Oltre a moltiplicazioni per potenze di 2, seppur con maggior difficoltà, è possibile realizzare moltiplicatori per numeri generici, numeri binari qualsiasi. Il problema della moltiplicazione è tuttavia abbastanza complicato, e molte soluzioni "semplici" sono anche molto inefficienti. Verranno qui presentate le idee principali per la realizzazione di moltiplicatori di questo tipo.

- A partire da un insieme di sommatori (half adder e full adder), disposti secondo topologie atte a realizzare somme di prodotti parziali, utilizzando l'algoritmo "classico" (quello che si impara alle elementari) per fare il prodotto di due numeri, si moltiplica ciascuna cifra del secondo

fattore per tutte le cifre del primo; per ciascuno di questi "prodotti parziali" si otterrà un numero che, per ciascuna cifra da moltiplicare parzialmente, verrà shiftato a sinistra di una posizione, collegando in maniera opportuna i vari sommatori;

- Se i numeri sono senza segno, ciò è sufficiente; per numeri signed, di solito si complementano i fattori negativi, e si procede come prima;
- Variante più veloce è quella di usare una topologia di tipo carry-save: una catena somma i "riporti parziali" ottenuti ad ogni giro, dunque i ritardi ottenuti dai carry vengono ridotti al solo ritardo della catena.

4.5 Cenni a rappresentazioni numeriche alternative

Senza voler approfondire l'argomento, già affrontato normalmente in un corso di Fondamenti di Informatica, si vuole accennare all'esistenza di due particolari formati per la rappresentazione dei numeri binari: "fixed point" e "floating point".

- Fixed point: il numero è diviso in una parte intera ed in una parte decimale (frazionaria), suddivise simbolicamente da una "virgola". La tecnica di conversione del numero è quella di considerare la seguente relazione:

$$V(B) = \sum_{i=-K}^{n-1} b_i 2^i$$

  Il problema di questi numeri è il seguente: al momento di usarli è necessario denormalizzarli/normalizzarli, ma allo stesso modo: una normalizzazione errata provocherebbe errori enormi nella fase di calcolo. La circuiteria dovrà essere appropriata per questo genere di informazioni.

- Floating point: si tratta di numeri formati da un bit di segno, e da un certo numero di bit di esponente e di mantissa. Sono formati molto più difficili da gestire rispetto ai numeri fixed point, ma comunque fondamentali quando la rappresentazione deve essere in grado di presentare grossi range di numeri.

Capitolo 5

Circuiti sequenziali

La seconda grande categoria di circuiti e sistemi elettronici digitali, dopo quella dei circuiti combinatori, è quella dei circuiti sequenziali. In sistemi combinatori non è possibile memorizzare in alcun modo stati, dati o qualsiasi altro elemento. Dal momento che in un sistema elettronico si introduce una retroazione, in esso si introduce di fatto un elemento di memoria. In maniera più "sistematica" sono stati sintetizzati negli anni alcuni elementi fondamentali atti a memorizzare dati: i latch e i flip-flop.

5.1 Principali blocchi circuitali con memoria

Un elemento con memoria deve essere in grado, come suggerisce la parola, di memorizzare una certa informazione, conservandola nel tempo. I principali elementi utilizzati saranno i D-flip-flop , ossia flip-flop dotati di una particolare configurazione. Si presenteranno a questo punto le principali idee nascoste dietro a questi blocchi, in modo da poterli conoscere in modo almeno qualitativo.

5.1.1 SR-Latch

Un SR-Latch è un dispositivo costituito da questo set di porte logiche:

Tendenzialmente si può vedere che S (set) imposterà il valore dell'uscita positiva Q_a; reset, o R, dualmente, imposterà il valore dell'altra uscita. Si noti che, di solito, è buona cosa che $S = \overline{R}$: come si riprenderà tra breve, esistono configurazioni che "sarebbe meglio non usare", quali in questo caso $S = R = 1$. Al contrario, $S = R = 0$ è una configurazione molto interessante, in quanto introduce la novità rispetto ai circuiti finora analizzati:

quando entrambi gli ingressi sono negati, il circuito in uscita mantiene i valori precedentemente introdotti, ottenendo di fatto un effetto memoria.

5.1.2 Gated D-Latch

Ciò che si fa di solito, rispetto al blocco appena presentato (che deve solo rappresentare l'idea di base di retroazione controllata), è introdurre due elementi di novità: prima di tutto, introdurre un segnale di enable En, che verrà utilizzato al fine di segnalare al circuito il desiderio di introdurre o meno una variazione delle uscite in seguito ad una variazione degli ingresso. L'altro elemento è quello di non utilizzare più due ingressi set e reset tra loro separati, bensì un unico ingresso di dato, detto per l'appunto D. Una realizzazione circuitale di questo blocco potrebbe nella fattispecie essere la seguente:

Essendo questo blocco uno dei più utilizzati, si vuole anticipare un elemento che presto tornerà più chiaro. Come detto, questo elemento è in grado di memorizzare un certo ingresso di dato, D. Fondamentale è che questo ingresso, al momento della cattura, della memorizzazione, del "campionamento", sia stabile, ossia che, al momento in cui il latch entra in funzione di memoria (quando il segnale di clock di riferimento passa dalla modalità campionamento, '1', a quella di memorizzazione, '0'), il segnale D non subisca transizioni o deformazioni che potrebbero comprometterne il riconoscimento.

5.1.3 Edge-Triggered D flip-flop

Quello che probabilmente è l'elemento più importante nell'elettronica digitale sequenziale sincrona tuttavia è il D flip-flop, ossia un elemento in grado di campionare i dati e mantenerli in memoria, ma con una modalità differente rispetto a quella di un normale latch. Come il titolo della sottosezione suggerisce, si parla di dispositivi edge-triggered, ossia "comandati mediante spigolo". Questa definizione non può essere ben compresa, prima di aver analizzato il circuito:

In sostanza, questo circuito campiona, acquisisce (a meno di ritardi interni che verran dopo introdotti) un dato solo al momento in cui il clock "sale", ossia al momento in cui il segnale di clock commuta da basso a alto (o viceversa, a seconda di come il dispositivo è progettato). Si parla di "edge" dunque in termini di "fronte", di salita o discesa, del clock: al momento in cui vi è il fronte del clock, il flip-flop acquisisce il segnale di dato che in quell'istante è presente nell'ingresso, e dopo un ritardo lo propone sull'uscita, mantenendolo in memoria fino al successivo istante di campionamento.

In blocchi di questo tipo è possibile introdurre segnali di preset o reset sincroni o asincroni, a seconda delle necessità del sistema. Tipicamente, buona cosa sarebbe introdurre sistemi di reset sincroni, anche se assolutamente non è detto sia una soluzione obbligatoria, come tra poco si esporrà. Sicuro è un fatto: tipicamente i circuiti sequenziali devono essere in qualche modo inizializzati, mediante un reset; potrebbe poi essere necessario, durante l'uso, re-inizializzarli, o a causa di guasti o a causa di altri fenomeni di vario genere. Esistono fondamentalmente, come già annunciato, due filosofie per il reset del sistema: reset sincrono e reset asincrono. Meglio utilizzare reset sincroni o asincroni? Come annunciato, si tratta di un problema di implementazione: a volte, resettando, è necessario far tutto solo dentro i colpi di clock, a volte è necessario un reset immediato. Si analizzano comunque aspetti positivi e negativi.

Qua riportati i circuiti per il reset sincrono e asincrono:

- Nel caso asincrono, esistono percorsi combinatori più lunghi: un reset sincrono si può di fatto realizzare semplicemente mediante una porta AND, dunque la semplicità circuitale implementativa nonchè il corrispondente vincolo sulla frequenza massima di funzionamento del circuito saranno di sicuro un punto a favore per il sistema sincrono.

- Il clock nel reset sincrono per quanto rallentante "filtra" variazioni indesiderate dei segnali, ossia i "glitches": il fatto che si campioni esclusivamente in un istante di tempo anzichè in continuazione, fa sì che la probabilità di campionare un disturbo sia assolutamente minima. Nel caso di circuiti con reset asincrono, dove si campiona sostanzialmente in qualsiasi momento, un glitch può rappresentare un grosso problema: un disturbo indesiderato può di fatto provocare reset indesiderati del sistema sequenziale, che potrebbe dunque diventare non funzionante.

- Tipicamente, quando si resetta la macchina, nel caso sincrono, una volta smesso di premere il "pulsante" di reset, la macchina smette di resettare e si riavvia dallo stato predefinito iniziale. Nel caso asincrono, tolto il pulsante, se non sono stati verificati i tempi di setup e di hold, il reset può far entrare la macchina in uno stato proibito (come si accennerà meglio in seguito). Ciò che può sostanzialmente capitare è che, entrati in stati proibiti, questi possono "fregare" più flip flop, comunicando informazioni errate, dunque causare problemi di questo tipo. Se dunque durante la deattivazione del segnale di reset il sistema è prossimo al fronte attivo del clock, si rischia di ripristinare il sistema introducendo però in memoria uno stato indesiderato.

Si supponga ora di progettare un sistema, e di volerlo realizzare su di un circuito integrato. Degli integrati prodotti sicuramente non tutti funzioneranno, a causa di impurità presenti sui substrati di silicio piuttosto che a causa di altre motivazioni tecnologiche. Quello che si fa comunemente dopo la produzione dei circuiti integrati è provarli, introducendo dei dati "corretti" per poi vedere come reagisce il circuito alle eccitazioni. Per rendere più veloce il progetto della macchina di test spesso si aggiunge dell'hardware per fare esclusivamente i test di controllo (questa procedura di progetto è solitamente nota col nome di "design for testability"). Aver a che fare con circuiti sincroni è sempre facile; un reset asincrono al contrario potrebbe provocare probemi, poichè, se qualcosa fa girare il reset, allora questo cambia radicalmente e immediatamente il comportamento della macchina. Testare un reset asincrono è abbastanza complicato.

L'unico vantaggio (o uno degli unici, oltre alla velocità) potrebbe interessare casi particolari, quali il fatto che le uscite dei flip flop vengano usate per pilotare dei buffer tri-state: se per caso il clock non riesce a temporizzare correttamente il reset, i tri-state possono danneggiarsi dal momento che il circuito rimane in uno stato qualunque. Un reset asincrono all'accensione del circuito potrebbe risolvere questo problema, imponendo immediatamente e indipendentemente dal clock uno stato al circuito, evitando danni.

5.2 Metastabilità

Oltre a introdurre tante belle cose quali la possibilità di introdurre la memoria di uno stato di un sistema, in modo da permetterne l'evoluzione al variare del tempo, i sistemi sequenziali introducono un enorme problema: l'esistenza di stati particolari del sistema, in cui di fatto si hanno malfunzionamenti. Gli elementi con memoria spesso presentano "stati proibiti", ossia stati che possono portare al non rispetto del principio di complementarietà delle uscite. Per questo motivo è fondamentale introdurre tecniche e nozioni atte a evitare di rientrare in casistiche di questo tipo, mediante la definizione di tempi, di ritardi che permettano al sistema di operare in maniera corretta.

Il fronte del clock, in un flip-flop, ha un certo andamento, e a seconda che si violino o meno i tempi di mantenimento del segnale precedentemente accennati è possibile che si abbiano "risposte strane". Fondamentalmente, quando i cosiddetti "stati proibiti" vengono utilizzati, o quando si campiona l'uscita mentre essa varia, si entra nel cosiddetto "stato di metastabilità". La metastabilità è un fenomeno assolutamente negativo per i circuiti sequenziali: se un flip-flop che deve pilotare altri due elementi con memoria, essi possono per esempio interpretare in modo diverso il valore dell'uscita, dunque il siste-

ma inizia a sbagliare clamorosamente i valori, raggiungendo stati imprevisti; per far funzionare in modo corretto un flip-flop è necessario aumentare i tempi di ritardo per tenere conto della violazione dei tempi di mantenimento e di impostazione del segnale di ingresso, in modo da permettere un corretto campionamento e non l'ingresso nello stato proibito.

5.2.1 Parametri temporali del D-flip-flop

Come appena detto, l'elemento alla base della progettazione di sistemi elettronici digitali sequenziali sarà il D-flip-flop; data la sua importanza, fondamentale sarà introdurre i parametri temporali che lo coinvolgono. Come già detto, al fine di evitare la metastabilità, è necessario che nei dispositivi campionatori, i dispositivi con memoria, si faccia in modo da mantenere gli ingressi e/o le uscite per determinati tempi, in modo da evitare di campionare o propinare dati errati al resto del sistema, garantendo una naturale e corretta evoluzione degli stati.

Per quanto riguarda dunque questo dispositivo, si consideri il seguente disegno:

In questo schema sono rappresentati tutti i parametri temporali da conoscere al fine di utilizzare correttamente un sistema dotato di D-flip-flop. A partire da questo schema è possibile ricavare un'informazione fondamentale del sistema: a quale frequenza massima può lavorare. Ogni sistema, al fine di non entrare in stati proibiti quali metastabilità, può lavorare ad una determinata frequenza, che in questo caso è dettata da una combinazione lineare di questi tempi. Si classificano dunque questi tempi in modo da attribuirvi un significato fisico, e capire come utilizzarli.

- t_{CK} : si tratta del periodo di clock, ossia la durata di un segnale di clock (considerato come somma di parte alta e parte bassa);

- $t_{pd,comb}$: si tratta del tempo di propagazione combinatorio tra un flip-flop che fornisce in uscita un determinato dato, ed uno che accetta un certo dato in ingresso; dato in pratica un flip-flop che fornisce un certo dato in uscita, dato che viene elaborato da una logica puramente combinatoria costituita da un certo insieme di porte, questo tempo quantifica il ritardo introdotto da tutto ciò che sta tra il flip-flop che fornisce l'uscita e quello che campionerà il dato elaborato;

- t_{slack} : margine temporale, che aumenta la "sicurezza" rispetto al tempo combinatorio minimo necessario per l'elaborazione del dato tra i flip-flop;

- t_s : tempo di setup: tempo che serve al flip-flop per immagazzinare, campionare il dato in ingresso; si tratta in pratica dell'ampiezza temporale dell'ingresso, minima, necessaria perchè il flip-flop possa campionare senza "fraintendimenti" il dato. Si può dire che il tempo di setup quantifichi, rispetto al colpo di clock, il tempo necessario perchè il dato sia memorizzato nel flip-flop prima del fronte di campionamento. Si tratta di un tempo che deve garantire la stabilità del segnale prima del colpo di clock. Questo tempo solitamente va sempre considerato, e calcolato sul percorso di propagazione combinatorio "massimo", quello che porta ad una più lenta propagazione del segnale (dunque in funzione del periodo minimo, per calcolare la massima frequenza di lavoro).

- t_h : tempo di hold : si tratta del tempo durante il quale, in seguito al fronte di campionamento del clock, il dato deve rimanere costante, stabile, affinchè si abbia una corretta memorizzazione. Dualmente al precedente, questo tempo quantifica e stabilisce vincoli sulla stabilità del segnale dopo il fronte di clock, anzichè prima di esso. Esso spesso è automaticamente rispettato, se si è dotati di una logica sufficientemente "lenta", che possa permettere una lunga propagazione del segnale. Se la logica fosse troppo veloce, sarebbe necessario in qualche modo introdurre latenze nel circuito, per esempio mediante una catena composta da un numero pari di inverter logici. Come si può intuire da questa affermazione, il tempo di hold dunque deve essere calcolato soprattutto per quanto riguarda i rami "rapidi" del circuito, per fare in modo che esso sia rispettato (ossia che il percorso combinatorio sia almeno più lento di esso, in modo da garantire la stabilità del disturbo, allungando i tempi di propagazione di eventuali variazioni).

Al fine di calcolare le frequenze massime di funzionamento di un circuito sequenziale, è necessario lavorare con i casi peggiori. Il tempo totale minimo richiesto per il segnale sarà sostanzialmente una somma di alcuni di questi tempi (supponendo che il tempo di hold sia rispettato):

$$t_{min} = t_{C2Q} + t_{pd,comb} + t_s$$

Si somma il tempo che il flip-flop di partenza impiega per produrre, per presentare, a partire dal colpo di clock di campionamento, l'uscita Q; questa si dovrà a questo punto propagare per un percorso combinatorio, impiegando un tempo dipendente dalla famiglia logica delle porte, dal tipo e dal numero di porte (uno XOR sarà più lento di un inverter per esempio), quindi sarà necessario aspettare il tempo di setup proprio del flip-flop di uscita, in cui si deve campionare il dato derivante dall'altra parte del circuito.

$$f_{max} = \frac{1}{t_{min}}$$

Nel caso di più percorsi combinatori presenti, sarà necessario scegliere il maggiore. Il fatto che si parli di "tempo minimo" tendenzialmente esclude la presenza di tempi di slack, ossia di margini di tempo in cui il segnale viene mantenuto costante, in modo da poter documentare una frequenza di lavoro minima ma per qualsiasi casistica reale del circuito.

Una nota conclusiva: l'identificazione dei vari percorsi tra due flip-flop (si noti che i percorsi da identificare devono essere sempre e comunque fatti tra due elementi di memoria: uno che campiona l'ingresso di un circuito sequenziale, l'altro che campiona e propone l'uscita del medesimo, o quantomeno uno stato intermedio) può essere di diverso tipo, e fornire informazioni di tipo diverso: a seconda del fatto che i flip-flop campionino sul fronte di salita o di discesa, si possono trarre informazioni di tipo diverso:

- Se il flip-flop di ingresso campiona in salita e quello di uscita in discesa, l'informazione riguarderà il tempo minimo di durata del valore alto del clock;
- Nella situazione duale, ossia flip-flop di ingresso campionante in discesa e quello di uscita in salita, l'informazione riguarderà il tempo minimo di durata del valore basso del clock;
- Nella situazione di due flip-flop che operano sullo stesso tipo di campionamento, l'informazione sarà sul periodo minimo di clock;
- Si noti che per un elettronico è facile produrre segnali con duty cycle pari al 50 %; nel caso le informazioni sui periodi minimi differissero, si sceglie quella più vincolante, e l'altra verrà scelta di conseguenza, per esempio raddoppiando questo periodo scelto. Se poi il periodo risultante da questa operazione fosse maggiore del periodo massimo, si sceglierebbe esso come periodo di clock.

5.3 Macchine a stati finiti

Ciò che si può fare con i circuiti sequenziali è sostanzialmente costruire delle macchine in cui si possa stabilire quale deve essere l'evoluzione degli stati del sistema. A seconda degli stati già presenti e/o degli ingressi, il sistema evolverà passando in altri stati, presentando determinate uscite a seconda degli altri elementi del sistema. In questo momento si sta parlando esclusivamente

di macchine a stati sincrone, dunque non si hanno variazioni dello stato e delle uscite che non siano in qualche modo indipendenti dal clock.

Esistono sostanzialmente due tipi di macchine a stati:

- Macchine di Moore: si tratta di macchine le cui uscite dipendono esclusivamente dallo stato attuale del sistema;
- Macchine di Mealy: si tratta di macchine le cui uscite dipendono dallo stato attuale del sistema e dagli ingressi attuali del sistema; si noti che il fatto che si parli di "ingressi attuali" potrebbe far pensare a uscite asincrone. Ciò non è corretto: la variazione delle uscite dipende esclusivamente dal colpo di clock. Ad ogni colpo di clock vi sono variazioni delle uscite, senza colpi di clock non possono esservene. La differenza tra macchine di Moore e di Mealy è che, a seconda del valore dell'ingresso, possono esservi diverse variazioni dell'uscita, ma questo sempre a seconda del colpo di clock.

Si propone uno schema in grado di far meglio comprendere quale sia la sostanziale differenza tra macchine di Moore e macchine di Mealy:

Sostanzialmente, una macchina a stati è composta da alcuni blocchi "concettuali":

- Una logica di transizione dello stato, ossia una rete puramente combinatoria in grado di far "evolvere" lo stato nella direzione interessata dal progettista: a seconda del comportamento che si intende attribuire alla macchina a stati, sarà necessario che essa, da uno stato ad un altro, effettui determinate operazioni logiche; queste vengono realizzate mediante funzioni logiche, a partire da una rete combinatoria introdotta tra i vari elementi con memoria;
- Registri di stato: di fatto i vari "passi", i vari "stati" della macchina a stati vanno memorizzati, in modo da poter evolvere per l'appunto in vari passi. Sarà dunque necessario introdurre elementi sequenziali, in modo da poter mantenere traccia, memoria della storia precedente del circuito.
- Logica di decodifica dell'uscita: di fatto, a seconda di ciò che il progettista sceglie, è necessario introdurre, per ciascuno stato, una certa codifica numerica, basata su codici di vario tipo: sequenziali (spesso scelta triste), piuttosto che casuali (altrettanto triste), piuttosto che one-hot (ottima, per macchine piccole), piuttosto che Grey (intelligente rispetto alle prime ma che comunque richiede una logica non banale

di decodifica). Il fatto di dover utilizzare codifiche richiede che, per l'esterno del sistema, le informazioni siano semplicemente decifrabili; per questo motivo lo stadio finale di una macchina a stati solitamente è proprio una logica di decodifica.

Come si può osservare, si è introdotto un "tratteggiamento": esso rappresenta, di fatto, la differenza tra le macchine di Moore e le macchine di Mealy: nel caso delle macchine di Mealy, dove è presente anche il "tratteggiamento", esiste in sostanza un collegamento diretto tra ingressi e logica di decodifica, collegamento che potrebbe introdurre condizioni aggiuntive per quanto riguarda l'evoluzione della macchina a stati; questa è la cosiddetta dipendenza sia dai registri di stato, ossia dalla memoria passata del sistema, sia dagli ingressi "presenti" del medesimo.

Si noti che, nel caso di progetti particolari, è possibile inglobare la logica di decodifica degli stati alla logica di evoluzione degli stati, inglobando di fatto due blocchi in uno. Ciò potrebbe ridurre i tempi di propagazione del clock verso l'uscita, aggiungendo da un lato stati e dunque elementi sequenziali, ma riducendo drasticamente d'altro canto i tempi morti, i tempi di propagazione combinatoria.

In un certo senso le macchine di Mealy possono "guadagnare un colpo di clock" rispetto alle macchine di Moore: le uscite di Mealy sono uscite che variano a seconda dello stato, e a seconda dell'ingresso presente a un certo colpo di clock. Se ogni variazione di Moore deve aspettare una certa configurazione degli stati, quella di Mealy può "saltare" alcuni passi, proponendo un'uscita di fatto allo stesso colpo di clock di altre di Moore, e non al colpo di clock successivo.

I passi di progetto per macchine a stati, di Moore e di Mealy, sono abbastanza simili, per quanto cambino al loro "interno":

1. Comprendere le specifiche richieste dal committente;
2. Disegnare il diagramma di transizione degli stati (pallogramma), con le modalità proprie delle macchine di Mealy o di Moore;
3. Calcolare e produrre la tabella di transizione degli stati.

In questo procedimento, ovviamente, è permesso l'uso di don't care, quando se ne ha la possibilità e si intende snellire il progetto, introducendo semplificazioni. Si noti che ciò potrebbe portare, tuttavia, alcuni problemi, che è necessario quantomeno tenere a mente:

- Nel caso i flip-flop comandino dei tri-state, se non si introduce il reset e se si hanno stati don't care tali da produrre un '11' in uscita, si ha il rischio di farli scoppiare. Anche negli stati non raggiunti serve fare attenzione ai don't care, poichè mettere condizioni proibite o rischiare di farlo può essere molto dannoso, in certi casi patologici quali questo;

- In una macchina con reset sincrono, se per qualche motivo esistono stati scorrelati dagli altri e collegati solo a sè stessi (per errore), capita che non se ne esce mai. Per questo motivo di solito le macchine presentano dei power-on-reset asincroni, e reset sincroni: il power-on-reset viene utilizzato solo all'avvio e poi disabilitato, il reset utilizzato. I don't care potrebbero produrre situazioni di questo tipo.

Da un lato le macchine di Mealy sono belle in quanto servono meno flip-flop per realizzarle, sono più veloci e il pallogramma è più compatto, dal momento che si ha a che fare con meno stati; una macchina di Mealy inoltre è più reattiva anche agli eventi esterni. D'altra parte, le uscite sono dipendenti dagli ingressi istantanei, dunque se vi sono glitch, il rischio che si propaghino è non nullo: una macchina di Mealy è poco robusta. Può capitare che le uscite varino, in una macchina di Moore, a ogni colpo di clock; quelle di Mealy, come già detto, sono tali da avere un dato di ingresso tale per cui la connessione combinatoria non permetta di sapere quanto tempo ci vuole per "tornare indietro", dunque la frequenza di funzionamento della macchina. Il suggerimento è: quando si può, si usi Moore; quando si deve per forza andare veloci si faccia pure uso di Mealy, facendo però attenzione agli accorgimenti appena proposti.

Ci si potrebbe chiedere: finora tutte le tecniche di progetto note e che si possono trovare nei testi più comuni fanno uso di D-latch edge triggered. Si possono usare anche altri elementi, quali ad esempio latch? La risposta è sì, ma non conviene: elementi come i latch sono trasparenti, dunque può capitare che lo stato cambi in modo desiderato, ma si abbia comunque campionamento. Per questo motivo lo studio e il progetto dei circuiti sequenziali verrà soprattutto fatto utilizzando, come blocchi di memoria, i D-flip-flop.

5.4 Alcuni semplici esempi di circuiti sequenziali

Verranno ora presentati, almeno in modo sommario, alcuni esempi di circuiti sequenziali usati tipicamente nel progetto di sistemi elettronici digitali.

5.4.1 Shift register

Uno shift register è un circuito in grado di memorizzare più informazioni allo stesso tempo. Se da un lato un flip flop è in grado di memorizzare e mantenere dunque sull'uscita un singolo bit, un registro è in grado di memorizzare una sequenza di bit, un vettore. La modalità più semplice con cui un registro può essere realizzato è la seguente:

Collegando tutti i flip-flop allo stesso clock si sincronizzano tutti i blocchi sequenziali tra loro; per ogni colpo di clock si "shifta" l'informazione verso destra (in questo caso): il dato in ingresso viene campionato dal primo flip-flop, l'uscita di questo passa in ingresso al secondo, e così via, fino all'uscita dell'ultimo flip-flop che viene proposta in uscita. Questo circuito presenta alcuni ovvi problemi: è molto lento da caricare, per quanto semplice da utilizzare.

Una variante a questo circuito, per quanto concettualmente simile, potrebbe essere la seguente:

Con un circuito di questo tipo è possibile effettuare il caricamento sia seriale sia parallelo del circuito, del registro, rendendolo di fatto più costoso sotto il punto di vista dell'area di integrazione, ma molto più versatile. Questo circuito è in grado di fornire uscite in parallelo (prelevando dai pin delle uscite ciascuna informazione), caricare in ingresso in parallelo l'informazione (mediante la circuiteria presentata), ma anche in serie, in modalità classica. La selezione delle modalità viene effettuata mediante un segnale di controllo.

5.4.2 Flip-flop tipo non-D

Il flip-flop finora presentato e che comunque sarà sempre alla base dei progetti sequenziali nella stra-grande maggior parte dei casi è il flip-flop tipo D. Si presenteranno a questo punto alcuni blocchi alternativi, per cultura generale.

Flip-flop SR

Il flip-flop di tipo più "basilare" (anche più del D) è il cosiddetto flip-flop SR (Set-Reset, esattamente come per il SR-Latch): anzichè presentare un singolo ingresso di dato, ne presenta due: uno di set, uno di reset. Dualmente vengono presentate, nelle implementazioni più comuni, due uscite, una la negata dell'altra.

Questo circuito presenta sostanzialmente lo stesso limite degli SR-Latch: è necessario introdurre due dati anzichè uno, rendendo un po' più complicata la progettazione; inoltre esiste lo stato proibito, ossia lo stato nel quale

non si può entrare, pena il rischio di mandare il dispositivo in metastabilità oscillatoria. Considerando tuttavia come ingressi uniti:

$$D = S = \overline{R}$$

Si ottiene semplicemente il ben noto flip-flop tipo D.

Flip-flop JK

Un ulteriore modo di implementare flip-flop è il cosiddetto "JK". Come già detto, il grosso problema dei flip-flop tipo SR è la presenza di uno stato proibito, non ammissibile. L'idea fondamentale alla base dei flip-flop tipo JK è quella di eliminare questo tipo di problema, introducendo, al posto dello stato proibito, un'ulteriore funzionalità del circuito:

Come funziona questo schema? Beh, per le configurazioni di ingressi "00", "10" e "01", esattamente come un normale flip-flop SR; la circuiteria introdotta nel sistema tuttavia permette, al posto dello stato proibito, l'introduzione di una nuova funzionalità: il fatto che si retroazioni l'uscita negativa nella circuiteria combinatoria, permette di fatto un'inversione logica delle uscite, facendole "cambiare": in altre parole, Q diventa $\overline{Q}$, e $\overline{Q}$ diventa Q, in seguito all'introduzione della combinazione di ingressi "11".

Flip-flop T

Ultimo blocco piuttosto interessante e spesso utilizzato è il flip-flop tipo T (dove T sta per "Toggle", "scambiante"): come appena visto, in un flip-flop JK si introduce la possibilità di negare i valori delle uscite, conservando i valori ma negandoli logicamente. Introducendo negli ingressi una piccola modifica, ponendo $J = \overline{K}$, è possibile ottenere qualcosa di questo tipo:

Questo flip-flop, detto tipo T, si comporta nella seguente maniera: quando $T = 0$, l'uscita memorizzata rimane tale; quando $T = 1$, al colpo di clock l'uscita viene invertita, diventando dunque la negata rispetto alla precedente.

5.4.3 Contatori asincroni

Una prima applicazione degli ultimi blocchi appena introdotti, i flip-flop tipo T, può essere la seguente:

Questo schema rappresenta un semplice contatore asincrono: l'asincronicità deriva dal fatto che ogni flip flop, o meglio ogni uscita di essi, funge di fatto da clock per il flip-flop successivo nella catena. Quando si intende contare, si introduce un "1" permanente sull'ingresso del primo flip-flop, che quindi cambierà il proprio valore. A sua volta il colpo di clock successivo la

presenza di un "1" sull'uscita farà commutare l'uscita del secondo flip-flop e così via, ottenendo di fatto un contatore.

Questi contatori non sono il massimo: i ritardi sequenziali, infatti, tendono ad accumularsi, rendendo di fatto inefficiente questo tipo di circuito. Quando si realizza qualcosa del genere conviene mettere pochi stadi sequenziali, anche in modo da evitare transitori sulle uscite.

5.4.4 Contatori sincroni

I contatori sincroni possono essere implementati nella seguente maniera:

In un circuito di questo tipo tutti i flip-flop commutano in dipendenza dallo stesso clock, dunque per questo motivo sono sincroni. In questo caso la massima frequenza di funzionamento si può calcolare in funzione dei ritardi combinatori introdotti dalle porte logiche utilizzate. Utilizzando un'implementazione ottimizzata, è possibile vedere che il tempo minimo di funzionamento è sostanzialmente quello di due porte logiche:

$$t_{comb,max} = t_{C2Q} + 2t_{AND}$$

A questo punto, sommando i tempi di setup, è possibile calcolare il periodo minimo, e da esso la frequenza di lavoro.

Si noti che questo circuito, e tutti gli altri, sono sintetizzabili mediante le classiche tecniche basate sull'uso di mappe di Karnaugh e procedimenti classici, o mediante VHDL.

5.4.5 Contatori a caricamento parallelo

Un altro modo di realizzare contatori "particolari" potrebbe essere il seguente:

Questo tipo di topologia può essere molto utile e interessante per realizzare contatori non modulo 2^n: il fatto che sia possibile caricare un valore iniziale nei contatori, e introdurre un istante di termine conto, permette di limitare i bound da contare, realizzando di fatto contatori di modulo qualsiasi. Concettualmente, il segreto per la realizzazione di circuiti di questo tipo è l'uso dei multiplexer, che permette di introdurre il dato o da un contatore stesso o da un dato esterno, imposto dunque da un utente in qualche modo.

5.4.6 Contatori a frequenza elevata

I contatori finora implementati e presentati funzionano, ma non sono perfetti per alcune situazioni: i lavori ad alte frequenze.

Esistono almeno due particolari topologie, che ora verranno quantomeno presentate sommariamente, in grado di lavorare a frequenze ben maggiori rispetto agli schemi più tradizionali appena mostrati.

Ring counter

La prima idea per realizzare un contatore veloce potrebbe essere la seguente:

In sostanza, l'idea alla base di questo tipo di circuito è quella di usare uno shift register retroazionato, in cui l'uscita dell'ultimo flip-flop viene portata indietro fino al primo. In questa situazione, si inizializzano tutti i flip-flop mediante un reset a zero tranne uno, pilotato da un sistema di complementazione. Dopo un certo numero di colpi di clock si torna al punto di partenza. Di tanto in tanto si possono trovare circuiti di questo tipo associati ad un encoder, anche se non si tratta di una scelta molto felice: un encoder introduce grossi ritardi su un circuito così veloce, dunque bisogna stare attenti nell'inserire logiche che potrebbero rallentare il circuito.

Johnson counter

Il concetto del Johnson Counter è il seguente:

Si resettano contemporaneamente e assieme tutti i flip flop, dunque il ritorno al primo, la retroazione, viene fatta con l'uscita $\overline{Q}$ anzichè Q, dell'ultimo flip-flop. Il circuito dunque concettualmente è molto simile a quello del ring counter, se non per questa piccola differenza.

I ritardi del circuito sono sostanzialmente pari al tempo di setup e al tempo compreso tra il colpo di clock e la propagazione dell'uscita: un'uscita è sempre diversa dalle altre, e questa continua a variare a seconda del valore dell'uscita attuale del contatore, in modo da variare continuamente il valore delle uscite.

5.5 Macchine a stati algoritmiche - ASM Charts

Quando si devono progettare sistemi sequenziali complicati, l'approccio noto dal corso di Calcolatori Elettronici, basato sulla sintesi del grafo di transizione degli stati, potrebbe essere troppo limitante rispetto alla complessità dei progetti da effettuare.

I passi da seguire sono, come sempre, sostanzialmente i seguenti:

1. Comprendere le specifiche;
2. Introdurre una rappresentazione grafica del comportamento della macchina a stati;

3. Semplificare gli stati e attribuire una codifica ai medesimi;

4. Procedere con un metodo di sintesi.

Per quanto riguarda la rappresentazione grafica, si dedicherà una sottosezione a parte; si vogliono prima di essa riprendere alcuni concetti, che potrebbero tornare utili per macchine relativamente semplici, riguardo la codifica degli stati.

5.5.1 Codifica degli stati

Nel caso di procedimenti di sintesi manuali o semi-manuali, uno dei passi che possono rivelarsi più fatali è quello di attribuire, per ciascuno stato, una codifica, ossia un valore numerico in grado di identificare lo stato della macchina. Nel caso di macchine a stati relativamente semplici, fino a 8 o 10 stati, la scelta più sicura è quella della codifica one-hot: identificare ciascuno stato con la posizione di un '1', rispetto a tutti '0' per gli altri stati. Questo tipo di approccio è molto interessante per due motivi:

- Si può sostanzialmente prevedere l'evoluzione logica dello stato in modo semplice, per "ispezione", identificando l'1 tra gli 0. Il senso è che si può ricavare le funzioni logiche combinatorie e sequenziali rappresentanti l'evoluzione degli stati semplicemente "guardando" la tavola di transizione degli stati, "scrivendo" le condizioni che permettono di passare dall' stato Q_{n-1} a Q_n immediatamente e con semplicità;

- Il fatto di avere una codifica così semplice permette sostanzialmente di non utilizzare alcuna logica di decodifica, rendendo di fatto molto semplice e veloce la macchina a stati.

La sintesi della macchina a stati può dunque essere effettuata in diversi modi, seguendo le tecniche precedentemente proposte: mediante LUT, ROM, MUX o altri tipi di tecniche.

5.5.2 ASM Charts

Quando si ha a che fare con progetti di una certa complessità, quali le ASM (Algoritmic State Machines), ossia macchine a stati basate sull'implementazione elettronica digitale di un certo algoritmo di una certa complessità (un algoritmo di divisione piuttosto che di ordinamento piuttosto che altre cose), le tecniche basate sul "pallogramma" risultano essere inefficienti, poichè non in grado di documentare in maniera efficiente il progetto. Quella che si vuole

a questo punto introdurre è una migliore e più semplice schematizzazione, in grado di contenere un maggior numero di informazioni e presentante un maggior riscontro su quello che sarà il comportamento della macchina a stati finale.

Questo tipo di rappresentazione è basata sostanzialmente sull'uso di tre blocchi, più indicazioni sui segnali in esse contenute:

Sostanzialmente bisogna tenere presente il significato e la temporizzazione legate a questi tre blocchi, in modo da poterli utilizzare in maniera corretta:

- State box (forma "rettangolo"): si tratta sostanzialmente di blocchi contenenti indicazioni sull'evoluzione delle uscite secondo una logica di Moore. Ciò significa che tutte le uscite e le indicazioni sui segnali contenute nei "rettangoli", al colpo di clock in cui si arriva nel suddetto stato, variano. Si noti che solo al colpo di clock successivo, ossia in prossimità del successivo "rettangolo", le uscite saranno a disposizione della macchina o dell'utenza. Il significato sotto il punto di vista della temporizzazione è: per ogni rettangolo è associato un colpo di clock, e solo al colpo di clock successivo a quello di un dato "rettangolo" si avran a disposizione le uscite richieste nel rettangolo stesso.

- Decision box (forma "rombo"): si tratta di blocchi in grado di introdurre una "scelta", a seconda dello stato in cui ci si trova. Entrati in uno stato, dati i valori di diverse uscite, l'evoluzione della macchina a stati può portare a uno stato piuttosto che a un altro; questa box semplicemente, riferendosi al colpo di clock del rettangolo "appena precedente", propone una biforcazione, inviando la macchina in uno stato anzichè in un altro.

- Conditional output box (forma "ovale") : si tratta sostanzialmente della modellizzazione, in ASM Charts, delle "uscite di Mealy": a seconda dello stato in cui ci si trova e a seconda del valore di determinati ingressi, le uscite contenute nei rettangoli potranno essere o meno attivate e/o portate a determinati valori. Queste uscite devono essere tendenzialmente introdotte dopo un "rettangolo"; il vantaggio di queste uscite è che saranno già a disposizione, se attivate, al colpo di clock successivo a quello del rettangolo: essendo uscite di Mealy esse saranno sincrone, in quanto dipendenti comunque dal clock, ma potranno essere o meno attivate nello stesso istante delle condizioni contenute nel "rettangolo" appena precedente, dunque essere pronte per il colpo di clock seguente.

Tenendo bene a mente queste indicazioni, a partire da questi grafici è possibile ricavare con semplicità un'implementazione VHDL, ottenendo così una macchina a stati facilmente sintetizzabile mediante strumenti informatici.

5.5.3 Progetto di sistemi elettronici digitali complessi

In questa sottosezione si suggerirà sostanzialmente un metodo per progettare un sistema elettronico digitale, a partire dagli strumenti per ora presentati. Il fatto di aver introdotto gli ASM Chart fornisce sostanzialmente uno strumento molto interessante: un ASM chart rappresenta, di fatto, una traduzione di un algoritmo, pensato e proposto ad esempio in pseudocodice (piuttosto che in un linguaggio particolarmente semplice quale un pseudo-C piuttosto che pseudo-BASIC o pseudo-pascal, a seconda dei gusti).

La tattica da seguire quando si progettano sistemi di tipo complesso, è quella di dividere il progetto in diverse sezioni:

- Traduzione dell'algoritmo in ASM chart, in modo da fornire una prima idea di quello che sarà il progetto definitivo (parte di fatto opzionale, ma che di sicuro non guasta);
- Progetto della parte puramente "hardware" del sistema, ossia del cosiddetto "datapath": si definiscono tutte le strutture fisiche (descritte a diversi livelli, a seconda di ciò che il sistema e le specifiche richiedono), e si collegano tra di loro (descrivendo il sistema graficamente, e poi in seguito in VHDL utilizzando i costrutti del linguaggio di descrizione / programmazione a seconda del livello di astrazione);
- Progetto della parte puramente "comportamentale" del sistema: una volta definite tutte le strutture fisiche nel sistema, mediante un altro ASM chart si definisce il comportamento dei vari segnali, in modo da definire il comportamento e l'evoluzione degli stati della macchina, operando però non più in modo astratto, algoritmico come per il primo ASM chart, bensì definendo esattamente il comportamento che i segnali devono avere nei diversi stati, dunque ai diversi colpi di clock.

Una volta definito dunque un algoritmo che descriva il funzionamento del sistema elettronico, lo si traduce in un primo ASM chart, descrittivo; a questo punto, dall'algoritmo, è possibile capire quante e quali strutture dati servano, per realizzare il sistema elettronico digitale; queste vengono dunque introdotte e connesse opportunamente tra loro nel datapath, definendo un certo numero di segnali di supporto che permetteranno di controllare la macchina a stati. Un terzo blocco, un altro ASM chart, molto simile al primo, sarà in grado di controllare il datapath, modificando i valori dei segnali ai vari istanti di clock. A questo punto, terminata la schematizzazione, essa viene implementata o manualmente o (più comunemente) mediante una descrizione in VHDL, che quindi potrà essere sintetizzata dal calcolatore.

5.6 Pipelining

Per quanto breve, si vuole dedicare una sezione ad un concetto che sempre più frequentemente torna utile nell'ambito della progettazione dei sistemi elettronici digitali: il pipelining. Il pipelining è uno dei concetti più importanti creati nell'ambito dell'elettronica digitale, ed è sempre più frequentemente utilizzato al fine di incrementare le prestazioni di un sistema di questo tipo.

La normale architettura di un sistema elettronico è formata da blocchi di vario tipo: blocchi computazionali quali ALU, e registri; si prelevano i dati dai registri, si elaborano, si spostano in altri registri, si leggono, e così via. Quello che ci si può chiedere è: dati tutti gli studi sui ritardi, sulle elaborazioni, sui tempi di memorizzazione, di mantenimento del segnale, quante operazioni al secondo si possono fare con un certo circuito? Beh, molto semplice: una volta fatti tutti i nostri calcoli, si ricavava una frequenza massima di lavoro, f_{max}. Il "throughput", ossia il numero di elaborazioni al secondo, sostanzialmente coincide con la frequenza di lavoro: per ogni istante si estrae un numero di prodotti pari alla frequenza di lavoro del sistema.

La domanda a questo punto è: come si può migliorare le prestazioni del sistema? Beh, proviamo a proporre alcune strade:

- Migliorare la qualità, l'efficienza della rete combinatoria, utilizzando porte più reattive, con minori parametri parassiti ad esempio, diminuendo così il tempo combinatorio e di conseguenza aumentando la massima frequenza di lavoro;

- Introducendo nuovi registri, nuovi stati della macchina, "in mezzo tra due stati".

Questa seconda idea, apparentemente inutile, è in realtà una delle idee più geniali e utilizzate nell'ambito dell'elettronica digitale. Si è detto che si vuole introdurre un elemento sequenziale tra altri due, in modo dunque da aumentare gli stati della macchina. A che serve ciò? Beh, molto semplice: per ogni operazione, di fatto, si "dimezza" il ritardo combinatorio: anzichè attendere che ogni operazione finisca per iniziarne una nuova, prima se ne inizia una, poi la si porta avanti, ma contemporaneamente se ne inizia un'altra, e così via, portando avanti più lavori, più processi per volta, ottenendo però di fatto un'efficienza, o meglio, un throughput molto più elevato. Questo tipo di operazione è detta "pipelining" (nella fattispecie, si sta parlando di "data pipelining").

Ovviamente questo processo si può iterare, finchè è possibile: si continua a introdurre nuovi stati, continuando a triplicare, quadruplicare e così via il

numero di stati, ma dunque continuando a prendere sempre più operazioni per volta, portandole avanti indipendentemente.

Capitolo 6

Complementi di VHDL per circuiti sequenziali

Precedentemente sono stati introdotti alcuni concetti fondamentali sul VHDL, al fine di introdurre alcune tecniche per la descrizione di circuiti di tipo combinatorio. Potrebbe essere molto importante avere la possibilità di descrivere, mediante tecniche non ancora citate (che comunque qua verranno solo citate in modo approssimato; il VHDL Cookbook può essere molto più esaustivo dei brevi cenni che "mostreranno l'esistenza" di alcuni costrutti che verranno ora proposti), sistemi digitali di tipo sequenziale.

6.1 Processi

Ciò che finora non è stato ancora spiegato, è come "mantenere" un'informazione. Nella fattispecie, il costrutto più utilizzato per la descrizione di circuiti in VHDL è stato qualcosa del tipo:

```
A <= B AND C;
```

Ossia descrizioni dei collegamenti tra segnali, tra "pins".

Cosa significa nella realtà, per l'interprete del linguaggio, questo costrutto? Beh, la risposta è abbastanza semplice: fino a quando B o C non cambiano, il sistema non subisce variazioni di alcun tipo. Dopo aver variato uno di questi segnali, nella event list si schedula il fatto che A dovrà cambiare, eventualmente dopo un certo tempo (se sono state introdotte informazioni sui delays). Nel caso si introducano più di queste istruzioni, come già visto esse variano "tutte assieme", nello stesso istante di tempo, in parallelo tra

loro. Questo perchè le assegnazioni effettuate in questo modo rappresentano processi impliciti, in cui l'elaborazione avviene ogni volta che qualcuno modifica il valore degli ingressi, Si parla di "assegnazioni concorrenti" in quanto non esiste una valutazione delle assegnazioni: non conta come e quando si dichiarano, dal momento che, in questo tipo di funzionalità, il linguaggio non opera in modo sequenziale. Si è parlato di "processi impliciti" ma, a questo punto, ci si potrebbe chiedere: esistono tecniche per realizzare processi in maniera "esplicita" ? La risposta ovviamente è sì, e su di essa si basa il più potente metodo per la descrizione di circuiti di qualsiasi tipo, soprattutto sequenziale.

Il concetto di base è il seguente: esiste (come si dirà tra breve) una lista di elementi che, se toccati, fanno "scattare" il processo, introducendolo nella ben nota event list. Come qualcuno tocca uno di questi "ingressi sensibili", il processo si avvia e tutte le istruzioni contenute nella sua descrizione vengono effettuate. Questa lista di "ingressi sensibili" è detta "sensitivity list": tutti e soli i segnali contenuti in essa sono in grado di far "svegliare" il processo. Si noti che in un processo fondamentale diventa l'ordine delle istruzioni: all'interno della descrizione del processo, tutto viene eseguito in ordine rigorosamente sequenziale. In esso si può usare una sintassi simile a quella dei vari linguaggi di programmazione come C, introducendo la possibilità di inserire assegnazioni condizionate, cicli, operazioni di tipo aritmetico (se si utilizzano le librerie apposite), e così via. Si noti che in questo ambito VHDL diviene molto simile ad un linguaggio di programmazione, anche se il sintetizzatore certo non va messo da parte: il ruolo del sintetizzatore sarà, a questo punto, quello di identificare a partire dai costrutti utilizzati per il processo un insieme di hardware atto a realizzare le funzioni in esso descritte, e a collegarlo in maniera appropriata. Rimandando a un riferimento quale VHDL Cookbook, si noti che esistono e sono utilizzabili i seguenti comandi associati a relativi costrutti (e non solo):

- FOR;
- CASE;
- IF;
- LOOP;

Si noti che alcuni costrutti quale ad esempio "WHEN" potevano essere utilizzati anche come processi impliciti; questo è naturale, quando si vogliono descrivere in VHDL elementi di selezione quali decoder o multiplexer;

elementi di questo genere possono essere anche realizzati mediante processi espliciti, utilizzando i costrutti prima proposti.

Potrebbe essere utile, nel processo esplicito, utilizzare variabili; data una variabile A alla quale si vuole assegnare ad esempio il valore "1", si utilizza un costrutto di questo genere:

```
A := 1;
```

Il significato di variabile è molto differente da quello di segnale, come si può intuire dal fatto che si utilizzi un metodo di assegnazione differente dal precedente: la variabile è sostanzialmente uno strumento del VHDL come linguaggio di descrizione, in termini di processi espliciti (quindi più prossimo ai classici linguaggi di programmazione); una variabile non sarà direttamente legata all'hardware come potrebbe essere un segnale, descrizione in VHDL di un pin, bensì è semplicemente uno strumento atto a descrivere il comportamento del circuito, comportamento che verrà in qualche modo implementato dal sintetizzatore.

Come è fatto nella pratica un processo? Beh, si veda ciò:

Fondamentalmente, un processo si può strutturare nella seguente maniera:

1. Attribuire un'etichetta, una "label" al processo (in modo da facilitare il debug);

2. Utilizzare il costrutto:

   ```
   LABEL PROCESS(A, B, C, D);
   ```

   Dove "LABEL" è l'etichetta citata, e A, B, C, D, ossia i contenuti della parentesi relativa a PROCESS, sono la "sensitivity list", ossia l'insieme degli ingressi che, se "toccati, provocano l'inserimento del processo nell'event list, dunque il suo avvio;

3. Definizione di eventuali variabili nel caso ve ne fosse bisogno;

4. Body del processo, ossia insieme dei costrutti atti a descrivere il comportamento.

Una regola d'oro da utilizzare per i processi è la seguente: scrivere due processi che pilotino la stessa uscita, forzandola magari a due diversi valori, è brutto: si mettono infatti in conflitto le due uscite, provocando errori (che però magari le librerie saranno in qualche modo in grado di risolvere). A meno di casi specifici, ogni esame deve essere pilotato da uno e un solo processo.

Il processo deve considerare in modo esaustivo tutte le combinazioni delle uscite al variare degli ingressi, ossia ad esempio nel caso in cui si voglia implementare una qualche tavola di verità, introdurre tutte le possibili combinazioni di ingressi, tutte le possibilità (a meno che non si voglia esplicitamente fare il contrario): se non si considerano possibili ingressi, il circuito non cambia i valori, introducendo un errore del tipo "latch inferred": poichè il circuito non sa come comportarsi in questa situazione, dal momento che non è stato definito un comportamento per l'ingresso, il circuito tende a non cambiare niente, facendo introdurre al sintetizzatore dei latch (o comunque elementi con memoria).

Come già detto, le assegnazioni nel process vanno eseguite sequenzialmente. Nel caso vengano assegnati più valori allo stesso segnale, ora, non si ha concorrenza, bensì "vince l'ultimo": solo l'ultima istruzione relativa al segnale verrà eseguita.

6.2 Idee sul progetto di circuiti sequenziali

Senza andare troppo nei dettagli, si vuole dire come rappresentare flip-flop e latch. Un'idea (negativa in parte) è già stata proposta, ossia quella di "non dire di variare l'uscita", in modo da mantenerla tale. Ciò però introdurrebbe dei latch, dispositivi trasparenti, sincroni ma trasparenti. Quello che non è stato ancora in alcun modo introdotto in VHDL è il "concetto di fronte": nel caso si voglia far produrre al sintetizzatore un flip-flop, è necessario avere una sintassi in grado di introdurre qualcosa del genere. Si può utilizzare il costrutto IF, nella seguente maniera:

```
PROVA PROCESS(Clock, Reset)
BEGIN
IF Clock'EVENT AND Clock = '1' THEN
BEGIN
...
...
END IF;
END PROCESS;
```

Nella sensitivity list si introducono due elementi, di cui uno fondamentale: il Clock. Esso è un SIGNAL, un segnale, il collegamento a un pin. Il costrutto presentato, basato su 'EVENT, fornisce il fatto che di tutto il segnale di clock non si consideri il valore, bensì esclusivamente il fronte, l'evento di variazione e l'istante a esso relativo. Imponendo poi mediante AND che in fondo il clock valga 1, permette di specificare il fatto che il fronte di campionamento sia quello di salita. L'apice " ' " indica l'attributo del segnale, in questo caso il "fronte", il fatto che il clock vari. Esistono nel VHDL altri attributi, che però ora non ci riguarderanno.

Mediante una descrizione di questo genere è possibile definire non solo blocchetti quali flip-flop ma circuiti complicati quali contatori o altro. Non si analizzeranno nella trattazione esempi pratici, ma si esporrà in termini di linguaggio di descrizione l'implementazione di uno degli elementi più importanti in un sistema digitale: il RESET.

Come visto, si è già introdotto un segnale "Reset" nella sensitivity list. Ciò significa che, quando qualcuno "preme il pulsante del reset", il processo in questione sarà avviato. A questo punto, come ben noto, il reset si può gestire in due maniere, ossia "sincrono" e "asincrono"; a partire da questa idea si possono trovare due implementazioni ben differenti in VHDL, in grado di implementare le idee precedentemente espresse.

- Per quanto riguarda il reset sincrono, esso deve essere per l'appunto "sincronizzato al colpo di clock", dunque la condizione sul clock prevale su quella sul reset. Si dovrà utilizzare dunque un costrutto di questo genere, in modo da "dare la priorità al clock":

```
PROVA PROCESS(Clock, Reset)
BEGIN
IF Clock'EVENT AND Clock = '1' THEN
BEGIN
IF Reset = '1' THEN .... ;
...
...
END IF;
END PROCESS;
```

- Per quanto riguarda il reset asincrono, esso deve essere indipendente dal clock; è possibile dunque utilizzare un costrutto di questo genere:

```
PROVA PROCESS(Clock, Reset)
BEGIN
IF Reset = '1' THEN ..... ;
ELSIF Clock'EVENT AND Clock = '1' THEN
BEGIN
...
...
END IF;
END PROCESS;
```

In questo modo il Reset è indipendente, scorrelato dal clock, e può partire indipendentemente da esso.

Approccio alternativo alla sensitivity list

Esiste un metodo alternativo di esprimere il concetto di clock in un processo, non sfruttando la sensitivity list. Introducendo un process con sensitivity list nulla, sostanzialmente il processo si avvierà sempre, in continuazione. Si può fare qualcosa di questo genere:

```
PROVA PROCESS()
BEGIN
WAIT UNTIL Clock = '1' AND Clock'EVENT;
...
...
...
END PROCESS;
```

In questo caso il processo è sempre nella event list, ma il suo contenuto, il body, verrà eseguito esclusivamente a patto che il clock soddisfi le condizioni, ossia nel caso del fronte di salita. Si noti che questo metodo è "brutto" da usare in quanto snatura il concetto di sensitivity list, tuttavia, per quanto sconsigliato, si voleva presentare, in grado di mostrare l'esistenza del costrutto "wait", nonchè per cultura generale.

Capitolo 7

Memorie

Un'azione estremamente importante per un progettista di sistemi elettronici digitali è memorizzare informazioni di vario genere; mediante registri ed elementi di memoria di vario genere è possibile memorizzare dati, ma non in grandi quantità: è necessario introdurre, a tal fine, meccanismi ed elementi in grado di memorizzare in maniera "intelligente" i dati, e di recuperarli in modo altrettanto efficiente: è necessario introdurre memorie in grado di contenere dati di una certa entità.

Per "memoria" si intende un insieme di celle in grado di immagazzinare dati, e di circuiti in grado di facilitare l'accesso in lettura/scrittura alle varie celle. Una memoria può essere organizzata, come vedremo, in diverse maniere, a seconda dei tipi di lavori cui è destinata. Sarà necessario introdurre dei metodi di riconoscimento delle celle, a seconda dell'organizzazione, basati sull'uso di indirizzi, codificati in diverse maniere. Le memorie sono comunemente organizzate come collezioni di bit, di byte (insiemi di 8 bit), o di parole (generici insiemi di n bit), a seconda dell'architettura del sistema.

Per questioni di integrazione, non si considerano memorie di tipo vettoriale, ossia costituiti da un solo insieme di righe o di parole: integrare una "riga" sarebbe molto difficile, rispetto all'integrare una matrice di celle; si introducono, a tal fine, metodi di decodifica "a due passi", in grado di identificare univocamente una cella mediante un meccanismo di decodifica orizzontale (per le colonne) e uno verticale (per le righe). Si predilige dunque, a un'architettura monodimensionale, una di tipo matriciale, bidimensionale; nel caso di particolari necessità, quali memorie di una certa dimensione a partire da blocchi fondamentali di dimensione inferiore, è possibile spesso aver a che fare con architetture di tipo tridimensionale, basate su tre stadi di decodifica: i due "classici", e uno in grado di selezionare il blocco interessato. Ciò può essere nella fattispecie molto utile se:

- Il parallelismo della memoria è piuttosto basso;
- Se le allocazioni di memoria di un singolo blocco non sono sufficienti, dunque se, rispetto alla capacità di indirizzamento della logica di decodifica, si abbiano pochi blocchi di memoria.

Si aggiunge quindi rispetto al caso bidimensionale una logica di selezione del blocco, raggiungendo i tre già citati blocchi di decodifica.

Il vantaggio derivante da questo tipo di architettura è il fatto che, unendo più matrici in un singolo circuito integrato (anzichè utilizzare dunque diversi integrati), si può inviare un dato che non passerà più per "tutti i blocchi", ma, se si organizza in maniera opportuna la logica di decodifica, solo per "certi", guadagnandoci in efficienza di decodifica e in energia necessaria per l'alimentazione del sistema di memoria.

Come vedremo meglio nel dettaglio in seguito a quest'introduzione, esistono sostanzialmente due metodi, due filosofie, due scuole di pensiero per la gestione dei dati (in ingresso e in uscita) in una memoria:

- Avere un grosso numero di pin, in modo da gestire mediante terminali differenti segnali differenti;
- Avere una logica di decodifica più complicata con diversi segnali di controllo, in modo da risparmiare sul numero di terminali.

Analizziamo nei dettaglio diversi tipi di memorie.

7.1 Random Access Memory (RAM)

Tra le memorie una categoria molto "famosa" e utilizzata è quella delle RAM, ossia memorie ad accesso casuale. Si tratta di memorie molto veloci, utilizzate dunque per questo motivo come memorie primarie in ambito informatico (ossia memorie in grado di contenere anche una quantità ridotta di dati, al costo però di variarne spesso il contenuto, in modo da poter effettuare velocemente elaborazioni di vario tipo).

Esistono sostanzialmente, tra le RAM, due sotto-categorie:

- Static RAM (SRAM)
- Dynamic RAM (DRAM)

Analizziamole con maggior dettaglio.

7.1.1 Static Random Access Memory

Le SRAM sono una categoria di RAM che si distingue per la propria capacità di mantenere (se alimentate) anche per un tempo idealmente illimitato le informazioni; questo fatto è dovuto alla robustezza dei componenti su cui questa categoria di memorie si basa: l'elemento alla base di una SRAM è un latch, elemento estremamente robusto e potente; si tratta di memorie che richiedono una bassa alimentazione, ed estremamente veloci, ma con un difetto che le rende inutilizzabili per molte applicazioni: il latch, per quanto robusto, è un elemento che richiede una grossa area di integrazione, dunque inutilizzabile per costruire, in poco spazio, grosse celle di memoria; vengono spesso utilizzate come base per memorie di tipo cache (eventualmente dopo introdotte).

Un esempio di realizzazione circuitale di una SRAM potrebbe essere il seguente:

La logica combinatoria ha lo scopo di abilitare la memoria, per un'operazione di lettura/scrittura. Introducendo logiche combinatorie aggiuntive, è possibile selezionare il tipo di operazione da fare, ed effettuarla.

A questi tipi di logiche si può associare un decoder, in grado di, a partire da una codifica di ingresso, selezionare una sola delle linee (abilitando uno solo dei segnali di "select"), in modo da scrivere in una certa cella, indirizzata dall'esterno.

Al fine di semplificare la logica di decodifica, è possibile utilizzare vari stratagemmi, quale quello già citato di introdurre un'architettura di tipo bidimensionale; sotto-stratagemma potrebbe essere quello di utilizzare due decoder che puntino ad un certo indirizzo, uno per le righe e uno per le parole (ossia per gli insiemi di colonne).

7.1.2 Dynamic Random Access Memory

Le DRAM sono le memorie più utilizzate al fine di immagazzinare informazioni volatili; il principio di funzionamento sul quale si basano è memorizzare in un condensatore un'informazione il quale, se è caricato con una certa tensione, verrà mantenuto in quello stato mediante un interruttore (allo stato solido); aprendo l'interruttore, la carica può essere eliminata o mantenuta.

Questa cella si basa su di un principio molto semplice, ed estremamente elementare da realizzare; esso inoltre minimizza l'area da utilizzare, sovrapponendo di fatto capacità e interruttore.

Ciascuna cella di memoria è collegata ad una bitline, la quale è un "contenitore" molto più capace della memoria stessa, dunque piccole variazioni di tensione causate dallo svuotamento del condensatore di memoria a favore

di un riempimento della bitline devono essere percepite, mediante un sense amplifier, ossia un amplificatore in grado di "sentire" la variazione di tensione e renderla distinguibile, in modo da discriminare la presenza di un "1" o di uno "0" presente nella cella di memoria all'istante di lettura.

La lettura di una cella di memoria dinamica è distruttiva; una volta letta l'informazione, se si intende mantenerla, è necessario ripristinare il livello in seguito alla lettura, e non solo: il transistore utilizzato come interruttore è non-ideale, dunque anch'esso presenta perdite di vario genere: la memoria, in altre parole, deve essere rinfrescata sia a ogni lettura sia dopo un certo periodo.

Questa è la struttura di una memoria DRAM: mediante l'uso di due indici, uno indicante la riga e uno indicante la colonna, è possibile selezionare la cella di memoria interessata.

Questa memoria presenta più svantaggi che vantaggi, ma il suo vantaggio principale la rende così appetibile da renderla la più utilizzata in ambito commerciale: le dimensioni delle celle sono estremamente ridotte, quindi il numero di celle realizzabili in un circuito integrato è estremamente elevato, nell'ordine del miliardo. Parlando di questi ordini di grandezza, sarebbe necessario indirizzare su più di 30 linee di indirizzo, cosa che potrebbe provocare limiti nella realizzazione; l'idea alla base della realizzazione di memorie di questo tipo, dunque, è quello di selezionare prima una riga, poi una colonna, in due istanti differenti.

Il problema enorme delle DRAM è il fatto che esse perdono, a causa dei diversi elementi di non-idealità in esse presenti, informazione. In decine di millisecondi infatti il contenuto del condensatore, per un motivo o per un altro, si perde: essendo la capacità molto ridotta, la carica in esso contenuta è molto piccola, quindi facile da perdersi. Ciò costringe il progettista a introdurre un meccanismo molto scomodo: il refresh (rinfresco). Esso consiste nel rinfrescare continuamente il contenuto della memoria, in modo da non permettere mai che il contenuto sparisca, rendendo tuttavia la memoria non utilizzabile durante il periodo di rinfresco. Ciò introduce inoltre un certo numero di segnali e blocchi di controllo, nonchè una logica di controllo, basata su di un controllore del procedimento di rinfresco (refresh controller) e su di un contatore (refresh counter); ciò che si fa nella pratica è una serie di letture fittizie delle celle, in ordine, dove l'ordine viene controllato dal contatore che contiene l'indice della riga di memoria in fase di rinfresco; ad ogni rinfresco di riga, il contatore viene incrementato di un'unità, in modo da "tenere il conto", quindi si passa alla successiva lettura fittizia. Ad ogni colpo si rinfresca una cella per volta, in modo da non bloccare per troppo tempo l'accesso alla CPU.

Si osservi il seguente schema:

Si nota la presenza di due segnali non ancora discussi, RAS e CAS; si tratta di segnali di temporizzazione per quanto concerne la riga e la colonna. Come già detto, si usa la logica di decodifica applicata a righe e colonne in due istanti differenti; al fine di utilizzare questo tipo di temporizzazione, si ha una FSM che gestisce gli indirizzi e l'istante in cui devono essere acquisiti.

Una memoria può dunque essere utilizzata in modo sincrono; nella fattispecie, le più note tra le DRAM sono, per l'appunto, sincrone.

Analizziamo più nel dettaglio le più note di queste memorie.

Synchronous DRAM

Le SDRAM sono semplicemente delle DRAM in cui il trasferimento di dati coinvolgente la memoria (in ingresso e/o in uscita) è dettato da un clock. Da quando si fornisce il segnale di ingresso, dopo un certo tempo si ha un'uscita, dettata dal clock, o viceversa: un clock forza la validazione delle operazioni, in modo da rendere di fatto inutilizzabile, senza colpi di clock, le operazioni.

DDR

Aumentando la tecnologia si è andati verso una memoria in grado di elaborare (nel senso di campionare o proporre in uscita), per ogni fronte, sia esso di salita o discesa, due dati: le DDR (Double-Data Rate Synchronous DRAM). Il concetto, di base, si può riassumere in due affermazioni:

1. Per come la memoria è strutturata, si può selezionare un'intera riga, memorizzarla in un registro, e lavorarci in serie, utilizzando dunque un meccanismo di lettura multipla; quest'idea permette di aumentare la banda della memoria, facendola lavorare molto più velocemente. Si leggono quindi in un solo accesso molti più dati rispetto a prima, riducendo di molto il tempo di accesso medio del dato;

2. Ciò che permette di usare in modo organizzato questa struttura è il fatto che l'organizzazione sia fatta a blocchi, cosa che permette di adattare le dimensioni dei blocchi a quelli delle memoria da cui si legge/scrive, ideando un nuovo tipo di organizzazione della memoria: le cache.

Si noti che il processo di temporizzazione introduce tuttavia un fatto fondamentale da conoscere: rispetto all'uso classico, asincrono di una memoria, ossia al tradizionale tipo di trasferimenti, si introduce una distanza temporale tra l'inserimento dell'indirizzo e la ricezione del dato in uscita; questo, a causa dell'elaborazione degli indirizzi di riga e di colonna, temporizzati

mediante i segnali RAS/CAS. Aumentare la velocità delle memorie, a questo stato, significa semplicemente ridurre i tempi morti; da qui, le attuali evoluzioni delle memorie.

RAMBUS

Un tipo alternativo di memoria, che tuttavia non ha preso piede nelle architetture in uso, è il cosiddetto RAMBUS. Esso si basa sull'uso di un insieme di tre linee di bit di indirizzo, cinque di riga e cinque di colonna, multiplexate; mediante un clock da 500 megahertz si fa viaggiare un pacchetti che contiene informazioni di indirizzamento e di dato sul bus, dividendo in modo da permettere accessi consecutivi a banchi differenti, aumentando l'efficienza del sistema.

Il fatto che un sistema di questo tipo necessiti l'uso di una circuiteria piuttosto complicata non ne ha permesso la commercializzazione, rendendolo quindi inutilizzato.

7.1.3 Content-Addressable Memory

La struttura di una CAM è molto simile (per non dire uguale) a quella di una RAM: una matrice contiene un insieme di celle il cui contenuto è un bit di informazione.

In cosa si differisce dunque questo tipo di architettura dalle RAM statiche tradizionali? Il fatto che per ogni locazione si aggiunge un EXOR tra il valore che è stato memorizzato e quelli che si inseriscono in ingresso, collegati in logica wired-or.

Volendo scrivere dei dati sulle bitline, i dati si propagano, ma vengono di fatto scritte solo variazioni rispetto ai dati già presenti: gli exor hanno il compito di riconoscere le variazioni e introdurre nella memoria esclusivamente queste, quindi quello di evitare di ri-scrivere informazioni già presenti. In uscita si ha un segnale che vale 1 solo se si ha un match completo tra la parola inserita e quella già presente, e questo per ogni riga.

La grossa innovazione della memoria è il fatto che le locazioni vengono riconosciute a seconda del loro contenuto: la memoria viene indirizzata a seconda del contenuto che si intende introdurvi (cosa che potrebbe ricordare, in un certo senso, la struttura di una hash table). Questo tipo di tecnologia si utilizza ogni volta che è necessario di effettuare ricerche molto veloci in hardware.

Collegando i match ad una SRAM tradizionale, con tante righe quanti sono i segnali di match, tutto viene collegato alle linee di selezione delle RAM, ottenendo dati di uscita. Ciò permette di risolvere in modo velocissimo le

ricerche in doppia chiave, ma ancor meglio di associare all'indirizzo un dato, inventando così le memorie tanto citate: le cache.

Le cache, di fatto, sono memorie CAM (Tag Memory) in cui si ha il dato, e una parte che contiene l'indirizzo; la CAM verifica il fatto che l'indirizzo sia presente in memoria, e se vi è un match conferma immediatamente la sua presenza; la velocità di ricerca è enorme poichè le linee operano in parallelo, fornendo dunque immediatamente un risultato.

7.2 Read Only Memory (ROM)

Le ROM in realtà non sempre sono fedeli al loro nome: il fatto che siano a sola lettura è tendenzialmente vero, anche se, a seconda della tecnologia con la quale sono realizzate, possono essere, in maniera più o meno semplice, essere quantomeno "riscritte".

Sostanzialmente, le scuole di pensiero per la realizzazione di memorie di questo tipo sono due:

- Memorie realizzate cn dati già presenti a partire dal processo di integrazione (mask programming): ROM. Molto veloci, ma altrettanto costose;
- Memorie programmabili in diverse maniere.

Memorie di questo tipo vengono utilizzate sostanzialmente per due motivazioni:

- Memorizzare del codice da eseguire che, per certo, non dovrà più essere cambiato;
- Realizzare funzioni logiche di vario tipo, generalizzando il concetto di mappe di Karnaugh e tavole di verità.

Uno degli usi "principi" di queste memorie è ad esempio realizzare funzioni di tipo non lineare, permettendo, al variare degli ingressi, di proporre uscite del tutto bizzarre (quali logaritmi, coseni, o simili).

Il concetto base è avere una word-line orizzontale, una bit-line verticale, che, a seconda delle connessioni, permettono o meno la propagazione del segnale logico per il circuito. Esistono diverse topologie di memoria, in grado di realizzare differenti tipi di strutture e di "piani" di lavoro.

Esaminiamo diverse soluzioni di realizzazione delle ROM, a livello più o meno commerciale:

PROM

Per programmare una ROM dunque si può semplicemente introdurre, in serie al collegamento tra i source dei transistori e il potenziale di riferimento, un fusibile: questa è l'idea alla base delle PROM, ossia delle Programmable ROMs: si fonde in modo selettivo e irreversibile il collegamento.

Questo tipo di tecnologia presenta un vantaggio e uno svantaggio:

- La programmazione di questo tipo di dispositivi è piuttosto semplice, e la può fare anche un utente, dotato delle macchine (comunque non troppo costose);
- Un fusibile, in termini di area di integrazione, è molto più grosso di un MOS, cosa che rende poco appetibile, in termini commerciali, memorie di questo tipo.

EPROM

Le EPROM (Erasable Programmable ROM) sono memorie basate su tecnologie di tipo differente, in grado di realizzare memorie programmabili e cancellabili, in modo più o meno semplice, anche da un utente. L'idea alla base di questo tipo di memorie è quella di usare particolari tipo di transistori, detti FAMOS: si tratta di transistori MOS in cui si introduce una sorta di schermo, in grado di inibire la creazione del canale di conduzione.

L'idea è la seguente: inserendo un gate aggiuntivo, detto "floating gate", tra il gate principale e il canale di conduzione di un normale MOSFET, in condizioni normali il floating gate è neutro, di fatto, a meno di alcuni piccoli effetti parassiti dettati dall'introduzione di capacità.

Se si alimentano il drain e il gate con una tensione molto alta, mettendo il source a 0 V, gli elettroni tendono ad andare verso il drain, ma anche verso il gate, spinti da un forte campo elettrico; essendo l'ossido di spessore limitato, gli elettroni tendono a "schiantarsi" verso il floating gate, che si carica negativamente; l'ossido di fatto è ancora intatto, poichè la rottura è solo elettronica, tuttavia l'energia degli elettroni non è sufficiente a permettere di ri-superarlo, dunque il gate anche in stabilità rimane carico negativamente. La tensione di soglia del transistore è cambiata: cambiando la tensione al gate, non si riesce più a condurre, a differenza di prima, dunque, a meno che non si mettano tensioni molto elevate, il transistore non reagisce più come prima. Questo, ha dunque "programmato" il transistore.

Il fatto che si possa "eliminare" l'informazione dipende dal fatto che, se si illumina il gate del transistore con una radiazione ultravioletta sufficientemente intensa (la luce del sole per alcuni minuti è sufficiente), gli elettroni

nel gate flottante acqusiscono l'energia sufficiente per andarsene, e tornare in situazione normale.

EEPROM

L'inconveniente delle memorie appena descritte consiste nel fatto che da programmare sono abbastanza balorde: sia la cancellazione che la scrittura, di fatto, si basano su operazioni dannose, che dunque limitano la vita della memoria, che non potrà essere riprogrammata per un numero infinito di volte.

Le EEPROM (Electricaly Erasable PROM) sono memorie cancellabili mediante i soli elettroni, mediante l'uso delle "leggi di Kirchhoff": il campo elettrico è tale da "cacciare" gli elettroni. Ciò si può in pratica realizzare in due maniere:

- FLOTOX: un processo più sofisticato di quello precedente, basato sull'effetto tunnel: utilizzando un ossido sufficientemente sottile, facendo quasi toccare la zona n, si riesce a far spostare gli elettroni con un semplice campo elettrico, per quanto esso debba comunque essere intenso; si avvicina la parte prossima al drain, con processi tecnologici piuttosto costosi (nei tempi in cui venivano realizzati), rendendo la cancellazione elettrica, ma comunque ripetibile per un numero limitato di volte.

- Flash EEPROM: con il miglioramento della tecnologia si assottiglia l'ossido sia dalla parte del source che da quella del drain, rendendo quindi la cancellazione effettuabile con il solo source, la scrittura con il solo drain. Questo tipo di cella permette di ottenere in modo elettrico sia scrittura che cancellazione, rendendo ripetibile l'operazione per un numero molto maggiore di volte rispetto alle tecnologie precedenti.

Tutte quelle finora presentate sono memorie di diverso tipo, sotto il punto di vista tecnologico; dal punto di vista dell'organizzazione, di fatto, non importa più di tanto quale sia la tecnologia: la parte di selezione/decodifica sarà sempre uguale, a meno di alcune eventuali aggiunte (a seconda delle possibilità di cancellazione o meno); ciò che cambia è il modo di trattare i segnali internamente alle memorie, ma la logica "esterna" ad esse concettualmente non dovrebbe cambiare particolarmente.

Capitolo 8

Introduzione ai circuiti sequenziali asincroni

In questo capitolo si presenterà un'introduzione al progetto e alle idee nascoste dietro i circuiti sequenziali asincroni, o macchine a stati asincrone.

Una macchina a stati asincrona differisce dalla sua parente sincrona poichè l'elemento di memoria campiona i segnali in modo puramente combinatorio, ossia senza clock. Anzichè un ritardo introdotto dalla differenza temporale tra l'istante di variazione del segnale e il tempo di campionamento, si ha un elemento di ritardo che traduce lo stato futuro calcolato dalla macchina a stati (next state, NS) nello stato presente (present state, PS), facendo evolvere in questo modo la macchina a stati. Invece di memorizzare lo stato mediante flip flop, gli elementi non vengono temporizzati, ma mandati direttamente in retroazione.

Si ha qualcosa di questo tipo:

Come accennato si introducono elementi di ritardo, quali il Δ. Si noti tuttavia che i ritardi, in ambito asincrono, acquisiscono un significato diverso: se precedentemente la reazione era a tempo discreto, ora di fatto è a tempo continuo, dal momento che, a meno dei vari delay di cui si sta parlando, il campionamento dovrebbe essere pressochè istantaneo.

Le macchine sincrone rappresentano una forza molto elevata, come già detto in precedenza: i glitch, fenomeni transitori, possono essere eliminati, mediante il pipelining: riducendo i percorsi combinatori, si riducono le probabilità di glitch e aumenta la frequenza di lavoro della macchina. Nel mondo asincrono i glitch non vengono eliminati gratuitamente come in quello sincrono, bensì devono essere trattati, in modo da evitare malfunzionamenti della macchina a stati. Senza un'opportuna trattazione i glitch possono provocare commutazioni spurie, dal momento che non esiste un tempo di campionamento in grado di "filtrarli".

Ai fini della trattazione considereremo le seguenti ipotesi semplificative, purtroppo non sempre presenti in un sistema:

- Si considera un solo ingresso variabile alla volta: quando un ingresso cambia, gli altri restano stabili;
- Il fatto che lo stato "torni indietro" dopo la variazione di un ingresso deve essere inibito: si deve aspettare la stabilizzazione nello stato nuovo, prima di cambiare ancora gli ingressi.

Esiste la possibilità di sfruttare a proprio favore questo fatto, come ad esempio in questo caso:

Questo circuito è un semplice oscillatore, la cui frequenza totale è il ritardo della catena: di fatto, in un sistema di questo genere, non esiste uno stato stabile. Questo è un classico esempio di progetti da evitare. Come nel caso dell'elettronica analogica, incappare in sistemi oscillanti è la cosa peggiore che possa capitare ma, purtroppo, potrebbe capitare spesso a un progettista non espertissimo.

Esiste un modo di evitare le macchine a stati asincrone oscillanti? Beh, esistono, come nel caso dei circuiti sincroni, tecniche in grado di progettare in maniera abbastanza sistematica circuiti di questo tipo. La differenza fondamentale è la seguente: esiste sempre una transizione tra stati, ma in questo caso, anzichè essere governata da un clock, è governata da questo fatidico Δ. Un'idea può essere quella di costruire la tabella di eccitazione, tenendo conto di questo fatto: l'evoluzione della macchina dipende dai ritardi reali, ossia dal fatto che vari prima un segnale Y_1 piuttosto che un Y_2. Una volta aver costruito la tabella in modo furbo, si fa l'allocazione degli stati, cercando di utilizzare una codifica degli stati sicura. I segnali del circuito devono tutti essere trattati come segnali che variano uno alla volta, cosa che nei sistemi reali spesso non è vera, dato che gli ingressi tendenzialmente possono cambiare un po' come vogliono. Da queste problematiche si può intuire che progettare sistemi asincroni sia veramente complicato.

Esiste un qualche vantaggio, dunque nella progettazione di questo tipo? Beh, uno di sicuro: l'estrema velocità che presentano, dal momento che agiscono in tempo pressochè reale. Le macchine inoltre spesso sono piccole: dal momento che gli elementi di memoria non sono presenti (poichè si utilizzano direttamente blocchi combinatori retroazionati) l'area del sistema sarà ridotta. Il circuito inoltre è realizzato da porte che commutano in tempi "distribuiti": nelle macchine sincrone le commutazioni sono tutte concentrate in un singolo istante di tempo, cosa che qua non capita, dal momento che le porte commutano solo quando i segnali variano, cosa che potrebbe di

fatto capitare di continuo, anzichè secondo una certa temporizzazione. Ciò implica che, se da un lato una macchina sincrona ha bisogno di un'alimentazione impulsiva, la cui ampiezza deve dipendere dal numero di porte che commutano contemporaneamente, una macchina asincrona crea meno disturbi sull'alimentazione di rete. A causa di questi problemi di alimentazione le piste ed i cavi vengono progettati di conseguenza, rendendo sia il progetto sia la funzionalità del sistema molto meno efficiente.

Cercare tecniche per realizzare circuiti asincroni stabili è utile anche dunque sotto questo punto di vista.

Può capitare (in realtà, capita di certo) di dover lavorare badando a dei glitch. Si parla di "static hazard" quando un impulso fa de-stabilizzare uno stato stabile. Si parla di "dynamic hazard" quando un glitch si forma durante una transizione del segnale, provocando variazioni strane di stato.

Il problema sta nel fatto che nelle macchine asincrone hazard di questo tipo, se non si previene in alcun modo, possono essere nefasti, in quanto presenti con una certa frequenza.

Come si possono evitare i glitch in sistemi di tipo asincrono? Beh, innanzitutto, cerchiamo di capire perchè si formano: essi si formano perchè si passa da una configurazione ad un'altra, passando da un implicante ad un altro (modellizzando il sistema mediante la mappa di Karnaugh) senza che si abbia una contiguità tra un implicante e un altro. Per evitare ciò, spesso, è sufficiente introdurre una certa ridondanza nel sistema: considerando più implicanti di quelli che servono, creando però contiguità tra i vari implicanti, si riesce ad evitare questo tipo di problema. In questo modo l'uscita non si troverà mai in una condizione a rischio, perdendo qualcosa sotto il punto di vista dei componenti presenti nel sistema ma guadagnando molto sotto il punto di vista della stabilità.

Può capitare quindi molto frequentemente l'uso di macchine asincrone. Ciò che può capitare è anche un qualcosa di tipo diverso: l'interazione tra ingressi (che potrebbero essere il mondo reale piuttosto che altro) asincroni con macchine sincrone, o viceversa. Esistono quattro casistiche, che richiedono tendenzialmente diversi tipi di trattamenti:

- Interazione asincrono - asincrono: caso generalmente non trattato, dove l'unico modo per creare un canale di comunicazione è usare sistemi quali l'handshake.

- Sincrono - asincrono: al fine di rendere tutto "a posto", è sufficiente che le uscite sincrone non presentino glitch.

- Asincrono - sincrono : se si riesce a sincronizzare gli ingressi mediante il clock, si riesce a rendere comunicabili le macchine.

- Sincrono - sincrono : è necessario o che i due clock siano già sincroni, o sincronizzare gli ingressi mediante il clock del circuito ricevente.

Per sincronizzare i circuiti esistono sotto-sistemi detti "sincronizzatori", il cui scopo è per l'appunto quello di eliminare situazioni di metastabilità e glitch vari. Ciò potrebbe tuttavia rallentare il sistema, introducendo la dipendenza da questi clock aggiuntivi. Si devono inoltre introdurre blocchi con memoria nel sistema:

Sostanzialmente un sincronizzatore è costituito da una cascata di flip-flop i quali evitano la metastabilità: introducendo tre flip-flop si rende quasi impossibile il fatto che il terzo della catena vada in uno stato di metastabilità, rallentando la macchina ma rendendola estremamente più stabile di prima.