

UNIVERSITÀ^{DEGLI} STUDI DI
NAPOLI FEDERICO II
Scuola Politecnica e delle Scienze di Base

Architettura dei Sistemi di Elaborazione

Appunti tratti dalle lezioni del Prof. N. Mazzocca

A cura di:

Rocco di Torrepadula Franca

Somma Alessandra

Indice

I Appunti	5
1 Ciclo di sviluppo in VHDL	6
1.1 Concetti introduttivi	6
1.2 Fattori che influiscono sul progetto	7
1.3 Classificazione delle macchine	8
1.4 Il linguaggio VHDL	9
1.5 Livelli di astrazione del VHDL	11
1.6 Progettazione in VHDL	12
1.7 Unità di Controllo e Unità Operativa	14
2 Macchine combinatorie	17
2.1 Ottimizzazione delle reti combinatorie	18
2.1.1 Metodo esatto di McCluskey per funzioni ad una sola uscita su due livelli	19
2.1.2 Branch and Bound	21
2.1.3 Metodo esatto di McCluskey per funzioni a più uscite su due livelli	22
2.2 SiS	26
2.2.1 Funzioni ad una uscita	26
2.2.2 Funzioni a più uscite	29
2.3 Classificazione delle macchine combinatorie	30
2.3.1 Macchine a priorità	30
2.3.2 Macchine di connessione	31
2.3.3 Implementazione VHDL di un multiplexer 8:1	35
3 Macchine Sequenziali	40
3.1 Classificazione delle macchine sequenziali	40
3.2 Schema di una macchina sequenziale	41
3.3 Flip-Flop RS	43
3.4 Flip-Flop D	45
3.5 Tempificazione delle macchine sequenziali	45
3.5.1 Flip Flop Edge-Triggered e Master Slave	46
3.6 Costruzione di un flip flop Master Slave	49
3.7 Costruzione di un flip flop Edge-Triggered	50
4 Contatori	53
4.1 Progetto del contatore mediante automa	53
4.2 Progetto tramite composizione di più contatori	54
4.2.1 Implementazione VHDL	57
4.3 Progetto del contatore ad incremento variabile	58
4.3.1 Indirizzamento relativo	60
4.3.2 Cronometro con intertempo	60
4.3.3 Implementazione VHDL	60

4.4	Principio di funzionamento del contatore	61
4.5	Contatore tramite registro a scorrimento	62
5	Registri a scorrimento	64
5.1	Registro a scorrimento serie-serie	64
5.1.1	Implementazione VHDL del SISO	65
5.2	Registro a scorrimento parallelo-parallelo	66
5.2.1	Implementazione in VHDL dell'SR PIPO	66
5.3	Registro a scorrimento serie-serie circolare	67
5.3.1	Implementazione VHDL del registro SISO circolare	68
5.4	Applicazione degli Shift-Register	70
5.5	Ricevitore seriale	71
5.6	Esempio di progettazione di un'unità operativa e unità di controllo	73
5.6.1	Come è fatto il clock della rete di controllo?	75
5.7	Protocolli	76
6	Macchine aritmetiche	79
6.1	Addizionatori	79
6.1.1	Half Adder	79
6.1.2	Full Adder	80
6.1.3	Ripple Carry Adder	81
6.1.4	Sottrattore	82
6.1.5	Sommatore e sottrattore	82
6.1.6	Carry Look Ahead	84
6.1.7	Sommatore veloce: Carry Select	86
6.1.8	Sommatore veloce: Carry Save	87
6.2	Moltiplicatori	89
6.2.1	Moltiplicatore binario a due bit	89
6.2.2	Prodotto come somma per righe	89
6.2.3	Prodotto come somma per diagonali	91
6.2.4	Prodotto come somma per colonne	91
6.2.5	Moltiplicatore MAC	93
6.2.6	Moltiplicatore di Robertson	94
6.2.7	Moltiplicatore di Booth	98
6.3	Divisori	100
6.3.1	Modalità restoring	102
6.3.2	Modalità non restoring	103
7	Comunicazione seriale	105
7.1	Comunicazione tra dispositivi tramite interfaccia seriale: UART	107
7.2	Analisi del codice	108
7.3	Ricevitore	112
7.3.1	Controllo degli errori	116
7.3.2	Incoming Data Counter	118
7.3.3	Receiving Shift Register	118
7.4	Trasmettitore	119
7.4.1	Transfer Data Counter	123
7.4.2	Transfer Shift Register	123

8 Dispositivi programmabili e FPGA	124
8.1 Programmable Logic Device (PLD)	124
8.2 FPGA	126
8.2.1 Famiglia Spartan 3E	128
8.3 Protocollo JTAG	133
8.4 Timing Analysis	136
8.5 Timing Analysis: CLA puramente combinatorio	139
8.6 Timing Analysis: CLA con registri	140
9 Il processore	142
9.1 La logica microprogrammata	142
9.2 Esempi di processore	143
9.2.1 Processore a 5 blocchi	144
9.2.2 Architettura a registri generali	146
9.3 Il processore MIC-1	148
9.3.1 Datapath	148
9.3.2 Unità di controllo	151
10 La memoria	155
10.1 Memorie associative	158
II Appendici	161
A Workflow dal problema alla sua soluzione	162
B Sistemi complessi: analisi e sviluppo	174
C Flip Flop Master-Slave ideale	204
D Overview dei dispositivi logici programmabili	210
E JTAG	226
F Il processore MIC-1	236

Parte I

Appunti

Capitolo 1

Ciclo di sviluppo in VHDL

1.1 Concetti introduttivi

In questo corso, trattiamo la progettazione su schede molto piccole, fatte in questo modo: esse presentano una serie di connettori ed all'interno hanno svariati componenti, quali il processore, la memoria, dispositivi di I/O ed un blocco che chiamiamo “*altro*”.

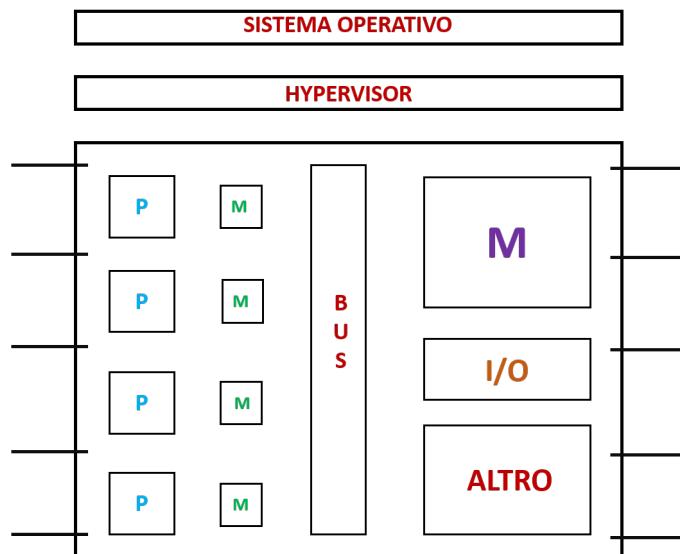


Figura 1.1: La scheda.

Per la precisione, queste schede sono tipicamente caratterizzate da molteplici processori, in particolare possono essere *multi-core* (più processori dello stesso tipo) o *many-core* (più processori di tipo diverso). Questi sono oggetti che devono fare calcolo, per cui dispongono sicuramente di una memoria (le M verdi), tuttavia se avessero solo questa a disposizione il sistema non funzionerebbe. Per interscambiare informazioni, sono, infatti, collegati ad un bus, a cui è a sua volta collegata una memoria (la M viola). Sappiamo che il processore è una macchina sequenziale, in grado di elaborare un programma che si trova all'interno della memoria e che possiamo caricare in modo permanente o con un programma. L'intero sistema deve portare dati verso l'esterno, motivo per cui sono necessari i dispositivi di I/O.

Una scheda ha un numero di connettori limitato, mentre il numero di dispositivi di I/O è più elevato. Questo perchè tra I/O e fili di connessione c'è una matrice programmabile che decide

quali sono realmente i dispositivi di I/O che vanno in output. A seconda dell'applicazione è possibile decidere quanti dispositivi devono andare verso l'uscita, è dunque possibile programmare la matrice in maniera coerente e si otterranno sui connettori tanti dispositivi pari a quelli selezionati.

“Altro” non è un programma, può essere realizzato con dell'hardware dedicato, macchine combinatorie (senza memoria) o macchine sequenziali (con memoria). La sfera complessa è allora questo “altro”, dove è possibile programmare l'uscita verso i pin, i dispositivi di I/O e così via. Addirittura, se questo “altro” è particolarmente complesso, è possibile scaricare in esso un processore con 4 cuori, cioè quattro unità attive all'interno della stessa scheda. Il processo realizzativo di un processore è lo stesso di una macchina sequenziale, ovviamente con una complessità maggiore. I sistemi complessi si chiamano **system-on-chip**, un sistema all'interno dell'integrato, cioè possiamo costituire un'intera macchina mettendo tutto all'interno di “altro”; i componenti messi all'interno sono detti **IP-core**.

“Altro” si può realizzare in due modi diversi:

- **ASIC - Application Specific Integrated Circuit**: componenti elettronici sviluppati in modo completamente programmabile. In altre parole, è come se avessimo una piastra di silicio libera, su cui possiamo, implementando il ciclo di sviluppo (descritto a partire dalle sezioni successive di questo capitolo) realizzare tutte le porte AND, OR, NOT e così via all'interno. E' complesso ed è il dominio dell'ingegneria elettronica, per cui non ci soffermeremo su questo.
- **FPGA - Field Programmable Gate Array**: sistema in cui è possibile avere programmabilità, ma la piastra di silicio non è vuota, ci sono già dei componenti all'interno, che devono essere collegati, per cui **il nostro obiettivo è programmare i collegamenti**. E' chiaro che è una tecnica meno efficiente di quella precedente, poiché (almeno in questo corso) non ci troveremo mai nella condizione di dover usare tutti i componenti.

Tutto questo deve poi essere utilizzato, si crea un livello software, **hypervisor**, che deve conoscere come è fatta la scheda altrimenti non potrebbe gestire le risorse fisiche. Il sistema operativo è al di sopra di questo strato al fine di creare una separazione tra gestione delle risorse fisiche ed i suoi compiti. Il software che noi alla fine vediamo si pone sopra lo strato del sistema operativo ed il linguaggio è ancora più sopra. Se ci poniamo ad un livello molto alto, è difficile vedere fattori come il tempo, la velocità, il consumo di un componente o addirittura la sua rottura. Ecco perché se vogliamo fare un qualcosa in relazione a questi sistemi, che rientrano nel ramo dei **sistemi embedded**, dobbiamo conoscere questo modello.

1.2 Fattori che influiscono sul progetto

Quando realizziamo un progetto, per prima cosa è necessario trovare un **modello di definizione**, tramite cui descrivere il sistema da realizzare. Le forme di tale modello possono essere diverse, possiamo descriverlo tramite una tabella di verità, tramite un automa o tramite un linguaggio. Il modello, attraverso un opportuno **ciclo di sviluppo**, deve essere realizzato su una **tecnologia realizzativa**. La tecnologia e il modello scelti possono polarizzare il ciclo di sviluppo, ad esempio, a seconda della tecnologia potrebbe essere o meno necessario effettuare la minimizzazione delle funzioni booleane: se realizziamo la funzione booleana tramite una memoria ROM, e non tramite porte logiche, la minimizzazione non porta vantaggi, in quanto la memoria ROM realizza la tabella di verità stessa. Negli FPGA, in cui le funzioni booleane vengono realizzate con memorie ROM allora non ha importanza la minimizzazione, negli ASIC invece, in cui progettiamo e inseriamo le porte logiche che ci servono, ha senso la minimizzazione perché ci permette di utilizzare meno porte. Ciò ci consente di capire che **ciò che si fa durante il ciclo di sviluppo dipende dalla tecnologia target**.

Un primo fattore che influisce sul progetto è il **tempo di risposta** di un componente (T_R). Ogni circuito è caratterizzato da tale parametro ed è fondamentale, in quanto indica quanto tempo impiega il sistema a produrre una variazione dell'uscita a una variazione dell'ingresso. Noto T_R conosciamo anche il suo reciproco, ovvero $\frac{1}{T_R}$, la frequenza massima a cui può lavorare il sistema. Sul tempo di risposta incide come sono fatti i componenti e i collegamenti tra di essi. **Tali collegamenti rappresentano infatti un circuito RC o RCL, dunque un filtro** che all'aumentare della frequenza agisce, rendendo impossibile il funzionamento del sistema (figura 1.2). Per migliorare il tempo di risposta possiamo agire sui componenti e sui collegamenti.

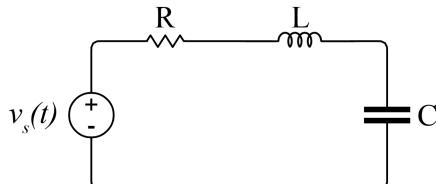


Figura 1.2: Circuito RLC

Supponiamo, ad esempio, che nel circuito non ci sia l'induttanza L , che è presente in un sistema con una spira, nel filo avremo sicuramente la resistenza R e il condensatore C , in quanto non esiste un oggetto puramente resistivo. L'impedenza è data da $R = \omega C$, se aumenta la frequenza ω , aumenta l'impedenza. Tipicamente i circuiti lavorano a frequenze elevate, se l'impedenza aumenta non vengono fatte passare le alte frequenze, secondo il comportamento di un filtro passa-basso. Le alte frequenze sono le frequenze responsabili del rapido passaggio del segnale da 1 a 0 e più il sistema le blocca, più l'onda che riceve viene smussata, si deforma, fino a diventare talmente piatta da non poter distinguere il valore 0 dal valore 1. C'è quindi un limite fisico di frequenza, oltre cui il circuito non può andare, e ciò dipende dalla lunghezza del filo.

Anche la **temperatura** è un fattore che influenza il progetto. Ciascun oggetto lavora più o meno bene a seconda del range di temperatura a cui si trova. Se varia il sistema termico variano le caratteristiche dei componenti. Per tale motivo in alcuni sistemi si utilizza la *conduzione forzata*, si sfruttano delle ventole per riportare il sistema alle condizioni di funzionamento nominale.

L'ultimo fattore invece riguarda le **schede**, ovvero dove mettiamo i componenti. Se parliamo di un sistema on-board allora tutti i componenti vanno su una board. Se parliamo di SoC (System on Chip), tutto va sul componente integrato. In ogni caso, più lontani sono i componenti e più lunghi sono i fili con cui li collegiamo, dunque gli effetti capacitivi pesano di più e la frequenza diminuisce maggiormente.

1.3 Classificazione delle macchine

Quando dobbiamo risolvere un problema possiamo utilizzare due approcci, o utilizziamo una **macchina programmabile**, dunque una macchina general purpose, oppure utilizziamo una **macchina dedicata**, che comunque a sua volta potrebbe essere fatta da macchine programmabili. La macchina programmabile stessa, in realtà, è un sistema dedicato ad elaborare un programma. Una macchina dedicata può essere realizzata, a sua volta, come una **macchina combinatoria** - descritta dalla relazione $U = f(I)$ - o come una **macchina sequenziale** - descritta dalla relazione $U = f(I, S)$, nel caso di *macchina di Mealy*, oppure da $U = f(S)$, nel caso di una *macchina di Moore*. Matematicamente possiamo descrivere le macchine combinatorie e le macchine sequenziali tramite **tabelle di verità** e **automi**, rispettivamente. Questi modelli però **non contengono il concetto di tempo**, relativo alla tecnologia realizzativa, dobbiamo in qualche modo tenerne conto. Dal punto di vista realizzativo la tabella di verità si implementa con un insieme di porte logiche, l'automa invece oltre a ciò richiedere anche dei registri, tramite cui mantenere il concetto di stato. Dunque, le macchine sequenziali al loro interno hanno una parte comune comunque combinatoria:

il modello di macchina combinatoria influenza anche quello di macchina sequenziale. Quando inseriamo il concetto di tempo nelle reti combinatorie si pone il **problema delle alee**. Dall'altro lato, nelle macchine sequenziali bisogna rispettare la regola $U = f(I, S)$ e per farlo occorre tempificare opportunamente, bisogna risolvere il **problema di tempificazione**. Per far fronte a ciò come registri non usiamo delle **macchine latch**, che catturano sempre il segnale quando sono abilitate, ma delle **macchine edge-triggered**, che modificano il loro stato solo alla variazione del livello del segnale di abilitazione. Una macchina sequenziale inoltre, come sappiamo, può essere **sincrona** o **asincrona**, in quest'ultimo caso bisogna anche fare attenzione all'evoluzione libera, dovuta alla retroazione senza registri, anche qua si possono avere problemi di alee, risolvibili con un'opportuna codifica degli stati.

1.4 Il linguaggio VHDL

Nella pratica, un problema complesso si risolve con l'approccio “**divide et impera**”, *si decompon* il problema in sotto-problemi sempre più piccoli, così da dominarne la complessità. È fondamentale il concetto di **gerarchia**: dobbiamo individuare i moduli che compongono il problema e i collegamenti tra di essi. Risulta importante anche il concetto di modulo esistente, ovvero una **libreria**, con cui il sistema oltre che modulare diventa riusabile.

Il nostro obiettivo è capire se tramite un certo livello di astrazione, che però ci consenta di esprimere il concetto di gerarchia, possiamo descrivere il sistema in una modo più semplice, con l'intento di sintetizzarlo su una tecnologia. Lo strumento che ci permette di fare ciò è un linguaggio. Il linguaggio VHDL è uno degli strumenti con cui possiamo descrivere il modello di definizione del nostro sistema. Esso è un **linguaggio di descrizione dell'hardware (HDL)**, a differenza dei linguaggi visti fino ad ora, non descrive un algoritmo ma moduli e segnali, dunque non è compilato: non produce un eseguibile che gira su una macchina ma la descrizione formale di un sistema. Ad ogni attività corrisponde un'azione fisica, il nostro obiettivo è imparare a descriverla.

Il linguaggio che utilizziamo è finalizzato a ciò che dobbiamo realizzare, a livello tecnologico. I livelli di gestione della tecnologia sono due, gli FPGA e gli ASIC, la differenza fondamentale è la libertà che ci forniscono. Dobbiamo conoscere il livello di integrazione che ci fornisce la tecnologia (ovvero il numero di componenti che possiamo inserirvi), eventuali problemi di velocità, di temperatura. Nel costruire il sistema dobbiamo tener conto di tutti questi parametri, come visto nel paragrafo 1.2. Un altro fattore importante è il **costo**, che è funzione anche del tempo in quanto una aumento del tempo implica un aumento del costo. Rispetto all'FPGA, l'**ASIC ha un maggiore livello di integrazione, è più veloce, da meno problemi di propagazione e la temperatura è minore**. Ciò è dovuto al fatto che possiamo mettere sulla scheda solo ciò che effettivamente ci serve, dunque consuma di meno. Di contro, l'**ASIC ha un costo maggiore** perché il silicio non presenta forme preconfigurate: il tempo di lavorazione è maggiore. Noi dobbiamo capire su che sistema target dobbiamo lavorare, in quanto è influenzato dalla tecnologia.

Il VHDL ci permette di descrivere sistemi complessi, V, nell'acronimo, sta per *Very High Speed Integrated Circuits*, descrive sistemi molto veloci. Tramite questo linguaggio possiamo esprimere gerarchie, definire i moduli e le loro interazioni, possiamo usare le librerie e infine descrivere le formule che caratterizzano il nostro sistema. Possiamo scegliere il livello di astrazione a cui spingerci, il più alto è il **livello behavioral**, il più basso è il **livello porta**.

Il VHDL è un **linguaggio concorrente**, in quanto l'hardware è concorrente. Non essendo un linguaggio compilato, non possiamo eseguirlo per capire se abbiamo scritto il codice corretto, dobbiamo effettuare una **simulazione**. Valutiamo il comportamento reale del sistema in un mondo “ideale”, quanto più questo è simile all’ambiente reale e tanto meno avremo problemi in fase di effettiva sintesi. Per effettuare la simulazione allora inseriamo il concetto di tempo, teniamo conto dei ritardi che i componenti reali hanno, e se il risultato non è corretto dobbiamo descrivere nuovamente il sistema per poi risimularlo. Effettuiamo un *loop* di progettazione.

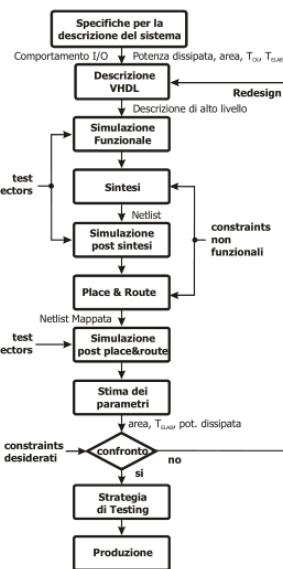


Figura 1.3: Ciclo di sviluppo VHDL

La figura 1.3 mostra il **ciclo di sviluppo del VHDL**. Esistono diversi livelli di simulazione, in primo luogo abbiamo la *simulazione funzionale* che può essere raffinata e può portare a una *sintesi*. Possiamo poi simulare il sistema sintetizzato per vedere se funziona effettivamente come avevamo pensato di farlo funzionare, dunque facciamo una *simulazione post-sintesi*. Poi possiamo fare il *place & route* (allocare il sistema in un sistema fisico) ed effettuare la *simulazione post-place & route*. Questa simulazione è diversa da quella precedente perché, una volta allocato il sistema fisico, possiamo simulare i parametri del circuito e se sono corretti abbiamo raggiunto lo scopo, possiamo fare il testing e mandare il sistema in produzione. Se non sono corretti dobbiamo ritornare sopra, risimulare funzionalmente il circuito per vedere se di fatto si trova o meno. Esistono delle problematiche di creare dei vettori di test (per testare il sistema) e problematiche relative a vincoli non funzionali, che consentono di dire se il sistema è fisicamente realizzabile o meno.

Dobbiamo capire come descrivere cose che poi si possano sintetizzare, **dobbiamo distinguere ciò che è simulabile tutto da ciò che invece è sintetizzabile**. Tutto è simulabile ma non tutto è sintetizzabile. Se vogliamo realizzare un componente possiamo decidere di descriverlo in modo molto verticale, così che sia sintetizzabile, e tutto l'ambiente intorno in modo più generico, in modo che, anche se non fosse sintetizzabile, *comunque sia provabile in simulazione*. Per fare il componente dobbiamo comunque descrivere anche tutto il sistema complesso in cui si trova, dunque **descriviamo il sistema più complesso a un alto livello di astrazione e poi il componente lo facciamo molto specifico, così da poterlo realizzare**. Se vogliamo simulare può bastare anche un livello di descrizione alto, per sintetizzare serve un livello di descrizione più basso, per far sì che poi il compilatore possa fare le operazioni di sintesi. Dunque, **più precisa è la descrizione e più è probabile che il sistema fisico lavori come quello logico**. Il sistema logico deve generare una *netlist*, ovvero *l'insieme dei componenti e della loro descrizione in termini comportamentali*. La netlist va sulla tecnologia, se simuliamo male avremo problemi quando andremo sulla tecnologia. **Simulare bene vuol dire fornire una buona descrizione fisica del sistema e un avanzamento del tempo logico opportuno** (che potrà essere quantizzato o ad eventi). Il nucleo allora è il simulatore, il GHDL o Xilinx, la differenza è che il primo non genera la netlist, dunque non ci permette di andare sulla tecnologia, mentre Xilinx sì. La figura 1.4 riassume quanto considerato finora.

Il vantaggio del sistema simulato, rispetto al sistema reale, è che possiamo facilmente vedere

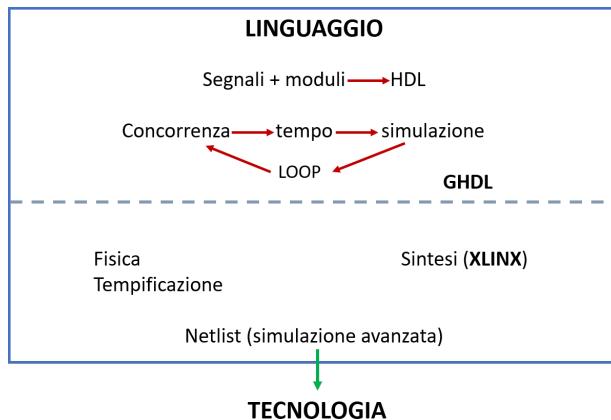


Figura 1.4: Schema concettuale del ciclo di sviluppo

qualunque grandezza fisica, non solo segnali di ingresso e uscita ma anche punti intermedi. Nei sistemi reali invece vedere un segnale intermedio è complesso, sono infatti segnali interni. Dunque, fare il test sul sistema simulato risulta più semplice, nel sistema reale invece dovremmo inserire dei criteri di test interni, altrimenti è testabile solo a livello di ingresso/uscita, ciò vuol dire che se c'è un problema lo notiamo, perché l'uscita è sbagliata, ma non sappiamo determinare cos'è successo. **Il simulatore invece ci consente di prendere facilmente i segnali interni, fondamentali per il test.**

I sistemi che devono generare la netlist devono gestire i collegamenti tra le reti combinatorie. Notiamo che in realtà anche i registri sono fatti tramite reti combinatorie, dunque tutto si può ricondurre al modello di macchina combinatoria. Tali reti possono essere realizzate in due modi, tramite porte AND/OR o tramite memorie ROM. Nel primo caso configureremo opportunamente le porte per fare ciò che ci interessa, nell'FPGA però il problema è che il livello di integrazione e la velocità non sono alti, c'è infatti un sistema di interconnessione che passa tra le reti, impiegando un certo tempo. Il consumo non è ottimo intollerabile perché non utilizziamo molti dei componenti che comunque si trovano sulla scheda. La distanza tra le porte richiede fili lunghi, c'è dissipazione, aumenta la temperatura e tutto ciò va gestito. L'alternativa è realizzare la rete combinatoria attraverso una memoria ROM, in questo modo il tempo è costante. La minimizzazione, come visto, non serve perché la ROM realizza la tabella di verità stessa, non le porte logiche, sarà però poi fatta di porte logiche.

1.5 Livelli di astrazione del VHDL

Possiamo decidere a che livello di astrazione descrivere il sistema. Il livello più basso è il **livello porta**, descriviamo il sistema in termini delle porte AND e OR che lo costituiscono. In realtà, il livello ancora più basso sarebbe il **livello fisico**, che riguarda proprio gli elettroni nei transistor. Grazie al concetto di gerarchia, di modularità, possiamo anche descrivere i differenti componenti a livelli diversi. Ad esempio, il componente che vogliamo effettivamente realizzare, lo descriviamo a livello basso, così che sia sintetizzabile, tutto ciò che invece ci serve per testarlo, come ad esempio il sistema che gli fornisce l'abilitazione, lo descriviamo in termini più astratti, anche in termini **algoritmici (livello comportamentale)**. In questo modo sarà simulabile, che è ciò che ci serve, anche se non è detto sia sintetizzabile, ma d'altronde nella realtà quel pezzo sarà sostituito dal sistema che effettivamente fornisce l'abilitazione al nostro.

A livello strutturale descriviamo il sistema in termini di sotto-componenti e loro interconnessioni. Esistono 3 possibilità per fare ciò:

- **Livello Data-Flow:** descriviamo la struttura *in termini di flussi di dati* che entrano ed escono da essa. Descrivendo ciò con gli operatori logici, è probabile che il sistema risulti sintetizzabile;
- **Livello Transfer-Register (RT):** descriviamo il sistema in termini di *registri e reti combinatorie*, che modificano i valori che passano tra due registri (figura 1.5). Attiviamo il registro, la rete combinatoria fa un calcolo e manda il valore risultante al registro successivo. Ciò è simile a quello che accade nell'FPGA, per cui è probabile che sia sintetizzabile;
- **Livello Porta:** descriviamo il sistema in termini di porte, utilizzando delle librerie. Il vantaggio è che le librerie oltre che riusabili, sono sicuramente sintetizzabili.

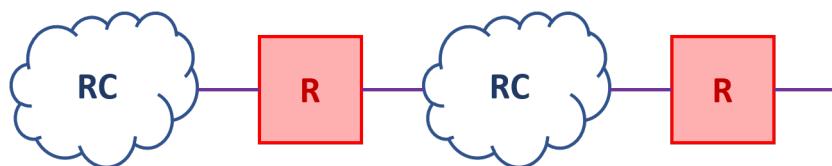


Figura 1.5: Modello transfer-register

N.B: Per approfondimenti riguardanti questo argomento, si consiglia di consultare l'appendice A.

1.6 Progettazione in VHDL

Per progettare un sistema tipicamente ci muoviamo su 3 livelli:

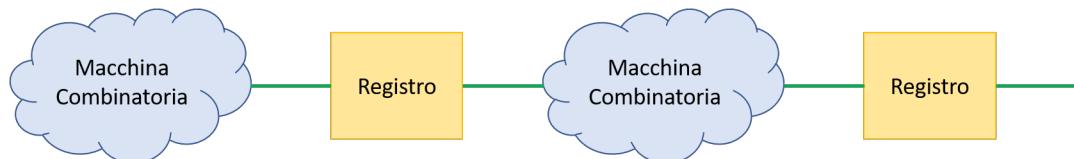
- Specifica del problema;
- Individuazione del modello;
- Realizzazione tecnologica.

Da un lato dobbiamo specificare il problema opportunamente, poi dobbiamo trovare un modo per rappresentarlo e infine dobbiamo scegliere la struttura tecnologica tramite cui realizzarlo. Quando vogliamo risolvere un determinato problema per prima cosa dobbiamo **identificare l'architettura**, serve allora un modello che indichi se si tratta di una *macchina combinatoria* o di una *macchina sequenziale*. Tipicamente, per le macchine combinatorie un primo modello è la *tabella di verità*, per le macchine sequenziali invece l'*automa*. In realtà ciò non basta, se pensiamo, ad esempio, a un processore, che è una macchina sequenziale, otterremmo un automa troppo grande. Se abbiamo infatti un processore i cui codici operativi sono rappresentati su stringhe da 16 bit, avremmo una macchina con 2^{16} ingressi, quindi un automa con 2^{16} archi, diventerebbe troppo complesso. **L'identificazione del modello dipende dalla complessità dell'architettura da realizzare** e, se la macchina risulta particolarmente complessa, conviene utilizzare il modello dell'automa o della tabella di verità a livello più basso, per i componenti più piccoli, a livello più alto, invece, conviene dividere concettualmente la macchina in due parti: **unità operativa (UO)** e **unità di controllo (UC)**, la prima decide cosa fa la macchina e la seconda invece controlla e comanda ciò che avviene. Va sottolineato che queste due unità sono comunque costituite da un insieme di macchine combinatorie e macchine sequenziali, quindi non cambia l'architettura ma il modello di rappresentazione: *individuiamo prima la struttura globale e poi ragioniamo sui componenti*.

L'ultimo livello è la tecnologia, il sistema deve essere sintetizzato e per poterlo fare è necessario diminuire il gap tra il modello e la tecnologia. Serve uno strumento per descrivere opportunamente il sistema e quello che usiamo è un linguaggio, in particolare il VHDL. Il linguaggio infatti è dotato di un potere descrittivo maggiore rispetto a diagrammi e tabelle e, soprattutto, non è ambiguo nella codifica, rispetto a una particolare tecnologia. Il problema consiste nel fatto che, se non descriviamo correttamente il sistema, potrebbe non essere facile farlo diventare un'attività tecnologica, sintetizzarlo. Nella progettazione partiremo dall'architettura e la scomporremo in UO e UC, poi procederemo dividendole a loro volta in sotto-unità. Stiamo definendo un approccio strutturale, dunque serve un linguaggio che ci consenta di descrivere gerarchie, strutture, come del resto fa il VHDL.

Descriviamo i blocchi costituenti l'architettura e i collegamenti tra di essi. Possiamo inoltre scegliere a che livello di astrazione effettuare tali descrizioni: a livello comportamentale o a livello data-flow. Nel primo caso ci manteniamo a un livello di astrazione alto, per cui il gap tecnologico aumenta, lasciamo al compilatore l'onere di individuare la struttura per procedere alla sintesi e talvolta potrebbe essere complicato. Se invece definiamo noi stessi la struttura e poi i vari componenti, mediante una descrizione data-flow, riduciamo il gap e per il compilatore sarà più agevole procedere alla sintesi.

Durante il ciclo di sviluppo incontriamo due problemi differenti, da un lato dobbiamo capire se il circuito funziona, dobbiamo simularlo, dall'altro dobbiamo fornire opportune informazioni allo strumento di sintesi, per arrivare alla completa realizzazione. Dobbiamo allora capire cosa facciamo per implementare il circuito, per simularlo, e cosa invece per realizzare la sintesi. Ad esempio, strumenti come il testbench e la parola chiave *after* servono unicamente alla simulazione, introduciamo dei ritardi con *after* e poi tramite il testbench generiamo dei segnali per verificare in simulazione il corretto funzionamento. Quando andiamo a sintetizzare il sistema sulla scheda troviamo dei componenti fisici per cui il ritardo è già intrinsecamente presente. Allo stesso modo, generiamo i segnali simulati nel testbench ma, quando andremo a utilizzare il sistema sintetizzato, dovremo generare dei veri segnali di input e ciò avviene tramite gli switch, i clock, i tasti, etc. Tutti questi strumenti servono in simulazione per risolvere il problema del tempo ma è importante sottolineare che tempo simulato e tempo reale sono differenti ed è più complesso gestire il tempo reale. Il tempo costituisce la maggiore difficoltà nelle macchine reali, per questo motivo le rappresentiamo con il modello con macchine combinatorie e registri, mostrato in figura 1.6.



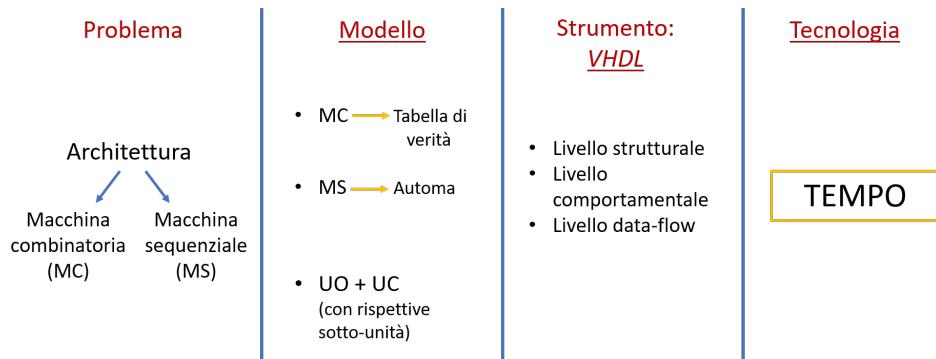


Figura 1.7: Schema concettuale

In fase di simulazione indica che l'oggetto è sensibile alla variazione del clock, quando il clock poi diventa 1 (dunque alla variazione 0-1, sul fronte di salita); al circuito che sintetizza indica di dover usare un registro che sia sensibile alla variazione 0-1 e non a quella 1-0. Il criterio, quindi, lo descriviamo tramite il linguaggio, lo usiamo in simulazione e lo portiamo avanti, analogamente, nella sintesi. **La cosa più importante è esprimere il concetto di abilitazione.** Ciò è presente anche nel process, nell'interfaccia definiamo infatti la *sensitivity list*, l'insieme di segnali a cui è sensibile l'oggetto. Il process lavora alla variazione di uno di tali segnali, se non avvengono variazioni rimane fermo. Ricordiamo che il process rappresenta un oggetto che va montato all'interno della scheda, per questo la sua descrizione non può essere concorrente. Al suo interno ovviamente sarà fatto di porte logiche e quindi realizzato in modo concorrente ma *la sua descrizione rimane non concorrente*. **Il process, dal punto di vista descrittivo, rappresenta un oggetto non concorrente ma la sua realizzazione, in termini di porte logiche, sarà comunque concorrente.**

Lo schema in figura 1.7 riassume quanto esposto fino a questo punto.

1.7 Unità di Controllo e Unità Operativa

Supponiamo di voler realizzare un processore che riceve istruzioni, formate da codice operativo e operandi, e le svolge utilizzando una memoria. **Se abbiamo 8 codici operativi e 4 possibili modi di indirizzamento abbiamo una macchina con 3+2 bit, ovvero 5 bit, dunque vi sono $2^5 = 32$ possibili ingressi.** È evidente che progettare questa macchina attraverso un automa, con la logica del riconoscitore di stringhe, risulterebbe complicato, avremmo 32 archi differenti. Inoltre, se consideriamo processori più articolati, a 16 o 32 bit, la complessità cresce in modo esponenziale con le potenze di 2. Dobbiamo trovare una strada alternativa all'automa per procedere agevolmente alla progettazione. L'idea è separare i problemi, da un lato mettiamo la memoria, dall'altro il processore e infine un bus che permetta loro di comunicare (figura 1.8). In questo modo abbiamo identificato un'unità operativa, ovvero la memoria, contenente dati e istruzioni, e un'unità di controllo, il processore, secondo il *modello di Von-Neumann*. Il modello dell'automa presupponeva che fossimo noi a dare le stringhe in ingresso al processore, adesso invece le informazioni sull'istruzione che il processore deve eseguire sono presenti in memoria, dunque serve qualcosa per andare in memoria e prelevarle, aggiungiamo allora il registro **Memory Address (MA)** e il **Memory Buffer (MB)**. I registri servono infatti a separare le unità, permettono di leggere e scrivere i dati in memoria. Serve anche un **Program Counter (PC)** che mantiene l'indice dell'ultima locazione a cui si è acceduti in memoria. Per facilitare l'uso delle informazioni in memoria, inseriamo anche un ulteriore insieme di registri. Quello che abbiamo fatto è **definire l'unità operativa e le sue sotto-unità**, essa è in grado di leggere e scrivere in memoria, sposta i dati. Se vogliamo realizzare qualcosa di più complesso, come somme, moltiplicazioni o riporti, dobbiamo inserire un ulteriore

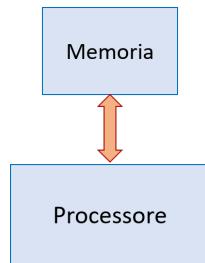


Figura 1.8: Memoria e processore

componente, l'**ALU**, e un **registro di stato** a suo supporto, per mantenere eventuali risultati o carry generati. Per riuscire a far funzionare tale unità operativa serve un **bus**, su cui più oggetti possono mettere i dati e un solo oggetto alla volta può prelevarli, non sapendo dove si trova è come se avessimo un sistema con n sorgenti e m destinazioni, dunque è costituito da un mux/demux in cascata.

L'unità di controllo invece decide cosa fare, in base alle istruzioni che riceve, serve mantenere l'istruzione da eseguire (IS). Riconosciuta l'istruzione, la sequenza di bit, genera una serie di abilitazioni per far muovere sorgente e destinazione, in modo da svolgere l'operazione. **L'unità di controllo quindi è un generatore di segnali di abilitazione**, in base ai segnali che riceve. Una volta progettata l'unità operativa allora dobbiamo capire, dato il codice operativo, che segnali abilitare per far avvenire gli scambi. *L'unità di controllo va progettata sempre dopo l'unità operativa, altrimenti non sappiamo cosa controllare.* Lo schema è mostrato in figura 1.9.

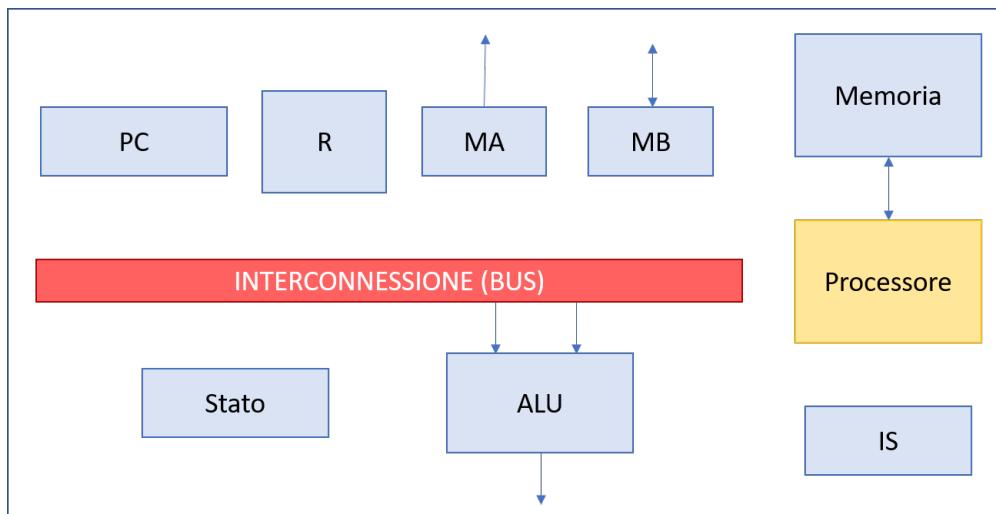


Figura 1.9: Schema del processore.

Abbiamo essenzialmente due modi per realizzare l'unità di controllo, se usiamo una memoria ROM si parla di **logica microprogrammata**, per cui eseguiamo la sequenza preregistrata nella ROM; se la realizziamo mediante porte logiche si parla invece di **logica cablata**. Se stiamo progettando un microcontrollore in cui vogliamo poter cambiare ciò che fa usiamo la logica microprogrammata, in questo modo basta riscrivere il contenuto della ROM. È importante sottolineare che la stessa unità operativa può avere un'unità di controllo in logica cablata o microprogrammata, essa infatti produrrà comunque i segnali di abilitazione, l'unica differenza è in come avviene tale generazione. Dobbiamo riuscire a capire se l'oggetto che descriviamo verrà sintetizzato in logica microprogrammata o in logica cablata. A livello comportamentale non siamo in grado di dedurlo, scendendo a un livello di astrazione più basso invece possiamo. *Se descriviamo l'unità di controllo in termini di automa probabilmente verrà realizzata in logica cablata; se invece la descriviamo in forma di tabella probabilmente verrà realizzata tramite una ROM in cui precarichiamo tutte le possibili sequenze, che serviranno a muoversi sul bus.* La difficoltà consiste ora nell'individuare il punto della ROM in cui è mantenuta la sequenza che ci interessa. **Quando progettiamo un sistema dobbiamo prima di tutto disegnare l'unità operativa e capire come funziona, anche tramite un diagramma temporale. Successivamente, progettiamo l'unità di controllo e scegliamo se realizzarla in logica cablata o in logica microprogrammata.**

N.B: Per approfondimenti riguardo a questo argomento, si rimanda all'appendice B.

Capitolo 2

Macchine combinatorie

Un problema è combinatorio quando l'uscita in ogni istante di tempo dipende solo dall'ingresso e non da quelli passati. Un problema è di tipo sequenziale quando l'uscita dipende sia dall'ingresso che dallo stato.

La macchina combinatoria realizza una funzione:

$$U=f(I)$$

dove I ed U sono rispettivamente gli insiemi limitati dei valori di ingresso e di uscita.

Se le funzioni sono definite su un insieme finito di punti, sono rappresentabili su una **tabella di verità**, da una parte le variabili indipendenti e dall'altro lato i valori che la funzione assume in corrispondenza delle variabili indipendenti. Le funzioni si possono scrivere sotto forma di somma di prodotti (disgiuntiva di congiunzioni), i cui fattori componenti si chiamano termini elementari o clausole, detta **forma normale di tipo P**, oppure come prodotto di somme (congiuntiva di disgiunzioni), i cui fattori si chiamano fattori elementari, detta **forma normale di tipo S**. Ogni variabile prende il nome di *letterale*. La somma di tutti i termini P è 1, il prodotto di tutti i termini S è 0.

Un **mintermine** è un termine prodotto in cui compaiono tutte le variabili di ingresso corrispondenti alle combinazioni per cui la funzione booleana assume valore 1. In particolare, il mintermine è composto dalle variabili che assumono valore 1 prese in forma naturale e da quelle che assumono valore 0 prese in forma complementata. L'**implicante** è il termine ridotto ottenuto dalla somma di più mintermini.

Un **maxtermine** è un termine somma in cui compaiono tutte le variabili di ingresso corrispondenti alle combinazioni per cui la funzione booleana assume valore 0. In particolare, il maxtermine è composto dalle variabili che assumono valore 0 prese in forma naturale e da quelle che assumono valore 1 prese in forma complementata.

Per **ottimizzazione/minimizzazione** di una funzione si intende la sua trasformazione, attraverso passi successivi, con lo scopo di ottenere un'espressione equivalente, ma migliore rispetto ad una data metrica di valutazione (area occupata, tempo necessario a produrre un dato risultato, potenza o energia assorbita, etc.) Possibili metriche di area sono il numero di porte logiche generiche, il numero di porte logiche a due ingressi, il numero di implicanti o di implicantati, il numero di letterali. Si definiscono tre costi:

- **Costo di letterali:** è pari al numero delle variabili indipendenti della funzione, ciascuna moltiplicata per il numero di volte che essa compare nella forma.
- **Costo di porte:** è pari al numero delle funzioni elementari fi che la compongono, che per reti unilaterali è uguale al numero complessivo di porte adoperate.

- **Costo di ingressi:** è pari al numero delle funzioni f_i che la compongono, ciascuna moltiplicata per le variabili (dipendenti o indipendenti) di cui è funzione. Per reti unilaterali tale costo equivale al numero complessivo di porte adoperate, ciascuna pesata per il numero di ingressi (fan-in).

Viene dato per scontato che, se è data una variabile, viene dato anche il suo negato (quindi sia sui livelli, sia sul costo della funzione non viene considerato il not).

Solitamente si considera l'ottimizzazione dal punto di vista dell'area occupata, ma non è l'unica prospettiva importante, poiché oggi è fondamentale l'asse temporale, ovvero **il ritardo di propagazione di segnali all'interno del circuito**. Se la rete viene compattata fisicamente, solitamente si riduce anche il ritardo temporale. Il minimo che possiamo avere è una rete su due livelli, in cui il ritardo di propagazione interessa solo due porte. Il fan-in e fan-out di una porta influenzano anche il ritardo di propagazione della porta, ammesso che sia realizzabile. Nella realtà in base alla tecnologia realizzativa che utilizziamo potrebbe non esistere una porta con il numero di ingressi di cui abbiamo bisogno oppure potrebbe esistere ma comporta ritardi non accettabili.

Osservazione: *l'ottimizzazione di una funzione può avere o meno senso a seconda del contesto, o meglio della tecnologia realizzativa.* Negli anni '70 esistevano pochi componenti in logica discreta, si compravano le porte, vendute in un integrato, per cui la minimizzazione era vantaggiosa, perché trovare una forma minima significava comprare e collegare meno oggetti. Oggi invece esistono due mondi sostanzialmente, ASIC o FPGA. Chi usa l'FPGA monta la funzione e non la minimizza poiché implementa la tabella così com'è. La macchina combinatoria su FPGA può essere implementata sia con le porte logiche, ma anche con una memoria di cui le stringhe di n bit di ingresso rappresentano l'indirizzo delle celle e il loro contenuto è il valore assunto dalle variabili di uscita. Se la stringa di ingresso è da 4 bit e le uscite possibili sono due (A e B per esempio), serve una memoria da 16 celle, ognuna da 2 bit per cui la minimizzazione non avrebbe senso, a meno che non sia tale da riuscire a risparmiare una variabile (memoria con 8 celle piuttosto che 16). In realtà, se davvero con la minimizzazione riuscissimo a risparmiare una variabile, vuol dire che qualcosa nella progettazione era sbagliato e che quella variabile non era necessaria dall'inizio. Nel caso dell'ASIC, invece, la minimizzazione ha senso perché si ha una piastra di silicio su cui meno porte vengono istanziate meglio è. Generalmente, quando si usa l'ASIC, lo si fa perché la funzione è difficile e quindi la tabella è grande. Sarebbe impensabile fare la minimizzazione manualmente, è allora possibile sfruttare uno strumento (SiS) per l'ottimizzazione.

2.1 Ottimizzazione delle reti combinatorie

L'ottimizzazione è caratterizzata da due fasi importanti:

- **Espansione:** procedura che a partire dai mintermini (prodotti di letterali in corrispondenza dei quali la funzione è alta) cerca di ottenere un certo numero di **implicanti primi**. Si spera che siano in numero minore di mintermini (solitamente è così), hanno un numero minore di letterali e, essendo primi, non contengono nessun altro implicante (è quello che facciamo quando prendiamo i sottocubi di area massima nelle mappe di Karnaugh). La fase di espansione si serve delle proprietà dell'algebra di Boole, quali quella del complemento (una variabile più il suo negato è uguale ad 1) e quella di idempotenza (una variabile sommata a sé stessa è uguale alla variabile stessa), che utilizziamo quando un certo uno è condiviso da più raggruppamenti. Infatti, se abbiamo la somma di due termini che differiscono per una sola variabile, mettiamo in evidenza il termine comune e rimane il termine per uno, cioè il termine stesso. Se abbiamo due termini di lunghezza n , applicando la proprietà di complemento, riduciamo la lunghezza da $n+n$ ad $n-1$. La fase di espansione termina con una lista di

implicanti primi, tuttavia la procedura di minimizzazione non termina qui, poiché tra tutti gli implicanti primi potremmo non prenderne alcuni magari superflui.

- **Copertura:** si occupa di trovare gli implicanti primi che servono per coprire tutta la funzione. Sono detti *essenziali*, perché vanno per forza presi nella copertura, altrimenti qualche uno della funzione non risulterebbe coperto. Presi gli essenziali, può succedere che per altri uno ci siano varie scelte, motivo per cui si sceglie quello più piccolo o nel caso di stessa dimensione uno a caso. Gli implicanti essenziali costituiscono il *nucleo* della funzione, mentre quelli non essenziali che vengono scelti per la copertura prendono il nome di *residuo*.

Per i circuiti a due livelli e un'uscita le tecniche di ottimizzazione di *Karnaugh* e *McCluskey* sono metodi esatti, ma esistono anche metodi che possono dare soluzioni subottime, come il *Branch and Bound*. In generale sono tecniche che arrivano alla soluzione ottima con iterazione successive, se si ha tempo e risorse di computazione e sono adatti a reti molto grandi, per cui Karnaugh per esempio è complesso da utilizzare (*McCluskey* si può usare fino a 20 variabili). Le reti possono anche essere a due livelli ma con più uscite, anche in questo caso esistono sia metodi esatti che euristici. Infine le reti possono essere anche a più di due livelli, per cui non c'è l'ottimizzazione classica di somma di prodotti, ma sono realizzate con dei grafi. Possono presentare più livelli e più uscite, in tal caso si utilizzano metodi euristici.

2.1.1 Metodo esatto di McCluskey per funzioni ad una sola uscita su due livelli

Consideriamo la tabella di verità in figura 2.1 come esempio. Nella fase di espansione, prendiamo

x	y	z	v	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Figura 2.1: Esempio di tabella di verità.

i mintermini della funzione, combinazione di variabili per cui la funzione vale uno o don't care (-), e li mettiamo ordinati in base al numero di uno che contengono. La fase di espansione utilizza le due proprietà viste, per cui si cercano i mintermini che differiscono di un solo letterale. Guardiamo due gruppi contigui, perché la *distanza di Hamming* (numero di 1 per cui differiscono due mintermini) è uguale a 1. Se per caso sono tutti uguali e differiscono di un solo letterale, si dice che i due mintermini generano consenso. Se generano consenso non sono primi, perché li possiamo espandere, motivo per cui li marchiamo. I due mintermini che generano consenso danno

luogo ad un nuovo termine, che sarà un implicante. Al termine del primo passaggio della fase di espansione otterremo una **nuova lista di implicanti**, che avranno almeno un simbolo di $-$ (hanno generato consenso). Continuiamo con la procedura sulla nuova lista; quando non si può più applicare, prendiamo i mintermini che non hanno generato consenso (non sono stati marcati), questi sono **implicanti primi**. Fa eccezione solo un caso: se generano consenso due mintermini

m_i	x	y	z	v	
1	0	0	0	1	✓
4	0	1	0	0	✓
5	0	1	0	1	✓
6	0	1	1	0	✓
9	1	0	0	1	✓
7	0	1	1	1	✓
11	1	0	1	1	✓
14	1	1	1	0	✓
15	1	1	1	1	✓

$\{m_r \dots m_s\}$	x	y	z	v	
1, 5	0	-	0	1	A
1, 9	-	0	0	1	B
4, 5	0	1	0	-	
4, 6	0	1	-	0	✓
5, 7	0	1	-	1	
6, 7	0	1	1	-	✓
6, 14	-	1	1	0	✓
9, 11	1	0	-	1	C
7, 15	-	1	1	1	✓
11, 15	1	-	1	1	D
14, 15	1	1	1	-	✓

$\{m_r \dots m_s\}$	x	y	z	v	
4, 5, 6, 7	0	1	-	-	E
6, 7, 14, 15	-	1	1	-	F

Figura 2.2: Fase di espansione del metodo di McCluskey.

che avevano tutti don't care, quello che abbiamo appena generato non è un implicante primo, per cui va marcato a priori. Da notare che i mintermini non generano consenso se non hanno il trattino (-) nella stessa posizione. Se marchiamo un termine vuol dire che quel mintermine non è primo per definizione, è stato inglobato in qualcosa di più grande. Tutti quelli non marcati sono primi. La funzione, se non volessi procedere ad ottimizzarla, potrebbe essere scritta come $A+B+C+D+E+F$, di cui i primi 4 termini contengono 3 letterali, mentre E ed F solo 2. Abbiamo un miglioramento rispetto alla funzione iniziale, ma il procedimento non è ancora concluso.

Si procede con la **fase di copertura**, in cui si utilizza la **matrice di copertura**: sulle righe gli implicanti (abcdef), sulle colonne i mintermini (tutto ciò che va coperto), **mettiamo una x nella casella ij se l'implicante i -esimo copre il mintermine j -esimo**. Per esempio, 4-5-6-7 è un raggruppamento che raggruppa il 4, 5, 6, 7, per cui prendiamo le colonne relative e mettiamo le x. La prima cosa è **determinare il nucleo della funzione: gli implicanti essenziali** (gli uno coperti da un solo implicante in Karnaugh), **nella matrice di copertura, sono quelli che che presentano una sola x in colonna**. Questa condizione ci dà l'essenziale, per cui possiamo già cominciare a scrivere la funzione. Se per esempio A copre il singolo mintermine ma anche altri, vuol dire che ho coperto anche tutti quelli che stavano dentro, si identificano tutte le colonne dei mintermini coperti da A e si cancellano, così da ridurre le dimensioni della matrice. Facendo le cancellazioni, possiamo scoprire che ci sia un implicante con una sola x, in tal caso parliamo di **essenzialità secondaria**, significa che l'implicante non è essenziale, tuttavia va preso nella copertura. Se non ci sono più colonne con una sola x dentro, dobbiamo decidere come coprire tutte le colonne; si applicano allora i **criteri di dominanza di riga e di colonna**. La dominanza di riga si ha quando, date due righe, una ha tutte le x dell'altra più almeno un'altra, più precisamente la riga i domina la riga j quando l'implicante i copre tutti gli implicanti coperti da j più almeno uno. Se abbiamo scoperto che tra due implicanti A copre le stesse cose e di più

di B, cancelliamo B: non significa che automaticamente prenderemo A, siamo però sicuri che il costo della copertura sarà non maggiore, perché prendendo A abbiamo preso in un solo colpo 4 mintermini, mentre prendendo B ne avremmo coperti solo 2 e avremmo poi dovuto pensare agli altri 2 (costo più grande). Quindi possiamo operare la semplificazione perché il costo della copertura finale che avremmo prendendo A e non B è sicuramente non maggiore di quello che avremmo facendo la copertura finale. Per quanto riguarda le colonne, il mintermine uno domina il due: si

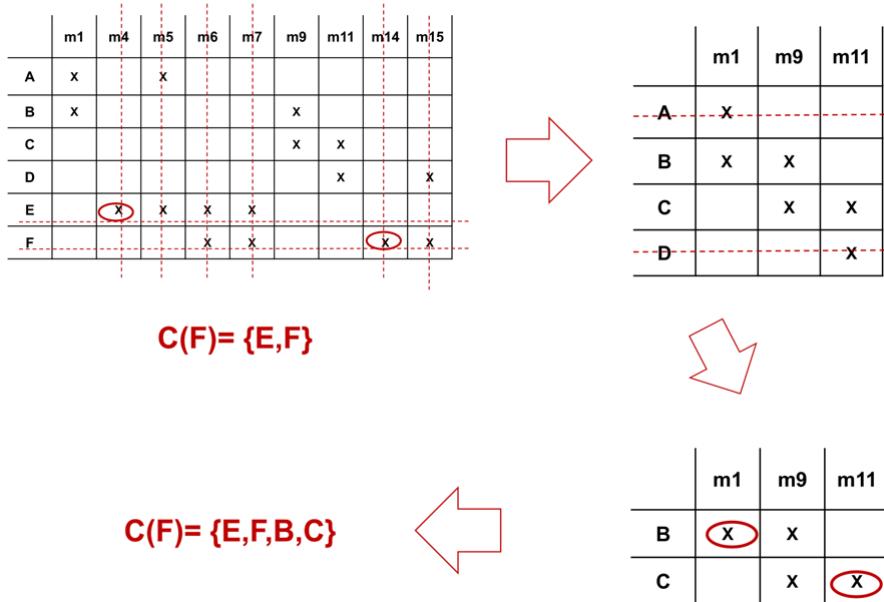


Figura 2.3: Fase di copertura del metodo di Mccluskey.

deve cancellare una colonna, che significa poter dimenticare un mintermine che va coperto. Se dobbiamo coprire il 2 possiamo coprirlo o scegliendo B o C. Se scegliamo B o C il 2 è coperto, mentre per coprire l'1 possiamo scegliere A, B, C. Se scegliamo A abbiamo coperto l'1 ma non il 2, per cui il costo potrebbe essere uguale ma anche maggiore; allora, cancelliamo la colonna dominante, così che il costo della copertura non sia maggiore di quello che avremmo se facessimo il contrario.

Quando ci sono i don't care, durante l'espansione, se generano consenso due mintermini di don't care li marchiamo a priori ma procediamo nell'espansione, semplicemente non saranno mai primi. Nella fase di copertura, in queste colonne non dobbiamo mettere anche i don't care, perché non siamo obbligati a coprirli, servono solo a fare i cubi più grandi, per cui nella fase di copertura non si mettono.

2.1.2 Branch and Bound

Il branch and bound si applica in quelle situazioni in cui non c'è né essenzialità né si può applicare dominanza, per cui si deve procedere iterativamente a tentativi. Costruiamo tanti sottoinsiemi e li esploriamo per vedere se possono essere soluzioni valide, per coprire tutta la funzione, e che costo hanno. Prendiamo la soluzione migliore, cosa che ovviamente dipende anche dal tempo a disposizione. Quando il problema è molto complesso, potremmo trovare anche una soluzione molto lontana dall'ottimo (problema di tutti gli algoritmi euristici). Nell'esempio in figura 2.4, l'albero si dirama in 4 foglie, nessuno dei 4 implicanti è una soluzione valida, per cui va esplorato il grafo. A può essere combinato con B, C, D; iniziamo a guardare tutte le soluzioni con due implicanti dentro. Se prendiamo A con C non copro Q, se prendiamo A con C non copriamo R,

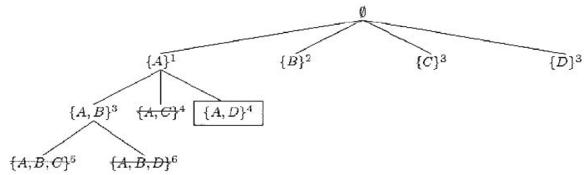


Figura 2.4: Esempio del metodo euristico Branch and Bound.

S; se prendiamo A-D copriamo tutto, quindi A-D è potenzialmente una soluzione, di cui dobbiamo calcolare il costo (somma di letterali). Le altre due casistiche sono incomplete, in tal caso l'albero continua. Andiamo avanti e continuiamo con il ramo AB, per cui troviamo costi più grandi del minimo già trovato, quindi le scartiamo a priori.

Adesso vediamo il **caso di funzioni caratterizzate da più uscite**, che fino ad ora abbiamo minimizzato separatamente. Per funzioni più complesse, si può fare di meglio perché si può scoprire che una porta può essere sfruttata da più uscite, semplicemente tirando un filo in più da una porta come fan out. Innanzitutto, minimizziamo le due funzioni separatamente con Karnaugh, cosa che corrisponde ad applicare l'algebra di Boole. È utile scrivere le funzioni per esteso perché

$$\begin{aligned} f_1 &= x'yz' + xyz' + xyz = xy + yz' \\ f_2 &= x'y'z' + x'y'z + x'yz' = x'y' + x'z' \end{aligned}$$

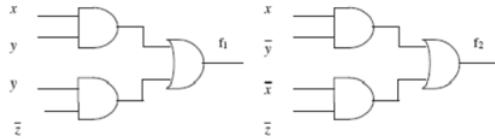


Figura 2.5: Esempio di funzione con più uscite.

notiamo che il termine rosso in figura 2.5 sta in entrambe le funzioni, per cui se lo scrivessimo in forma di circuito, è una porta a tre ingressi. Se minimizziamo le due funzioni separatamente otteniamo due circuiti da 3 porte ciascuno, per un totale di 6 porte, al contrario se guardiamo le due funzioni insieme, possiamo notare la presenza del termine comune e realizzare un unico circuito che conta un totale di 5 porte.

2.1.3 Metodo esatto di McCluskey per funzioni a più uscite su due livelli

Si può applicare una versione alternativa di McCluskey per funzioni a più uscite. La rete combinatoria è sempre espressa con la tabella di verità, caratterizzata da più colonne di uscita. Nel nostro esempio, ne consideriamo 3. Prendiamo tutti i mintermini ed anche i termini laddove la funzione vale don't care ed effettuiamo **una suddivisione in classi**. Al fianco di ogni mintermine mettiamo una stringa binaria, detta **marcatore**, che è l'unione dell'uscite delle tre funzioni (la prima vale 1, le altre due 0, quindi mettiamo come marcatore 100).

Fase di espansione

Non possiamo banalmente applicare le regole del metodo di McCluskey per capire se due mintermini generano consenso, ma dobbiamo fare delle **considerazioni in base alla AND bit a bit tra i marcatori di ogni mintermine**. Ci sono, infatti, quattro casi possibili (l'approccio che usiamo in questa spiegazione è esporre il caso generale e presentare un esempio relativo alla tabella in figura 2.6):



mi	x	y	z	v	f1	f2	f3
0	0	0	0	0	1	1	0
1	0	0	0	1	0	0	0
2	0	0	1	0	1	0	1
3	0	0	1	1	0	0	0
4	0	1	0	0	1	1	0
5	0	1	0	1	1	1	0
6	0	1	1	0	1	0	1
7	0	1	1	1	1	1	0
8	1	0	0	0	0	0	0
9	1	0	0	1	0	0	0
10	1	0	1	0	0	1	1
11	1	0	1	1	0	1	1
12	1	1	0	0	0	0	0
13	1	1	0	1	0	0	0
14	1	1	1	0	0	1	1
15	1	1	1	1	0	1	1

mi	x	y	z	v	f1	f2	f3
0	0	0	0	0	1	1	0
2	0	0	1	0	1	0	1
4	0	1	0	0	1	1	0
5	0	1	0	1	1	1	0
6	0	1	1	0	1	0	1
10	1	0	1	0	0	1	1
7	0	1	1	1	1	1	0
11	1	0	1	1	0	1	1
14	1	1	1	0	0	1	1
15	1	1	1	1	0	1	1

Figura 2.6: Tabella di verità e classificazione per una funzione a più uscite.

0000 \Rightarrow 1100
AND \rightarrow 0000
 0001 \Rightarrow 0011

CASO 1.

1100 \Rightarrow 1110
AND \rightarrow 110- \Rightarrow 0110
 1101 \Rightarrow 0111

CASO 2.

0000 $\xrightarrow{\text{AND}}$ 1100 X
 0001 $\xrightarrow{\text{AND}}$ 1101 \rightarrow

CASO 3.

1100 $\xrightarrow{\text{AND}}$ 1100 X
 1101 $\xrightarrow{\text{AND}}$ 1100 \rightarrow

CASO 4.

Figura 2.7: I 4 casi delle AND bit a bit tra i marcatori.

- 1. **AND bit a bit restituisce 0000:** vuol dire che non capita mai che i due mintermini siano insieme nella stessa funzione (non abbiamo “uno” adiacenti in Karnaugh, per cui non possiamo costruire il sottocubo). Prese singolarmente, nessuna funzione fa fare il raggruppamento: non è un implicante valido.
- 2. **AND bit a bit restituisce una stringa che non corrisponde a nessuno dei due marcatori:** in poche parole, vuol dire che i due mintermini hanno qualcosa in comune, ma non dappertutto. L’implicante è valido, poiché, per dire che è valido, non abbiamo bisogno che stia su tutto, ma che ci sia qualcosa in comune. L’unico problema è se marcare questi mintermini o meno, ricordando che marcare un mintermine vuol dire che non è primo. Se guardassimo solo F2 ed F3, li marcheremmo perché non sono primi, per F1 e F4 non possiamo

dire che sicuramente non sono primi, perché ancora dobbiamo procedere con l'espansione. Non li possiamo marcare. Il nuovo implice viene introdotto ma nessuno degli implicanti di partenza viene marcato (gli implicanti potrebbero ancora essere primi per alcune funzioni).

- 3. **AND bit a bit restituisce una stringa che corrisponde ad uno dei due marcatori:** per esempio in F1 e F2 c'è il primo, il secondo sta in F1, F2 e F4. In F1 e F2 c'è il raggruppamento, negli altri casi no. Se però guardiamo meglio, il mintermine 0000 sta in F1 e F2; *ogni volta che lo troviamo nella funzione 1 e 2 partecipa sempre ad un raggruppamento*, in F3 e F4 non c'è proprio. Questo vuol dire che *laddove il mintermine è presente, viene sempre raggruppato*, cosa diversa rispetto al caso due in cui in alcuni casi c'era ma non partecipava a raggruppamenti. Allora in questa terza ipotesi, **l'implicante non può mai essere primo**, perché, ogni volta che c'è, lo usiamo per raggruppare. 0000 può allora essere marcato. Vediamo il secondo mintermine, questo sta nella F1, F2 e lo raggruppiamo, in F3 non c'è e in F4 c'è ma non lo raggruppiamo, non possiamo marcarlo perché nella funzione in cui è presente non dà luogo a raggruppamenti, per cui potrebbe essere primo. *Marchiamo solo l'implicante il cui marcitore è uguale al risultato della AND bit a bit.*
- 4. **AND bit a bit restituisce una stringa che corrisponde ad entrambi i marcatori:** questo quarto caso vuol dire che, ogni volta che i mintermini sono in una funzione, stanno sempre accoppiati e quindi li raggruppiamo sempre. 1100 e 1101 compaiono solo in F1 e F2, sempre insieme, non possono essere mai primi, per cui vanno raggruppati e marcati.

mi	x	y	z	v	f1	f2	f3		mi	x	y	z	v	f1	f2	f3	
0,2	0	0	-	0	1	0	0	X	0,2,4,6	0	-	-	0	1	0	0	P5
0,4	0	-	0	0	1	1	0		2,6,10,14	-	-	1	0	0	0	1	P6
2,6	0	-	1	0	1	0	1	P1	4,5,6,7	0	1	-	-	1	0	0	P7
2,10	-	0	1	0	0	0	1	X	10,11,14, 15	1	-	1	-	0	1	1	P8
4,5	0	1	0	-	1	1	0	P2									
4,6	0	1	-	0	1	0	0	X									
5,7	0	1	-	1	1	1	0	P3									
6,7	0	1	1	-	1	0	0	X									
6,14	-	1	1	0	0	0	1	X									
10,1 1	1	0	1	-	0	1	1	X									
10,1 4	1	-	1	0	0	1	1	X									
7,15	-	1	1	1	0	1	0	P4									
11,1 5	1	-	1	1	0	1	1	X									
14,1 5	1	1	1	-	0	1	1	X									

IMPICANTI:

P0=0-0 =x'z'v' 0,4	110-> f0,f1
P1=0-10 =x'zv' 2,6	101->f0,f2
P2=010- =x'yz' 4,5	110->f0,f1
P3=01-1 =x'yv 5,7	110->f0,f1
P4=-111 =yzv 7,15	010 ->f1
P5=0--0 =x'v' 0,2,4,6	100->f1
P6=-10 =zv' 2,6,10,14	001->f2
P7=01-- =x'y 4,5,6,7	100->f0
P8=1-1- =xz 10,11,14,15	011->f1,f2

Figura 2.8: Termine della fase di espansione dell'esempio.

Il procedimento viene ripetuto finché non è più possibile determinare consensi; gli implicanti che risulteranno non marcati alla fine della prima fase sono gli implicanti primi della funzione vettoriale.

Fase di copertura

Affianchiamo le matrici di copertura singole. Sulle righe troviamo gli implicanti determinati, sulle colonne i mintermini delle F1, F2, F3 e F4. Andiamo a vedere le essenzialità, può succedere che abbiamo una sola X in una colonna, dobbiamo capire se sia possibile cancellare. Se A è implicante e copre m3 di F1 ma copre anche altri mintermini di altre funzioni, che succede? Se fosse essenziale

per tutte lo prenderemmo nella copertura e lo cancelleremmo. Se è essenziale solo in una funzione, non possiamo cancellarlo del tutto perché potrebbe servire anche alle altre. In F1 vediamo delle essenzialità, che sono P0 e P8. *La riga P8 è essenziale per F1 e F2, laddove c'è è essenziale sempre, in F0 non c'è. Dato che ogni volta che c'è è essenziale lo mettiamo nella copertura e possiamo cancellarlo*, così come visto per la procedura ad una sola uscita, cancelliamo la riga e le colonne coperte. P0 sta in F1 e F0, in F0 non è essenziale, per F1 sì. Dato che solo in F1 è essenziale non possiamo cancellare la riga. Possiamo fare le semplificazioni in F1, ma **non a livello globale**. La dominanza ha una piccola differenza: la dominanza di colonna si vede sempre all'interno delle

	f0							f1							f2						
	0	2	4	5	6	7		0	4	5	7	10	11	14	15	2	6	10	11	14	15
P0	X	X					(X)	X													1
P1	X			X																	1
P2		X	X					X	X												1
P3			X	X	X			X	X												1
P4						X				X											1
P5	X	X	X		X																1
P6												X	X	X	X						1
P7		X	X	X	X	X		(X)	X	(X)	X	(X)	X	(X)	X						1
P8												X	(X)	X	(X)	X	(X)	X	(X)	X	1

	f0							f1		f2	
	0	2	4	5	6	7	5	7	2	6	
P0	X	X									0
P1	X			X					X	X	1
P2		X	X				X				1
P3		X		X	X	X					1
P4											1
P5	X	X	X	X							1
P6		X	X	X	X	X					1
P7		X	X	X	X	X					1

Figura 2.9: Prima parte della fase di copertura.

singole funzioni, anche perché non avrebbe senso altrimenti. *La dominanza di riga invece si guarda a livello di implicante*, che invece interessa più funzioni. I domina J se ha più X, bisogna però fare un discorso sul costo: *chi ha più X domina l'altro, ma potrebbe succedere che il costo dell'implicante di per sé sia maggiore di quello da cancellare, in quel caso non domina*. Il costo è in termini di letterali, normalmente, ma non è l'unico, può essere anche in termini di porte. Ragioniamo in termini di porte: alla fine della tabella abbiamo una colonna di tutti 1, vuol dire che se dovessimo prendere quell'implicante, in termini di numero di porte totali quanto costa? Costa 1 perché lo prendiamo una volta, dato che ogni volta che prendiamo un implicante dobbiamo instanziare una porta. Se prendiamo P3, la porta è presa e instanziata, se lo abbiamo preso per F2 ma poi scopriamo che va bene anche per F0 ormai la porta è presa, per cui il nuovo costo è 0. Durante il procedimento quindi dobbiamo modificare la colonna dei costi, all'inizio avremo tutti 1 e man mano che li utilizziamo costeranno 0.

	f0							f1		f2	
	0	2	4	5	6	7	5	7	2	6	
P0	X	X									0
P1	X			X					X	X	1
P2		X	X				X				1
P3		X		X	X	X					1
P5	X	X	X	X							1
P7		X	X	X	X	X					1

	f0						
	0	2	4	5	6	7	
P0	X						0
P1		X			X		0
P2			X	X			1
P3			X		X	X	0
P5	X	X	X	X	X		1
P7		X	X	X	X	X	1

	f0			
	0	2	7	
P0	X			0
P1		X		0
P3		X	X	0
P5	X	X		1
P7		X	X	1

	f0		
	0	2	
P0	X		0
P1		X	0
P3			0
P5	X	X	1

Figura 2.10: Seconda parte della fase di copertura, considerando il costo in termini di porte.

Vediamo l'esempio con le porte, poniamo il costo di P0 a 0, perché era essenziale solo per una

funzione (per quella funzione lo abbiamo scelto e messo nel circuito), l'abbiamo preso e ogni volta ci costerà come numero di porte 0 perché ormai è stata istanziata la porta. Adesso non abbiamo più essenzialità, procediamo con la dominanza. P1 ha tutte le X di P6 più altre, hanno lo stesso costo, quindi la dominanza vale e cancelliamo P6. Lo stesso vale tra P3 e P4, cancelliamo P4 perché il costo è lo stesso. Facciamo allora la nuova tabella (prima riga a destra in figura 2.10), senza P4 e P6. Individuiamo poi delle essenzialità P1 e P3, vediamo se possiamo cancellarli (se sono essenziali per tutti). P3 copre F1 ma anche F0, è essenziale solo per F1 per cui non possiamo eliminarlo, possiamo solo prendere la porta, metterla nella copertura e mettere il costo a 0. P1 compare in F0 e F2, il problema è lo stesso, dato che copre anche F0 per cui non è essenziale, non lo cancelliamo, mettiamo il costo a 0 e lo prendiamo nella copertura. Facciamo la nuova tabella (seconda riga a sinistra), c'è dominanza tra P7 e P2, cancelliamo P2. Vediamo anche le colonne, la 0 domina la 4, va cancellato il 4, non ci sono problemi perché siamo all'interno di F0. Cancelliamo 4, 6 e 5. Poi vediamo la dominanza, P5 e P1, P5 sembra dominare ma ha un costo più alto, per cui non lo prendiamo, significherebbe instanziare un'altra porta, non è una dominanza. P3 domina P7 perché hanno stesso numero di X ma il costo è 0: cancelliamo P7. Poi scopriamo che P3 è un'essenzialità secondaria. Alla fine potremmo coprire tutto con P5 o uno alla volta con P0 e P1: prendiamo P0 e P1 perché li abbiamo già.

Se avessimo invece un costo in termini di letterali, sulla colonna a destra della matrice, invece di tutti 1, avremmo il numero di letterali dell'implicante e faremmo esattamente la stessa cosa, quando prendiamo l'implicante instanziamo la porta per cui se poi vogliamo usarla anche per altre funzioni dobbiamo solo tirare un filo, che andrà in ingresso ad un'altra porta, dunque l'espressione dell'altra porta avrà un letterale in più: il costo diventa da 4, ad esempio, 1, perché ogni volta che vogliamo usarla pagheremo solo 1 (il filo).

2.2 SiS

2.2.1 Funzioni ad una uscita

Consideriamo una funzione su due livelli ad una uscita definita in questo modo:

ONSet=1,4,5,6,7,9,11,14,15; DCSet=0.

```
.model esempio1
.inputs x y z v
.outputs f
.names x y z v f
0001 1
0100 1
0101 1
0110 1
0111 1
1001 1
1011 1
1110 1
1111 1

.end
```

Figura 2.11: Definizione di una funzione per utilizzare SiS.

Vediamo come risolvere questo esercizio con SiS. Per prima cosa dobbiamo capire come si definisce la funzione in ambiente SiS. Possiamo scrivere il testo con un editor qualunque, *il formato deve avere estensione .blif*, rispettando le seguenti regole:

- Il nome del circuito è preceduto da **.model**.
- Le variabili di ingresso (nel nostro caso la funzione è a 4 variabili che indichiamo con x y z e v) si definiscono con **.inputs**.
- La variabile di uscita (nel nostro esempio è f) si indica con **.outputs**.

La tabella di verità si descrive con **.names** seguito dalle variabili di input e di output, con quest'ordine e sotto vanno messi i mintermini (solo quelli in cui la funzione è alta). Dopo il mintermine, va inserito uno spazio e poi i valori dell'uscita (quindi 1). Carichiamo il file da SiS:

```
c:\sis-1.2\ESERCIZI>sis
UC Berkeley, SIS 1.3 (compiled Jun 11 2003)
sis> read_blif monof.blif
sis> write_blif
.model esempio1
.inputs x y z v
.outputs f
.names x y z v f
1111 1
0111 1
1011 1
0001 1
1110 1
0110 1
0100 1
.end
sis> write_eqn
INORDER = x y z v;
OUTORDER = f;
f = !x*y!z*v + !x*y*z!v + x*y*z!v + !x*y!z*v + x*y!z*v + !x*y!z*v + x*y!z*v
y*z*v + !x*y*z*v + x*y*z*v;
sis> print_stats
esempio1 pi= 4 po= 1 nodes= 1 latches= 0
lits(sop)= 36
```

Figura 2.12: Comandi SiS, prima parte.

per poterlo fare c'è il comando **read_blif**, seguito dal path. Se vogliamo vedere lo script, utilizziamo il comando **write_blif**. Se vogliamo vedere l'espressione booleana della funzione, possiamo dare il comando **write_eqn**. Otteniamo infatti la forma somma di prodotti, non ancora minimizzata, per cui ci sono tutti i mintermini. Se vogliamo fare statistiche a livello di numero di letterali e così via,

```
sis> simplify
sis> write_eqn
INORDER = x y z v;
OUTORDER = f;
f = !y*z*v + x*z*v + y*z + !x*y;
sis> print_stats
esempio1 pi= 4 po= 1 nodes= 1 latches= 0
lits(sop)= 10
sis> write_blif
.model esempio1
.inputs x y z v
.outputs f
.names x y z v f
01-- 1
-11- 1
1-11 1
-001 1
.end
sis>
```

Figura 2.13: Comandi SiS, seconda parte.

digitiamo **print_stats**: pi indica le variabili di ingresso, po quelle di uscita, node il numero di nodi; il latches è presente perché possiamo minimizzare anche gli automi, lits(sop) è il numero di letterali

nella forma somma di prodotti. Vogliamo adesso semplificare questa funzione. Vengono messi a disposizione una serie di comandi. Il comando per la semplificazione è **simplify**, che effettua la minimizzazione di una rete combinatoria sfruttando un'implementazione ottimizzata di McCluskey, una loro versione dell'algoritmo Espresso. Per avere l'output diamo di nuovo write_eqn, osserviamo che i letterali da 36 diventano 10. Se digitiamo di nuovo write.blif è come se sovrascrivesse (non su disco) il file di partenza e indica chi sono gli implicanti trovati. Quando ci sono dei don't care, nello scrivere il file .blif, i mintermini dove la funzione vale 1 si definiscono come fatto precedentemente; scriviamo poi .exdc (che significa external dc), ripetiamo la funzione della definizione ma solo dove la funzione assume valore don't care. Alla fine mettiamo il .end.

```
.exdc
.inputs x y z v
.outputs f
.names x y z v f
0011 1
0101 1
0110 1
0111 1

.end
```

Figura 2.14: Esempio SiS con i don't care.

Se consideriamo la funzione che ha questa espressione:

ONSet=4,10,11,13,14,15; DCSet=3,5,6,7

scrivendo correttamente lo script, possiamo dare il comando di simplify. Mentre la funzione di partenza aveva 24 letterali, mediante la simplify otteniamo una funzione di 9 letterali. Possiamo

```
sis> read_blif monofDC.blif
sis> write_eqn
INORDER = x y z v;
OUTORDER = f;
f = !x*y*z*v + x*x*y*z*v + x*x*y*z*v + x*x*y*z*v + x*x*y*z*v + x*x*y*z*v;

Don't care:
INORDER = x y z v;
OUTORDER = f;
f = !x*y*z*v + !x*x*y*z*v + !x*x*y*z*v + !x*x*y*z*v;

sis> print_stats
esempioIDC pi= 4 po= 1 nodes= 1 latches= 0
lits<sop>= 24
sis> simplify
sis> write_eqn
INORDER = x y z v;
OUTORDER = f;
f = !x*y*z*v + x*x*y*v + x*x*z;

Don't care:
INORDER = x y z v;
OUTORDER = f;
f = !x*y*z*v + !x*x*y*z*v + !x*x*y*z*v + !x*x*y*z*v;

sis> print_stats
esempioIDC pi= 4 po= 1 nodes= 1 latches= 0
lits<sop>= 9
sis> full_simplify
sis> write_eqn
INORDER = x y z v;
OUTORDER = f;
f = !x*y*z*v + !x*x*y*z*v + !x*x*y*z*v + !x*x*y*z*v;

sis> print_stats
esempioIDC pi= 4 po= 1 nodes= 1 latches= 0
lits<sop>= 6
```

Figura 2.15: Script di SiS per l'esempio con i don't care.

anche usare la **full_simplify**, che utilizza in modo ancora più efficiente i don't care interni. In alcuni casi migliora il simplify, ma non sempre.

2.2.2 Funzioni a più uscite

```
.model multif1
.inputs x y z v
.outputs f1 f2 f3      .names x y z v f2      .names x y z v f3
                        0000 1                  0010 1
                        0100 1                  0110 1
.names x y z v f1      0001 1                  0101 1                  1010 1
                        0010 1                  0111 1                  1011 1
                        0100 1                  1010 1                  1110 1
                        0101 1                  1011 1                  1111 1
                        0110 1
                        0111 1
                        1111 1
.end
```

Figura 2.16: Esempio minimizzazione con SiS per funzioni a più uscite.

Vediamo il caso di funzioni a più uscite, in particolare l'esempio presenta 3 uscite ed è definita come segue:

```
ONSet1=0,2,4,5,6,7; DCSet1=0
ONSet2=0,4,5,7,10,11,14,15; DCSet2=0
ONSet3=2,6,10,11,14,15; DCSet3=0
```

Scriviamo 3 volte il .names e indichiamo tutti i punti in cui la funzione vale 1, esattamente come fatto per il caso delle funzioni ad una uscita (figura 2.16) *Utilizzando il comando simplify o il full_simplify si ottiene una minimizzazione esatta, in cui però le variabili vengono minimizzate separatamente, senza valutare possibili sovrapposizioni di porte.* La descrizione del circuito in uscita è tipicamente su più livelli. Dopo aver dato full_simplify, quando scriviamo write_eqn ci aspettiamo una funzione su due livelli, in questo caso è vero, ma non lo è sempre. Potremmo ottenere una rete su due livelli dove però i nodi si intrecciano tra loro.

```
.i 4
.o 3
.ilb x y z v      1110 010
.ob f1 f2 f3      1010 010
.p 20
0100 010
0000 010
0111 100      1111 001
0101 100      1011 001
0110 100      1110 001
0010 100      0110 001
0100 100      1010 001
0000 100      0010 001
1111 010
0111 010
1011 010
0101 010
```

Figura 2.17: Formato .pla.

Se vogliamo una funzione su due livelli, possiamo usare lo strumento ***Espresso***, che è un algoritmo iterativo, il quale dà una soluzione esatta ma non prende in ingresso il file con estensione .blif, ma il file con estensione **.pla**, che è una particolare rete combinatoria fatta da due sezioni, una di AND e una di OR. Se dobbiamo costruire una funzione di 2 livelli, con 4 variabili, il pla

ci dà una serie di combinazioni tra di esse, una serie di AND già preinstallate, con una serie di collegamenti già fatti, noi dobbiamo solo bruciare quelli che ci interessano. Sotto ci dà un'ulteriore banco di OR, allo stesso modo bruciamo quello che ci interessa e otteniamo così la rete a due livelli. È un primo tentativo per fare rete customizzata ai bisogni dell'utente, in modo molto semplice. Un file .blif può essere convertito in formato .pla con i seguenti comandi: `read_blif <nome_file.blif>`, seguito da `write_pla <nome_file.pla>`. La descrizione pla è quella in figura 2.17.

Il formato pla è molto simile a quello che abbiamo visto per il procedimento di Mccluskey, mettiamo i mintermini e accanto la maschera delle funzioni. **Espresso è un programma esterno a SiS**, va richiamato da linea di comando come `"espresso -s file.pla"`. Questo programma esterno minimizza su due livelli e cerca di trarre vantaggio da eventuali sovrapposizioni di implicanti fra diverse uscite.

```
C:\sis-1.2\ESERCI~1\espresso -s multif_pla.pla
# c:/sis-1.2/sis/bin/espresso.exe -s multif_pla.pla
# UC Berkeley, Espresso Version #2.3, Release date 01/31/88
# PLA is multif_pla.pla with 4 inputs and 3 outputs
# ON-set cost is c=20<20> in=80 out=20 tot=100
# OFF-set cost is c=7<6> in=14 out=11 tot=25
# DC-set cost is c=0<0> in=0 out=0 tot=0
# ESPRESSO      Time was 0.00 sec, cost is c=4<0> in=11 out=8 tot=19
.i 4
.o 3
.ilb x y z v
.ob f1 f2 f3
.p 4
01-1 110 P3
0-10 101 P1
0-00 110 P0
1-1- 011 P8
.e



Costi in termini di letterali  

    rispettivamente dell'ON-set,  

    OFF-set e DC-set



Costi della soluzione minimizzata


```

Figura 2.18: Comandi per minimizzazione con Espresso.

2.3 Classificazione delle macchine combinatorie

Le macchine combinatorie possono essere di **connessione**, dato che dobbiamo poter connettere i sistemi tra loro. Se è senza memoria, la connessione tra una macchina A ed una macchina B è ancora combinatoria. Ci sono poi le macchine combinatorie di **codifica** e **decodifica**, quelle che gestiscono la **priorità** e le macchine **artimetiche**.

2.3.1 Macchine a priorità

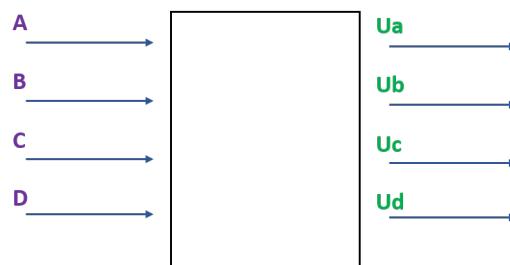


Figura 2.19: Macchina combinatoria a priorità con 4 ingressi.

E' una macchina che necessita di un criterio per poter gestire la priorità. Consideriamo l'esempio in figura 2.19, macchina combinatoria con 4 ingressi: l'uscita U_a è alta se A è abilitato, l'uscita U_b è alta se B è abilitato e non A , l'uscita U_c è abilitata se C è alto ma non A e non B , l'uscita U_d è alta se D è alto ma non A , non B e non C .

In altre parole, se volessimo realizzare la tabella di verità dovremmo farne una per ogni uscita, con 4 ingressi ciascuna, tuttavia per logica si può ottenere lo stesso risultato: oni

$$\begin{aligned} U_a &= A \\ U_b &= (\neg A) * B \\ U_c &= (\neg A) * (\neg B) * C \\ U_d &= (\neg A) * (\neg B) * (\neg C) \end{aligned}$$

Gli ingressi non possono essere alti contemporaneamente, perchè sono tutti casi mutamente esclusivi. Questo è un chiaro esempio di come il metodo (scrittura della tabella di verità) conduca allo stesso risultato della logica, ma in questo caso scegliere la seconda piuttosto che il primo ha reso la progettazione più veloce.

2.3.2 Macchine di connessione

In generale, le macchine combinatorie di questa categoria possono essere di diverso tipo, cioè possono realizzare connessioni n:1 (n sorgenti ad una destinazione), 1:m (1 sorgente ad m destinazioni) ed n:m (n sorgenti ad m destinazioni). Ci sarebbe anche il caso 1:1 che è il filo.

Multiplexer

La macchina combinatoria che realizza la connessione n:1 è il multiplexer, che sarà molto ricorrente anche nella realizzazione di architetture più complesse. *Questo perchè il multiplexer è la macchina principe di qualunque sistema dato che realizza un if-then-else, ma mediante questo costrutto sequenziale ed il go to possiamo implementare qualunque sistema.*

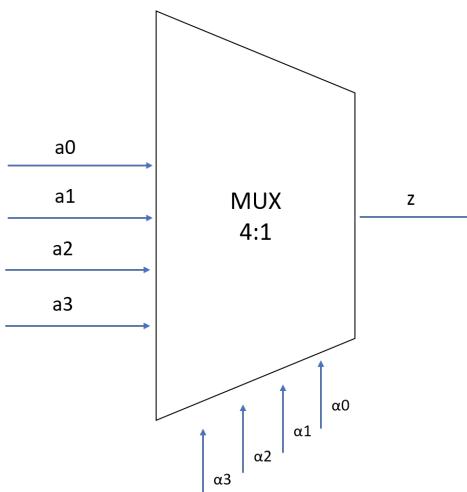


Figura 2.20: Esempio di un multiplexer a 4 ingressi ed 1 uscita.

Il multiplexer rappresentato in figura 2.20 è di tipo **lineare**, che presenta tante linee di selezione quanti sono gli ingressi ed al più è attiva una alla volta. L'uscita z è unica ed assume valore pari ad a_0 se α_0 è attivo, pari ad a_1 se invece è attivo α_1 e così via. È una macchina combinatoria utilizzata quando più linee devono essere convogliate verso un'unica linea di uscita (*bus*), **il multiplexer è un convogliatore.**

Quando gli ingressi del multiplexer sono binari, si parla di *multiplexer binario*. Possiamo scrivere la funzione di trasferimento di questa macchina come:

$$z = \alpha_0 * a_0 + \alpha_1 * a_1 + \alpha_2 * a_2 + \alpha_3 * a_3$$

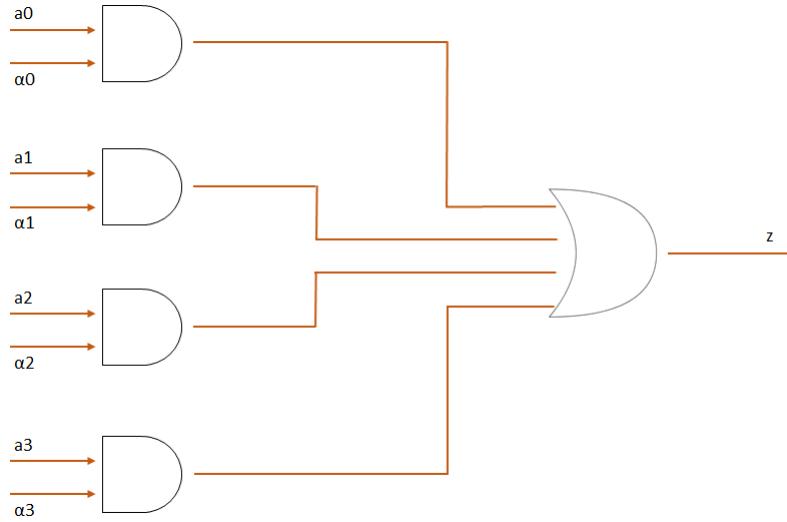


Figura 2.21: Realizzazione del mux 4:1 con porte AND ed OR.

Da questa capiamo che è possibile realizzare il mux 4:1 con 4 porte AND al primo livello, ognuna da due ingressi, ed una OR da 4 ingressi al secondo livello, come mostrato in figura 2.21. Questo sistema ha un'uscita che vale 0 o 1, sempre. Se ad esempio a_0 è 0 e α_0 è alto, viene selezionato a_0 e l'uscita è 0; se nessuna delle 4 abilitazioni è alta comunque l'uscita è bassa. In altre parole qualora tutte le abilitazioni fossero basse o l'ingresso corrispondente all'abilitazione alta fosse basso, l'uscita del multiplexer è sempre 0. Se volessimo distinguere lo zero del segnale dal caso in cui l'abilitazione non c'è, dovremmo aggiungere una OR al secondo livello che prende in ingresso tutte le α . E' possibile realizzare la macchina anche in **logica tristate** (figura 2.22):

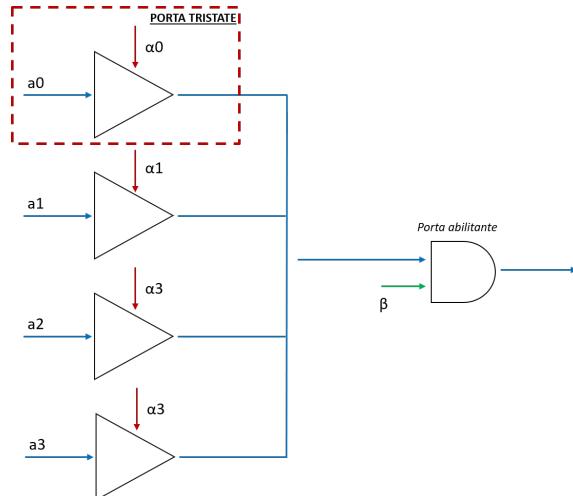


Figura 2.22: Realizzazione del mux 4:1 in logica tristate.

la singola porta presenta un'abilitazione, se questa è alta il filo è chiuso, se non c'è il filo è aperto.

E' un interruttore.

Non c'è più bisogno della porta OR poiché l'uscita in un dato istante di tempo è data da quella porta che risulta chiusa. La funzione di trasferimento della porta tristate è la seguente:

Tabella 2.1: Funzione di trasferimento della porta tristate.

ingresso ai	α	uscita
0	0	aperto
1	0	aperto
0	1	0
1	1	1

Quando la porta tristate è chiusa ammette in uscita 0 o 1 a seconda del valore del segnale di ingresso. Se il sistema è aperto (è un circuito aperto) non passa alcuna corrente, si può accumulare della carica ed a contatto con un conduttore la carica trova un sistema per poter fluire. Questo vuol dire che il circuito può assumere un valore di tensione qualsiasi, che potrebbe essere pericoloso, poiché si potrebbe interpretare l'uscita come uno 0 o un 1, quando in realtà non c'è quel valore dato l'instaurarsi di una tensione. Il vantaggio dei sistemi realizzati con le porte tristate è che il nodo di uscita risulta avere un potenziale fissato, dato che almeno una tra i 4 elementi sarà attivo e sarà quello che fissa il potenziale.

Un multiplexer lineare presenta un limite dovuto all'eccessivo numero di fili dei segnali di selezione. E' possibile aggiungere un **decoder** (*il decodificatore è una macchina che riceve in ingresso una parola codice su n bit e presenta in uscita la sua rappresentazione decodificata su m=2 elevato ad n bit*). L'architettura del multiplexer con 4 ingressi ed 1 uscita diventa quella in figura 2.23, dove il decoder prende due bit in ingresso e ne produce 4 in uscita. In questo caso il multiplexer è detto **multiplexer indirizzabile**.

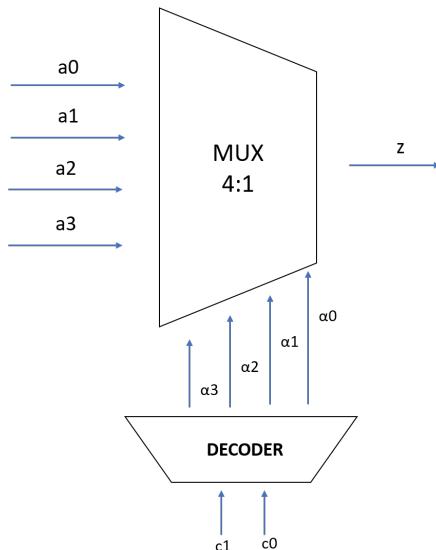


Figura 2.23: Multiplexer indirizzabile.

Demultiplexer

Il demultiplexer è la macchina che realizza la connessione 1:m, quindi prende un ingresso e fornisce un'uscita U uguale all'ingresso I moltiplicato per i segnali di selezione abilitati o negati. In altre parole il demux ha un ingresso dati I, n segnali di selezione dei quali al più uno è attivo ed n uscite dati, con l'i-esima uscita U_i pari ad $I \cdot \alpha_i$ se α_i è attiva. Così come per il multiplexer, il **demultiplexer**

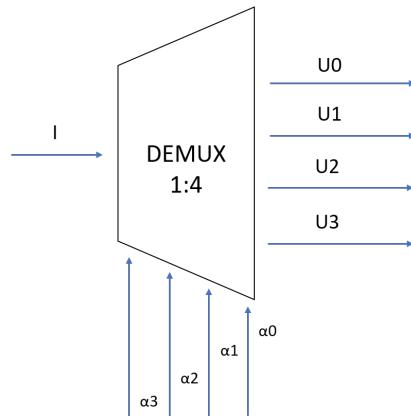


Figura 2.24: Demultiplexer lineare.

lineare presenta tanti segnali di selezione quante sono le uscite; il **demultiplexer indirizzabile** è uno lineare, i cui segnali di selezione sono l'uscita di un decodificatore.

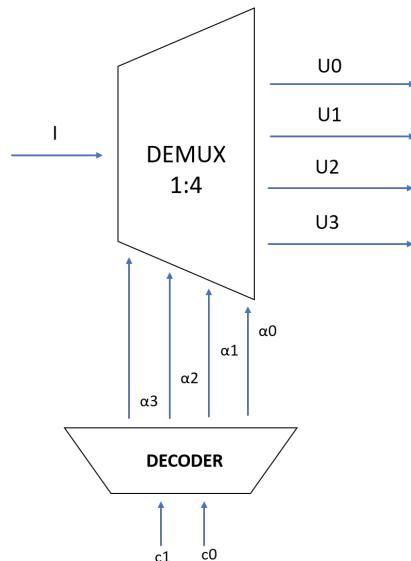


Figura 2.25: Demultiplexer indirizzabile.

2.3.3 Implementazione VHDL di un multiplexer 8:1

Vogliamo progettare ed implementare in VHDL un multiplexer 8:1 indirizzabile, utilizzando una descrizione di tipo *structural* che componga opportunamente multiplexer più piccoli.

Utilizzando l'approccio modulare, ovvero decomponendo la macchina da implementare in componenti più piccoli, procediamo con l'implementazione di un multiplexer 8:1 tramite la composizione di due multiplexer 4:1 e un multiplexer 2:1. I multiplexer 4:1 sono implementati, a loro volta, mediante la composizione di tre multiplexer 2:1. La cella fondamentale di questa macchina, quindi, è il multiplexer 2:1, grazie a cui realizzeremo prima i multiplexer 4:1 e poi il multiplexer 8:1. Per tale motivo, partiamo da una descrizione di tipo *dataflow* della cella fondamentale per poi procedere con descrizioni di tipo *structural* delle macchine da essa derivanti.

Schematici

Realizziamo un multiplexer di tipo *indirizzabile*, dunque per 2 ingressi serve un solo ingresso di selezione, in base al cui valore una tra le linee di ingresso sarà portata sulla linea di uscita. Nel complesso la macchina presenta 3 ingressi e un'uscita, come mostra la figura 2.26. Dalla composi-

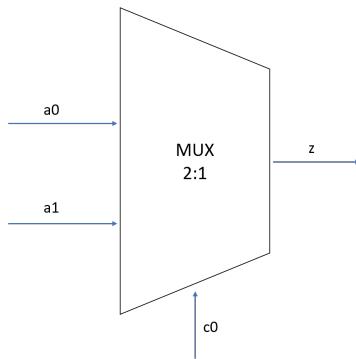


Figura 2.26: Schema multiplexer 2:1.

zione di tre multiplexer 2:1 otteniamo un **multiplexer 4:1**. Anche quest'ultimo è un multiplexer indirizzabile: prende in ingresso 4 segnali e di volta in volta ne porta uno solo in uscita, a seconda dei valori dei due ingressi di selezione. I 4 ingressi entrano in due multiplexer che, essendo 2:1, prendono 2 ingressi e producono un'uscita ciascuno. Tali uscite entrano nel terzo multiplexer che produce l'unico output finale. Lo schema è mostrato in figura 2.27. Concettualmente, stiamo

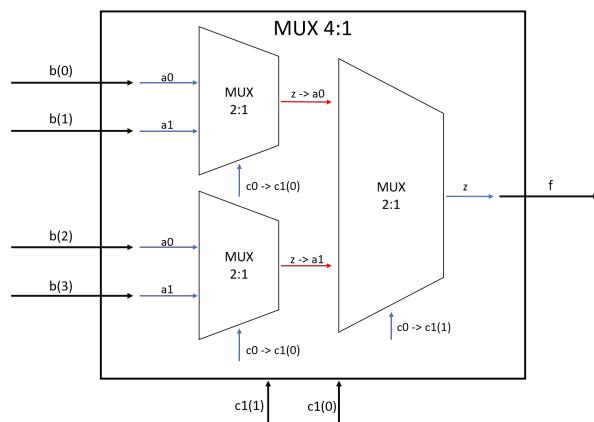


Figura 2.27: Schema multiplexer 4:1.

dividendo l'indirizzo in ingresso al multiplexer finale (c_1) in due parti:

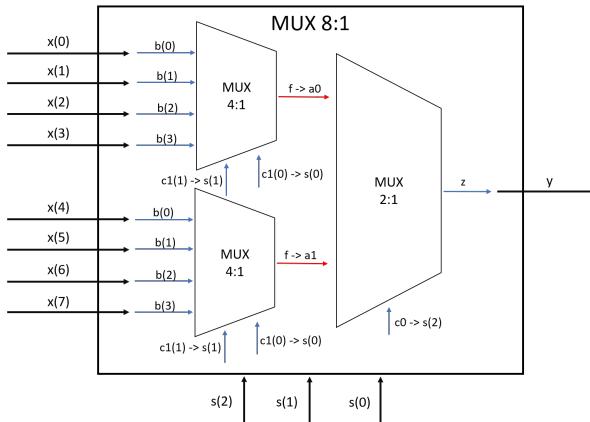


Figura 2.28: Schema multiplexer 8:1.

- la parte meno significativa ($c1(0)$) va in ingresso ai multiplexer del primo stadio e seleziona per ciascuno un filo in uscita, seleziona quindi il filo all'interno del singolo blocco;
- la parte più significativa ($c1(1)$) entra nel multiplexer del secondo stadio e decide quale dei due fili, provenienti dai due blocchi precedenti, andrà in uscita, dunque seleziona il blocco.

Ad esempio, se supponiamo di avere come ingresso di selezione la stringa “10” allora il bit meno significativo, pari a 0, fa uscire dai multiplexer del primo stadio l’ingresso a_0 , ovvero $b(0)$ per il primo e $b(2)$ per il secondo. Il bit più significativo, pari a 1, seleziona in uscita il filo a_1 del terzo multiplexer, dunque $b(2)$, selezionando il secondo blocco. Decomponendo l’indirizzo in questo modo, otteniamo in uscita l’ingresso atteso, ovvero quello avente come indice il numero binario “10”, pari a 2 in decimale ($b(2)$). Il procedimento per la composizione del **multiplexer 8:1** è analogo, questa volta oltre al multiplexer 2:1 sfruttiamo anche i multiplexer 4:1, appena realizzati. Il multiplexer 8:1 prende 8 segnali in ingresso che dividiamo in due gruppi da 4 e mandiamo a 2 multiplexer 4:1. Ciascuno di essi produce un’uscita e il multiplexer 2:1, al secondo stadio, decide quali tra di esse sarà l’uscita finale. Anche in questo caso abbiamo decomposto l’indirizzo in due livelli: al primo stadio abbiamo due dei multiplexer 4:1 a cui quindi manderemo due bit di selezione, i due bit meno significativi ($s(1)$ e $s(0)$); il multiplexer del secondo stadio invece è 2:1, per cui prenderà come unico ingresso di selezione il bit più significativo ($s(2)$). I due bit meno significativi selezionano quale tra i 4 fili in ingresso va in uscita a ciascuno dei multiplexer 4:1, del primo livello, mentre il bit più significativo seleziona uno tra i due blocchi, al secondo livello. Lo schema analizzato è presente in figura 2.28.

Codice

Multiplexer 2:1

Partiamo dall’implementazione del multiplexer 2:1. Per prima cosa, bisogna definire l’ *entity* mux_21, al cui interno dichiariamo le porte.

In questo caso, come visto, abbiamo 3 ingressi (a_0 , a_1 e c_0) e un’uscita (z), di tipo std_logic. Dato che tale oggetto è il componente fondamentale della nostra architettura, l’abbiamo descritto a un livello più basso di astrazione, a *livello RTL*, descrivendo esplicitamente le trasformazioni che i dati subiscono. In particolare, il valore dell’uscita dipende dal valore assunto dall’ingresso di selezione: è pari ad a_0 quando c_0 vale 0, ad a_1 quando c_0 è pari a 1. Per descrivere ciò abbiamo utilizzato un *costrutto when*, dunque un’*assegnazione condizionata*. Notiamo che è presente un caso aggiuntivo, di default, in quanto abbiamo definito i segnali come std_logic, c_0 non può assumere solo i valori 0

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity mux_21 is
    port
        ( a0, al: in std_logic;
          c0: in std_logic;
          z: out std_logic
        );
end mux_21;

architecture rtl of mux_21 is
begin
    z<= a0 when c0='0' else
        al when c0='1' else
        '-';
end rtl;

```

Figura 2.29: Codice VHDL multiplexer 2:1.

o 1 ma ciascuno dei 9 valori definiti per questo tipo: per qualunque altro valore di c_0 , diverso da 0 o 1, l'uscita assumerà valore non specificato (-).

Multiplexer 4:1

Il multiplexer 4:1 è stato descritto in modo strutturale (figura 2.30). L'entity è un oggetto con 6 ingressi, di cui 2 di selezione, e un'uscita. Anche qui il tipo è std_logic ma notiamo che gli ingressi sono stati dichiarati come vettori per facilitare l'implementazione, così facendo infatti abbiamo potuto istanziare più oggetti analoghi tramite un *ciclo for*.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity mux_41 is
5      port
6          ( b: in std_logic_vector(0 to 3);
7            cl: in std_logic_vector(1 downto 0);
8            f: out std_logic
9          );
10 end mux_41;
11
12 architecture structural of mux_41 is
13
14     signal u: std_logic_vector(0 to 1);
15
16     Component mux_21 is
17         port
18             ( a0, al: in std_logic;
19               c0: in std_logic;
20               z: out std_logic
21             );
22     end component;
23
24 begin
25     mux0to1: FOR i IN 0 TO 1 GENERATE
26         m: mux_21
27             port map
28                 (b(i*2),
29                  b(i*2+1),
30                  cl(0),
31                  u(i)
32             );
33     end GENERATE;
34
35     mux2: mux_21
36         port map
37             ( u(0),
38               u(1),
39               cl(1),
40               f
41             );
42
43 end structural;

```

Figura 2.30: Codice VHDL multiplexer 4:1.

In realtà, sarebbe più corretto definire gli ingressi singolarmente e non come un unico vettore, in quanto nella pratica si potrebbe trattare di 4 fili provenienti da sorgenti differenti, che viaggiano indipendentemente, come mostrato in figura 2.31. Notiamo che la definizione dei versi dei vettori (0 to 3 per il vettore b , 1 downto 0 per il vettore c_1) è coerente con quanto schematizzato in figura 2.27. I componenti di questo oggetto sono i mux_21 realizzati precedentemente. Li definiamo nel blocco *component* e poi li istanziamo. Prima di fare ciò abbiamo definito un vettore u di 2 segnali, dobbiamo infatti collegare i mux del primo stadio al mux del secondo stadio, per farlo serve definire un segnale d'appoggio, un filo che esca dalla prima macchina ed entri nella successiva. I primi due mux sono istanziati in un ciclo for perché lavorano allo stesso modo, prendendo però in ingresso un gruppo diverso degli ingressi della macchina esterna. Il mux0 prende in ingresso i segnali che vanno da $b(0)$ a $b(1)$, il mux1 da $b(2)$ a $b(3)$, ognuno quindi prende in ingresso i segnali da $b(2*i)$ a $b(2*i+1)$. L'ingresso di selezione è lo stesso, $c_1(0)$, ovvero il bit meno significativo, e l'uscita viene data al segnale temporaneo $u(i)$, rispettivamente $u(0)$ e $u(1)$. Il mux2 invece prende in ingresso

il vettore u, quindi u(0) come a0 e u(1) come a1, prende come ingresso di selezione il bit più significativo (c1(1)) e manda in uscita l'uscita finale f.

```
entity mux_4_1 is
  port (b0,b1,b2,b3 : in std_logic;
        c1,c0 : in std_logic;
        f : out std_logic
      );
end mux_4_1;
```

Figura 2.31: Dichiaraione mutliplexer 4:1, senza vettori.

Multiplexer 8:1

In maniera molto simile abbiamo composto il multiplexer 8:1, la cui entity presenta un vettore di 8 ingressi (x), un vettore di 3 ingressi di selezione (s) e un'uscita (y), tutto di tipo std_logic (figura 2.32). Come component ora dobbiamo dichiare i mux_41, che serviranno per il primo livello, e il mux_21 che servirà per il secondo livello. Per l'architettura, analogamente a quanto visto per il multilplexer 4:1, istanziamo i primi due mux 4:1 in un ciclo for, ognuno prende in ingresso 4 segnali del vettore x, il primo ($i=0$) prende i segnali che vanno da $x(0)$ a $x(3)$, il secondo ($i=1$) da $x(4)$ a $x(7)$, dunque da $x(i*4)$ a $x(i*4+3)$. Entrambi prendono in ingresso i due bit meno significativi dell'ingresso di selezione (s(1) e s(0)) e mandano l'uscita nel vettore temporaneo (temp(i)). L'ultimo mux, questa volta 2:1, prende in ingresso le uscite dei primi due (temp(0) e temp(1)) e come bit di selezione il bit più significato (s2). Infine manda l'uscita nell'uscita finale del sistema, y.

```
entity mux_81 is
  port
    ( x: in std_logic_vector(0 to 7);
      s: in std_logic_vector(2 downto 0);
      y: out std_logic
    );
end mux_81;
architecture structural of mux_81 is
  signal temp: std_logic_vector(0 to 1);
begin
  mux0to1: FOR I IN 0 TO 1 GENERATE
    m: mux_41
    port map
      ( x(i*4 to i*4+3),
        s(1 downto 0),
        temp(i)
      );
  end GENERATE;
  m2: mux_21
  port map
    ( temp(0),
      temp(1),
      s(2),
      y
    );
end structural;
```

Figura 2.32: Codice VHDL multiplexer 8:1.

Simulazione

Per effettuare la simulazione per prima cosa dobbiamo creare il **testbench**, il cui codice è mostrato in figura 2.33. La prima cosa che possiamo notare è che il corpo della entity è vuoto, questo infatti non è un oggetto che realizziamo ma serve solo per effettuare la simulazione e quindi verificare che il sistema realizzato funzioni correttamente. Il testbench non ha segnali di ingresso né di uscita perché non può essere istanziato da nessun blocco ma istanzia al suo interno altri blocchi per effettuarne il test. Nel nostro caso, l'oggetto da testare è il mux 8:1 definito precedentemente, quindi lo istanziamo in questa architecture. Definiamo i segnali che mandiamo in ingresso al mux, stiamo definendo essenzialmente l'alimentatore. Istanziando il mux 8:1 e collegiamolo ad esso tali segnali, ovvero inputx, inputs e l'output, tramite cui controlleremo che il sistema funzioni correttamente. Adesso definiamo un *process*, un algoritmo che il nostro testbench esegue per effettuare il testing.

Aspettiamo 4 ns e diamo in ingresso il vettore “00001100” e come vettore di selezione “101”, che corrisponde al numero decimale 5. Ci aspettiamo allora di vedere in uscita il segnale x(5), che corrisponde a 1. Utilizzando le *assert* verifichiamo allora che l’output sia pari a 1, in caso contrario segnaliamo l’errore e arrestiamo il processo (riga 37-38). Notiamo che prima di effettuare tale verifica aspettiamo 10 ns, ovvero aspettiamo che i transitori si stabilizzino e l’uscita vada a regime, altrimenti rischiamo di vedere un valore differente e arrestare la simulazione. Come secondo caso di test abbiamo modificato il vettore in ingresso, abbassando x(5), ma non modificando l’ingresso di selezione, ci aspettiamo allora di vedere in uscita il valore 0 e lo verifichiamo con un altro assert, dopo un’attesa di 10 ns. Infine modifichiamo l’ingresso di selezione in “100”, per cui in uscita ci aspettiamo il valore di x(4), pari a 1. Verifichiamo anche ciò e poi terminiamo il processo.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.all;
3
4 entity mux_81_tb is
5 end mux_81_tb;
6
7 architecture behavioural of mux_81_tb is
8 begin
9 component mux_81
10 port
11   ( x: in std_logic_vector(0 to 7);
12     s: in std_logic_vector(2 downto 0);
13     y: out std_logic
14   );
15 end component;
16
17 signal inputx: std_logic_vector(0 to 7);
18 signal inputs: std_logic_vector(2 downto 0);
19 signal output: std_logic;
20
21 begin
22 uut: mux_81
23   port map
24     ( inputx,
25       inputs,
26       output
27     );
28
29 stim_proc: process
30 begin
31
32   wait for 4 ns;
33   inputx <= "00001100";
34   inputs <= "101";
35
36   wait for 10 ns;
37   assert output='1'
38   report "errore0"
39   severity failure;
40   inputx <= "00001000";
41
42   -- varia l'ingresso ma non il segnale di controllo
43   wait for 10 ns;
44   assert output='0'
45   report "errore1"
46   severity failure;
47   inputs <= "100";
48
49   wait for 10 ns;
50   assert output='1'
51   report "errore2"
52   severity failure;
53
54 end process;
55
56 end behavioural;

```

Figura 2.33: Codice VHDL testbench.

Lanciando la simulazione possiamo verificare quanto appena spiegato. In figura 2.34 vediamo che inizialmente i segnali assumono valore indeterminato, in quanto non li abbiamo inizializzati perchè nella realtà non ne conosciamo i valori iniziali. Dopo 4 ns modifichiamo i valori di inputx e inputs, l’ingresso di selezione vale 101, la linea x(5) è alta e infatti notiamo che l’uscita si porta a un valore alto. Dopo altri 10 ns abbassiamo x(5) e l’uscita si porta bassa. Infine, dopo altri 10 ns modifichiamo l’ingresso di selezione in 100 e l’uscita torna alta, in quanto x(4) è ancora alta. Osserviamo che ogni volta che modifichiamo gli ingressi l’uscita varia a tempo 0, istantaneamente, questo perchè nell’implementazione dei componenti non abbiamo aggiunto ritardi. Se li aggiungessimo potremmo notare che l’uscita varierebbe solo in seguito a un certo ritardo, deltaT.

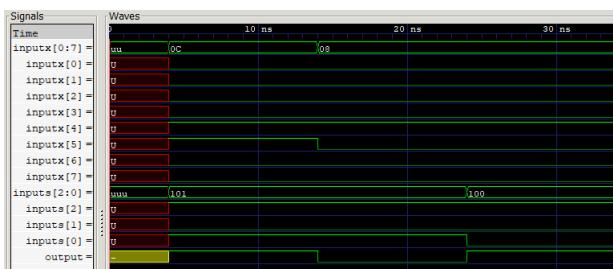


Figura 2.34: Simulazione in gtkwave del multiplexer 8:1.



Capitolo 3

Macchine Sequenziali

Un problema è di tipo sequenziale quando l'uscita dipende sia dall'ingresso che dallo stato, le macchine sequenziali sono macchine che hanno memoria, in cui lo stato rappresenta l'evoluzione di esse. Il modello matematico che consente di descrivere una macchina sequenziale è l'automa, il quale ha la potenza espressiva della macchina di Turing. È possibile realizzare la macchina sequenziale anche mediante il linguaggio (VHDL nel nostro caso): se implementiamo l'automa o descriviamo la macchina mediante il linguaggio stiamo dal punto di vista espressivo realizzando la stessa cosa, tuttavia ciò non significa che dal punto di vista implementativo abbiamo fatto due cose uguali. Infatti, quello che fuoriesce dal linguaggio non è detto che sia l'automa realizzato con carta e penna.

3.1 Classificazione delle macchine sequenziali

Un **ASF - Automa a stati finiti** è una quintupla $\langle Q, I, U, t, w \rangle$ con:

$$q \in Q \quad i \in I \quad u \in U$$

dove Q , I ed U sono rispettivamente gli insiemi finiti degli stati interni, ingressi ed uscite. La funzione t è la **funzione di transizione** definita come:

$$t: Q \times I \rightarrow Q$$

Infine, w è la **funzione di uscita** definita differentemente a seconda del modello di automa a stati finiti.

- **Modello ASF di Mealy:** si rappresenta la variazione dell'uscita a fronte di ingresso e di stato.

$$w: Q \times I \rightarrow U$$

- **Modello ASF di Moore:** L'uscita dipende solo dallo stato, vale a dire che l'ingresso modifica lo stato e l'uscita è espresso in funzione dello stesso stato.

$$w: Q \rightarrow U$$

Si rappresenta pertanto l'uscita a fronte della variazione dello stato, che a sua volta varia in funzione dell'ingresso.

Una macchina sequenziale può essere descritta mediante la **tavola degli stati**: gli indici di colonna sono i simboli di ingresso; gli indici di riga sono i simboli di stato che indicano lo stato presente. Gli elementi sono la coppia (q', u) dove $q' = t(i, q)$ è il simbolo stato prossimo e $u = w(i, q)$

- Macchine di Mealy

i_1	i_2	...
S_j^{**t} / u_j	S_k^{**t} / u_k	...
S_m^{**t} / u_m	S_l^{**t} / u_l	...
...

- Macchine di Moore

i_1	i_2	...	u_t
S_j^{**t}	S_k^{**t}	...	u_t
S_m^{**t}	S_l^{**t}	...	u_2
...

Figura 3.1: Tabella degli stati per le macchine di Mealy e di Moore.

è il simbolo di uscita, nel caso della macchina di Mealy. Se la macchina è di Moore, $q' = t(i, q)$ è il simbolo stato prossimo. Nelle macchine di Moore i simboli d'uscita sono associati allo stato presente.

Spesso, la stesura della tabella degli stati è preceduta da una rappresentazione grafica ad essa equivalente, denominata **diagramma degli stati**. Il Diagramma degli stati è un **grafo orientato** $G(V, E, L)$ con:

$$v \in V \quad e \in E \quad L \in \mathbb{L}$$

dove V è l'insieme dei nodi, ognuno dei quali rappresenta uno stato. Inoltre, ad ogni nodo è associato un simbolo d'uscita (macchine di Moore). E è l'insieme degli archi, dove ogni arco rappresenta le transizioni di stato. Infine, L è l'insieme degli: ingressi e uscite (macchine di Mealy), ingressi (macchine di Moore).

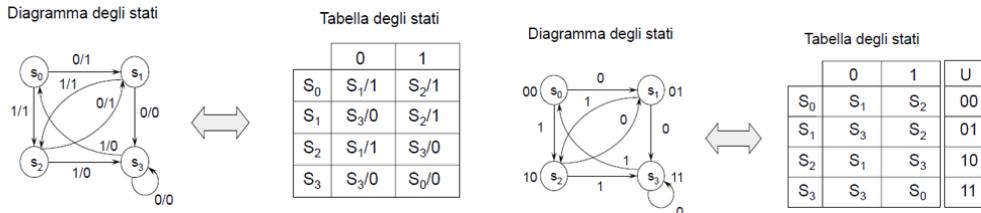


Figura 3.2: Equivalenza tra la macchina di Mealy e quella di Moore.

Le macchine sequenziali possono essere **asincrone**, intrinsecamente stabili poiché partono da uno stato stabile ed arrivano sempre in uno stato stabile, oppure possono essere **sincrone**, cioè macchine per cui questa proprietà di stabilità non è verificata per ogni ingresso e per ogni stato.

3.2 Schema di una macchina sequenziale

Il grande problema di queste macchine è la tempificazione. La matematica consente di progettare gli automi senza tener conto del tempo, tuttavia nella realizzazione di queste macchine rientra il problema della **fisica realizzabilità**, motivo per cui non è necessario solo il segnale di ingresso, ma anche un **segnale di sincronismo** (**macchina a sincronizzazione esterna**), che consenta di stabilire gli istanti di tempo in cui la macchina evolve in modo significativo.

Supponiamo che il segnale di ingresso sia quello in figura 3.3, senza un segnale di sincronismo che renda l'ingresso significativo solo in alcuni istanti di tempo, non siamo in grado di poter sempre distinguere ingressi successivi, quindi non possiamo dire di aver ricevuto tre volte l'1.

Potremmo pensare di realizzare la macchina sequenziale come in figura 3.4: è una macchina combinatoria che prende in ingresso I, segnale di ingresso, ed S, lo stato (realizza una tabella). Dall'ela-

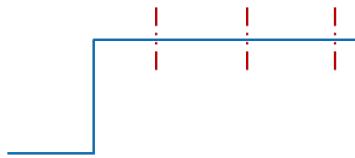


Figura 3.3: Esempio segnale di ingresso.

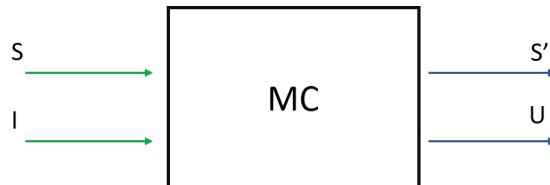


Figura 3.4: Primo schema di una macchina sequenziale.

borazione della macchina combinatoria, si ottengono l'uscita e il nuovo stato S' . Questa macchina realizza sempre la trasformazione tra lo stato precedente e l'ingresso producendo l'uscita ed il nuovo stato.

Il problema è che, se la facessimo realmente così, non sapremmo in quali istanti di tempo i segnali sono significativi. È necessaria una **memoria** che conservi lo stato (*proprietà intrinseca del sistema*), è come se fosse costituita da due sottoreti, una che calcola l'uscita e magari il segnale U va verso un'altra macchina ed un'altra che calcola lo stato, il quale viene conservato. Se la macchina deve avere la capacità di distinguere gli ingressi **occorre retroazionare lo stato** (figura 3.5), perché se mettiamo in ingresso uno stato S ed I , la macchina dopo un certo intervallo di tempo raggiunge l'equilibrio, producendo il nuovo stato, ma se lo stato S di ingresso è sempre lo stesso, S' deve sempre essere uguale.

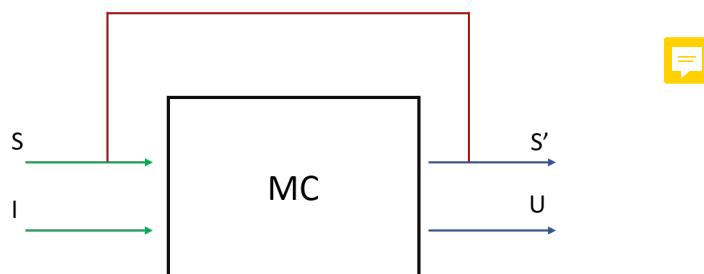


Figura 3.5: Secondo schema di una macchina sequenziale.

Come cambia lo stato? È necessario cambiare l'ingresso, perché la macchina è intrinsecamente stabile, ma, dato che I è significativo solo in alcuni istanti di tempo, se non diamo un segnale di sincronismo la macchina non è in grado di evolvere. C'è bisogno di un altro elemento, il **registro**, mediante cui è possibile memorizzare lo stato, contribuendo alla sua evoluzione, così che se viene rimosso il segnale di abilitazione la macchina è ferma. Lo schema della macchina sequenziale diventa quello in figura 3.6. È un modello *transfer register*, perché nella parte combinatoria facciamo i calcoli e di volta in volta trasferiamo il nuovo stato nei banchi di registri. In realtà, c'è un ulteriore problema. Se l'impulso di sincronismo è ideale, il registro prende lo stato, ad esempio ad ogni fronte di salita. Se invece l'impulso dura molto, la macchina passa da uno stato q_0 ad uno q_1 . Se l'abilitazione è ancora alta, cattura nuovamente, per cui con un solo impulso di sincronismo, invece di muoversi da q_0 a q_1 , la macchina si muove da q_0 a q_2 . Bisogna introdurre una variabile, il **tempo**.

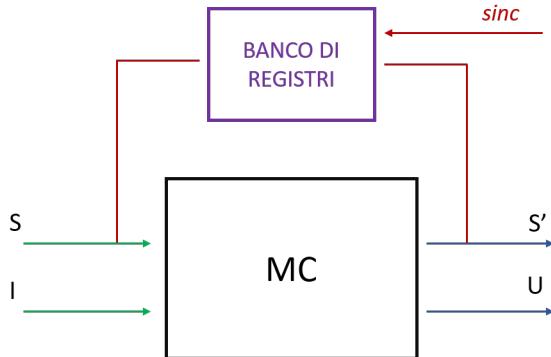


Figura 3.6: Schema definitivo della macchina sequenziale.

Osservazione: alcuni fenomeni che sono all'interno della macchina sono **transitori** (ad un certo punto finiscono). In un sistema retroazionato senza registri (figura 3.5), c'è un problema di transitori che non si può controllare, bisogna fare in modo da non averli. Con i sistemi a sincronizzazione esterna invece, possiamo fare in modo che il transitorio si estingua e solo dopo venga data l'abilitazione, tenendo conto anche del tempo di risposta dei blocchi della macchina. Se la macchina combinatoria ritarda di un **tempo Tr**, dobbiamo aspettare almeno questo tempo per poter dare l'impulso di sincronismo. In realtà il lasso di tempo dura:

$$T_r + T_h$$

se la rete non riesce a sentire l'ingresso, non è in grado di reagire, quindi ci sarà un **tempo minimo che è quello fisico per poter far reagire la rete ed un tempo in cui la rete risponde**; dopo $T_r + T_h$ possiamo dare l'abilitazione. Si sottolinea che la frequenza a cui possiamo lavorare **non è**

$$\frac{1}{T_r + T_h}$$

perchè **il banco di registri impiega un tempo T_d per memorizzare** ed è caratterizzato da **un tempo in cui i registri percepiscono il segnale ed uno di ritardo**.

Questo sistema si basa su ritardi e tempificazione, è importante perchè nella realtà noi effettivamente dobbiamo dare il segnale di ingresso e quello di abilitazione. Se quest'ultimo viene dato con troppo anticipo, l'intero sistema non funziona. La tempificazione viene affrontata con maggior dettaglio nella sezione 3.5. C'è anche il problema della scelta dei registri: se il banco di registri è fatto da flip-flop di tipo D, se dobbiamo scrivere 0 basta scrivere 0, mentre se utilizziamo un FF RS, per scrivere 0 dobbiamo mettere il set a 0 ed il reset ad 1 (ci sono due fili).

Si evidenzia anche che, dato il modello in figura 3.6, **il FF base con cui si costruiscono le celle di memoria non può essere sincrono** (se fosse sincrono avremmo bisogno di una cella di memoria per realizzarlo) **ma è asincrono, retroazionato, intrinsecamente stabile.**

3.3 Flip-Flop RS

Dato che è una macchina bistabile, ha due stati, q0 (codificato con 0) e q1 (codificato con 1). Gli ingressi possibili sono due, il set e reset. Il valore neutro dell'ingresso è 00, cioè quando S=0 ed R=0, il FF continua a memorizzare il valore precedente. Prima di tutto disegniamo l'automa di questa macchina, che deve essere asincrona: dallo stato q0, se mettiamo in ingresso 10 (il primo valore è di reset ed il secondo di set) rimaniamo in q0, se mettiamo in ingresso 01 passiamo da q0 a q1, perchè dobbiamo memorizzare un 1. La macchina, come detto, è asincrona; infatti se continuiamo a mettere 01, la macchina rimane dov'è (q1 nell'ultimo caso). Se da q1 mettiamo in

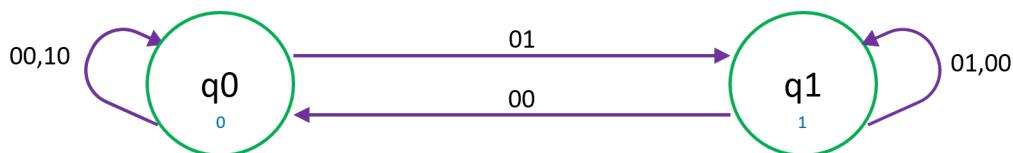


Figura 3.7: ASF del Flip Flop RS.

ingresso 10 ritorniamo in q_0 . Per quanto riguarda l'ingresso neutro, in entrambi i casi il sistema rimane dove. La sequenza 11 non è ammessa. Dato che lo schema che viene fuori è quello della figura 3.5, occorre evitare il transitorio.

Il problema delle macchine asincrone prende il nome di **alee**, comportamenti non voluti nella macchina combinatoria. L'alea si risolve facendo in modo che gli ingressi non varino mai di due o più variabili alla volta. Questo vuol dire che non possiamo passare dal valore 01 a 10 (da reset a set) poiché stanno variando due bit in un'unica transizione, dobbiamo effettuare un passaggio di questo tipo 01 - 00 - 10, cioè *dobbiamo sempre passare per il valore neutro*. Se non riusciamo, aggiungiamo una variabile per la codifica degli stati al fine di garantire che la codifica vari di un solo bit. Aumentare il numero di bit d'altro canto complica la rete combinatoria, del resto quando realizziamo una rete asincrona non sempre la macchina combinatoria che viene fuori è minima, perché per far sì che la variazione tra i bit di un ingresso e quelli del successivo sia al più di un valore, è necessario portare in conto tutti i termini adiacenti nella minimizzazione.

	00	01	11	10
0				1
1		1	1	1

Figura 3.8: Problema delle alee in una rete asincrona.

Se prendiamo l'esempio della mappa di Karnaugh in figura 3.8, ai fini della minimizzazione di una rete combinatoria basterebbe prendere i due mintermini verdi, tuttavia, se questa serve per realizzare una rete asincrona, rischieremmo di avere una transizione 110 - 101, in cui variano due bit e non uno. Questo vuol dire che staremmo introducendo un'alea nel sistema, per evitare ciò prendiamo anche il termine di congiunzione rosso.

N.B: per approfondimenti sulle tipologie di alee si rimanda all'appendice A.

Possiamo realizzare questo FF aggiungendo un **segnale abilitante**, come mostrato in figura: quando l'abilitazione A è alta, il segnale si propaga. Il FF realizzato è sempre bistabile ed asincrono.

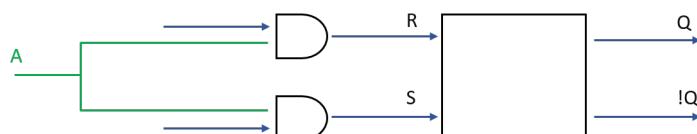


Figura 3.9: Realizzazione flip-flop RS abilitato.

crono, solo che funziona abilitato. E' come se ci fossero due valori neutri, sia quando A=0 e sia quando A=1 ma RS=00.

3.4 Flip-Flop D

Dal FF RS della sezione precedente, vogliamo progettare il FF D che ha abilitazione e dato in ingresso. È una macchina ancora **asincrona**, ma viene chiamato **flip-flop sincrono bistabile** perché si muove se c'è un segnale di sincronismo. Bisogna distinguere tra la classificazione di una macchina (asincrono) e quella riguardante gli ingressi (sincrono).

Il FF D deve memorizzare 1, se il dato è 1 e l'abilitazione è alta, vuol dire che *il set è uguale al*

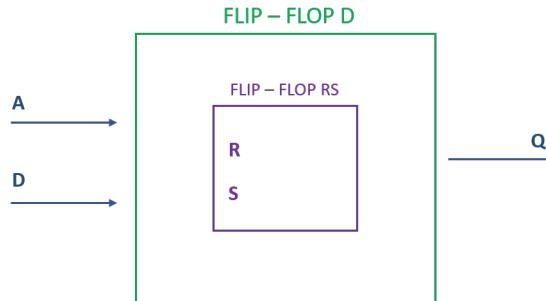


Figura 3.10: Realizzazione flip-flop D a partire dall'RS.

dato ed il reset è pari al dato negato.

Tutti i nostri discorsi partono dal presupposto che il circuito sia inizialmente in uno stato s0, perciò **il reset è fondamentale affinché il sistema parta da uno stato noto**. Questo vuol dire che sia l'RS che il D devono essere realizzati con il reset, sono sempre **macchine asincrone**, tuttavia si dicono a **caricamento sincrono**, poiché se non c'è l'abilitazione, che funge da sincronismo, **non funzionano**. In figura 3.11 è rappresentata la realizzazione del flip-flop D con il segnale di reset: la macchina che consente di scegliere tra un valore o un altro, cioè tra il dato o il reset è il **multiplexer**, che produce l'uscita $U=R*O+(!R)*D$.

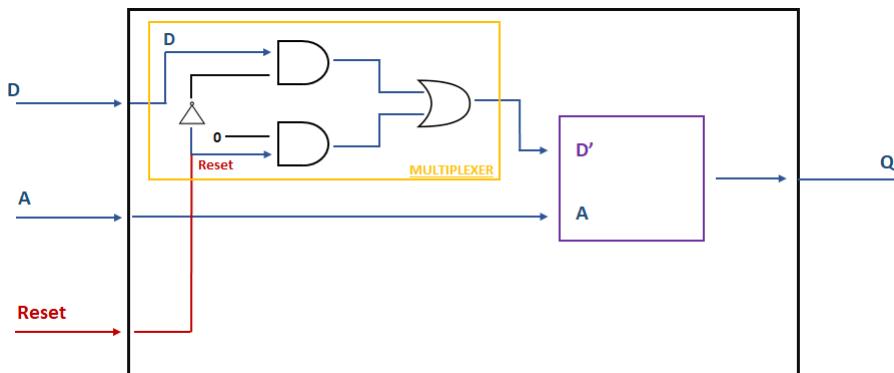


Figura 3.11: Realizzazione flip-flop D con segnale di reset.

3.5 Tempificazione delle macchine sequenziali

Il modello sincrono, come abbiamo visto, è formato da una parte combinatoria, di calcolo, e da registri, che memorizzano lo stato, come mostra la figura 3.12. In particolare, parliamo di **macchina di Moore** se l'uscita dipende solo dallo stato, dunque $U = f(q)$. Ciò significa che **l'uscita è a livelli**, finché lo stato si mantiene costante l'uscita non varia. Tra le macchine

notevoli, la macchina per antonomasia in cui l'uscita è funzione solo dello stato è il **flip flop**. L'uscita del flip flop corrisponde al suo stato. Lo stesso discorso vale per il contatore, l'uscita di conteggio non cambia finchè il contatore non cambia stato. Nelle **macchine di Mealy** invece l'uscita dipende dallo stato ma anche dal particolare ingresso, dunque $U = f(q, i)$. Una **macchina combinatoria** è una particolare macchina di Mealy per cui lo stato non cambia e l'uscita dipende solo dall'ingresso.

Possiamo anche avere una **macchina impulsiva**, in cui l'uscita è vera solo per un determinato stato e per un determinato ingresso, poi torna bassa. I riconoscitori di sequenza sono delle macchine di questo tipo. Supponiamo di riconoscere la sequenza nello stato q_2 se arriva l'ingresso i , allora solo nel passaggio da q_0 a q_2 , provocato dall'ingresso i , l'uscita sarà alta, poi tornerà immediatamente bassa in quanto non saremo più in q_2 . Questa è una macchina di Mealy, per farla diventare di Moore basta dire che quando siamo in q_2 , all'arrivo dell'ingresso i ci spostiamo in q_3 , lo stato che indica di aver riconosciuto l'uscita, che a quel punto sarà sempre alta. Il *funzionamento di queste macchine è semplice, a patto che si riesca a realizzare un comportamento completamente impulsivo*.

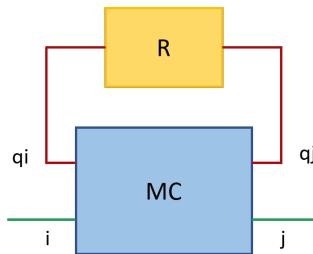


Figura 3.12: Schema di una macchina sequenziale.

Con riferimento alla figura 3.12, quando mettiamo l'ingresso i la macchina si porta nello stato q_j , che verrà memorizzato nel registro solo all'arrivo del segnale di sincronismo. Se consideriamo i bistabili base, i **latch**, questi continuano a memorizzare fintanto che l'abilitazione è alta. **Do-vremmo allora avere un segnale di sincronismo sufficientemente stretto per far sì che la macchina si sposti solo da q_i a q_j , quindi consideri l'ingresso una sola volta.** Se non riusciamo a realizzare ciò, il sistema oscillatorà, senza mai convergere. Il problema fondamentale allora risulta quello di **tempificare le macchine, per fare in modo che si muovano un passo alla volta**.

3.5.1 Flip Flop Edge-Triggered e Master Slave

Un **flip flop D** è un bistabile, può trovarsi in due soli stati (q_0 o q_1). Vediamo come farlo lavorare in modo impulsivo. Quando l'abilitazione, A , è alta, il flip flop sente le variazioni del dato.¹ La figura 3.13 mostra il tipico funzionamento di un flip flop, nell'intervallo in cui A è alto l'uscita del flip flop (Y) segue il dato (D).

A partire da questo oggetto, vogliamo realizzare un oggetto che abbia un comportamento impulsivo. Vogliamo realizzare o un **comportamento Edge-Triggered (ET)**, per cui l'uscita cambia solo in corrispondenza del fronte di salita o di discesa dell'abilitazione, o un **comportamento Master-Slave (MS)**, per cui la macchina campiona sul fronte di salita ma mostra ciò che ha campionato sul fronte di discesa. La figura 3.14 mostra il funzionamento di tali macchine. In particolare, l'ET campiona sul fronte di salita, trova il valore 0 e l'uscita non cambia fino al fronte successivo. Il MS invece, quando A si alza campiona, quindi campiona 0, ma lo presenta solo sul fronte di salita.

¹In realtà, con il termine flip flop si fa riferimento a macchine **edge-triggered**, ovvero macchine che commutano sul fronte di salita o discesa del segnale abilitante. Le macchine che sentono le variazioni del dato, quando abilitate, sono i **latch**. In questo capitolo il termine flip flop è utilizzato nel suo senso generale di bistabile.

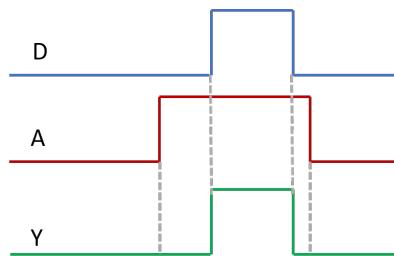


Figura 3.13: Diagramma temporale di un flip flop.

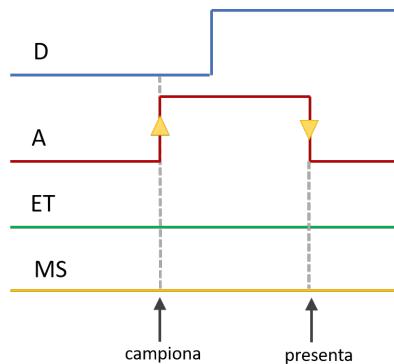


Figura 3.14: Confronto ET/MS.

Una macchina di questo tipo non può essere un bistabile, servono più di due stati. In particolare, per l'ET serviranno meno stati rispetto al MS, perché una volta commutato, sul fronte di salita, non dovrà fare altro, invece il MS deve aspettare il fronte di discesa per presentare. È preferibile utilizzare ET sul fronte di discesa, in quanto sul fronte di salita la macchina campiona prima ma il suo valore cambia quando l'impulso è ancora alto. Eventuali macchine a valle vedranno il cambiamento quando l'impulso è ancora alto e questo può portare a problemi (un esempio di ciò è il contatore seriale). Sul fronte di discesa, invece, tale problema non si presenta perché il segnale di sincronismo è già terminato, dobbiamo però essere sicuri che l'ingresso rimanga stabile fino a quel momento, altrimenti viene memorizzato un valore non corretto. La sincronizzazione delle macchine è un problema fondamentale in ogni sistema. Le macchine che lavorano come visto, sono macchine asincrone: cambiano comportamento solo se varia un determinato segnale, altrimenti rimangono dove si trovano, sono intrinsecamente stabili. Realizzare questo comportamento allora è complesso, oltre a richiedere un maggior numero di stati richiede di gestire tutti i problemi legati alle macchine asincrone.

Vediamo un modo alternativo per realizzare questo comportamento. Rendere una macchina attiva solo sul fronte di salita, impulsiva, equivale ad avere un'abilitazione molto stretta, in modo che sia come un impulso, rispetto alla nostra base dei tempi. Dev'essere stretta sufficientemente da far effettuare una sola memorizzazione, quando l'impulso è alto. Rendere un segnale stretto significa farne la derivata e ciò va bene per la specifica velocità a cui gira il circuito, se cambia qualcosa che prima era considerabile impulsivo potrebbe non esserlo più. Fare la derivata va bene ma la dimensione a cui dobbiamo arrivare dipende dalla base dei tempi. In particolare, dobbiamo fare la derivata digitale, un circuito che effettua ciò è mostrato in figura 3.15(a). Il filo superiore fa viaggiare il segnale così com'è, il filo inferiore invece lo fa passare per due porte NOT, ritardandolo. Se facciamo il prodotto tra i due segnali (figura 3.15(b)), otteniamo un segnale più stretto, tanto più stretto quanto più sarà il ritardo introdotto. Notiamo, dunque,

che il risultato è fortemente legato alla tecnologia, se cambia la tecnologia, cambia il ritardo introdotto e cambia il segnale risultante. In realtà, questo comportamento non è effettivamente ET, è semplicemente una macchina che lavora ricevendo un segnale di abilitazione stretto. Inserendo un opportuno ritardo, il circuito sembra lavorare come un ET, rispetto alla nostra base dei tempi, ma solo per l'inserimento del blocco derivatore, togliendolo lavorerebbe come un comune bistabile.

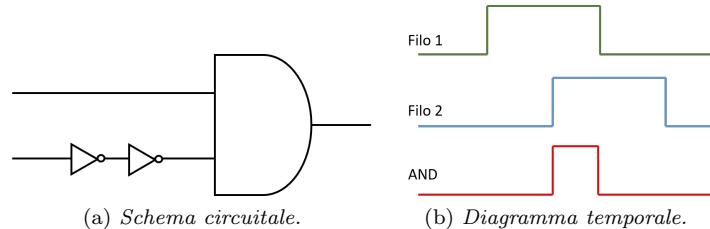


Figura 3.15: Derivata digitale.

Per realizzare il comportamento MS, invece, si utilizzano due bistabili, uno *master* e uno *slave*, come mostra la figura 3.16. Se ciascun bistabile ha due stati, la macchina complessiva ha 4 stati. È comunque più complesso di un semplice bistabile perché lavora differentemente in due istanti, uno di acquisizione e uno di presentazione. Vediamo come funziona, fintanto che l'abilitazione A è alta, funziona il master (M), che segue il dato. Lo slave (S) riceve A negato ($\neg A$), quindi fintanto che A è alto, non funziona. Quando A si abbassa, $\neg A$ si alza, quindi il master non funziona, mantiene un valore fisso, mentre lo slave acquisisce e presenta il valore del master. Se il dato (D) varia, il master ora non lo sente, lo sente solo se A è alto.

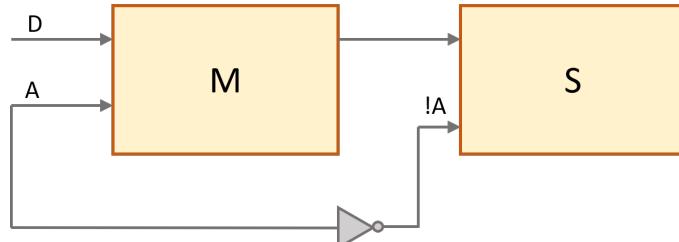


Figura 3.16: Master Slave.

A stretto rigore questo comportamento non è master-slave perché quando l'abilitazione è alta il master continua a seguire il dato. Non campiona sul fronte di salita ma segue il dato finché A è alto, quando scende lo presenta. Rispetto al circuito visto prima, con la derivata digitale, è comunque migliore, in quanto il funzionamento non è legato a un ritardo. È un sistema che lavora in opposizione di fase, la porta NOT rappresenta uno sfasatore. Con questa realizzazione dobbiamo fare attenzione, il dato non deve variare nell'intervallo in cui l'abilitazione è alta, altrimenti l'uscita risulterebbe impredicibile.

La figura 3.17 mostra un confronto tra flip flop MS e ET realizzati con i modelli visti (figura a) e secondo il loro comportamento ideale (figura b). Il latch quando l'abilitazione A è alta segue il dato D, che prima è alto e poi si abbassa, quando l'abilitazione si abbassa mantiene l'ultimo valore memorizzato, 0. Il comportamento dell'ET non ideale non è predicibile a priori, dipende dal ritardo della replica di A. Abbiamo una replica di A (A_2) ritardata di un certo Δt , a seconda del quale avremo un'abilitazione più o meno stretta. Solo guardando A e A_2 possiamo sapere ciò che accade, perché dipende dal ritardo. Infine, l'MS non ideale ha il master sensibile

quando A è alto, segue il dato che prima è 1 e poi diventa 0. Quando A scende, lo slave funziona e presenta l'ultimo valore memorizzato dal master, ovvero 0. Vediamo invece il comportamento ideale: l'ET sul fronte di salita campiona il dato quando A sale, quindi campiona 1; l'ET sul fronte di discesa campiona il dato quando A scende, quindi campiona 0; infine, il MS campiona quando A sale, quindi campiona 1, ma lo presenta quando A scende. Notiamo, in particolare, che *le uscite degli ET ora sono perfettamente predibibili, l'uscita del MS invece è diversa dal caso non ideale, quindi infatti campiona 1 in quanto non è sensibile al livello ma alla variazione dell'abilitazione*.

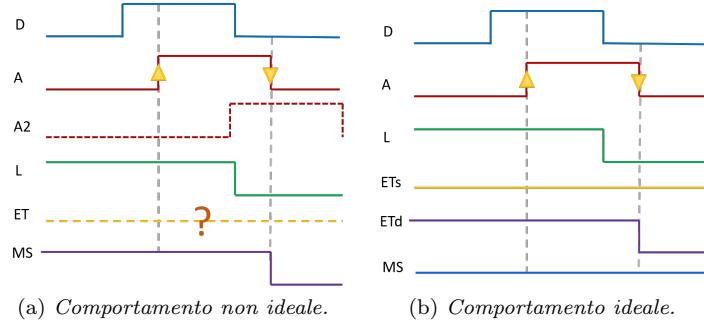


Figura 3.17: Confronto tra comportamento ideale e non.

3.6 Costruzione di un flip flop Master Slave

N.B: Per lo studio del comportamento e della realizzazione di un flip flop MS ideale, si rimanda all'appendice C.

Il comportamento del MS ideale è tale perché campiona quando sale e presente quando scende, dunque non vede tutto ciò che accade in mezzo. Nel caso non ideale, invece, usiamo degli oggetti reali, i bistabili, realizzarlo è più semplice, tuttavia, il suo comportamento è più simile a un ET sul fronte di discesa che a un vero MS. Il motivo per cui si parla di MS è che vi sono due componenti: **il comportamento MS non è dato dall'idealità dell'impulso ma dalla presenza di due componenti.** Vediamo come costruire questa macchina, ricordiamo che quando l'abilitazione sale deve campionare, quando scende deve presentare. Supponiamo di partire dallo stato q0 e che la macchina riceva come primo bit l'abilitazione, A, come secondo bit il dato, D. La macchina può presentare 0 o 1, dunque abbiamo uno stato q0 in cui presenta 0 e uno stato q1 in cui presenta 1. Quando presenta 0 e l'abilitazione si alza può campionare 0 o 1 (q00 o q01), analogamente, quando presenta 1 e l'abilitazione si alza, può campionare 0 o 1 (q10 o q11). **Il MS ideale allora non ha 4 stati, come ipotizzato precedentemente, ma 6.** Questo però significa che è una macchina che lavora meglio, in quanto ricorda più cose.

La figura 3.18 mostra l'automa fino allo stato q1, il comportamento successivo è uguale ed è descritto in appendiceC. Finchè A è bassa, qualunque sia il valore D, la macchina rimane ferma in q0. Quando A si alza (variazione 0→1), la macchina può andare in due stati, se D=0 va in q00, se D=1 va in q01. Ricordiamo che *la macchina, cambiando questo stato, non cambia il valore presentato ma campiona*. Supponiamo allora di aver dato 10 (prima A e poi D) e di essere passati in q00, se ora diamo come ingresso 10 rimaniamo ancora in q00, anche se mettiamo 11 perchè ormai il fronte di salita è passato, il MS ha già campionato. Se invece A si abbassa (variazione 1→0), indipendentemente dal valore di D (quindi con 00 o 01) si torna in q0. Torniamo a quando si alza A la prima volta, se il dato è D si passa in q01 (ingresso 11), qualunque sia il successivo valore di D, da questo stato quando si abbassa A (00 o 01) si passa allo stato q1, ovvero si presenta 1.

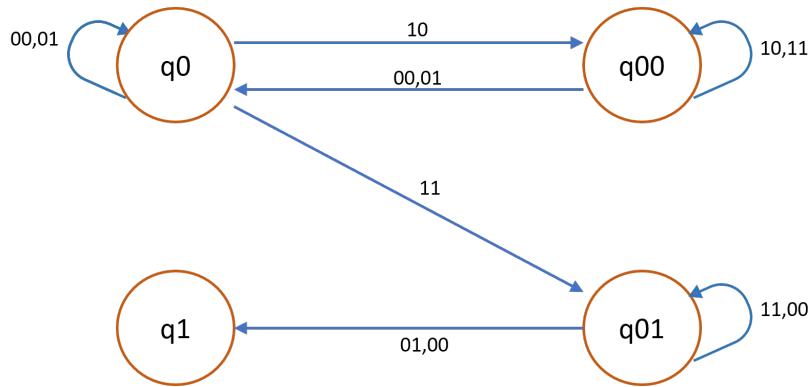


Figura 3.18: Automa del flip flop MS.

Questa macchina è asincrona perché si ferma sempre, inoltre è a variazione sul fronte, se non c'è variazione si ferma. I segnali di ingresso sono sempre adiacenti, varia solo un bit, ma anche la codifica degli stati dev'essere tale. Proviamo ad effettuare una codifica, come quella in tabella 3.1. Tra q00 e q01 vediamo che variano 2 bit ma questo non è un problema perché non sono stati adiacenti. Tra q10 e q0 invece variano 3 bit, non va bene perché sono stati adiacenti. È difficile anche trovare una codifica opportuna per q11 che deve variare di 1 solo bit rispetto a q1, non è facile risolvere questi problemi.

Tabella 3.1: Codifica degli stato del flip flop MS.

Stato	Codifica
q0	000
q00	001
q01	010
q1	011
q10	111
q11	?

Se utilizzando 3 variabili, ovvero quelle necessarie per codificare 6 stati, non riusciamo a risolvere il problema, **possiamo aumentare il numero di variabili**. Il problema però è che le AND/OR che dovremo usare avranno più ingressi e non è detto sia più semplice. Dunque, la codifica minima non è sempre la più vantaggiosa, a volte non da spazio per avere delle proprietà in più.

3.7 Costruzione di un flip flop Edge-Triggered

Vediamo come costruire un flip flop ET sul fronte di salita. Dobbiamo capire il senso fisico degli stati, **solo dopo aver trovato gli stati possiamo procedere alla codifica, che dev'essere tale che nel passaggio da uno stato all'altro vari al più un bit, altrimenti si verificano le alee**. La macchina si trova nello stato q00 se si trova in 0 e se arrivasse l'abilitazione presenterebbe 0; si trova in q01 se si trova in 0 e se arrivasse l'abilitazione presenterebbe 1; analogamente, si trova in q11 o q10 se si trova in 1 e se arrivasse A presenterebbe, rispettivamente, 1 o 0. **La macchina si predisponde per quando arriva l'abilitazione**. Vediamo la figura 3.19, quando D è 0 e A è 0, siamo in q00, siamo in 0 e leggiamo 0, quando il D si alza passa in q01, poi quando anche A si alza, passa in q11.

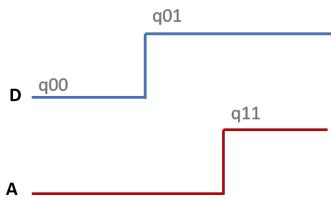


Figura 3.19: Diagramma temporale.

La figura 3.20 mostra il comportamento dell'automa. Se la macchina si trova in q_{00} e arriva l'ingresso 10 (prima A e poi D), rimane in q_{00} . Non possiamo mettere l'ingresso $00 \rightarrow 11$ perché si avrebbe un'alea, cambierebbe più di un bit nell'ingresso. Possiamo allora mettere la sequenza $00 \rightarrow 10 \rightarrow 11$ ma comunque la macchina permane nello stato q_{00} perché lo stato cambia quando A da 0 va a 1. Quando arriva 01 ci dobbiamo predisporre, passiamo in q_{01} . A questo punto, se D si riabbassa (00), si torna in q_{00} . Se invece in q_{01} diamo ancora 01 rimaniamo in q_{01} . Notiamo che questa è una macchina asincrona, infatti nell'automa troviamo degli anelli, che indicano la permanenza in uno stato. Se poi l'ingresso diventa 11 (A si alza) allora dobbiamo effettivamente cambiare uscita, che diventa 1, dunque passiamo nello stato q_{11} . Osserviamo che nello stato q_{01} non possiamo mettere l'ingresso 10, ci arriviamo con lo stato 10 e se mettessimo 10 avremmo un'alea. Se da q_{11} mettiamo ancora 11 rimaniamo in q_{11} . Se mettiamo 10 (quindi si abbassa il dato quando l'abilitazione è ancora alta) la macchina non è sensibile, quindi rimane in 11: la macchina si predisponde solo nel caso in cui A è bassa e il dato è l'opposto di quello che abbiamo. Se abbassiamo anche A (00) passiamo nello stato q_{10} , ci predisponiamo a presentare 0. Con un altro ingresso 00 si rimane in q_{10} . Se si alza D, 01, si passa in q_{11} , se invece si alza A (10) si torna in q_{00} . In q_{10} , allora, non è possibile avere l'ingresso 11, altrimenti si avrebbe un'alea.

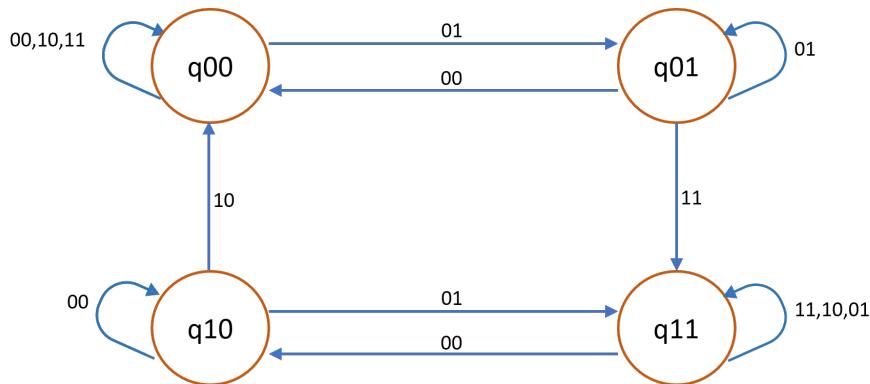


Figura 3.20: Automa del flip flop ET.

Inoltre, possiamo notare che gli stati codificati in questo modo non danno problemi di alee, i possibili passaggi infatti sono:

- da $00 \rightarrow 01$, cambia un bit;
- da $01 \rightarrow 00$ o da $01 \rightarrow 11$, cambia un bit;
- da $11 \rightarrow 10$, cambia un bit;
- da $10 \rightarrow 00$ o da $10 \rightarrow 11$, cambia un bit.

Il problema della codifica si pone perchè si tratta di una macchina asincrona, se fosse stata sincrona invece avremmo fatto finire le alee e solo dopo avremmo dato il segnale di abilitazione. La codifica non è un problema perchè con l'abilitazione avremmo evitato di vedere i fenomeni transitori.

Capitolo 4

Contatori

Per progettare un contatore, con un certo modulo, abbiamo essenzialmente tre possibilità:

1. **Automa:** il contatore è una macchina sequenziale, dunque la prima possibilità è sicuramente effettuare il progetto tramite un automa, in particolare di Moore. L'inconveniente di tale strategia è che l'automa ha un numero di stati pari al numero del conteggio del modulo che vogliamo realizzare. Il vantaggio è che abbiamo sicuramente una macchina minima, perché per contare fino ad 8 servono 8 stati, ma se vogliamo fare un contatore più grande, che conti fino a 16, a 32 e così via, il progetto risulta infattibile. Inoltre, definendo la macchina in questo modo, stiamo affidando l'intera sintesi al compilatore e non è detto sia in grado di sintetizzarla facilmente.
2. **Composizione di più contatori:** possiamo realizzare il contatore mediante la composizione di contatori più piccoli. In tal modo, passiamo da un approccio metodologico a uno progettuale, individuiamo i moduli e le connessioni tra di essi così da ottenere una macchina più facilmente sintetizzabile. Utilizziamo allora un contatore per contare le potenze di 2^0 , un altro per le potenze di $2^1, 2^2$, etc. Il progetto che facciamo quindi non è più comportamentale, come l'automa, ma diventa strutturale, stiamo definendo quali sono gli oggetti di cui è composta l'architettura e come questi vengono collegati tra loro. Il vantaggio di questa strategia è ottenere modularità e simmetria.
3. **Contatore ad incremento:** Supponiamo di voler incrementare il contatore non più di 1 ma di un valore arbitrario. Ad esempio, il Program Counter (PC) del processore si incrementa di 1 quando eseguiamo istruzioni in sequenza ma, nel momento in cui c'è un salto, deve incrementarsi della quantità del salto. Sarebbe difficile progettarlo con le prime due strategie viste, tramite la composizione di contatori infatti possiamo decidere solo il punto di partenza del conteggio ma non l'incremento, mediante l'automa invece dovremmo differenziare un numero troppo elevato di stati. Lo realizziamo inserendo, all'interno dell'architettura, un adder.

4.1 Progetto del contatore mediante automa

Dato che l'automa diventa sempre più complesso all'aumentare del modulo del contatore, analizziamo questo approccio nel caso di un contatore modulo 4. In questo caso, infatti, l'automa è semplice: ogni volta che arriva l'ingresso di conteggio si sposta nello stato successivo e, al quarto ingresso, torna nello stato iniziale, come mostra la figura 4.1.

Essendo una macchina a 4 stati, per la sintesi saranno necessari solo 2 flip-flop. Tuttavia, in questo progetto è presente un **problema di temporizzazione** in quanto se mettiamo un ingresso di conteggio che dura troppo, ad esempio nello stato q_0 , l'automa non passa nello stato successivo,

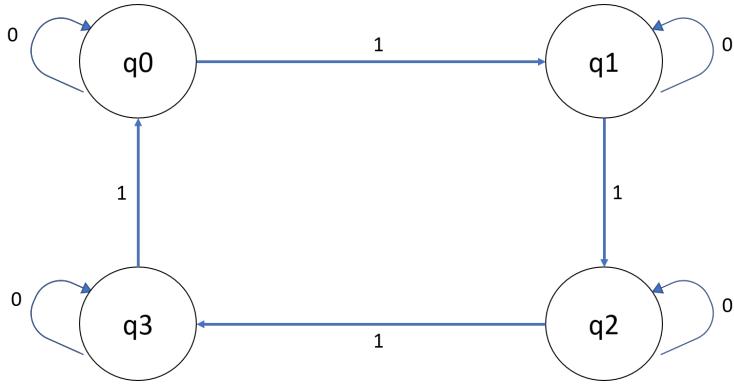


Figura 4.1: Automa del contatore modulo 4

q1, ma arriva fino a q2, dunque *la macchina conta due volte per un solo ingresso di conteggio*. Per risolvere tale problema, quando facciamo la sintesi della macchina **dobbiamo inserire sulle linee di retroazione dei flip flop edge-triggered**, che lavorano al variare del fronte e non sul livello. In questo modo ci assicuriamo che la commutazione avvenga solo una volta, per **ciascun ingresso di conteggio**. Se invece usassimo dei latch, il problema non sarebbe risolto in quanto il loro comportamento dipende dalla durata del segnale in ingresso.

4.2 Progetto tramite composizione di più contatori

Vogliamo progettare il contatore nella seconda tra le possibilità viste. **Consideriamo un contatore modulo 8**, le 8 configurazioni che può assumere sono mostrate in tabella 4.1.

Tabella 4.1: Configurazioni contatore modulo 8

X1	X2	X3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

È una macchina notevole in cui la cifra unità (X3) varia sempre, il bit centrale varia ogni 2 e il bit più significativo ogni 4, perché siamo in un sistema binario. Possiamo allora prendere **le macchine che contano rispettivamente 2^0 , 2^1 e 2^2** e collegarle tra di loro. Possiamo realizzare i singoli oggetti con dei flip flop a commutazione, che valgono 0 o 1, dunque useremo dei **flip flop T o JK**. Il problema è capire come gestire il tempo. Il contatore delle unità, 2^0 , commuta sempre, quindi il clock entra nel primo contatore e **dev'essere attivo sul fronte di discesa**, in modo che funzioni quando l'impulso non c'è più. Ovviamente *non possiamo farli latch perché altrimenti dipenderebbero dalla durata del segnale*. Stiamo mettendo i contatori in cascata, secondo il **modello seriale**, come mostra la figura 4.2.

Secondo questo schema, quando l'unità scende, il secondo commuta (perchè lavorano sul fronte di discesa) e quando la sua uscita scende, commuta il terzo. *Se fossero invece attivi sul fronte di salita*, quando il primo conta anche il secondo commuterebbe in quanto sarebbe sensibile al fronte

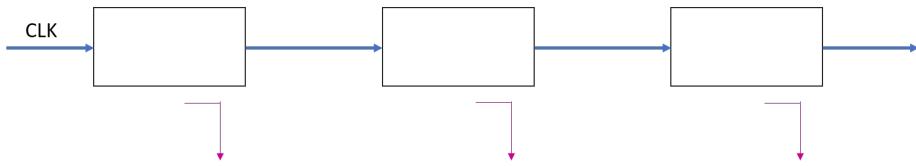


Figura 4.2: Modello seriale

di salita dell'uscita del precedente, dunque *commuterebbero tutti insieme, sullo stesso fronte, pur avendo avuto un solo segnale di conteggio.* Utilizzando tutti flip flop attivi sul fronte di discesa funziona, è modulare, è simmetrico e i fili non si incrociano. Tuttavia, si verifica un **problema di propagazione** in quanto il clock viene dato solo al primo componente, poi si propaga al secondo e dopo ancora al terzo. Consideriamo di nuovo la tabella 4.1, supponiamo di trovarci al conteggio 011 (3 in numero naturale), al prossimo impulso ci aspettiamo di contare 100 (ovvero 4 in numero naturale), a causa della propagazione però ci sarà un momento in cui i primi due contatori commutano, passando da 0 a 1, mentre l'ultimo deve ancora commutare, perché è in attesa dell'ingresso. Invece di passare da 011 a 100, si passerà da 011 a 000 e solo dopo a 100, dunque da 3 si conterà 0 per poi contare 4. Inoltre, **maggior è il numero di cifre, più forte è l'effetto della propagazione**, in quanto la catena di propagazione si allunga, quindi il sistema funziona in modo peggiore.

Dobbiamo inserire il reset per poter azzerare il contatore (figura 4.3), possiamo farlo sincrono o asincrono. Se lo facciamo **sincrono**, il primo colpo di clock viene usato per resettare; se lo facciamo **asincrono**, quando non c'è il reset al primo colpo di clock il primo contatore inizia a contare, però dobbiamo fare attenzione a gestire bene tutta la temporizzazione.

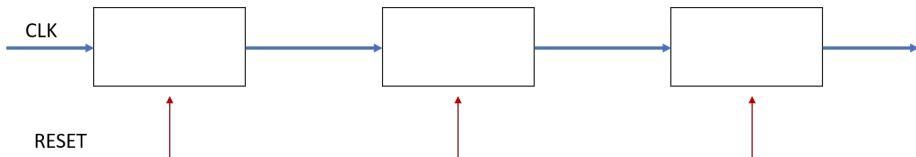


Figura 4.3: Modello seriale con reset

Possiamo realizzare il contatore anche in un altro modo, secondo il **modello parallelo**. In questo caso, il primo contatore commuta ogni volta che arriva il clock, il secondo invece quando arriva il clock e il primo è alto, il terzo quando arriva il clock e sono alti sia il primo sia il secondo. Dobbiamo però fare attenzione, **devono essere attivi sul fronte di discesa** perché, se fossero attivi sul fronte di salita, all'arrivo del primo clock commuterebbe il primo ma anche il secondo, perché troverebbe il clock ancora alto. Il problema è causato dalla combinazione del conteggio precedente con il clock che l'ha generato: **il modo per evitare che qualcosa si ricombini con ciò che l'ha generato è generarlo quando quest'ultimo non c'è più, dunque sul fronte di discesa**. Per realizzare questo schema mettiamo l'uscita del primo componente in **AND** con il **clock**, in questo modo, se sono attivi sul fronte di discesa, quando il clock si abbassa l'uscita del primo si alza (commuta) e il secondo commuta solo al colpo di clock successivo, quando trova alto sia il clock sia l'uscita del primo. **All'ingresso del terzo componente invece mettiamo in AND le uscite dei primi due e il segnale di clock**, in questo modo commuta solo quando le due uscite sono alte e arriva il clock, come mostra la figura 4.4. In questo modo il clock viene dato in parallelo ai tre componenti, quindi **non c'è il problema di propagazione**.

Vediamo le differenze, dal punto di vista del modello, tra gli schemi visti. **Nel primo modello**

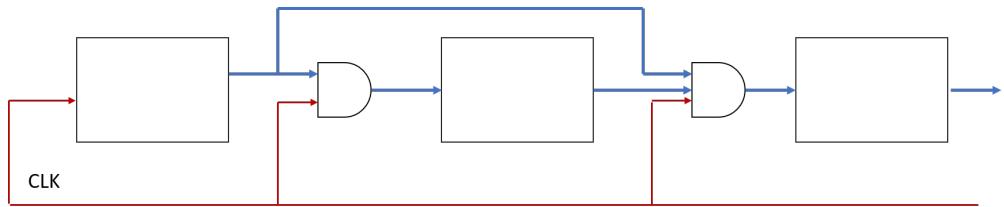


Figura 4.4: Modello parallelo

(seriale) abbiamo 3 macchine separate che lavorano in cascata: *il modello dell'automa si trova all'interno di ciascuna di esse*. Il comportamento è **asincrono** perché quando arriva la variazione le macchine raggiungono uno stato stabile. Il modello dell'automa classico (figura 4.5) invece prevede di dare l'abilitazione complessiva a tutti i banchi di registri, l'oggetto diventa attivo quando arriva il clock, per cui tutta la batteria di registri commuta contemporaneamente. **È ciò che accade nel secondo modello** visto (parallelo) in cui i 3 registri rappresentano lo stato e in relazione allo stato precedente viene calcolato il nuovo stato, quando arriva il clock.

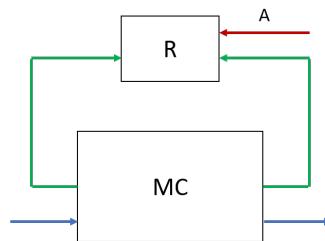


Figura 4.5: Modello di automa

Se volessimo realizzare un contatore modulo 6 a partire da uno modulo 8, vuol dire che non deve contare fino a 7 ma fino a 5, ovvero fino a quando l'uscita del primo contatore è 1, del secondo è 0 e del terzo è 1. Facciamo entrare 101 in una AND, in modo tale che quando questi tre valori sono veri abbiamo contato fino a 5 e dobbiamo resettare, lo schema è mostrato in figura 4.6. In realtà dobbiamo resettare quando 5 torna basso, dunque **anche il reset dev'essere attivo sul fronte di discesa**.

Questo modello non vale solo per potenze di due ma in generale, il primo conta i secondi, il secondo i minuti e il terzo l'ora. Bisogna però riuscire a contare fino a 60, nel nostro caso contiamo fino a 64 e poi aggiustiamo in modo da dare il reset dopo 59.

Se prendiamo le uscite dei singoli contatori, S1, S2 e S3, in realtà stiamo dividendo il clock: il primo è un contatore modulo 2, dunque divide la frequenza a metà (S1), il primo e il secondo formano un contatore modulo 4, dunque dividono la frequenza per 4 (S2), i 3 insieme formano un contatore modulo 8, dividono la frequenza per 8 (S3). Se prendiamo le uscite dei singoli blocchi, possiamo dividere la frequenza **sempre rispetto allo stesso clock perché sono isocroni**, il riferimento temporale è lo stesso. Possiamo usare un contatore non solo per contare ma anche con un obiettivo differente, per dividere la frequenza.

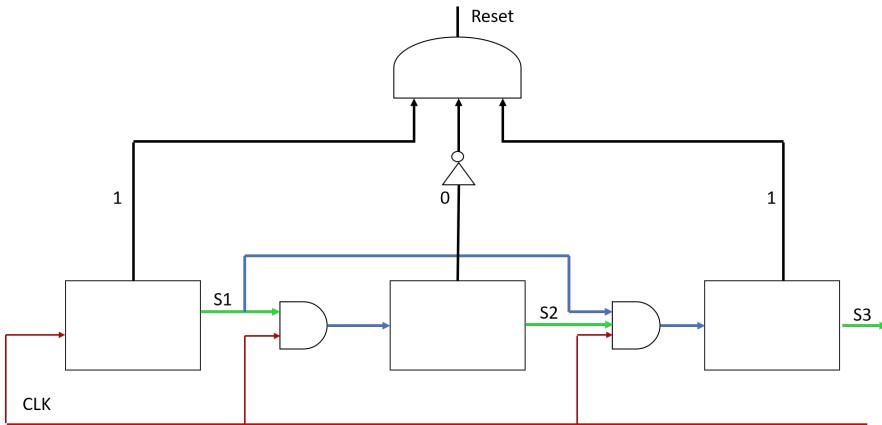


Figura 4.6: Contatore modulo 6 a partire da un contatore modulo 8

4.2.1 Implementazione VHDL

Di seguito riportiamo una proposta di implementazione in VHDL dei contatori descritti. In figura 4.7 è presente l'implementazione del flip flop T che, come visto, costituisce la macchina base del contatore.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity ffT is
5    port(
6      clk: in std_logic;
7      reset: in std_logic;
8      Y: out std_logic
9    );
10 end ffT;
11
12 architecture rtl of ffT is
13
14   signal TY: std logic;
15
16 begin
17   ff: process(clk,reset)
18   begin
19     if(reset='1') then
20       TY<='0';
21     elsif(clk'event AND clk='0') then
22       TY<=not TY;
23     end if;
24   end process;
25
26   Y<=TY;
27
28 end rtl;

```

Figura 4.7: Implementazione VHDL del FF T

In figura 4.8 troviamo l'implementazione del contatore vero e proprio, modulo 16, realizzato mediante la composizione di più contatori base, posti in cascata l'uno dietro l'altro, secondo lo schema seriale. La descrizione in questo caso è di tipo strutturale, il componente base utilizzato è il flip flop di tipo T.

```

5  entity cont_16_s is
6    port(
7      C: in std_logic;
8      R: in std_logic;
9      cont: out std_logic_vector(0 to 3)
10 );
11 end cont_16_s;
12
13 architecture structural of cont_16_s is
14
15 component ffT is
16   port(
17     clk: in std_logic;
18     reset: in std_logic;
19     Y: out std_logic
20   );
21 end component;
22
23 signal S1: std_logic;
24 signal S2: std_logic;
25 signal S3: std_logic;
26 signal S4: std_logic;
27
28 begin
29   ff1: ffT
30   port map(
31     C,
32     R,
33     S1
34   );
35
36   ff2: ffT
37   port map(
38     S1,
39     R,
40     S2
41   );
42
43   ff3: ffT
44   port map(
45     S2,
46     R,
47     S3
48   );
49
50   ff4: ffT
51   port map(
52     S3,
53     R,
54     S4
55   );
56
57   cont<=S4 & S3 & S2 & S1;
58
59 end structural;

```

Figura 4.8: Implementazione VHDL del contatore seriale

In figura 4.9 troviamo invece l'implementazione dello schema parallelo. In questa implementazione ogni flip flop riceve il clock, eccetto il primo, ciascuno lo riceve in AND con le uscite dei contatori precedenti. Abbiamo infatti inserito, oltre ai flip flop, le porte AND che realizzassero tale meccanismo.

```

4  entity cont_16_p is
5    port(
6      C: in std_logic;
7      R: in std_logic;
8      cont: out std_logic_vector(0 to 3)
9    );
10 end cont_16_p;
11
12 architecture structural of cont_16_p is
13
14   component ffT is
15     port(
16       clk: in std_logic;
17       reset: in std_logic;
18       Y: out std_logic
19     );
20   end component;
21
22   signal S1: std_logic;
23   signal S2: std_logic;
24   signal S3: std_logic;
25   signal S4: std_logic;
26   signal Y1: std_logic;
27   signal Y2: std_logic;
28   signal Y3: std_logic;
29
30 begin
31   ff1: ffT
32     port map(
33       C,
34       R,
35       S1
36     );
37
38   Y1<=S1 and C;
39
40   ff2: ffT
41     port map(
42       Y1,
43       R,
44       S2
45     );
46
47   Y2<=S1 and S2 and C;
48
49   ff3: ffT
50     port map(
51       Y2,
52       R,
53       S3
54     );
55
56   Y3<= S1 and S2 and S3 and C;
57
58   ff4: ffT
59     port map(
60       Y3,
61       R,
62       S4
63     );
64
65   cont<=S4 & S3 & S2 & S1;
66
67
68 end structural;

```

Figura 4.9: Implementazione VHDL del contatore parallelo

Infine, in figura 4.10, troviamo la simulazione effettuata mediante gtkwave dei due contatori, il cui esito è equivalente in quanto non abbiamo aggiunto ritardi nel componente base. Notiamo che il comportamento è quello desiderato: ad ogni colpo di clock l'uscita si incrementa e quando arriva a 15 poi ricomincia il conteggio da 0.

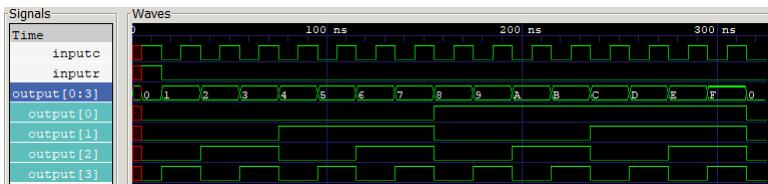


Figura 4.10: Simulazione del contatore serie/parallelo

4.3 Progetto del contatore ad incremento variabile

Per realizzare questo particolare contatore possiamo utilizzare un **adder**, secondo lo schema mostrato in figura 4.11(a).

Il valore rappresenta il conteggio a cui è arrivato il contatore, questo entra in ingresso all'adder che somma 1 e rimanda il valore incrementato. Il vantaggio di questo schema è che, se vogliamo fare un contatore classico, nell'adder mettiamo come secondo ingresso 1, se invece vogliamo effettuare un incremento particolare, in alcuni casi, basta mettere all'ingresso dell'adder un oggetto che ci permetta di effettuare una scelta. Serve un oggetto che consenta di scegliere se fare un incremento classico o di un valore particolare, è ovvio che tale oggetto è un **multiplexer**. Se nel multiplexer mettiamo un valore arbitrario e un 1, a seconda dell'ingresso di selezione l'adder sommerà il valore o farà un semplice incremento di 1. Lo schema è presente in figura 4.11(b).

Notiamo però che nel disegno **manca il tempo**, senza il quale non sappiamo determinare in che modo finisce il conteggio. Dobbiamo allora inserire un'abilitazione e **attivare la macchina**

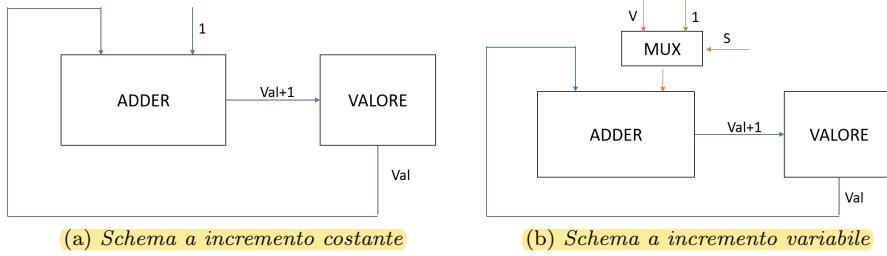


Figura 4.11: Schema contatore a incremento

sulla variazione: in questo modo siamo sicuri che prenda il valore e lavori una sola volta. Se non procedessimo in questo modo, dunque se la macchina non lavorasse sulla variazione, rischieremmo di ripetere l'incremento più volte, per un solo colpo di abilitazione. Se consideriamo di usare un clock come abilitazione, usiamo entrambi i periodi, entrambi i fronti di variazione: sul fronte di salita mettiamo il valore mentre sul fronte di discesa preleviamo il risultato. La durata di tale clock dev'essere almeno pari a:

$$T_{MUX} + T_{ADD}$$

Dove T_{MUX} è il tempo necessario al multiplexer per effettuare la selezione e T_{ADD} è il tempo necessario all'adder per fare la somma. Facendo così, realizziamo un circuito che funziona, non è in evoluzione libera e lavora su segnali di temporizzazione dati da noi (figura 4.12). Rispetto al modello precedente, realizzato per composizione di contatori, il numero di segnali di abilitazione che la macchina riceve complessivamente è maggiore. Per questo motivo, se ad esempio vogliamo realizzare un cronometro, che incrementa sempre della stessa quantità, conviene usare il modello visto al punto 2. Dobbiamo capire la complessità dell'oggetto e se aumenta conviene utilizzare un modello comportamentale che individua le parti ed i comportamenti tra di essi.

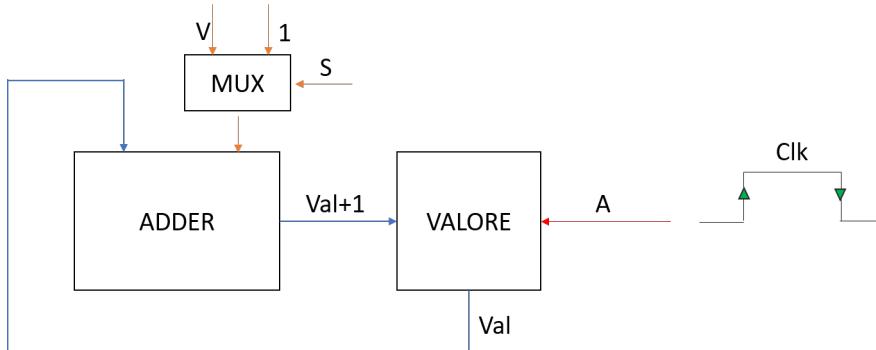


Figura 4.12: Schema completo contatore con incremento variabile

Una volta definito lo schema, dobbiamo capire come implementarlo in VHDL. Se definiamo l'addizione con l'operatore “+” o con le AND e le OR il risultato è diverso. Nel primo caso chiediamo al compilatore di sintetizzare un'operazione tra interi e di mettere qualcosa che faccia la somma. Se invece usiamo un addizionatore definito da noi, tramite le porte AND e OR, allora il blocco addizionatore non è un blocco comportamentale ma un blocco definito a livello più basso. Fintanto che il blocco che vogliamo realizzare non è complesso, come una somma tra interi, il compilatore lo riconosce e riesce fare la sintesi correttamente, ma se il blocco diventa più articolato, il compilatore dev'essere più intelligente, altrimenti rischiamo di riuscire a simulare ma non a sintetizzare. Se quindi vogliamo utilizzare un addizionatore più sofisticato, dobbiamo progettarlo

e non possiamo realizzare la macchina come un unico process, perchè in tal modo non chiediamo al compilatore di sezionare il modello nelle parti che controlliamo ma affidiamo al compilatore la realizzazione di tutta la macchina.

4.3.1 Indirizzamento relativo

Il contatore ad incremento variabile viene utilizzato anche per realizzare l'indirizzamento relativo o relativo e assoluto (con le istruzioni di jump e jump relativo). In quest'ultimo, dobbiamo poter sommare, all'interno del contatore, valori differenti provenienti da sorgenti differenti. Se abbiamo solo due alternative usiamo un multiplexer, se dovessero essere in numero maggiore potremmo usarne anche di più. Consideriamo l'istruzione in figura:



Figura 4.13: Istruzione di salto relativo

Supponiamo che il valore sia pari a 200, ciò vuol dire che il processore deve andare 200 istruzioni più avanti o eventualmente più indietro. La rete di controllo, quando vede l'istruzione di salto relativo, abilita il valore S in modo tale che il multiplexer non selezioni 1, ovvero l'incremento semplice, ma un particolare valore. Va sottolineato che non è importante l'esatto valore, potrebbe essere 200, 300 o altro ancora ma la rete di controllo deve fare sempre la stessa cosa, dunque solo la prima parte dell'istruzione serve per pilotare la rete di controllo, ovvero quella che identifica il salto e in base a cui pilotiamo il multiplexer. Se invece realizziamo la macchina con l'automa non risulta più vero quanto detto, il particolare valore diventa importante in quanto se vale 200 dobbiamo andare in uno stato che conti più di 200, se vale 300 dobbiamo andare in uno stato che valga più di 300: dobbiamo poter discriminare tutti gli ingressi.

4.3.2 Cronometro con intertempo

Supponiamo di voler costruire un cronometro con l'intertempo. L'intertempo effettua una fotografia del cronometro e la conserva. Oltre al contatore serve quindi un registro: prendere l'intertempo vuol dire chiedere al registro di memorizzare il valore assunto dal conteggio in quel determinato momento. Se invece vogliamo memorizzare più di un valore non basta un registro, serve un banco di registri perchè ogni volta dobbiamo memorizzare il valore in una locazione differente. Non conviene utilizzare una memoria in quanto dovremmo indirizzarla, conviene usare un insieme di registri a scorrimento, in cui ognuno memorizza un bit in una posizione differente ed è più facile da realizzare in quanto non serve fornire un indirizzo. Dunque, nel caso in cui vogliamo memorizzare solo un intertempo, dobbiamo scegliere una singola cella di memoria, è banale, nel caso in cui invece prendiamo più intertempi allora servono molteplici celle e la scelta non è più banale. Nel nostro caso l'accesso serve sempre sequenziale, dunque non conviene utilizzare una ROM, ad accesso casuale, perchè pur utilizzandola in modo sequenziale, per costruirla dovremmo gestire la logica per l'indirizzamento.

4.3.3 Implementazione VHDL

In figura 4.14 troviamo l'implementazione VHDL del contatore realizzato con l'adder. La versione implementata è quella a incremento fisso, pari al valore 1. Al full adder abbiamo allora passato come secondo addendo la stringa pari a 0001. Va sottolineato che non sarebbe bastato inserire, invece del full adder, il semplice incremento alla riga 46, ovvero $ty <= ty + "00001"$, in quanto l'operatore + non è definito per il tipo std_logic. Se vogliamo esser certi che il sistema sia

sintetizzabile dobbiamo inserire un oggetto che faccia la somma, da noi stesso definito, ovvero il full adder.

```

5  entity cont_add is
6    port(
7      clk, reset, T: in std_logic;
8      y: out std_logic_vector(0 to 3)
9    );
10 end cont_add;
11
12 architecture rtl of cont_add is
13
14 component full_ad_4 is
15   port(
16     O1, O2: in std_logic_vector(3 downto 0);
17     rip_in: in std_logic;
18     rip_out: out std_logic;
19     RIS: out std_logic_vector(3 downto 0)
20   );
21 end component;
22
23
24 signal ty,tx: std_logic_vector(0 to 3);
25 signal op2: std_logic_vector(0 to 3);
26 signal cin, cout: std_logic;
27
28 begin
29   op2<="0001";
30   cinc='0';
31   fa: full_ad_4
32   port map(
33     ty,
34     op2,
35     cin,
36     cout,
37     tx
38   );
39
40 prc: process(clk,reset)
41 begin
42   if(reset='1') then
43     ty<=(others>'0');
44   elsif(clk'event AND clk='0') then
45     if(T='1') then
46       ty<=tx;
47     end if;
48   end process;
49   y<=ty;
50
51 end rtl;
52

```

Figura 4.14: Implementazione VHDL del contatore con adder

In figura 4.15 è presente l'implementazione del contatore ad incremento variabile. In questo caso il valore del secondo operando in ingresso al full adder varia, è pari a 1 nel caso in cui l'ingresso di load è 0, altrimenti, se load è 1, è pari a un valore arbitrario. L'assegnamento condizionato alla riga 32 rappresenta un multiplexer all'ingresso del full adder, la cui uscita è op2 e il cui ingresso di selezione è il segnale load.

```

5  entity cont_add is
6    port(
7      clk, reset, T,load: std_logic;
8      V: in std_logic_vector(0 to 3);
9      Y: out std_logic_vector(0 to 3)
10 );
11 end cont_add;
12
13 architecture rtl of cont_add is
14
15 component full_ad_4 is
16   port(
17     O1, O2: in std_logic_vector(3 downto 0);
18     rip_in: in std_logic;
19     rip_out: out std_logic;
20     RIS: out std_logic_vector(3 downto 0)
21   );
22 end component;
23
24
25 signal ty,tx: std_logic_vector(0 to 3);
26 signal op2: std_logic_vector(0 to 3);
27 signal cin, cout: std_logic;
28
29 begin
30   cin<='0';
31
32   --assegnamento condizionato dell'ingresso dell'adder, tramite mux.
33   op2<="0001" when load='0' else
34   V when load='1' else
35   "0000";
36
37   fa: full_ad_4
38   port map(
39     ty,
40     op2,
41     cin,
42     cout,
43     tx
44   );
45
46   prc: process(clk,reset)
47 begin
48   if(reset='1') then
49     ty<=(others>'0');
50   elsif(clk'event AND clk='0') then
51     if(T='1') then
52       ty<=tx;
53     elsif(load='1') then
54       ty<=op2;
55     end if;
56   end process;
57   y<=ty;
58
59 end rtl;

```

Figura 4.15: Implementazione VHDL del contatore ad incremento variabile

Infine, in figura 4.16 possiamo osservare la simulazione del contatore ad incremento variabile. Inizialmente il segnale di load (l) è basso, per i primi 6 colpi di clock (c), dunque il contatore conta normalmente fino a 6. Al successivo colpo di clock si alza il segnale di load e l'ingresso vin, che rappresenta il valore passato al full adder, vale 9: il contatore invece di contare 1 conta 9, l'uscita passa da 6 a 15. Al colpo di clock successivo il load è ancora alto mentre il segnale vin vale 2, il contatore effettua due conteggi, passando da 15 a 1.

4.4 Principio di funzionamento del contatore

Per descrivere un oggetto contatore in un linguaggio come il C useremmo un ciclo for. **Nel caso del C infatti il for è temporale**, ad ogni ciclo calcola ed incrementa la variabile di conteggio. **Il for del VHDL invece è spaziale**, genera tanti oggetti analoghi, messi uno dietro l'altro, ma allora *come riesce a contare?* Il processo è un oggetto ed in quanto tale rimane attivo, ogni volta che arriva il segnale di abilitazione calcola la variabile che ha. La “i” del ciclo for allora è

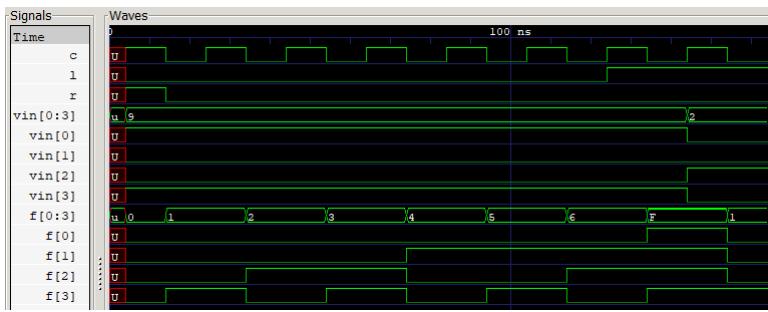


Figura 4.16: Simulazione del contatore ad incremento variabile

il segnale di alimentazione: quando arriva il segnale il processo si trova in uno stato, calcola e va in un altro stato. Ad ogni passo l'oggetto fa ciò che deve fare e si predisponde a lavorare per il passo successivo, dunque l'oggetto è in grado di contare perchè mantiene lo stato. Se il processo morisse non sarebbe possibile costruire un contatore ma tale processo non muore perchè ha uno stato, ha un registro e se arriva il segnale riesce a contare. Se inoltre dobbiamo contare con un certo modulo, per cui arrivati a un certo valore bisogna ripartire da 0, dobbiamo dare il *reset* e dobbiamo usare una condizione logica: serve una macchina che al verificarsi di una determinata condizione dia il *reset*. Stiamo costruendo un process sensibile al clock che quindi sarà presente nella sensitivity list. Ogni volta che arriva il clock prende la variabile interna che memorizza lo stato, y , e pone il conteggio pari a $y+1$, come mostra la figura 4.17.

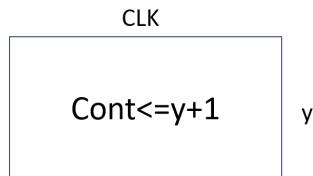


Figura 4.17: Process contatore

Facendo così però non stiamo realizzando un contatore modulo N ma un contatore che conta all'infinito. Per realizzare il **contatore modulo N**, ad esempio modulo 15, serve una particolare caratteristica circuitale. Se siamo arrivati a 15 il cont sarà pari a 1111, quando arriva il successivo incremento avremmo 10000 ma di questo numero **non carichiamo l'1** e otteniamo 0000, il valore di partenza. Per riuscire a realizzare il modulo dobbiamo scendere più in basso nel disegno dell'architettura e dire come caricare il contatore, in questo modo è come se si resettasse, bisogna capire come calcolare il nuovo stato.

4.5 Contatore tramite registro a scorrimento

Possiamo realizzare il conteggio anche con un **registro a scorrimento**, consideriamo ad esempio un registro con 8 locazioni e inseriamo un 1 in modo circolare, ad ogni colpo di clock l'1 si sposta di locazione in locazione e all'ottavo colpo ritorna nella posizione di partenza: è un contatore modulo 8. Osserviamo però che per realizzare il contatore modulo 8, nel modo visto precedentemente, abbiamo utilizzato solo 3 registri, adesso ne servono 8. Questo perchè il contatore è finalizzato proprio all'attività di conteggio, è realizzato non con veri e propri registri ma con registri a commutazione, che effettivamente contano. *Lo shift register invece viene realizzato con 8 registri classici, ha in sé il concetto di memorizzazione di sequenze che, in modo periodico,*

realizza un contatore. Di per sè quindi lo shift register svolge un lavoro differente. Il vantaggio di questa implementazione alternativa però è che possiamo far lavorare 8 macchine insieme ma in modo alternato, possiamo anche realizzare facilmente il concetto di sfasamento.



Figura 4.18: Contatore realizzato con shift register

Capitolo 5

Registri a scorrimento

Un registro a scorrimento, nella sua definizione classica, è fatto da un insieme di registri. È una macchina utilizzata per molte periferiche, serve per gli algoritmi che fanno crittografia, poiché shiftano bit e fanno poi operazioni di matematica; è utile perché shiftare a destra o sinistra significa dividere e moltiplicare in binario.

Se volessimo realizzare questi registri ad un basso livello di astrazione, vorrebbe dire scegliere il singolo elemento bistabile che li compone, ovvero un flip-flop di tipo T, RS e così via, anche se possiamo osservare che quasi tutte le schede hanno già a bordo FF di tipo D, per cui non conviene realizzare altre tipologie.

5.1 Registro a scorrimento serie-serie

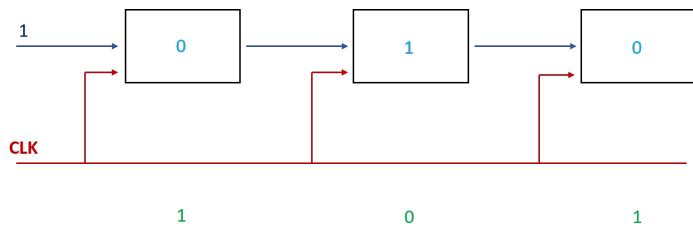


Figura 5.1: Esempio di registro a scorrimento con FF di tipo D.

Consideriamo il caso di tre registri, come in figura 5.1. Il segnale di abilitazione (**CLK**) è mandato in ingresso a tutti i sistemi in parallelo, mentre il dato si propaga da sinistra verso destra. Supponendo di partire da una situazione iniziale in cui i FF sono di tipo D e mantengono la sequenza 0-1-0, se in ingresso mettiamo 1, fintanto che l'abilitazione è 0, non succede niente; appena $CLK=1$, il primo flip-flop memorizza il valore. E' importante sottolineare che i FF debbano essere sensibili solo ad una variazione, ad esempio quella del fronte di discesa del clock: se il sistema fosse di tipo latch, non funzionerebbe sempre, perchè, dato l'ingresso pari ad 1, il primo flip-flop acquisirebbe 1 e dopo un certo intervallo temporale tutti gli altri flip-flop starebbero memorizzando 1, mentre noi vogliamo ottenere la configurazione che in figura è colorata in verde.

La variazione percepita potrebbe essere anche sul fronte di salita (funzionerebbe anche in questo caso), l'unica cosa che cambia è che, piuttosto che acquisire il valore dell'elemento di memoria precedente alla discesa del fronte, si effettua la stessa operazione alla salita del segnale di abilitazione. Realizzandolo nel primo modo (sul fronte di discesa) è possibile far sì che, se ci fosse una macchina a valle, questa potrebbe vedere l'evoluzione, il cambio degli stati quando l'abilitazione non c'è più. A questo punto, osserviamo l'evoluzione temporale del sistema, mediante il diagramma temporale

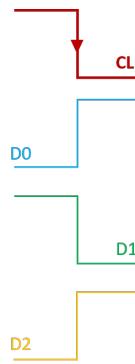


Figura 5.2: Diagramma temporale.

della figura 5.2: quando abbiamo la variazione del fronte di discesa del CLK, il primo FF commuta da 0 ad 1, il secondo da 1 a 0 ed il terzo da 0 ad 1.

Questo registro rientra nella categoria di **registro a scorrimento serie-serie** (SISO - serial input/serial output): è presente un solo ingresso, per cui i bit vengono caricati uno alla volta. Alla prima transizione (fronte di discesa o di salita a seconda di come lo definiamo) il dato presente in ingresso viene trasferito all'uscita del primo elemento di memoria, quindi del primo FF; all'impulso successivo il dato passa in uscita al secondo flip-flop e così via. Essendo anche a serial output, l'uscita è data da un singolo bit. ***Il comportamento complessivo di questo sistema è la propagazione del segnale di ingresso attraverso i vari flip flop al verificarsi di ogni fronte di clock.***

5.1.1 Implementazione VHDL del SISO

```

4  entity reg_ss is
5    port
6      (
7        x, clock, reset: in std_logic;
8        y: out std_logic
9      );
10 end reg_ss;
11
12 architecture rtl of reg_ss is
13
14   signal temp1, temp2, temp3, temp4: std_logic;
15
16 begin
17
18   RSS: process(clock, reset)
19   begin
20     if(reset='1') then
21       temp1<='0';
22       temp2<='0';
23       temp3<='0';
24       temp4<='0';
25     elsif(clock'event and clock='0') then
26       temp1<=x;
27       temp2<=temp1;
28       temp3<=temp2;
29       temp4<=temp3;
30     end if;
31   end process;
32   y<=temp4;
33 end rtl;

```

Figura 5.3: Implementazione registro SISO in VHDL.

Definiamo un'entità *reg_ss* caratterizzata da tre segnali di ingresso, il bit di dato x, il segnale di abilitazione della cascata clock ed il segnale di reset che è necessario affinché la macchina parta sempre da uno stato noto. Descriviamo poi l'architettura mediante il process, sensibile sia al segnale di clock che al reset. In questo caso abbiamo adottato un **reset sincrono**, cioè il primo colpo di clock (in questo caso il fronte di discesa) è utilizzato per resettare la macchina. Infatti, se c'è il segnale di reset, le 4 variabili temp (era possibile utilizzare anche una variabile temp di tipo std_logic_vector, come vedremo in altri esempi) vengono messe a 0, altrimenti se il reset non c'è e si verifica il fronte di discesa del segnale di clock, x viene shiftato e mandato in uscita al primo FF (ecco perché temp<=x), temp va in temp1 e così via. Alla fine del process si collega il segnale temp4 all'uscita. La struttura realizzata è alla fine un registro a scorrimento, simile a quello della figura 5.1, soltanto che avremo 4 FF, piuttosto che tre.

Realizzando un opportuno test_bench, possiamo simulare il sistema, il cui risultato è mostrato

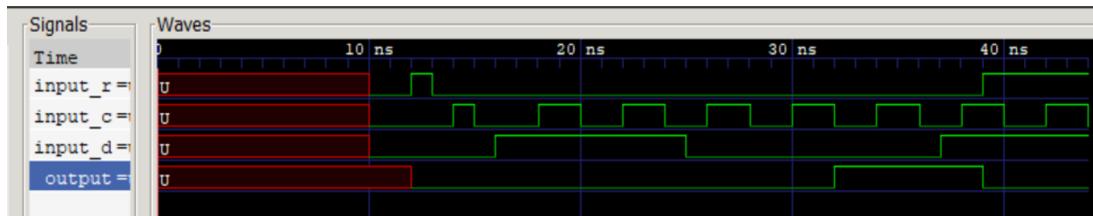


Figura 5.4: Simulazione registro serie-serie con *gtkwave*.

in figura 5.4: come abbiamo anticipato, all'istante di tempo $t=12$ ns arriva il segnale di reset (input_r), catturato al primo fronte di discesa del clock (input_c) nell'istante $t=15$ ns. In $t=16$ ns il dato (input_d) diventa alto, ma il primo shift avviene nell'istante temporale $t=20$ ns, ovvero al secondo fronte di discesa del clock. Dopo 4 colpi di clock, al quarto fronte di discesa in $t=32$ ns il registro presenta uscita alta, ovvero il dato alto si è propagato nei 4 flip-flop.

E' interessante osservare lo schematic fornito dall'ambiente di sviluppo ISE di Xilinx (figura

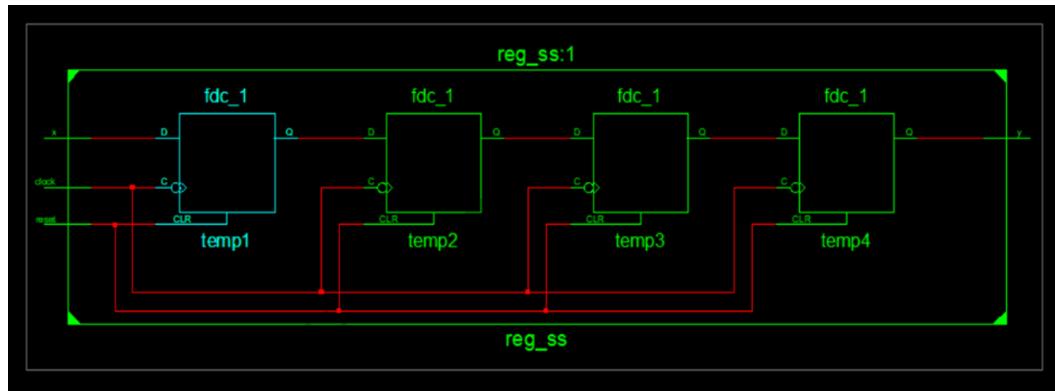


Figura 5.5: Schematic dello SR serie serie su ISE.

5.5): questo conferma esattamente quello che abbiamo detto all'inizio, cioè, sebbene sia possibile scegliere il singolo FF che compone il registro (in tal caso sarebbe necessario un approccio di tipo structural), **è inutile scegliere un bistabile che non sia di tipo D, perché gran parte delle schede porta proprio questi ultimi a bordo**, quindi quale che sia il flip-flop che sceglieremo sempre con il D finirà per essere realizzato. Infatti, mediante il tool di schematic viewer di ISE, è possibile notare come sia presente una cascata di 4 FF di tipo D.

5.2 Registro a scorrimento parallelo-parallelo

Lo Shift Register parallelo parallelo è quello tipicamente utilizzato ed è costituito da una catena di flip-flop di tipo D, sincronizzati con lo stesso segnale di clock e prendono in ingresso lo stesso segnale di reset. Il dato è suddiviso in più linee che entrano in parallelo ai FF, così come le uscite.

5.2.1 Implementazione in VHDL dell'SR PIPO

Nella figura 5.6 è presente una possibile implementazione VHDL del registro parallel input - parallel output. L'entità è pressoché la stessa, tranne per il fatto che questa volta x ed y sono rispettivamente vettori di ingresso e di uscita di 4 bit ciascuno, non più singole variabili, proprio perchè le linee di ingresso entrano in parallelo e quelle di uscita vengono prelevate in parallelo.

```

5  entity reg_pp is
6    port
7      (
8        x: in std_logic_vector(0 to 3);
9        clock, reset: in std_logic;
10       );
11      );
12    end reg_pp;
13
14  architecture rtl of reg_pp is
15  begin
16
17    --registro parallelo - parallelo di stringhe di 4 bit
18    RPP4: process(clock, reset)
19    begin
20      if(reset='1') then y<="0000";
21      elsif(clock'event and clock='1') then
22        | y<=x;
23      end if;
24    end process;
25
26
27  end rtl;

```

Figura 5.6: Implementazione registro PIPO in VHDL.

Ancora una volta l'architettura è descritta mediante un process, sensibile al clock ed al reset. Se il segnale di reset è alto, allora l'uscita viene posta a 0, altrimenti se non è 0 e si verifica il fronte di salita del segnale di clock ad y viene assegnato tutto il vettore x causando così lo shift dei 4 bit del vettore x attraverso i 4 flip-flop del registro.

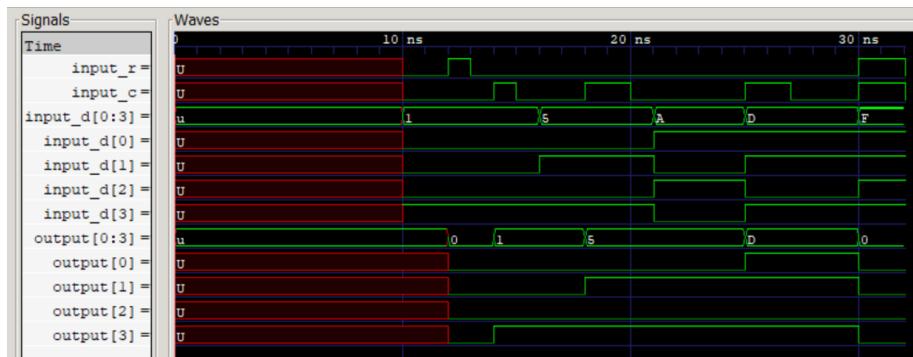


Figura 5.7: Simulazione del registro PIPO su gtkwave.

Guardiamo la simulazione fatta su gtkwave presenta in figura 5.7. All'istante $t=14$ ns, $\text{input_d}[3]$ è alto, si verifica il fronte di salita del segnale di clock (input_r), e quindi $\text{output}[3]$ si alza. All'istante $t=21$ ns, $\text{input_d}[0]$ si alza, ma questa variazione non viene percepita, perché il clock è ancora basso, soltanto all'istante $t=25$ ns, si presenta il fronte di salita del clock e $\text{output}[0]$ diventa alto.

5.3 Registro a scorrimento serie-serie circolare

Questo sistema può funzionare anche in **modalità circolare**, collegando opportunamente l'uscita del FF più a destra con l'ingresso del primo FF. Se volessimo farlo funzionare in **modalità circolare e non** dovremmo implementare il sistema mostrato in figura 5.8, che presenta l'aggiunta del multiplexer, il quale prende in ingresso il dato e l'uscita della cascata di FF, oltre che l'ingresso di selezione. Sulla base del valore di S, questo funziona come registro a scorrimento in modalità circolare o non.

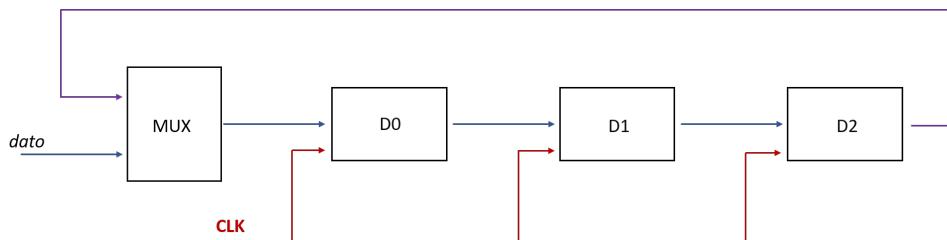


Figura 5.8: Registro a scorrimento in modalità circolare e non.

Questa macchina memorizza delle configurazioni: *può essere considerata uguale al contatore in cascata?* Assolutamente no, in quanto nel registro a scorrimento il segnale di CLK arriva in parallelo a tutti i registri, mentre nel contatore in cascata è mandato in ingresso solo al primo. Inoltre, i blocchi del contatore in cascata funzionano come contatori modulo 2 ed è un sistema che lavora sulle variazioni, mentre lo SR lavora sulle memorizzazioni.

5.3.1 Implementazione VHDL del registro SISO circolare

```

5  entity regss_circ is
6    port
7      ( x, sel, clock, reset: in std_logic;
8        y: out std_logic
9      );
10 end regss_circ;
11
12 architecture struct of regss_circ is
13
14 component mux_21 is
15   port
16     (
17       a,b,s: in std_logic;
18       y: out std_logic
19     );
20 end component;
21
22 signal temp: std_logic_vector(0 to 3);
23 signal xtemp: std_logic;

```

```

25 begin
26   -- se sel=0 prende l'uscita della catena, se sel=1 prende x
27   M21: mux_21 port map(temp(3), x, sel, xtemp);
28
29   RSSC: process(clock, reset)
30   begin
31     if(reset='1') then
32       temp(0 to 3)<="0000";
33     elsif(clock'event and clock='1') then
34       temp(0)<=xtemp; -- colleghiamo l'ingresso con l'uscita del multiplexer
35       temp(1 to 3)<=temp(0 to 2);
36     end if;
37   end process;
38
39   y<=temp(3);
40 end struct;

```

Figura 5.9: Implementazione registro PIPO in VHDL.

Le differenze implementative sostanziali rispetto al caso semplicemente SISO sono le seguenti:

- **sel:** il registro serie-serie circolare prende un segnale di ingresso che è il segnale di selezione del multiplexer; la scelta progettuale prevede che se sel vale 0, il multiplexer prende l'uscita della catena e la manda in ingresso al primo FF (è attiva la modalità circolare), altrimenti se sel vale 1 la modalità circolare è disattivata.
- **architecture:** istanziamo prima il componente MUX21 realizzando l'opportuno mapping. Infatti il mux prende in ingresso, oltre al segnale di selezione, x e temp(3), che sarebbe l'uscita della cascata, producendo in uscita xtemp. Il process che descrive il registro a scorrimento è sensibile al clock ed al reset, per cui se c'è il reset, il vettore temp è posto a 0, altrimenti al fronte di salita del clock in temp(0) viene inserito xtemp. In altre parole, in ingresso al primo FF D0 entra xtemp, che può essere temp(3), ovvero l'uscita della cascata, oppure x, l'ingresso, a seconda del valore di selezione del multiplexer. Poi vengono assegnati gli altri tre segnali (è una forma più compatta rispetto a quella della figura 5.3, ma il concetto è lo stesso).

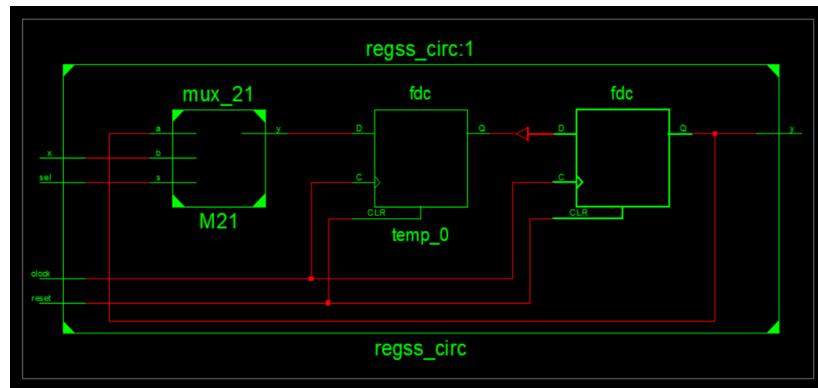


Figura 5.10: Schematic ISE del registro SISO circolare.

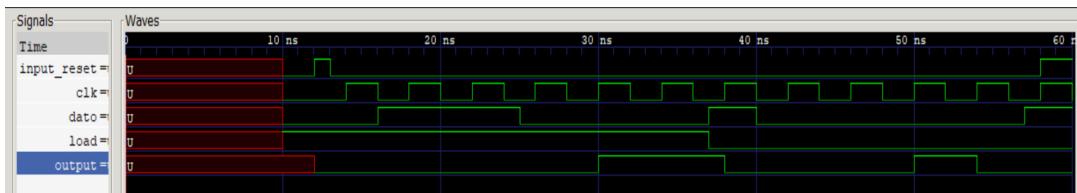


Figura 5.11: Simulazione registro SISO circolare con gtkwave.

Lo schematic che viene fuori dal tool di ISE è esattamente quello che ci aspettiamo (figura 5.10). Nella simulazione, vediamo lo stesso comportamento del SISO, soltanto che quando il load diventa 0, non viene prelevato il valore di ingresso al successivo fronte di salita, ma viene mandata in ingresso l'uscita della catena e dopo 4 fronti di salita l'uscita è nuovamente alta, poiché temp(3) era 1.

Osservazione: la scelta della tipologia di FF viene fatta a livello strutturale, non a livello comportamentale. Infatti, in quest'ultimo caso, non specifichiamo se i registri sono di tipo D o RS, perché già staremmo scendendo ad un livello data-flow, al contrario definiamo un vettore che “shifta” quando arriva un segnale di abilitazione, ovvero tre variabili che cambiano il loro valore quando si verifica un evento (variazione sul fronte di discesa del CLK), altrimenti mantengono il valore precedente.

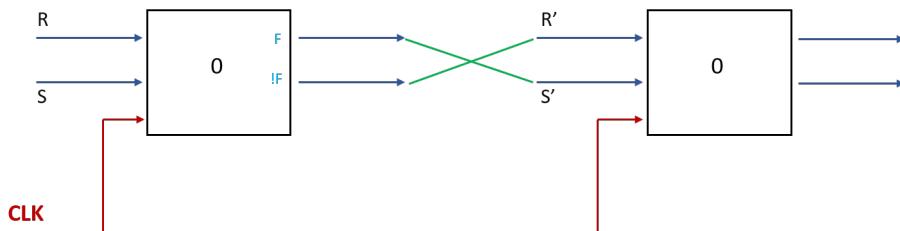


Figura 5.12: Realizzazione con flip-flop RS.

Supponiamo di voler realizzare *le due celle con dei FF RS* (figura 5.12): l'abilitazione deve essere uguale per tutti i flip-flop e quindi viene mandata in parallelo. Ipotizzando che nel primo FF ci sia 0, vuol dire che F è 0 ed F negato (\bar{F}) è 1; se vogliamo memorizzare uno 0 nel secondo FF, il reset R' deve essere 1 e il set S' deve essere 0. **Basta allora incrociare i fili d'uscita del primo FF con quelli di ingresso del secondo** (la parte contrassegnata in verde del disegno).

Se volessimo utilizzare lo Shift Register come memorizzatore di sequenze, l'implementazione sarebbe banale, ovvero si carica la sequenza mediante set e reset; tuttavia è un oggetto raramente usato da solo. Ricordando che esso realizza un **convertitore parallelo-serie** se è a caricamento parallelo ed uscita seriale o un **convertitore serie-parallelo** se è a caricamento seriale ed uscita parallela, il **registro a scorrimento** è una macchina fondamentale nella costruzione di una rete di calcolatori poiché consente di passare dal mondo del calcolatore, fatto di byte e quindi parallelo, al mondo delle telecomunicazioni, realizzato con canali e quindi seriale.

5.4 Applicazione degli Shift-Register

Abbiamo capito come progettare un singolo SR, tuttavia nel passaggio dalla progettazione del singolo componente a quella del sistema, bisogna capire come è possibile che un oggetto possa vivere insieme ad altri oggetti, quindi occorre affrontare un *problema trasmissivo*. Consideriamo un'entità E1 che riceve da A un dato in parallelo e lo trasmette in serie ed un'entità E2 che riceve il dato in serie e deve trasmetterlo in parallelo a B. Sono necessari due registri, uno ad ingresso parallelo ed uscita seriale e l'altro ad ingresso seriale ed uscita parallela. Collegando i due blocchi tra loro, dovrebbe teoricamente funzionare. In realtà in questo sistema c'è un *problema*



Figura 5.13: Comunicazione seriale.

di tempificazione: l'entità E1 trasmette in un tempo e l'entità E2 riceve in un altro tempo, per cui è necessario stabilire una convenzione affinché la comunicazione possa avvenire correttamente: E2 deve sapere che E1 sta per inviare un segnale.

Lo standard prevede, allora, di mantenere la linea in uno stato di riposo, successivamente viene tenuta attiva bassa per 2 quanti di tempo, così che E2 possa vederla e poi viene trasmessa una prima sequenza di bit che tipicamente sono di controllo. Completata questa fase, inizia la trasmissione (stiamo supponendo che il messaggio sia lungo un byte), al termine della quale viene inviato un bit di stop, sempre stabilito dalla convenzione, e dopo altri 2 quanti di tempo la linea trova attiva.



Figura 5.14: Esempio di comunicazione seriale con tempificazione.

C'è ancora dell'altro che stiamo trascurando, poiché, mentre la fase di trasmissione è alquanto semplice, prevedendo l'invio di una forma d'onda, *il vero problema sta in ricezione*. Pur essendo sicuri che i segnali di clock dei due sistemi viaggino alla stessa frequenza o a frequenze multiple (**isocronia**), non è detto che abbiano la stessa fase. *Uno sfasamento comporta un accumulo di ritardi che causano la perdita di campioni significativi*. Le soluzioni sono due:

- **utilizzare lo stesso clock**: possiamo inviare il clock all'interno del segnale, il che significa risolvere poi un problema di codifica, o inviare il segnale di clock su una linea differente e a questo punto bisognerebbe capire come gestire la presenza di più fili (per esempio il clock può avere uno skew, per cui il suo fronte si può allargare).
- **sovracampionamento della linea**: il ricevitore campiona ad una frequenza multipla della sorgente, prende più campioni così che, seppure ne dovesse sfasare uno, non accade nulla proprio per via del sovracampionamento. Una volta effettuato il sovracampionamento, sulla base di una decisione, se il valore è 1, lo SR memorizza il bit 1, altrimenti viceversa.

5.5 Ricevitore seriale

Implementiamo con una logica a blocchi la seconda soluzione, individuando la parte operativa e la parte di controllo: la sequenza di bit della figura 5.14 deve essere campionata da un apposito blocco, che prende in ingresso anche un segnale di *start_camp*, ed è inviata poi ad uno SR, che ha un suo segnale di abilitazione *ab_sr*. Il registro a scorrimento non riceve tutti i bit, ma soltanto gli 8 che rappresentano il messaggio trasmissivo. Per poter discernere il caso in cui i bit ricevuti appartengano alla sequenza di controllo, di stop o al messaggio è necessaria una macchina combinatoria notevole, il *demultiplexer*, che prende in ingresso anche un segnale di selezione *s_demux* e produce due uscite.

La macchina che decide quali bit vanno nel registro e quali no è la *rete di controllo* che ha il compito di generare i segnali di *start_camp*, *ab_sr* e *s_demux*. Per poter fare ciò, deve ovviamente ricevere delle informazioni in ingresso: il filo 1 di uscita del demultiplexer, che rappresenta i bit di controllo (non prende anche il filo 2 perché sono i bit del messaggio), il reset ed il clock, il quale deve essere mandato in parallelo anche al campionatore. ***La rete di controllo ed il campionatore devono lavorare su due fronti del clock differenti***, la prima deve visualizzare e quindi opera sul fronte di discesa, il secondo invece deve acquisire e lavora sul fronte di salita.

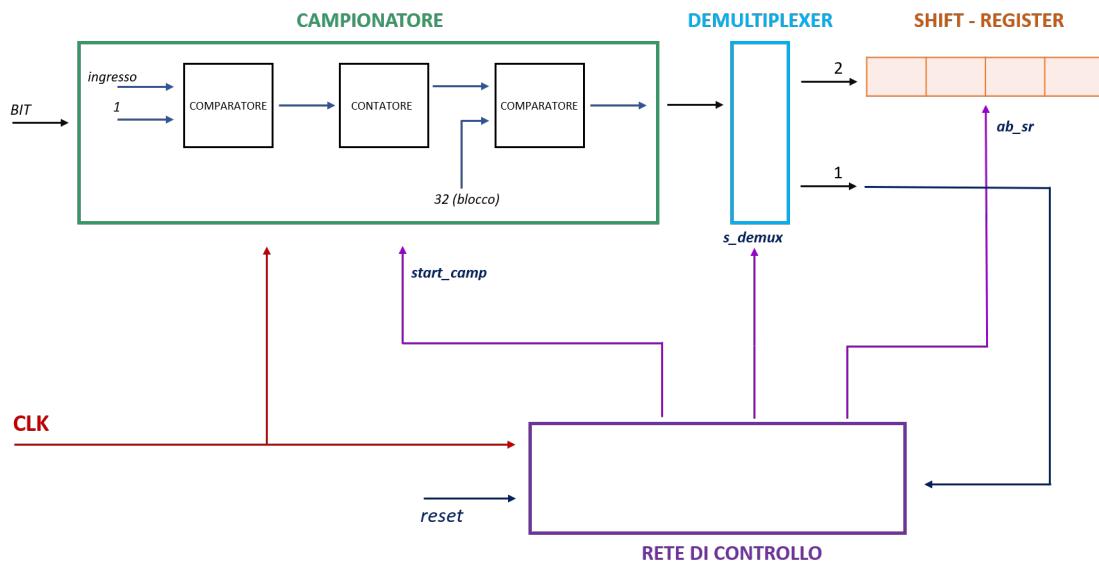


Figura 5.15: Schema a blocchi del ricevitore seriale.

Il blocco del campionatore è costituito da due comparatori ed un contatore: il primo comparatore serve per sapere se il bit in ingresso è 0 o 1, per cui si riduce ad una AND (ingresso AND 1=1 se l'ingresso è 1, altrimenti è 0); l'uscita del comparatore entra in ingresso ad un contatore che conta il numero di bit pari ad 1. Supponendo che il sovraccampionamento stabilisca di prendere 64 campioni in un periodo, il contatore potrà contare più di 32 (fattore di blocco) valori uguali ad 1 o meno di 32 bit alti. Se il conteggio è maggiore di 32, allora l'ingresso è un 1, altrimenti se il conteggio è minore di 32 l'ingresso è uno 0. Questo vuol dire che l'uscita del contatore deve entrare in ingresso ad un altro comparatore che effettuerà il confronto tra il conteggio e il fattore di blocco.

E' importante affrontare la realizzazione del progetto con un *approccio strutturale*, che comporta la divisione del problema in moduli; *suddividere i blocchi in unità operativa ed unità di controllo* significa dividere in moduli di due tipi, quelli dell'UO e quelli dell'UC. *Definire le relazioni tra moduli* significa dividere in sottomoduli l'unità operativa, mentre l'unità di controllo è tipicamente

un monolita. Effettuata la progettazione dei blocchi, occorre affrontare la tempificazione, in tal caso è d'aiuto il simulatore, anche se potrebbe accadere che il tempo simulato sia molto ottimistico rispetto a quello reale.

Nel progetto di questa periferica, sarebbe necessario inserire anche una *macchina che calcoli la parità*, per gestire il bit di controllo, ed un *contatore*, poiché, una volta che il sistema ha campionato, deve sapere ogni quanto vedere il risultato del campionamento. Inoltre, nel paragrafo precedente, in riferimento alla figura 5.13, abbiamo capito la necessità di un **protocollo di comunicazione**. Mentre un *protocollo parallelo* è più semplice, poiché, una volta che la sorgente ha trasmesso il dato, il ricevitore ha a disposizione l'informazione e può decidere quando prelevarla (a meno di ritardi di propagazione), essendo i fili in parallelo, un **protocollo seriale** è più complesso perché *il segnale non è statico, ma dinamico*. In altre parole, la sorgente inizia a trasmettere, il ricevitore non sa il momento preciso in cui parte la trasmissione, motivo per cui è necessario il protocollo; inoltre, se la sorgente invia un segnale ed il destinatario non legge il carattere, si rischia di sovrascrivere il valore ad una successiva trasmissione. Diventa fondamentale modificare lo schema e aggiungere un registro, in modo che alla lettura del dato il bit contenuto in esso passi da 1 a 0. Facciamo riferimento alla nuova figura 5.16, in cui la scatola rossa rappresen-

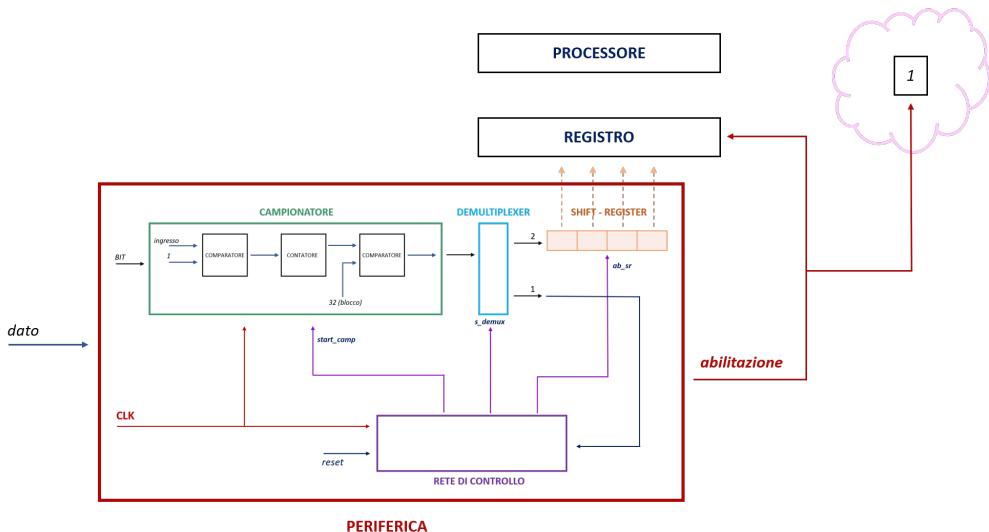


Figura 5.16: Schema a blocchi del ricevitore seriale con gestione errore sovrascrittura.

ta la periferica che abbiamo implementato nell'immagine 5.15: quando il demultiplexer produce l'uscita sulla linea 2, lo shift register prende il dato e lo memorizza in un banco di registri, che sarà poi utilizzato da un processore. E' la rete di controllo a dare il segnale di abilitazione per il trasferimento dall'SR ad R, che rappresenta il disaccoppiamento tra la periferica ed il processore. Quando la rete di controllo dà l'abilitazione, scrive anche 1 in un FF per comunicare che c'è un dato utile. Se all'interno del FF, la RC trova uno 0 vuol dire che il processore ha letto il dato, se trova un 1 vuol dire che non è stato letto. Nella nuvola rosa, ci sarà dunque un'altra logica che vede se il FF tra due scritture consecutive è diventato 1 o 0. Il flip-flop che una volta va a 0 ed una volta ad 1 è il FF a commutazione e serve a sapere se tra due scritture consecutive ci sia stata una lettura. Il programma che prende il dato e vede se ci sono stati errori è il driver (nuvola rosa).

Per approfondimenti in merito alla comunicazione seriale si rimanda al capitolo 7.

5.6 Esempio di progettazione di un'unità operativa e unità di controllo

Dobbiamo definire quali componenti servono per poter realizzare le due unità, ovviamente la progettazione parte dall'unità operativa, poiché non possiamo pensare di realizzare una rete di controllo senza prima conoscere cosa debba essa controllare. Vogliamo realizzare un ricevitore seriale diverso da quello precedente, infatti, piuttosto che ricevere un byte, riceve un messaggio, per cui treni di byte. La comunicazione prevede che vengano prima inviati sul canale due *segnali di sincronismo* (10101010), poi un *segnaletico di start* (00001111), il messaggio ed infine un *segnaletico di EOT* (01010101). Dal momento in cui viene ricevuto lo start, fino a quando non arriva il segnale di ter-

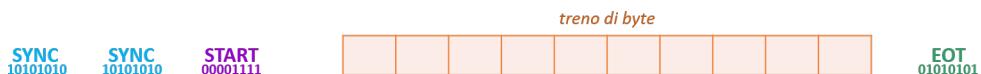


Figura 5.17: Sequenza di bit.

minazione, ogni 8 bit rappresentano un carattere, che va opportunamente memorizzato. E' chiaro che se l'EOT ha questa particolare codifica, nel messaggio non può essere presente una stringa di 8 bit uguale al segnale di stop, mentre può esserci la sequenza 10101010 perché dopo lo start parte il messaggio.

Prima della sequenza sync-sync-start ci sarà il protocollo di handshaking: *una volta stabilita la comunicazione con il protocollo asincrono, le due entità comunicanti devono per forza essere sincrone perché una invia i bit e l'altra li riceve*. Questo significa che il protocollo con cui le entità si sincronizzano è asincrono, tuttavia nel momento in cui entrano in gioco i registri a scorrimento in cui vanno messe e prelevate informazioni è opportuno che le entità siano sincrone, tempificate da un clock alla stessa frequenza. Per progettare l'architettura, partiamo da

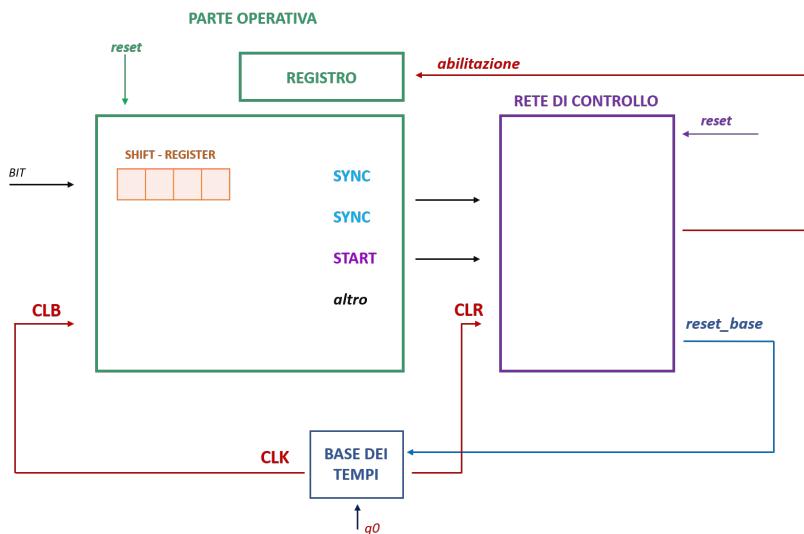


Figura 5.18: Schema di UO ed UC.

un blocco più esterno, che prende in ingresso i bit ed un clock che chiamiamo CLB. Questa è la parte operativa che deve memorizzare i bit in un oggetto al suo interno (lo shift-register) e dire se il segnale ricevuto è di sync, start oppure EOT. Dall'UO escono 2 fili, poiché 3 sono i casi che deve discernere e quindi sono necessari 2 bit per codificare tre informazioni. E' chiaro che in un altro blocco c'è la rete di controllo, che dopo la ricezione di due sync ed uno start, deve abilitare

il sistema alla memorizzazione di un dato in un registro R, che fa parte dell'unità operativa e che riceve l'abilitazione dalla RC. Anche la parte di controllo deve essere tempificata, per cui anch'essa riceve un segnale di clock che chiamiamo CLR. Poniamo attenzione su questo seguente aspetto,

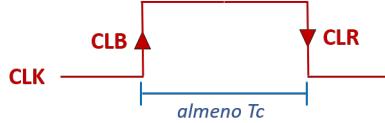


Figura 5.19: Segnale di clock del sistema.

facendo riferimento alla figura 5.19: **arriva il segnale di clock, sul fronte di salita l'unità operativa acquisisce il dato** e deve calcolare il risultato; la rete di controllo, per vedere il dato ed abilitarne la memorizzazione, deve essere sicura che sia assestato. Questo vuol dire che se l'unità operativa impiega un tempo T_c a calcolare l'uscita, **il segnale di clock deve durare almeno T_c** . Inoltre, **la rete di controllo prende il dato sul fronte di discesa** altrimenti il sistema non funzionerebbe.

Se andassimo sul VHDL, quest'ultimo non definisce il fronte di salita per il CLB e quello di discesa per il CLR, dobbiamo essere noi a specificarlo, così come il fatto che la durata del clock debba essere almeno T_c , cioè che *debba essere generato con un certo duty cycle altrimenti il sistema non funziona*.

Dal punto di vista implementativo, la parte operativa è fatta da uno SR che prende la linea dei bit e da una macchina combinatoria (codificatore) che classifica gli 8 bit ricevuti. La rete di controllo, invece, è un riconoscitore serie parallelo di sync, sync, start, questo vuol dire che è possibile **descrivere la RC con un automa**.

Inserito anche il reset, *l'automa a stati finiti della rete di controllo* è quello in figura 5.20. Quando

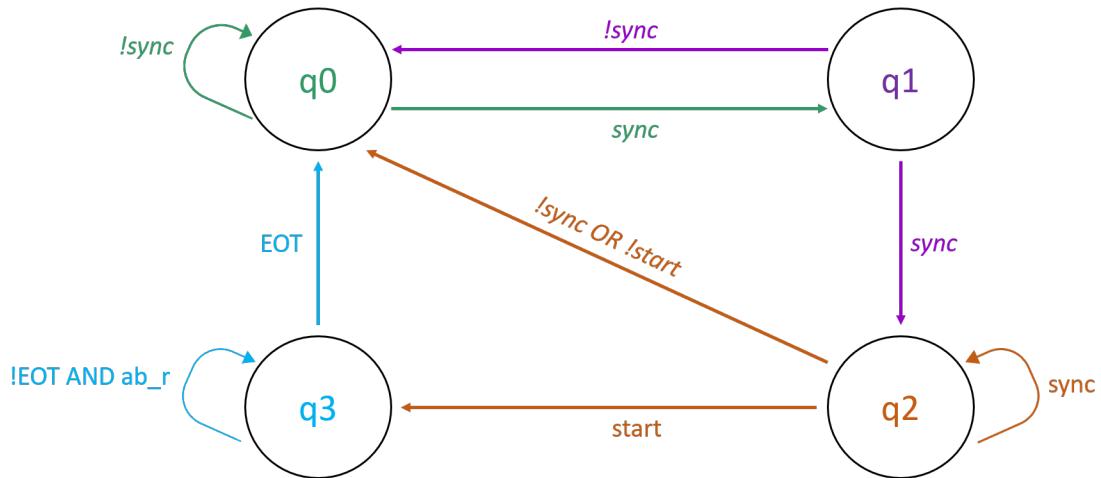


Figura 5.20: Automa a stati finiti della rete di controllo.

c'è il reset, va in q_0 ; finché l'unità operativa non dice che ha ricevuto un sync rimane in q_0 . Ricevuto il primo sync, andiamo in q_1 . Se il segnale che riceve successivamente non è un sync, ritorna in q_0 perché i sync da ricevere devono essere due sync. Se da q_1 riceve un altro sync, allora c'è la transizione verso q_2 . Il prossimo passaggio dipende dalla scelta del progettista: occorre decidere se, stando in q_2 , riceve un altro sync, non viene accettato e si ritorna in q_0 oppure se si permane in q_2 . Supponiamo che se arriva un altro sync rimanga in q_2 . **Se volessimo cambiare il protocollo, cambia l'unità di controllo, ma non la parte operativa**, perciò cambia solo l'automa. Se da q_2 arriva un segnale di start finisce in q_3 . Se non riceve né sync e nemmeno start torna in q_0 . Se

la macchina si trova in q3, vuol dire che ha ricevuto sync-sync-start, questo vuol dire che, fintanto che non arriva l'EOT, deve dare l'abilitazione al registro R così che possa memorizzare. Quando arriva l'EOT torna in q0.

Tutto questo è vero, ma **il sistema realizzato così non funzionerebbe.** Consideriamo la sequenza di bit in figura 5.21: il primo sync arriva dopo 8 bit, per riconoscere il successivo sync occorrono altri 8 bit. Il problema è che, se non operiamo sul clock, dopo i primi 8 bit che identifi-

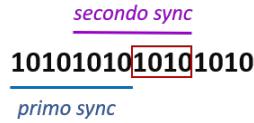


Figura 5.21: Sequenza di bit di ingresso.

cano il primo sync, mi basterebbe ricevere 4 bit (1010) per poter riconoscere un nuovo sync, quando noi in realtà non vogliamo un riconoscitore di sequenze sovrapposte, ma vogliamo un segnale ad ogni 8 bit ricevuti. Questo vuol dire che **q0 deve lavorare con la frequenza f di arrivo dei bit, q1 lavora dopo ogni 8 bit e quindi ad una frequenza che è f/8, così come q2 e q3.** E' necessaria una **base dei tempi** (ecco perchè presente il blocco sulla linea rossa nella figura 5.18), poiché l'unità operativa prende sempre i bit ma la rete di controllo non lavora sempre. Questo comporta anche una modifica all'automa, che diventa quello in figura 5.22, perché, quando l'unità operativa riconosce il primo sync, la rete di controllo dà un segnale alla base dei tempi di **"reset_base"**, in modo che da quel momento in poi la base dei tempi possa essere resettata e dare l'informazione ad ogni 8 colpi di clock.

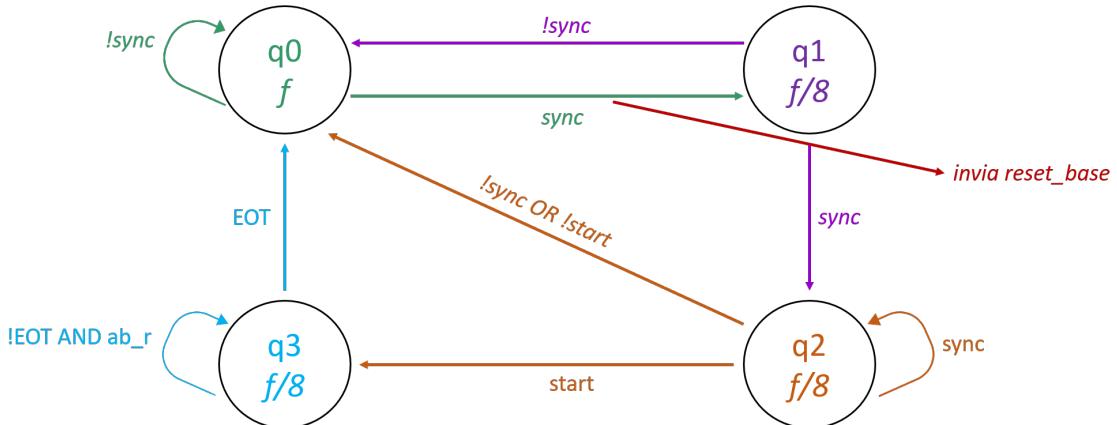


Figura 5.22: Automa a stati finiti corretto della rete di controllo.

5.6.1 Come è fatto il clock della rete di controllo?

Se CLK è il clock generale del sistema, noi sappiamo che se la rete di controllo sta in q0 deve lavorare a frequenza f e quindi vedere il clock del sistema CLK, ma se sta in q1, q2 o q3 (cioè non sta in q0) deve lavorare ad una frequenza f/8 e quindi vedere CLK8. L'equazione del clock della rete CLR è:

$$CLR = q0 * CLK + (!q0) * CLK8$$

La macchina che prende il segnale di clock e lo divide per 8 è un **contatore**, questo vuol dire che il segnale di “reset_base” va in ingresso al contatore. Per di più possiamo osservare che una volta il clock deve essere CLK e in altri casi deve essere CLK8, la macchina combinatoria che permette di scegliere tra due fili in ingresso è il **multiplexer**, il cui segnale di selezione è q0. Lo schema rappresentato in figura 5.23 non è altro che la base dei tempi.

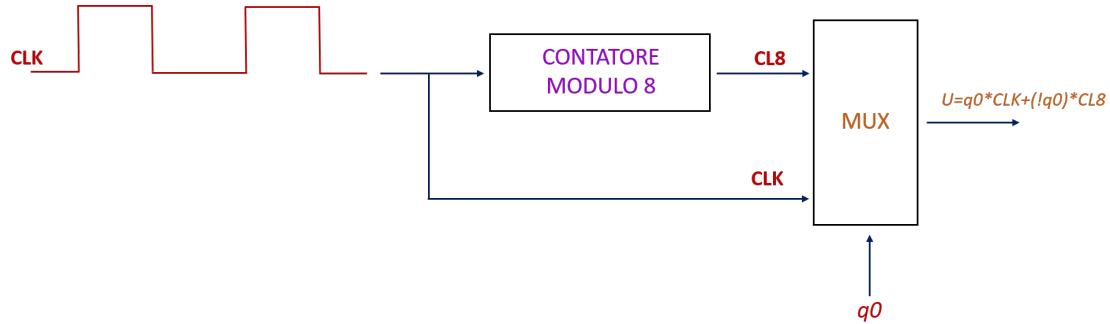


Figura 5.23: Blocco della base dei tempi.

5.7 Protocolli

Quando dobbiamo far comunicare tra di loro entità differenti, che magari devono cooperare verso un obiettivo comune, è necessario utilizzare dei protocolli per regolamentare tale comunicazione. L’obiettivo della progettazione, difatti, non è progettare un singolo componente ma inserirlo in un’architettura, nella quale deve interagire con altri componenti. Nasce allora un problema di sincronizzazione, un problema di protocollo. Possiamo classificare i protocolli in 3 categorie:

- Protocolli sincroni;
- Protocolli asincroni;
- Protocolli semisincroni.

Nei **protocolli sincroni** le unità lavorano con lo stesso riferimento temporale, noto e condiviso. Le due entità non si parlano direttamente: una volta dato il “via”, le loro azioni sono tutte motivate dall’esistenza del riferimento temporale comune. Ci sarà solo una fase di “prologo” iniziale, per far partire il lavoro. Il problema è che le attività possono sfasarsi e ciò va gestito, talvolta, infatti, conviene utilizzare un protocollo semisincrono.

I **protocolli asincroni** non sono basati sul concetto di tempo bensì su quello di **evento**. Le entità comunicano alla ricezione di eventi e finché non si verificano eventi, non trasmettono. Supponiamo, ad esempio, di avere due entità, A e B, che devono comunicare. In particolare, A deve mandare un dato a B. Se utilizzassero un protocollo asincrono, A invierebbe a B al colpo di clock. In questo caso deve avvenire uno scambio di eventi, allora A da il dato (messaggio 1), poi da il via (messaggio 2) e quando B riceve il dato risponde con un messaggio di ok (messaggio 3). A e B hanno eseguito un protocollo, il cosiddetto **handshaking**: si manda l’informazione, si aspetta di ricevere un messaggio di avvenuta ricezione e in questo modo il sistema converge e termina l’interazione, come mostra la figura 5.24.

Un altro protocollo asincrono è l'**handshaking completo**, in questo caso viene aggiunto un quarto messaggio: A manda il dato (messaggio 1), da il via (messaggio 2), B risponde di essere pronto (messaggio 3) e poi invia un ulteriore messaggio per avvertire di aver terminato l’operazione (messaggio 4), come viene mostrato in figura 5.25.

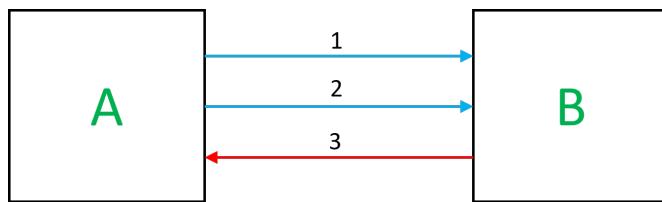


Figura 5.24: Protocollo handshaking

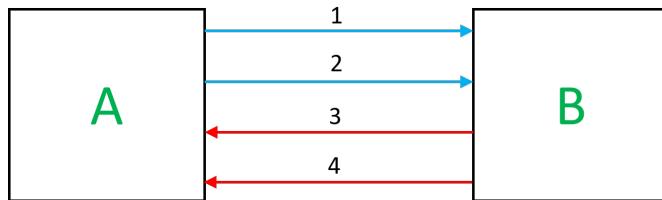


Figura 5.25: Protocollo handshaking completo

L'intento è fare avere subito ad A una risposta: nel protocollo precedente se B è un'entità meccanica, che impiega parecchio tempo per leggere il dato, A dovrà rimanere in attesa del messaggio 3 più a lungo. Per ricevere il messaggio 3, A deve aspettare che B termini l'operazione; in questo protocollo, invece, riceve subito il messaggio 3 e al termine delle operazioni riceve anche il messaggio 4. Questo protocollo aggiunge informazioni ma non è detto che quello precedente non funzioni. Infatti, anche se B non comunica di aver finito l'operazione, ovvero senza il filo 4, se A prova a mandare un nuovo dato comunque non riesce, B infatti non è ancora pronto a ricevere il dato successivo. In un protocollo asincrono la base sono gli eventi, in particolare eventi di richiesta, detti **comandi**, e eventi di risposta, detti **stati**, perché rappresentano lo stato in cui si trova la macchina che risponde. Questi protocolli possono essere arricchiti da ulteriori accorgimenti per gestire situazioni patologiche. Supponiamo di star utilizzando l'handshaking, non completo, e che si rompa la linea del messaggio 3. Possiamo inserire un *watchdog* così che, se B non risponde entro un certo tempo, si abortisce il sistema, notificando che la comunicazione non è avvenuta. Ciò non serve a terminare la comunicazione, che si conclude solo alla ricezione del messaggio 3, ma almeno a sbloccare l'entità A che così sa che la comunicazione non può effettuarsi.

Abbiamo visto che il numero minimo di fili, necessario a un'interazione asincrona, è 3, possiamo aggiungerne un quarto per avere l'handshaking completo (**protocollo interallacciato**) ma in realtà anche con 3 fili (**protocollo non interallacciato**) funziona ugualmente. Cerchiamo di capire se aggiungere ulteriori fili comporti qualche vantaggio. Supponiamo che l'entità A sia il **master**, M, e l'entità B lo **slave**, S. Il master invia un messaggio, da il via e lo slave risponde di aver correttamente letto il dato. A questo punto allora M sa che l'interazione è avvenuta correttamente, S non lo sa perchè non è certo che M abbia ricevuto il messaggio 3. Potremmo pensare di aggiungere un ulteriore messaggio, un filo, con cui M notifichi a S di aver ricevuto il messaggio 3, in questo modo S sa che l'interazione è avvenuta correttamente ma M no, non è certo che S abbia ricevuto il messaggio. Dovremmo aggiungere un altro filo ma torneremmo al problema opposto. È evidente che non è possibile arrivare a convergenza, anche se continuiamo ad aumentare il numero di fili, perchè si tratta di un **sistema distribuito**, lo stato è distribuito quindi se lo conosce un'entità l'altra non può conoscerlo. Questo deriva dal fatto che non c'è sincronizzazione, altrimenti non parleremmo di protocollo asincrono.

Infine, vi sono i **protocolli semisincroni**, che si comportano come i protocolli sincroni a meno che non si alzi un segnale speciale, un **flag**. In un protocollo sincrono, in cui le entità lavorano con lo stesso riferimento temporale – dunque si parla di **entità isocrone** – potrebbero verificarsi

delle situazioni di sfasamento, che vanno gestite. Se A e B comunicano ogni 2 colpi di clock ma per qualche motivo A non riesce ad essere pronta quando dovrebbe, alza un flag per avvertire B e interrompere il protocollo. Per capire meglio il concetto, consideriamo un esempio pratico: il processore e la cache comunicano con un protocollo semisincrono, condividono lo stesso clock e sul fronte di salita il processore richiede un dato, sul fronte di discesa la cache glielo fornisce, in modo sincrono. Questo è vero fin tanto che la cache contiene i dati richiesti dal processore ma se si verifica una situazione di *cache miss*, ad esempio a seguito di un salto, la cache impiegherà più tempo a fornire il dato, in quanto dovrà prelevarlo dalla memoria (figura 10.6).

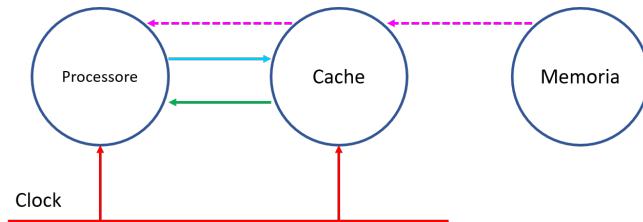


Figura 5.26: Protocollo semisincrono tra processore e cache

Per segnalare tale situazione, e interrompere la comunicazione sincrona, la cache alza un flag e il processore aspetta; quando la cache preleva il dato, abbassa il flag. Il processore nota che il flag è stato abbassato, solo al colpo di clock successivo, per questo è un protocollo semisincrono, non lo sarebbe se lo notasse proprio nel momento in cui il flag viene abbassato.

Capitolo 6

Macchine aritmetiche

In questo capitolo, affrontiamo lo studio e l'analisi di macchine aritmetiche, quali addizionatori, moltiplicatori e divisorì. Abbiamo scelto il posizionamento di questo argomento dopo aver affrontato macchine sequenziali, contatori e registri a scorrimento, poiché le architetture più complesse, come vedremo, fanno uso di questi componenti nella parte operativa.

6.1 Addizionatori

6.1.1 Half Adder

Il componente di base per la *somma di due bit* è l'half adder. Esso si progetta partendo dalla tabella di verità, come mostrato nella tabella 6.1. Del resto, questo componente realizza la somma

Tabella 6.1: Tabella di verità di un half adder.

X	Y	S	R
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

di due bit, che è alta quando uno dei due operandi in ingresso è alto, mentre il riporto in uscita è alto solo quando entrambi gli addendi sono 1. Con la logica o con la tabella di verità arriviamo allo stesso risultato, ovvero l'espressione delle funzioni S e R:

$$\begin{aligned} S &= !X*Y + X*(!Y) = X \text{ xor } Y \\ R &= X*Y \end{aligned}$$

E' possibile schematizzare questo blocco come in figura 6.1, sintetizzabile su una **rete a due livelli**.

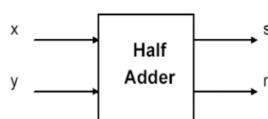


Figura 6.1: Half adder.

6.1.2 Full Adder

A partire dall'half adder possiamo realizzare il full-adder che presenta 3 ingressi, due bit e un riporto entrante, e ha due bit in uscita, sempre di somma e riporto. Anche in questo caso possiamo ricavare le funzioni booleane che esprimono S ed R a partire dalla tabella di verità.

Le funzioni sono le seguenti, mediante manipolazioni grazie alle proprietà dell'algebra di Boole e

Tabella 6.2: Tabella di verità di un full adder.

X	Y	C	S	R
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

eventuale minimizzazione con Karnaugh:

$$\begin{aligned} S &= X \text{ xor } Y \text{ xor } C \\ R &= X^*Y + C^*(X \text{ xor } Y) \end{aligned}$$

Per ricordare facilmente l'espressione del riporto, possiamo ragionare in questo modo: un riporto viene generato se entrambi gli addendi sono alti (and tra X ed Y) oppure (or) se c'è un riporto in ingresso e uno dei due addendi è alto, cioè abbiamo da sommare due bit alti e c'è un riporto sulla colonna (ecco perché $C^*(X \text{ xor } Y)$). Lo schematico di un full adder è quello in figura 6.2. Il

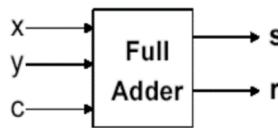


Figura 6.2: Full adder.

full-adder si può realizzare ad un livello di astrazione dataflow, oppure strutturale partendo dagli half adder.

Implementazione data-flow di un FA

Di seguito viene riportata una possibile implementazione in uno dei possibili livelli di astrazione, in cui non è stato fatto altro se non realizzare esattamente le due funzioni booleane S ed R.

```
-- implementazione dataflow di un full adder
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity full_adder is
    port
    (
        a,b, c_in: in std_logic;
        s, c_out: out std_logic
    );
end full_adder;
architecture dataflow of full_adder is
```

```

begin
    s <= ((a xor b) xor c_in);
    c_out <= ((a and b) or (c_in and (a xor b)));
end dataflow;

```

6.1.3 Ripple Carry Adder

Gli half adder ed i full adder sono in realtà fondamentali per l'introduzione del primo sommatore di stringhe di bit, il **sommatore a propagazione di riporto**. Si concatenano i vari full-adder,

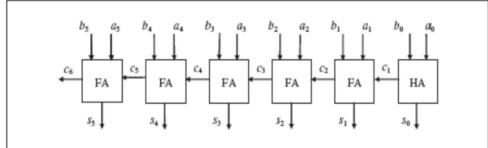


Figura 6.3: Architettura ripple-carry adder non omogenea.

come mostrato in figura 6.3, in cui ogni full-adder realizza la somma di una coppia di *bit omologhi* dei due operandi ed i riporti dell'uno vanno in ingresso a quello successivo. Abbiamo una *struttura non proprio omogenea*, perché tutti i componenti sono full-adder, tranne il primo che è un half adder, visto che non consideriamo la presenza di un carry in ingresso.

In teoria non sarebbe un problema, tuttavia dobbiamo sempre ragionare in virtù del fatto che nel mondo professionale non realizziamo qualcosa completamente da zero, ma ci ritroviamo a comprare dei componenti di libreria, ecco perchè **conviene usare un'architettura omogenea**. Possiamo allora utilizzare tutti blocchi full-adder e mettere il riporto entrante al primo pari a 0. In una

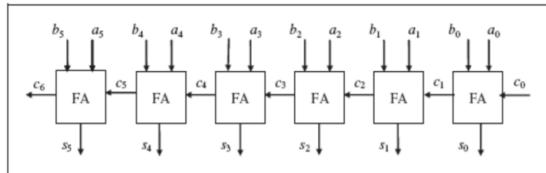


Figura 6.4: Architettura ripple-carry adder omogenea.

struttura del genere, è semplice valutare la propagazione dei riporti e quindi i ritardi totali. Se immaginiamo che ogni full adder, dato che è sintetizzabile su due livelli, abbia un ritardo complessivo di 2 tempi unitari sia per uscita che carry, ogni full adder ha ritardo 2T, quindi nel complesso **il ritardo è n*2T dove n è il numero di f-a**, proporzionale alla lunghezza della stringa. E' semplice ma non ha grandi prestazioni.

Questo circuito, fatto in questo modo, è in grado di *fare anche la somma di numeri relativi codificati in complemento a due*. Ci sono varie tecniche di codifica dei numeri relativi, normalmente si usa quella del complemento a due per fare queste operazioni, perché per altre tecniche, come il segno e modulo o in complementi diminuiti, bisogna fare degli aggiustamenti per gestire la somma. Per il complemento a 2 invece **consente di avere una somma corretta e determinare la condizione di overflow**.

Se stiamo sommando due numeri relativi in complemento a due e i numeri sono discordi non possiamo mai avere overflow, otteniamo sicuramente un numero rappresentabile, possiamo avere overflow quando sommiamo due bit positivi (primo bit 0) e il risultato è negativo (primo bit 1) o viceversa. Se invece vogliamo fare la somma di numeri naturali, su 4 bit ciascuno, potremmo dire che la macchina restituisce un numero di 4 bit, vuol dire che se diamo in ingresso delle stringhe

tali per cui l'uscita occuperebbe più di 4 bit la macchina deve dare una condizione di errore. Se invece vogliamo la somma di due stringhe su 4 bit e vogliamo il risultato vero, usiamo più bit in uscita. *Dipende sempre da dove dobbiamo inserire, in che sistema, il componente e a seconda di dove dobbiamo inserirlo cambiano le specifiche.*

Se prendiamo un registro con scritto 1111, se l'aritmetica è in interi è 15, se fosse un numero relativo in complemento a due, il primo 1 ci dice che è un numero negativo. Con 4 bit, possiamo rappresentare i numeri relativi da -8 a +7, poiché abbiamo 1-111 è -7. Se noi vediamo 1111 non sappiamo dire se è positivo o negativo, possiamo solo sommarlo; se stiamo lavorando con i numeri positivi allora abbiamo il carry, sistema di riporto, se invece sommiamo due numeri che iniziano per 1 ed abbiamo per risultato un numero che inizia per 0 vuol dire che abbiamo avuto un overflow, perché abbiamo sommato due numeri negativi e abbiamo ottenuto un numero positivo. Se infatti sommiamo 7 e 4 (rappresentabili su 3 bit), il risultato 11, che non possiamo rappresentare perché 11 è codificato come 1011. Se dobbiamo fare la somma per positivi vediamo solo la propagazione del riporto (quindi vedremmo 011 e riporto uscente alto), se dobbiamo fare una somma per numeri relativi e dobbiamo calcolare l'overflow, dobbiamo vedere il primo bit dell'addendo, il primo bit del secondo addendo e il primo bit del risultato. *L'espressione logica dell'overflow è $!bn!ancn + bnan!cn$: abbiamo sommato due numeri negativi e ottenuto uno positivo (cn) o viceversa, due numeri positivi e cn è negativo ($!cn$).*

6.1.4 Sottrattore

- Il complemento a 2 di B viene ottenuto aggiungendo 1 al complemento diminuito di B:
 $-B = b'_{n-1}b'_{n-2}\dots b'_0 + 1$

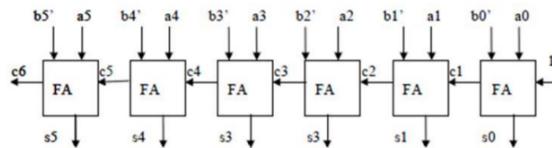
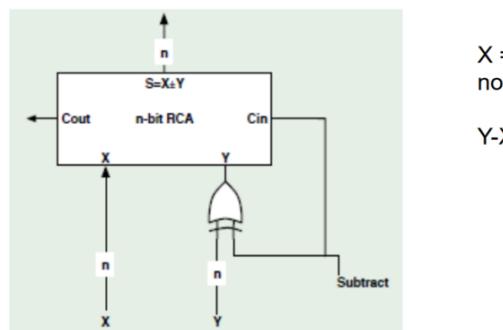


Figura 6.5: Sottrattore da un ripple-carry adder.

La differenza $a-b$ può essere vista come $a+(-b)$, quindi serve la codifica di b in complementi: si complementano tutti i bit e si somma 1. Se lo volessimo tradurre in circuito, potremmo sfruttare il ripple carry adder, dove all'addendo a mandiamo i vari bit di a , all'addendo b diamo tutti i bit negati (ci saranno delle porte not) e dil carry in ingresso è 1, perché poi si deve sommare a tutto il resto. **Questo circuito sa fare solo la sottrazione.**

6.1.5 Sommatore e sottrattore

Come vedremo, avere una macchina che sappia fare sia la somma che la sottrazione può essere molto utile. Per fare entrambe le operazioni, serve qualcosa di intelligente che metta il carry entrante ad 1 ed il negato di b quando vogliamo noi. Basta ricordare le **proprietà della porta xor**: $0 \text{xor} 0 = 0$, $0 \text{xor} 1 = 1$, **un numero in XOR con 0 fa il numero stesso, invece in XOR con 1 dà l'opposto del numero.** Se prendiamo un numero e facciamo XOR 0, otteniamo il numero stesso, invariato, mentre se facciamo la XOR con 1 otteniamo il complemento del numero. In questo modo otterremo i bit di b negati, facendo la XOR con 1, mentre se vogliamo fare la somma consideriamo la XOR di b con 0 così da ottenere proprio b . Questo valore con cui facciamo la XOR va in ingresso al RCA come **carry_in**. Analizziamo l'architettura a partire dalla figura 6.6: l'addendo x è mandato in ingresso all'RCA inalterato, mentre l'addendo y è in XOR con un



$$\begin{aligned}
 X &= X \text{ xor } 0 \\
 \text{not } X &= X \text{ xor } 1 \\
 Y-X &= Y+(-X) = \\
 &= Y+(X \text{ xor } 1) = \\
 &= Y+\text{not } X+1
 \end{aligned}$$

Figura 6.6: Sommatore e sottrattore da un ripple-carry adder.

segnale di subtract. Quando il subtract è pari ad 1, vuol dire che vogliamo fare la sottrazione, se è 0 vogliamo fare la somma. Supponiamo di voler fare la somma, vuol dire che subtract è 0, la XOR tra y e 0 mi restituisce y, per cui nel ripple carry entrano i due addendi x ed y. Come riporto entrante cin, entra il segnale di subtract, che era 0. Se vogliamo fare la sottrazione, mettiamo 1 in subtract, la XOR tra y ed 1 mi restituisce y con tutti i bit negati, allo stesso tempo l'1 del subtract va in ingresso al sommatore, Cin. Usiamo lo stesso circuito per realizzare due operazioni differenti, cambiamo con la rete di controllo il segnale di subtract. È un modello data-flow, in cui vediamo il flusso dei dati e non è un livello comportamentale.

Implementazione di un sottrattore/addizionatore

L'adder è formato da due blocchi, il ripple carry adder, ottenuto per composizione di full adder, e un blocco che effettua il complemento delle cifre, effettuando la xor bit a bit tra le cifre del secondo operando e il riporto entrante. Il segnale che chiamiamo cin nel port dell'entità sarà mappato con il segnale di subtract.

```
--implementazione di un sottrattore/addizionatore
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity adder_sub is
    port( X, Y: in std_logic_vector(7 downto 0);
          cin: in std_logic;
          Z: out std_logic_vector(7 downto 0);
          cout: out std_logic);
end adder_sub;
architecture structural of adder_sub is
    component ripple_carry is
        port( X, Y: in std_logic_vector(7 downto 0);
              c_in: in std_logic;
              c_out: out std_logic;
              Z: out std_logic_vector(7 downto 0));
    end component;
    signal complementoy: std_logic_vector(7 downto 0);

    begin
        complementoy_y: FOR i IN 0 TO 7 GENERATE
            complementoy(i)<=Y(i) xor cin;
        END GENERATE;
        RA: ripple_carry port map(X, complementoy, cin, cout, Z);
    end structural;
```

In figura 6.7, è possibile visualizzare una simulazione del funzionamento del componente: nel primo caso il segnale di subtract è 0, per cui facciamo la somma tra 0F e 28 in esadecimale, nel secondo caso il segnale di subtract è 1, per cui facciamo la differenza tra 28 e 17.

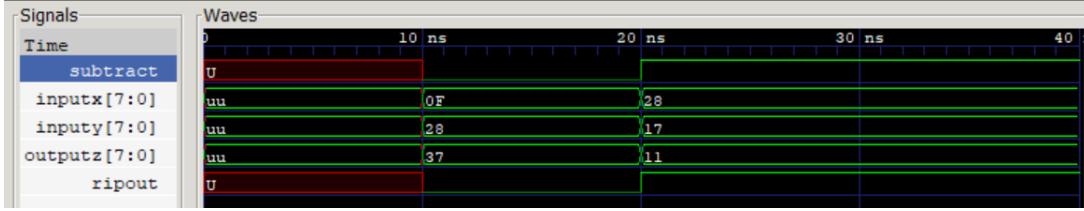


Figura 6.7: Simulazione sommatore/sottrattore.

6.1.6 Carry Look Ahead

Il carry look-ahead si basa su un semplice principio, ovvero **provare ad anticipare gli stadi di calcolo dei riporti**. Il problema di una macchina aritmetica, quale il ripple carry, è la lentezza, poiché per stringhe troppo lunghe occorre aspettare un tempo proporzionale agli stadi di somma. L'idea, invece, del CLA è calcolare i riporti a parte e darli quanto prima possibile insieme alle cifre da sommare. Le formule per il calcolo del riporto e della somma del full-adder sono mostrate in figura 6.8: si osserva che abbiamo un riporto in uscita se lo si sta generando in questo momento (**i due addendi sono alti**) oppure se lo si aveva già prima e almeno una delle due cifre è alta (in altre parole avremmo un 1 nella somma della colonna, ma dobbiamo sommare anche l'1 della colonna precedente). Queste due condizioni sono importanti da isolare, perché sono **le condizioni di generazione e propagazione del riporto**.

$$\begin{aligned} C &= xy + xz + yz = \\ &xy + z(xy' + x'y) = \\ &xy + z(x \oplus y) \end{aligned}$$

$$\begin{aligned} S &= x'y'z + x'yz' + xy'z' + xyz = \\ &x \oplus y \oplus z \end{aligned}$$

Figura 6.8: Calcolo riporto uscente e somma nel full-adder.

nerazione e propagazione del riporto. Normalmente la XOR scompare e diventa una OR, in realtà è la stessa cosa. Vediamo l'espressione del carry, **ci+1: il carry, generato nello stadio i-esimo e che insiste nello stadio i+1, è dato da $x^*y+(x+y)^*c$** . Va bene anche la OR perché l'unico punto in cui sono diverse tra loro è l'ingresso 11. Guardando l'espressione, **il primo termine della somma sarà alto, per cui la somma è un valore alto, è vero che la OR avrebbe dato 1 anche a destra, mentre la XOR avrebbe dato 0, ma il risultato di tutta la somma (il valore di ci+1) è comunque 1.** **La condizione di generazione è il prodotto, quella di propagazione è la somma.** Cominciamo ad isolare il termine del carry: avevamo un'espressione iniziale che era generica sul carry i+1 ed usava il carry i, perché ogni volta abbiamo bisogno del riporto generato allo stadio precedente (è quello che facciamo nel ripple carry adder). Se iteriamo il procedimento e calcoliamo le espressioni di Ci, facciamo il prodotto delle cifre i-1 e la somma delle cifre i-1 moltiplicata il riporto i-1; ad un certo punto otteniamo un'espressione che dipende da c0, il primo riporto. Se torniamo indietro e al posto della somma scriviamo P e al posto del prodotto scriviamo G, possiamo scrivere tutti i riporti in funzione di P e G che dipendono solo dagli addendi e da c0.

Osservazione: più andiamo avanti nella catena di carry, più l' espressione diventa complicata, teoricamente sono tutte somme di prodotti (tempo 2T perché su due livelli), ma non è esattamente

Il riporto in uscita dallo stadio i -esimo assume la forma:

$$c_{i+1} = x_i y_i + (x_i + y_i)c_i = G_i + P_i c_i \text{ dove } c_i \text{ può essere scritto come:}$$

$$c_i = x_{i-1} y_{i-1} + (x_{i-1} + y_{i-1})c_{i-1} = G_{i-1} + P_{i-1} c_{i-1} \text{ da cui:}$$

$$c_{i+1} = G_i + P_i c_i = G_i + P_i (G_{i-1} + P_{i-1} c_{i-1}) = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1}$$

Figura 6.9: Determinazione delle due condizioni.

così nella pratica. Innanzitutto dobbiamo avere delle porte con così tanti ingressi, per di più aumentando il fan-in delle porte, aumenta il ritardo, quindi di fatto il tempo per calcolare c_3, c_4 , e quelli che vengono dopo non sono uguali. *Idealmente abbiamo tolto il problema della propagazione perché saremo capaci di generare dopo un certo tempo, che è teoricamente sempre lo stesso, tutti i riporti che servono.* Come sommatore possiamo usare una rete di questo genere, al primo stadio

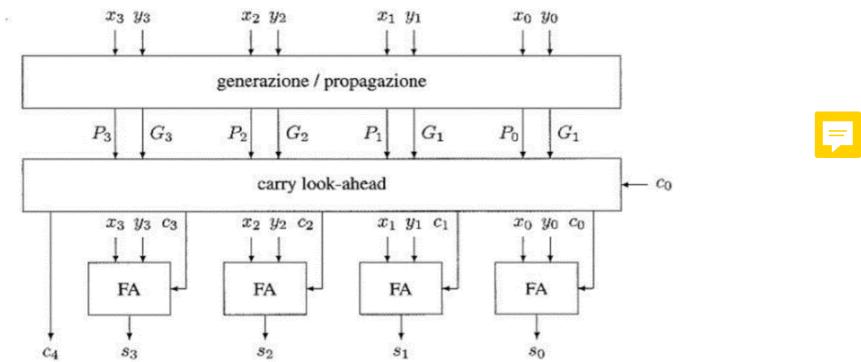


Figura 6.10: Schema 1 carry look ahead.

calcoliamo tutti i P e i G , poi calcoliamo tutti i riporti (rete carry look-ahead), all'uscita di questa rete otteniamo da qui c_1, c_2, c_3 , e così via. In quanto tempo? La prima rete è una porta, la seconda è data da due stadi, quindi siamo a 3 ritardi, la quarta rete può utilizzare dei full-adder perché per ogni porta abbiamo i due bit da sommare ed il riporto corrispondente. Il full adder è una rete a 3 livelli. In tutto sono 5 livelli, quindi almeno teoricamente abbiamo un ritardo fisso di 5T e non più k^*2T , come nel ripple carry adder.

Se ricordiamo che non c'era la OR ma la XOR e che l'espressione della somma del full-adder è la XOR tra i tre ingressi: $x \oplus y \oplus c$ è P , quindi possiamo tirar sopra i fili di $x \oplus y$ e fare la XOR con c che esce, ottenendo lo schema finale riportato in figura 6.11. **Riduciamo ancora di più il ritardo perché la XOR idealmente è una sola porta, passiamo da 5T a 4T.**

Nelle slice dell'FPGA c'è già la logica per implementare la propagazione del riporto, ci sono porte AND, XOR che sviluppano questo tipo di logica, di propagazione e generazione, in piccolo. Idealmente funziona con 5T o 4T, ma la rete di carry ahead è complicata da ottenere, in VHDL è anche difficile da generalizzare. Se lo proviamo sulla scheda abbiamo che più aumenta la grandezza delle stringhe, più le espressioni vengono spalmate su LUT di slice diverse, non avremo il comportamento ideale che ci aspettiamo. In realtà non lo avremmo nemmeno con il ripple carry adder, che in teoria potrebbe sfruttare la linea di carry che sta sulle slice (tutte le slice che stanno sulla stessa colonna condividono la linea di carry, per cui essendo una linea dedicata, il riporto sarebbe velocissimo) ma quando esauriamo le slice della colonna dobbiamo andare su un'altra colonna, entra in gioco la rete di interconnessione e quindi si introducono ritardi.

Ricordiamo che: $S = x \oplus y \oplus \text{Cin} = P \oplus \text{Cin}$

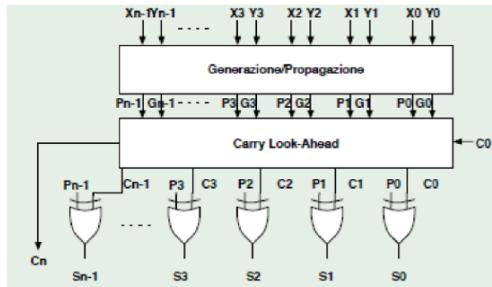


Figura 6.11: Schema 2 carry look ahead.

6.1.7 Sommatore veloce: Carry Select

Supponiamo di avere una serie di bit da sommare, per esempio stringhe a 32 bit. Se utilizzassimo un ripple carry adder, dovremmo considerare una catena di 32 full-adder, ognuno dei quali prende in ingresso il bit i -esimo dei due addendi, tuttavia la catena di propagazione si allunga ed in maniera ad essa proporzionale il ritardo cresce. Potremmo invece suddividere le due stringhe in gruppi di m bit ed ogni ripple carry realizza la somma di due addendi con un numero di bit inferiore. L'obiettivo è realizzare un **sommatore parallelo**. Per esempio, consideriamo la figura 6.12, in cui suddividiamo le stringhe in gruppi di 4 bit. Dopo 4T, otteniamo il riporto uscente c_4 . Invece

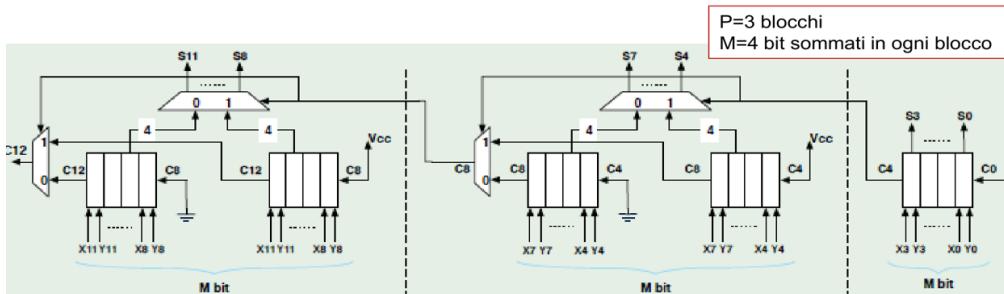


Figura 6.12: Sommatore carry select.

di aspettare i 4T, proviamo ad anticipare le somme, per cui al primo stadio replichiamo i ripple carry: il primo (a destra) effettua la somma tra gli altri 4 bit di X e di Y con riporto entrante 1 (V_{cc}); il secondo effettua la somma tra gli stessi 4 bit di X e di Y (da X_4 ad X_7 e da Y_4 a Y_7) con riporto entrante 0. Facendo così, se facciamo partire tutti i ripple carry allo stesso istante, dopo un tempo m^*T abbiamo il risultato del primo (partendo da sinistra), dei due al primo stadio, degli altri due al secondo e così via. Il risultato del riporto dello stadio precedente consentirà di prendere il risultato giusto, sia in termini di bit di somma che riporto.

Ecco perché sono necessari i multiplexer: quello in alto prende i 4 bit di somma generati dai due ripple carry dello stesso stadio ed il riporto dello stadio precedente consente di scegliere quelli corretti. Il mux sulla sinistra di ogni stadio prende in ingresso i due bit di riporto dei due ripple carry ed ancora una volta il riporto dello stadio precedente permette di selezionare il riporto in uscita corretto, che abiliterà la selezione dei multiplexer dello stadio successivo. *L'utilizzo del multiplexer consente una propagazione più veloce, infatti servono m tempi elementari per avere tutti risultati più un tempo che serve per settare i multiplexer di ogni stadio.* In figura 6.13, vediamo la formula che esprime il tempo di esecuzione di questa macchina: *avremo m volte il tempo elementare del full adder, perché partono tutti in parallelo, più tutte le volte che*

$$T = M * TFA + (P-1) * TMUX.$$

Figura 6.13: Tempo del sommatore carry select.

abbiamo dovuto aspettare il risultato del mux. Se abbiamo p stadi, i multiplexer sono in $p-1$ stadi, quindi il tempo totale è $M*TFA + (p-1)*TMUX$.

Osservazione: non è un sistema pipelined perché dovrebbe avere dei registri, tutti lavorerebbero in sequenza con un valore di clock. E' una macchina combinatoria con propagazione. Questa macchina è più veloce, però ha più del doppio delle porte che avrebbe avuto un ripple carry. Nella valutazione di una macchina, c'è da considerare sempre velocità, consumo ed occupazione. Ciò che va più veloce molto spesso consuma di più.

6.1.8 Sommatore veloce: Carry Save

Questo sommatore nasce per cercare di migliorare i tempi di propagazione, ma in realtà risulta particolarmente utile quando occorre sommare più stringhe, utile per esempio nelle moltiplicazioni. Se dovessimo sommare tre stringhe tra loro, potremmo pensare di usare banalmente un ripple carry adder per sommare le prime due ed il risultato parziale viene mandato in ingresso al secondo ripple carry insieme alla terza stringa. Scendendo a livello del singolo full-adder che compone il ripple

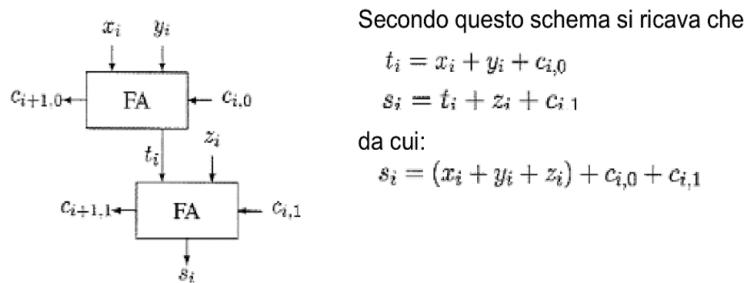


Figura 6.14: Somma tra x_i , y_i e z_i con un full adder.

carry, accade quanto mostrato in figura 6.14, prima sommiamo x_i ed y_i , con il riporto entrante $c_{i,0}$, il bit di somma temporaneo t_i entra nello stadio successivo insieme a z_i , generando così il bit i -esimo della somma. Se riorganizziamo gli addendi, possiamo vedere che la somma è fatta direttamente dando le cifre più significative, i -esime, e sommando a parte i riporti. In questo modo possiamo evitare la catena di riporti che si genera in un normale ripple carry.

Dati X , Y , Z , usiamo i 3 ingressi del full-adder per dargli x_0, y_0, z_0 e così via con tutti i bit dei tre addendi dello stesso peso. Quando facciamo le somme abbiamo una cifra con lo stesso peso delle cifre che abbiamo sommato e un riporto che avrà peso "+1", cioè un peso maggiore. Ottenuto il vettore delle somme parziali t e dei riporti c_s , dobbiamo sommarli tra loro. Non possiamo sommare t_0 e c_{s0} , perché hanno peso diverso: t_0 è la prima cifra della somma, non va sommata con niente; c_{s0} con t_1 , che è la cifra vera di peso 1 e così via.

In termini di tempo di propagazione tra il ripple carry e questo sommatore veloce, quando effettuiamo la somma tra due stringhe, non cambia nulla; nel momento in cui confrontiamo il comportamento del carry save quando le stringhe da sommare sono più di due con quello che avrebbe un insieme di ripple carry, il carry save consente di risparmiare tempo.

Se tra la *carry save logic* e il ripple carry adder mettessimo un registro, diventerebbe un'architettura pipelined, in cui possiamo sommare 3 cifre, aggiustarle, mentre sommiamo altre 3 cifre. Se non mettiamo i registri dobbiamo aspettare tutto il tempo necessario. Nel caso dell'architettura

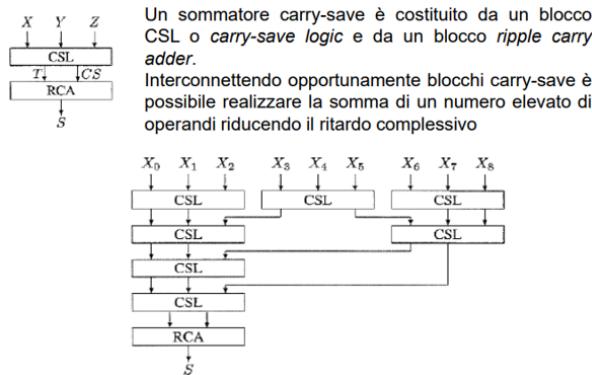


Figura 6.15: Sommatore carry save.

in figura 6.15, tutto il circuito lavora sempre, mentre nell'architettura pipelined la parte combinatoria calcola sempre, quella di memorizzazione opera quando il valore è significativo. La scelta tra un'architettura o l'altra dipende da eventuali vantaggi che si possono ottenere: se il tempo di memorizzazione del registro intermedio è vantaggioso rispetto al tempo di somma del blocco RCA allora conviene l'architettura pipelined, altrimenti no, perché vorrebbe dire che staremmo perdendo più tempo a memorizzare il dato che a calcolare la somma nonostante i tempi di propagazione. **In termini di tempo di attraversamento, è chiaro che va considerata la presenza del registro, tuttavia l'architettura pipelined consente di aumentare la produttività.**

Implementazione VHDL di un Carry Save

Di seguito viene riportata una possibile implementazione di un carry save per la somma di tre stringhe da 4 bit. E' stato realizzato ad un livello di astrazione strutturale, con un primo stadio di full adder che rappresenta la rete di carry save, quindi ogni full-adder genera il bit i-esimo di t (somma) e cs (riporto).

```
--sommatore carry save di 3 stringhe da N bit
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity carry_save is
    generic(N: natural:=8);
    port(    X, Y, Z: in std_logic_vector(N-1 downto 0);
              s: out std_logic_vector(N+1 downto 0));
end carry_save;
architecture structural of carry_save is
    component full_adder is
        port(    a,b,cin: in std_logic;
                  r,cout: out std_logic);
    end component;
    signal cs: std_logic_vector(N-1 downto 0);
    signal t,temp: std_logic_vector(N downto 0);
begin
    cs0ton1: for i in 0 to N-1 generate
        csi: full_adder port map(X(i),Y(i),Z(i),t(i),cs(i));
    end generate;
    t(N)<='0';
    temp(0)<='0';
    fa0ton1: for i in 0 to N-1 generate
        fa: full_adder port map(t(i+1),cs(i),temp(i),s(i+1),temp(i+1));
    end generate;
```

```

    s(N+1) <= temp(N);
    S(0) <= t(0);
end structural;

```

Il secondo stadio è la rete di ripple carry per sommare i bit di somma con i bit di carry con lo stesso peso.

6.2 Moltiplicatori

Se abbiamo due stringhe da moltiplicare, X moltiplicando codificato su n bit ed Y moltiplicatore codificato su m bit, il prodotto ha un numero di bit pari ad $n+m$. Vediamo sia moltiplicatori combinatori che sequenziali, di quelli paralleli vedremo un certo numero simili a quello che facciamo nella moltiplicazione a mano, mentre i seriali cercano di mettere in sequenza le operazioni elementari all'interno di un prodotto.

6.2.1 Moltiplicatore binario a due bit

Nei paralleli distinguiamo due fasi: in una calcoliamo i prodotti parziali, cioè tra ciascuna cifra del moltiplicando e ciascuna del moltiplicare. Essendo un prodotto fra due bit, possiamo farlo con la AND, per cui otteniamo una matrice di AND. La seconda fase è quella delle somme parziali, soggetta alle varie architetture. Per esempio, nel caso della moltiplicazione tra due bit, abbiamo

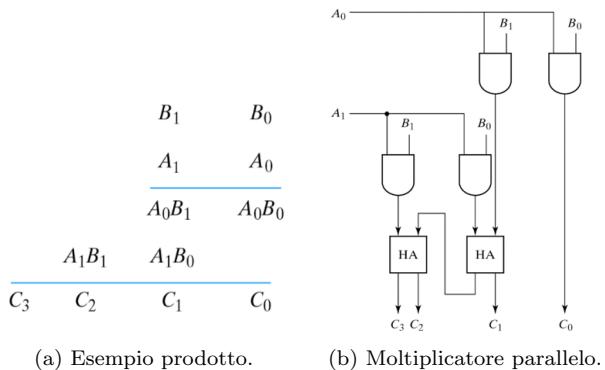


Figura 6.16: Moltiplicatore binario a due bit

le porte AND che fanno i prodotti parziali e poi usiamo half adder per fare la somma delle due righe. Sono half adder collegati in cascata, quindi alla fine stiamo usando il ripple carry.

6.2.2 Prodotto come somma per righe

Quando abbiamo molte righe da sommare, la parte di prodotto parziale può sempre essere fatta con le AND, quindi lo stadio iniziale dei moltiplicatori paralleli prevede sempre una matrice di AND. In questo caso proviamo a fare la somma per righe, utilizzando un ripple carry e ricordando quanto abbiamo già detto nel caso del carry save: se dovessimo sommare tre stringhe tra loro, possiamo usare un ripple carry adder per sommare le prime due ed il risultato parziale viene mandato in ingresso al secondo ripple carry insieme alla terza stringa. E' esattamente quello che facciamo in questa architettura, sommiamo prima le due stringhe, poi il risultato va con la terza e così via.

Nei nostri esempi, tipicamente consideriamo il caso in cui le stringhe da moltiplicare, cioè moltiplicando e moltiplicatore sono espressi sullo stesso numero di bit. In questo caso sono 4 bit;

nella moltiplicazione di due stringhe da 4 bit, vengono fuori 4 righe da sommare che si possono vedere come dei vettori di 4 bit ciascuno, solo che **le righe sono shiftate l'una rispetto all'altra poiché hanno un peso diverso**. Nella procedura manuale, noi incolumniamo le stringhe e aggiungiamo lo 0 per sommare cifre che hanno lo stesso peso. Quando utilizziamo il prodotto con la somma per righe non facciamo questa operazione, i nostri operandi sono ogni volta le stringhe da 4, per cui i componenti vanno istanziati e mappati in maniera opportuna. Sulle prime due righe, l'architettura è quella in figura 6.17, in cui il prodotto più a destra va

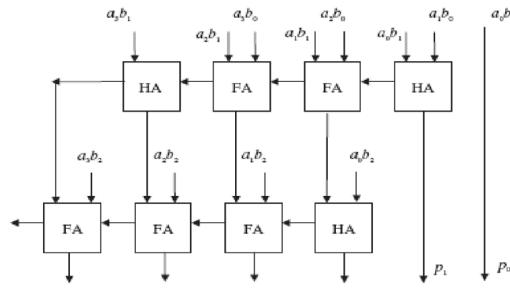


Figura 6.17: Prodotto come somma per righe.

direttamente giù, gli altri vanno incolumnati in modo giusto e con il sommatore a propagazione di riporti facciamo la somma delle prime due stringhe. Per eseguire la somma con la terza riga, dobbiamo realizzare una cosa del genere: aggiungiamo un'altra riga, mettendo opportunamente i prodotti con il peso giusto. **Bastano tre file di sommatori per sommare 4 righe, perché le prime due vanno insieme**. Otteniamo questa *architettura regolare* perché usa tutti componenti FA (il primo potrebbe essere FA con riporto entrante 0).

Questa architettura porta con sé tutti i problemi del ripple carry singolo. Se ci vuole un tem-

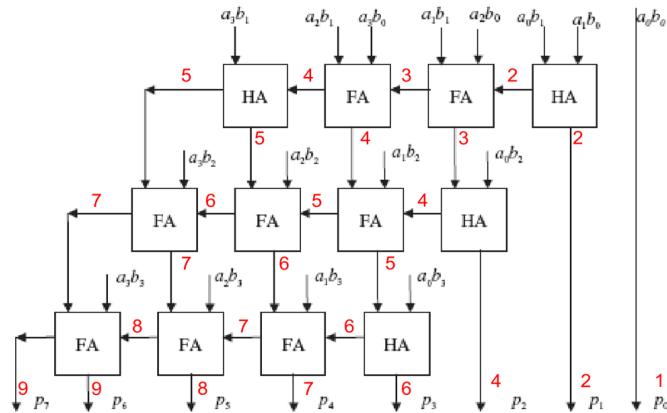


Figura 6.18: Prodotto come somma per righe completo.

po unitario per avere tutti i prodotti e ricordando che ogni FA è una rete su due livelli, per cui impiega 2T per fornire il risultato, partendo da destra verso sinistra, da sopra verso il basso, il primo componente HA e FA dà un risultato dopo 3 tempi elementari, 3T, perché il primo tempo si spreca per le AND e poi 2T per HA/FA. **Ogni componente, per poter dare un risultato corretto, deve avere i 2 o 3 ingressi stabili, per cui dobbiamo aspettare sempre il più lento degli ingressi**, per cui, anche se abbiamo il prodotto parziale a_0b_2 disponibile dopo un tempo unitario, dobbiamo aspettare che sia disponibile anche il risultato del FA che sta sopra. Il tempo complessivo che impieghiamo è pari a 17T.

6.2.3 Prodotto come somma per diagonali

Considerando di base gli stessi componenti e tenendo conto di quanto detto, *come si potrebbe migliorare la prestazione?* Possiamo usare un'architettura che propaga i riporti in diagonale. In termini di numero di componenti necessari, non cambia nulla tra questa macchina e quella che

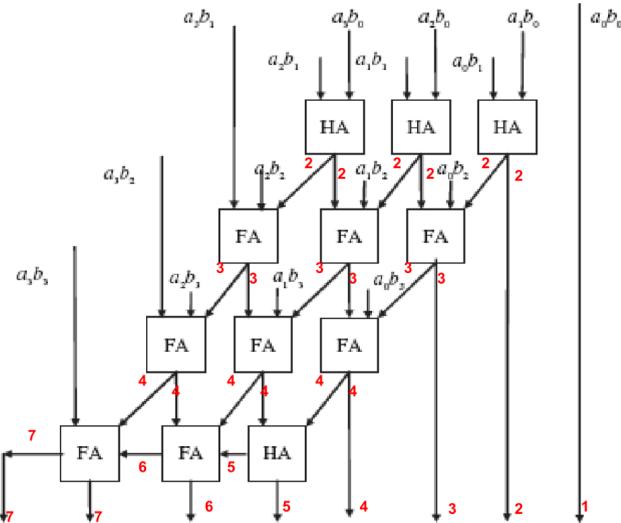


Figura 6.19: Prodotto come somma per diagonali.

realizza il prodotto come somma per righe: abbiamo 3 righe di 4 full adder ciascuno, nel moltiplicatore precedente, mentre adesso abbiamo 4 righe di 3 full adder ciascuno. Guardando la singola riga, ci accorgiamo che tutte, tranne l'ultima, non propagano il riporto tra loro ma lo mandano ogni volta allo stadio successivo, il riporto comunque si propaga non più sulla riga ma sulla diagonale. In questo modo, tutti i FA dello stesso livello daranno i risultati, sia somma che riporto uscente, dopo lo stesso tempo, quindi i ritardi che devono aspettare i blocchi del livello successivo, cioè l'input più lento, sono sempre gli stessi. Per esempio, il primo livello da gli output dopo 3, il secondo livello li prende dopo 3 e poiché l'input per ciascuna macchina è dato anche dal prodotto parziale che abbiamo già a disposizione, il secondo livello fornisce l'output dopo 5, quello successivo dopo 7 e così via. Solo l'ultimo stadio deve fare anche la propagazione.

Complessivamente risparmiamo qualcosa come tempo, ma dal punto di vista dell'occupazione spaziale non cambia nulla.

6.2.4 Prodotto come somma per colonne

Nella moltiplicazione su carta e penna, faremmo la somma sulle colonne e laddove non c'è un valore sommiamo 0. Se dovessimo fare questa macchina combinatoria, la somma degli elementi sulle colonne non potrebbe essere fatta utilizzando i ripple carry perché altrimenti dobbiamo fare la propagazione dei riporti anche sulle colonne. Consideriamo un **contatore di bit alti** (è una macchina combinatoria), l'uscita di questo è una macchina che fornisce un numero codificato su più linee. Per esempio, se dovessimo fare il contatore di 5 bit (cioè deve contare quanti bit alti ci sono in una stringa di 5 bit) l'uscita è definita su 3 bit.

Il **contatore di due e di tre sono rispettivamente HA e FA**, perché un oggetto che sa sommare tre bit e dà un'uscita su due bit o che sa sommare due bit e dà un'uscita su due bit è un contatore di 3 bit e 2 bit rispettivamente. A partire dagli HA e FA si possono utilizzare i contatori. Per ogni colonna, consideriamo come numero di bit in ingresso la somma dei bit che normalmente stanno sulla colonna, cioè la grandezza dell'operando, più tutti i possibili bit che potremmo avere

su quella colonna a causa di riporti generati dalle cifre precedenti. Facciamo riferimento a questa

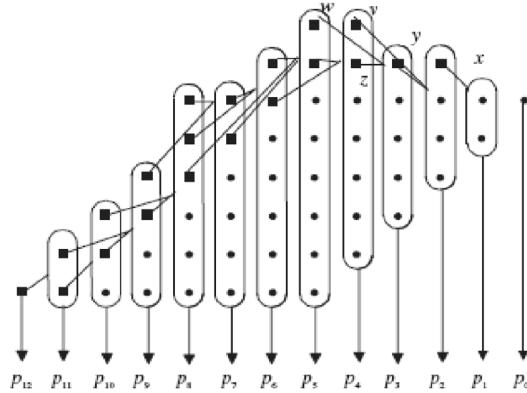


Figura 6.20: Esempio di somma per colonne.

figura: i pallini sono l'idealizzazione della cifra i -esima che dobbiamo sommare, i prodotti parziali $a0b0$, $a0b1$ e così via. Cominciamo da destra, il primo da destra sta da solo e lo portiamo già. Sulla seconda colonna abbiamo solo 2 elementi incolonnati e ne facciamo la somma, possiamo generare il riporto sulla cifra successiva. Parte la linea verso il rettangolo (indica il riporto dello stadio precedente) sopra che sta sulla cifra dopo. Quando arriviamo sulla 3 colonna, non abbiamo solo i 3 bit della colonna da sommare ma anche il riporto che potenzialmente potremmo avere dalla cifra precedente, per cui complessivamente dobbiamo saper fare la somma di 4, che può generare anche il numero 4 (100). Questo vuol dire che un bit viene tirato giù come termine i -esimo del prodotto, ma due saranno di riporto, uno che insiste sulla colonna successiva ed un altro che insiste su due colonne avanti, allora parte un possibile riporto sia su w che v . **Per ogni stadio dobbiamo considerare sempre un contatore più grande o più piccolo** (la matrice delle colonne man mano si riduce) **che tenga conto dei riporti generati dalla colonna precedente**. Questa architettura presenta comunque un problema, perché per ogni colonna dobbiamo istanziare un contatore con un numero di bit di ingresso differenti e per di più bisogna fare attenzione al mapping, dato che ogni contatore prende in ingresso le cifre della colonna ed eventuali riporti generati non solo dalla colonna precedente ma anche da quelle prima.

Possiamo fare qualcosa di diverso: come contatori per la colonna, istanziamo contatori con un numero di input pari a quelli della colonna; ciascuna di queste macchine non restituisce più una sola cifra, ma $\log_2(n)$ dove n sono gli ingressi. *I risultati del contatore sono disposti in diagonale e le cifre dei risultati del contatore insistono su cifre diverse del risultato.* I risultati dello stadio dei contatori devono comunque essere sommati, ma da una prima matrice in cui avevamo tutti contatori diversi tra loro e 6 righe in tutto da sommare, arriviamo ad una seconda matrice in cui le righe da sommare sono al più 3 ed utilizziamo contatori da 2 e da 3 (cioè sono HA e FA). Tra loro possiamo fare lo stesso discorso di prima, possiamo considerare i risultati delle somme opportunamente incolonnati tra loro. Procedendo in questo modo, passiamo dalla matrice $m1$ alla matrice $m2$ dove abbiamo soltanto due righe da sommare, per cui possiamo usare un ripple carry.

Per esempio, se abbiamo un prodotto di stringhe di 100 bit, dobbiamo fare 100×100 bit, quindi avremmo 100 righe da sommare. I 100 elementi sulla colonna devono andare in ingresso ad un contatore in grado di contare 100, ovvero fornisce un'uscita su 7 bit, per cui da una matrice di 100 righe passiamo ad una matrice di 7 righe e da quest'ultima ad una matrice di 3 righe, da cui possiamo ricavarne 2, sommate infine con un ripple carry. *Significa che in 4 stadi di computazione, fatti in questo modo, risolviamo un prodotto di 100×100 .* Ad ogni stadio istanziamo dei contatori, diversi, per cui **non c'è omogeneità**, ma sappiamo dire con certezza su ogni colonna quale contatore istanziare e dobbiamo fare attenzione a collegare opportunamente i contatori per l'ingresso

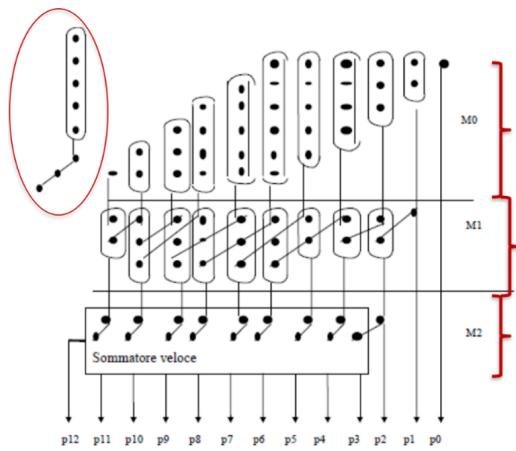


Figura 6.21: Esempio 2 di somma per colonne.

successivo del prossimo stadio che farà lo stesso.

Ci sono altre tecniche che migliorano questa dal punto di vista dell'omogeneità, ciascun contatore si può realizzare con HA e FA (per esempio un contatore di 7 si può fare collegando opportunamente i contatori di 3 e di 2), per cui si prova a utilizzare nel design solo componenti HA e FA, rendendo il progetto più complesso.

6.2.5 Moltiplicatore MAC

0	0	0	x_3y_0	x_2y_0	x_1y_0	x_0y_0
0	0	x_3y_1	x_2y_1	x_1y_1	x_0y_1	0
0	x_3y_2	x_2y_2	x_1y_2	x_0y_2	0	0
x_3y_3	x_2y_3	x_1y_3	x_0y_3	0	0	0

Figura 6.22: Prodotto.

Questa, in figura 6.22, è una schematizzazione di come è fatta la matrice dei prodotti parziali che dovremmo sommare in colonna se la facessimo a mano. Noi diamo per assunto che nello stadio iniziale facciamo una matrice di prodotti di porte AND in cui ogni elemento è il risultato di una porta AND, però *nel calcolo complessivo della moltiplicazione, da quali operazioni è interessato ogni elemento?* Se lo guardiamo, non solo dobbiamo fare attenzione al prodotto, ma poi lo dobbiamo sommare con tutti quelli che stanno sopra e con tutti i riporti che eventualmente vengono da destra.

Da questa considerazione, è stata creata una cella che è in grado di fare quest'operazione a livello del singolo elemento della matrice. Se guardiamo la cella, sulla riga orizzontale c'è y_1 . Se prendiamo y_2 della matrice, va su tutta la riga, sulla riga sopra c'è sempre y_1 , **il valore y è distribuito alla stessa maniera su tutte le righe, mentre il valore x cammina sulle diagonali**. Avremo tutte queste celle uguali, la y la prendono tutte e all'interno della cella la y serve sia in ingresso alla and insieme alla x (per realizzare il prodotto); poi prende in ingresso la somma dello stadio precedente (quella che viene da sopra) ed effettua una somma con un FA tra il prodotto generato e la somma presa da sopra. La somma generata viene presentata allo stadio successivo.

L'architettura complessiva è quella in figura 6.24, è **un'architettura regolare**, migliore rispetto a quella di somma per righe e diagonali perché queste prevedevano stadi diversi per fare le somme

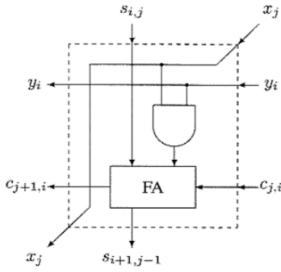


Figura 6.23: Cella MAC.

e i prodotti, mentre qui **il singolo elemento fa sia il prodotto che le somme**. Ha una propagazione di riporti, quindi comunque ci sarà un certo ritardo, ma ha il vantaggio di condensare tutto il comportamento in una cella. **Osservazione:** definiamo una cella, tra quella di sopra e

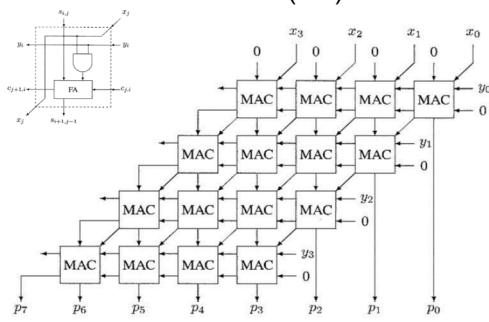


Figura 6.24: Moltiplicatore MAC.

sotto *c'è sempre una struttura iterativa, con il for possiamo fare l'istanziazione spaziale di tutti gli oggetti*. Se i livelli di integrazione non sono molto elevati non riusciamo a mettere tutto in un for.

6.2.6 Moltiplicatore di Robertson

Il moltiplicatore di Robertson è un moltiplicatore sequenziale, per capirne il funzionamento, analizziamo come si effettua la moltiplicazione manualmente, passo per passo, cominciando dall'algoritmo manuale di moltiplicazione binaria. Se dobbiamo moltiplicare 1010 per 1101, prendiamo il moltiplicando, lo moltiplichiamo per la cifra i -esima del moltiplicatore e poi dobbiamo sommare delle righe che *non hanno sempre lo stesso peso ma sono ogni volta shiftate di una posizione a sinistra*. Dunque, prima generiamo le righe e poi ne facciamo la somma, se entrambi gli operandi hanno dimensione N allora il prodotto ha dimensione $2N$. Ad ogni passo prendiamo la cifra i -esima, la moltiplichiamo per il moltiplicando, poi ogni volta moltiplichiamo per un fattore 2^i , ovvero facciamo uno shift a sinistra di i posizioni. Otteniamo la seguente sommatoria:

$$\sum_{j=0}^{n-1} x_j 2^j Y$$

Potremmo fare questa procedura manuale in formula, otteniamo la sommatoria presente a destra. Ogni termine è la cifra j -esima del moltiplicatore, per il moltiplicando Y , moltiplicato per 2^j . Vediamo cosa possiamo fare per modificare la procedura manuale in qualcosa di iterativo. Non conserviamo tutte le righe e poi facciamo la somma, come visto nelle architetture parallele, ma a ogni passo cominciamo a calcolare la somma intermedia: consideriamo come somma parziale

iniziale tutti 0 e poi sommiamo il prodotto $2^j x_j$, il prodotto ogni volta è shiftato di una posizione a sinistra. La somma non la facciamo alla fine per n righe ma iterativamente per ogni riga. Se dobbiamo trasformarlo in un algoritmo, ***al passo i-esimo il numero di shift dipende da i, non è fisso.*** Possiamo fare ancora meglio, invertiamo l'ordine della somma e degli shift, facciamo prima la somma parziale e poi lo shift a destra di una posizione:

$$P_i = P_i + x_i Y$$

$$P_{i+1} = P_i 2^{-1}$$

Osserviamo la figura 6.25, in rosso troviamo il prodotto tra la cifra i-esima del moltiplicatore e il moltiplicando, quindi se la cifra è 1 è pari al moltiplicando, se la cifra è 0 è tutti 0. P0 è il prodotto parziale al passo 0, dunque il prodotto parziale iniziale, pari alla stringa di 0. Dato che la prima cifra del moltiplicatore è 1, dobbiamo semplicemente sommare 1010. Normalmente alla successiva cifra considereremmo il valore (in questo caso 0000) shiftato di una posizione a sinistra. *Ora invece prendiamo la somma di 0000000 con 1010, shiftato di una posizione a destra (P1),* questo equivale ad allineare il successivo prodotto sotto le cifre giuste, per fare la somma. Facendo così ci ritroviamo che, con l'ultimo shift, abbiamo un prodotto che possiamo vedere sugli 8 bit in blu.

Y	1010		
X	1101		
		0000 0000	P0
		1010	
		0000 1010	ADD;shift
		000 0101	0 P1
		0000	
		000 0101	0 ADD;shift
		00 0010	10 P2
		1010	
		00 1100	10 ADD;shift
		0 0110	010 P3
		1010	
		1 0000	010 ADD;shift
		1000	
		0010	
A			
Q			

Figura 6.25: Esempio di moltiplicazione.

Cerchiamo di immaginare il datapath necessario a una macchina che opera in tal modo. A livello di registri ne serve uno per il moltiplicando, uno per il moltiplicatore e in teoria uno di lunghezza doppia per il risultato. Cerchiamo di ottimizzare le risorse: mettiamo il moltiplicatore in un registro da 4 bit, ad esempio, e il moltiplicatore invece in un registro che ha già lunghezza doppia, quindi 8 bit. Se pensiamo ai bit in blu in figura 6.25, è un registro in tutto grande 8 bit che si riempie piano piano, con A e Q. All'inizio esso potrebbe contenere la somma parziale che sono 8 zeri, poi ad ogni iterazione l'ultimo bit, il meno significativo, viene buttato, dunque man mano delle 8 cifre ne servono meno. ***Anziché usare un registro a parte per mantenere gli altri dati che ci servono, sfruttiamo il registro in cui le cifre a ogni iterazione si liberano.*** La porzione meno significativa del registro, Q, all'inizio è vuota e viene riempita man mano che avvengono gli shift, alla fine avremo sugli 8 bit che prima contenevano tutti 0 (prodotto parziale iniziale, P0) un pezzo di A e di Q che infine fanno il risultato. Possiamo fare questo algoritmo in modo sequenziale, inizializziamo P, poi moltiplichiamo il moltiplicando per una cifra, una volta 0 e

una 1 (possiamo anche usare un multiplexer per questo, che una da volta da 0 e una volta da il moltiplicando) poi abbiamo uno shift a destra e una somma, che otteniamo tramite un addizionatore. Come datapath allora avremo un sommatore, un multiplexer, che seleziona lo 0 o il moltiplicando, e il risultato della somma che deve andare in un registro che sia in grado di shiftare, uno shift-register.

Il problema dell'algoritmo è che funziona solo per i numeri positivi, quando abbiamo dei numeri negativi in *complementi a due*, ogni cifra del numero ha un peso, dato dalla notazione posizionale, ma la cifra più significativa ha un peso particolare. In notazione posizionale, se abbiamo n bit la cifra più significativa è quella moltiplicata per 2^{n-1} , **se il numero è negativo il peso della cifra significativa è negativo**, $-1 \cdot 2^{n-1}$, questo influisce perché l'algoritmo scandisce una cifra per volta ma non possiamo trattare una cifra che ha peso negativo come una cifra di peso positivo. In complementi a due, se dobbiamo fare $-X$ dobbiamo complementare tutti i bit e aggiungere 1. Il singolo bit del numero, in complementi, è 1 meno la cifra i-esima, ovvero il complemento:

$$-X = x'_{n-1}x'_{n-2}\dots x'_1x'_0 + 000\dots 1 \pmod{2^n} \text{ dove } x'_i = 1 - x_i \pmod{2}$$

Dunque otteniamo che

$$-X = (111\dots 11 - x_{n-1}x_{n-2}\dots x_1x_0) + 000\dots 01 \pmod{2^n}$$

Se X è positivo, dunque la cifra di peso più significativo vale 0, possiamo scrivere x tramite la seguente sommatoria:

$$\sum_{i=0}^{n-2} x_i \cdot 2^i$$

Dove i va da 0 a n-2 perché l'n-1 è 0. Se invece è negativo non possiamo fare così, il primo bit del segno sappiamo sarà 1 ma non sappiamo il suo peso, lo dobbiamo ricavare. Abbiamo determinato $-X$, dunque prima lo consideriamo con lo 0 davanti e poi sommiamo $10\dots 0$, perché sappiamo che quella cifra vale 1 (la più significativa, che determina il segno)

$$\begin{aligned} -X &= 111\dots 11 - (0 \cdot x_{n-2}\dots x_1x_0 + 100\dots 00) + 000\dots 01 = \\ &= (111\dots 11 - 100\dots 00 + 000\dots 01) - x_{n-2}\dots x_1x_0 = \\ &= 100\dots 00 - xx_{n-2}\dots x_1x_0 = \\ &= 2^{n-1} - x_{n-2}\dots x_1x_0 \end{aligned}$$

Per avere X in complementi moltiplichiamo per il segno meno, ottenendo:

$$X = -2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i$$

In complementi, un numero positivo si scrive solo con la sommatoria, un numero negativo invece la cifra più significativa ha peso -1. Abbiamo dovuto fare ciò perché l'algoritmo sequenziale guarda una cifra per volta per cui dobbiamo tener conto del peso negativo quando arriviamo alla cifra più significativa. Ad ogni passo moltiplichiamo tutto il moltiplicando per una singola cifra del moltiplicatore, *ci accorgiamo che il moltiplicatore era negativo solo all'ultimo passo*, dobbiamo fare qualcosa se ce ne accorgiamo, se invece il moltiplicando è negativo e il moltiplicatore è positivo, il prodotto parziale sarà sempre negativo, perché è una quantità negativa per una cifra. Appena ci accorgiamo che stiamo moltiplicando il moltiplicando negativo per una cifra i-esima diversa da 0, da quel momento in poi il risultato sarà negativo, **il bit più significativo del risultato parziale dev'essere 1**, per ricordarci ciò possiamo usare un latch. Ci sono due situazioni, da un lato se il moltiplicando è negativo ce ne accorgiamo subito; se invece lo è il moltiplicatore, lo sappiamo alla fine. In questo caso allora, ad ogni passo facevamo somma e shift, la somma perché erano cifre intrinsecamente positive, quando arriviamo all'ultima cifra invece dobbiamo fare la sottrazione (all'ultimo passo), al posto della somma, questo è il **passo di correzione**.

L'unità di controllo le sequenze di controllo all'unità operativa così da effettuare correttamente la moltiplicazione. In particolare, se pensiamo a come viene eseguita l'operazione, ad ogni passo (ad ogni cifra del moltiplicatore) si effettua l'addizione e si procede al right shift. Arrivata all'ultimo passo, si effettua il passo di correzione, necessario nel caso in cui il risultato è negativo. X e Y sono due numeri interi, in particolare possono essere entrambi positivi, entrambi negativi (quindi codificati in complementi a due) o di segno discorde, in quest'ultimo caso il risultato dev'essere negativo, quindi va considerato codificato in complementi a due. Se il moltiplicatore, X, è negativo, il risultato dev'essere negativo ma lo scopriamo solo all'ultimo passo, prima del quale abbiamo sempre effettuato l'addizione, presupponendolo positivo. Dunque, all'ultimo passo dev'essere fatto il passo di correzione per far sì che il risultato diventi negativo. In particolare, abbiamo 4 possibili casi da dover gestire:

- Se **X e Y sono entrambi positivi** e facciamo una moltiplicazione le somme sono sempre positive, opportunamente shiftate, non dobbiamo correggere niente;
- Se **X è positivo ma Y (moltiplicando) è negativo**, lo sappiamo subito, ogni volta che prendiamo la Y e la moltiplichiamo per una cifra non nulla il prodotto parziale è negativo, scriviamo 1 in un flip flop (F visto nell'unità operativa) e questo rimane il valore del segno, fino alla fine.
- Se **Y è positivo e X è negativo** fino all'ultima cifra non ce ne accorgiamo, facciamo le somme, all'ultima cifra ci accorgiamo che ha un peso negativo, invece di fare la somma facciamo la sottrazione (*passo di correzione*).
- Se **X e Y sono entrambi negativi**, dall'inizio per noi il prodotto parziale era negativo, solo alla fine ci accorgiamo che in realtà è positivo per cui invece di fare la somma facciamo la sottrazione, passo di correzione.

Tutto l'algoritmo prevede di fare moltiplicazione, somma e shift ad ogni passo, solo all'ultimo dobbiamo controllare se fare o meno la correzione.

Nota 6.1: Tempificazione

Non possiamo realizzare la moltiplicazione in un solo colpo di clock perché al primo colpo facciamo il prodotto, in quello dopo la somma (potremmo farle anche nello stesso clock più lungo), ma lo shift va fatto sicuramente dopo, altrimenti non saprebbe quale valore modificare; altrimenti potremmo far sì che i blocchi lavorino su fronti differenti.

In figura 6.26, vediamo l'architettura complessiva. Serve un contatore che a seconda delle dimensioni di X e Y sarà di 8, 16 o 32. A destra c'è Y che mettiamo nel registro M, ad ogni iterazione dobbiamo fare il prodotto di Y con il bit i-esimo di X, che può essere un 1 o uno 0, piuttosto che fare la and bit a bit usiamo un multiplexer. *In Q carichiamo all'inizio il valore del moltiplicatore, mentre A all'inizio è vuoto.* Nel disegno, sono due registri diversi ma si può realizzare come un unico registro che possiamo usare come shift register. In testa ad A c'è l'uscita del flip flop, serve perché se lo facciamo strutturale a un certo punto dobbiamo fare lo shift a destra, in testa che mettiamo? Fermo restando che in VHDL possiamo fare la concatenazione con 1, dal punto di vista strutturale dobbiamo mantenerlo da qualche parte e dobbiamo caricarlo dall'uscita di F alla testa di A. L'uscita dell'addizionatore va direttamente in A, in ingresso all'addizionatore va o M o 0, in un operando, e la stessa A, poi l'adder ha il carry_in che è collegato al segnale subtract della UC. Con la XOR, l'adder e il segnale di sub siamo in grado di realizzare un sommatore/sottrattore come visto precedentemente. **Il contatore va in ingresso alla UC, perché disciplina il passo di moltiplicazione a cui siamo, all'ultimo va fatta la correzione:** è la UC a dare il segnale subtract. La UC dà anche delle uscite verso il counter perché gli deve dare il conteggio.

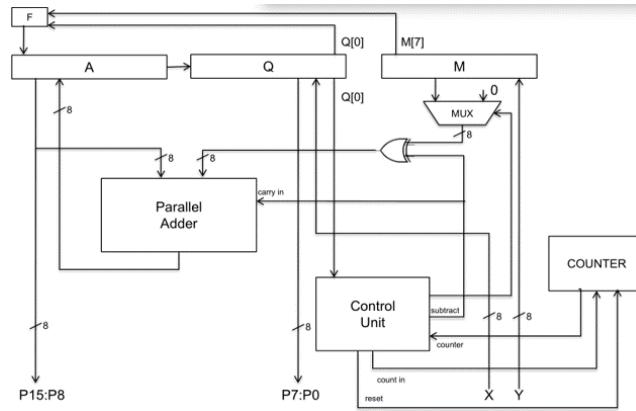


Figura 6.26: Architettura del moltiplicatore Robertson.

Una sequenza algoritmica del moltiplicatore potrebbe essere quella riportata in figura 6.27. Mettiamo in evidenza i passi di add e shift, ad ogni passo facciamo la somma. Ovviamente detto in termini algoritmici abbiamo proprio una moltiplicazione di M per $Q(0)$, non c'è il multiplexer. Facciamo un passo di somma e aggiustiamo F che mantiene il segno, basta che il moltiplicando sia negativo per dire che da quel momento in poi sarà negativo, bisogna vedere $M[7]$, il bit più significativo. Se è alto e lo stiamo moltiplicando per una cifra diversa da 0, era già alto da prima (F) allora F è alto ($F := (M[7] \text{ and } Q[0]) \text{ or } F;$). Facciamo Rshift (shift a destra) e incrementiamo il contatore. Ogni volta vediamo se è l'ultimo passo, se lo è facciamo la correzione, perché tanto se $q_0 = 0$, perché il numero era positivo non faremo niente, altrimenti facciamo la sottrazione. Come ultimo passo manteniamo il segno che avevamo, facciamo l'ultimo shift.

```

2CMultiplier: (in:INBUS; OUT:OUTBUS)
register A[7:0],M[7:0],Q[7:0],COUNT[2:0],F;
bus INBUS[7:0],OUTBUS[7:0];

BEGIN:   A:=0,COUNT:=0,F:=0,
INPUT:   M:=INBUS,Q:=INBUS;
ADD:     A[7:0]:=A[7:0] + M [7:0] x Q[0],
         F:=(M[7] and Q[0]) or F;
RSHIFT:  A[7]:= F, A[6:0],C:= A.Q[7:1],
INCREMENT: COUNT:=COUNT+1
TEST:    if COUNT<7 then go to ADD;
SUBTRACT: A[7:0]:=A[7:0]-M[7:0]xQ[0]; {l'ultima op è sempre SUB}
RSHIFT:  A[7]:= A[7], A[6:0],Q:= A.Q[7:1];
OUTPUT:  OUTBUS:=Q; OUTBUS:=A;
END 2CMultiplier;

```

Figura 6.27: Algoritmo di Robertson.

6.2.7 Moltiplicatore di Booth

Dato il numero X , rappresentato in complementi a due, costruiamo una successione, nel seguente modo:

$$\begin{aligned}
y_0 &= -x_0 \\
y_1 &= -x_1 + x_0 \\
&\dots \\
y_{n-1} &= -x_{n-1} + x_{n-2}
\end{aligned}$$

Moltiplicando il termine i -esimo di y , y_i , per la potenza 2^i , sommando ogni termine, otteniamo un'espressione del genere:

$$y_{n-1} \cdot 2^{n-1} + \dots + y_1 \cdot 2 + y_0 = -x_{n-1} \cdot 2^{n-1} + x_{n-2} \cdot 2^{n-2} + \dots + x_1 \cdot 2 + x_0$$

che è possibile scrivere in maniera più compatta, come segue.

$$Y = \sum_{i=0}^{n-1} y_i \cdot 2^i = -2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i = X$$

Per come è costruita, la **successione y non è binaria**, potrebbe valere 0 o -1 già dalla prima: stiamo rappresentando un numero binario in complementi a due in un numero le cui cifre sono terne, questa codifica, di Booth, ci aiuta nelle moltiplicazioni.

Consideriamo un generico numero X espresso in binario, prendiamone raggruppamenti di due bit alla volta e ragioniamo sulla sua codifica di Booth.

Per vedere il numero y corrispondente ad ogni due bit di X, aggiungiamo prima di tutto uno 0 finale per poter considerare tutte le possibili coppie e poi usiamo la tabella 6.3 che riassume la successione. Se avessimo un moltiplicatore X a 32 bit di tutti 1, con il moltiplicatore di Robertson,

Tabella 6.3: Tabella per la codifica di Booth.

x_{n-1}	x_{n-2}	y_{n-1}
0	0	0
0	1	1
1	0	-1
1	1	0

ad ogni passo facciamo, somma shift e solo all'ultimo la sottrazione, al contrario nel moltiplicatore di Booth, adoperando questa codifica, è possibile riconoscere tre casi:

- $x_{n-1} \cdot x_{n-2} = 00$ oppure **11**: nel passo di moltiplicazione occorre fare solo lo shift a destra di una posizione;
- $x_{n-1} \cdot x_{n-2} = 01$: nel passo di moltiplicazione occorre fare prima l'addizione e poi lo shift a destra, questo vorrà dire che il segnale di subtract in uscita dall'UC deve essere 0;
- $x_{n-1} \cdot x_{n-2} = 10$: nel passo di moltiplicazione occorre fare prima la sottrazione e poi lo shift a destra, questo vorrà dire che il segnale di subtract in uscita dall'UC deve essere 1.

Il moltiplicatore di Booth è particolarmente efficiente quando le stringhe da codificare presentano molti 1 o molti 0, perchè questo vuol dire che in molti casi dovremo fare solo lo shift, senza né somma né sottrazione. L'architettura di Booth (figura 6.28) è la stessa di Robertson, ma l'unità

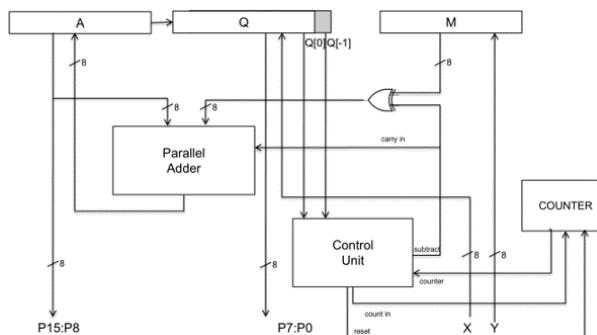


Figura 6.28: Architettura del moltiplicatore di Booth.

di controllo, invece di vedere solo l'ultima cifra del moltiplicatore, prende le ultime due, sulla

base delle quali decide se fare uno shift a destra, addizione e shift, sottrazione e shift. Dobbiamo aggiungere il bit grigio, lo 0, perché guardiamo la coppia, data da uno 0 iniziale che mettiamo noi. La parte operativa è la stessa. Usando il registro a scorrimento non variamo i due bit in ingresso alla UC, *lo scorrimento consente di non utilizzare un bus, poiché garantisce di vedere sempre i bit corretti.*

6.3 Divisori

Nel caso dei divisori affrontiamo solo lo studio di quelli sequenziali.

Partiamo da quello che sappiamo fare con la procedura manuale. **Se abbiamo un dividendo D su m bit e un divisore V su n bit**, quando facciamo la divisione intera il risultato è dato da due stringhe in uscita, **il quoziente e il resto**, li chiameremo rispettivamente Q e R. In generale come parallelismo degli output rispetto agli input, se il divisore è al più su n bit, il resto sarà al più pari a $V-1$, ovvero al più su n bit. Il massimo quoziente invece si può avere quando abbiamo il minimo divisore (cioè $V=1$). In realtà, **in generale i bit del quoziente saranno al massimo $m-n+1$.**

Quando facciamo le divisioni consideriamo i numeri “veri”, eventuali 0 in testa non li consideriamo, cominciamo ad abbassare le cifre vere. Nel campo binario non è banale, se un numero è su 10 bit però in realtà è 5, 101, abbiamo molti numeri a 0 in testa, se pensiamo ad un algoritmo iterativo che abbassa le cifre e fa i confronti dobbiamo stare attenti agli 0 in testa. Vediamo un esempio: il

$$\begin{array}{r}
 1\ 1\ 1\ 0\ 1\ 1 \\
 1\ 0\ 1 \\
 \hline
 1\ 0\ 0 \\
 0\ 0\ 0 \\
 \hline
 1\ 0\ 0\ 1 \\
 1\ 0\ 1 \\
 \hline
 1\ 0\ 0\ 1 \\
 1\ 0\ 1 \\
 \hline
 1\ 0\ 0
 \end{array}
 \quad
 \begin{array}{r}
 1\ 0\ 1 \\
 \hline
 1\ 0\ 1\ 1
 \end{array}$$

Figura 6.29: Esempio divisione in binario.

divisore è a 3 bit, allineiamo il divisore sotto il dividendo e abbassiamo le prime 3 cifre. *Quanto può essere grande al più il quoziente? Quando facciamo la sottrazione al più otteniamo una cifra perché il divisore non può essere contenuto nel dividendo per più di una volta, quindi al massimo abbiamo un 1 nella sottrazione.*

Concentriamoci sui passi che facciamo a mano, ad ogni iterazione: il passo iniziale è allineare il divisore sotto le prime n cifre del dividendo, poi confrontiamo il divisore con il dividendo parziale; se è contenuto scriviamo 1 nella cifra del quoziente e facciamo la sottrazione tra dividendo parziale e divisore. Quello che esce viene riportato sotto e continuamo così, abbassando di volta in volta 3 cifre, finché non troviamo un valore che non può essere espanso. Alla fine ritroviamo il resto R della divisione, le cifre che abbiamo man mano ottenuto rappresentano il risultato Q.

Cerchiamo di trovare l’algoritmo. Innanzitutto, abbiamo **un problema di allineamento del divisore** e poi abbiamo **un problema di determinazione della cifra i-esima di Q**, idealmente dobbiamo comparare (tramite un comparatore) il divisore con il dividendo parziale per capire quale dei due è maggiore, se il dividendo parziale è maggiore, allora la cifra i-esima di Q è 1. Dopo, facciamo la sottrazione, ovvero calcoliamo il nuovo dividendo parziale.

Al passo i-esimo prendiamo R_i , il dividendo parziale, e sottraiamo il prodotto $Q_i * V$ ovvero la cifra che abbiamo pensato di aver generato: se il divisore è contenuto nel dividendo, Q_i è 1, quindi

sottraiamo veramente V , altrimenti $Q_i=0$, sottraiamo 0, quindi non sottraiamo niente. Per la precisione, noi **sottraiamo** $Q_i \cdot 2^i \cdot V$, perché ogni volta dobbiamo posizionare il divisore parziale nel posto giusto. *Dobbiamo fare uno shift verso destra, sempre di posizioni diverse, a seconda del passo i avremo uno shift di i posizioni.*

Vediamo come rende iterativo l'algoritmo. Consideriamo un esempio, abbiamo 7 bit veri per

D	1 1 1 1 0 1 1	1 0 1 0	V
R0=D-D ₆ D ₅ D ₄	1 1 1 1 0 1 1	1 1 0 0	
conf	q ₃ =1		
V*=2 ³ *V	1 0 1 0		
R1=R0-V*q ₃	0 1 0 1 0 1 1		
conf	q ₂ =1		
V*=2 ² *V	1 0 1 0		
R2=R1-V*q ₂	0 0 0 1 1 1		
conf	q ₁ =0		
V*=2 ¹ *V	1 0 1 0		
R3=R2-V*q ₁	0 0 1 1		
conf	q ₀ =0		
V*=2 ⁰ *V	1 0 1 0		
R4=R3-V*q ₀	0 0 1 1		
			R

Consideriamo V e Q espressi su n bit e D su (2n-1) bit

Figura 6.30: Esempio divisione in binario.

D e 4 per V. Abbassiamo le prime 4 cifre, dato che il valore è contenuto, generiamo il bit 1 e facciamo veramente la sottrazione, allineiamo quindi sotto 1010, ovvero $Q_i * V$, facciamo la sottrazione e otteniamo un certo numero. Al passo successivo facciamo lo stesso, ma il dividendo parziale sarà la porzione cerchiata (tratteggiato in rosso), continuiamo con questo. Se è contenuto generiamo un altro 1, quindi $Q_2=1$, allineiamo di nuovo il divisore e facciamo la sottrazione. Nel terzo caso abbiamo un numero che non contiene il divisore, generiamo 0 e riportiamo il valore sotto, perché non facciamo la sottrazione. Poi andiamo avanti. Alla fine otteniamo R e Q, come mostra la figura 6.30.

Come possiamo migliorare queste iterazioni per fare qualcosa di più omogeneo ad ogni iterazione? Facciamo un confronto per determinare Q, allineiamo sotto la posizione giusta (Q^*V e poi shift di i a destra), poi facciamo la sottrazione. Possiamo **fare prima il left shift: non allineiamo il divisore sotto il dividendo parziale ma facciamo in modo che il dividendo parziale sia sempre allineato al posto giusto**, come per il moltiplicatore, solo che ora lo shift è a sinistra invece che a destra. Poi **facciamo la comparazione, idealmente con un comparatore, e poi la sottrazione**. In questo modo le operazioni di shift saranno sempre le stesse, ad ogni passo. Vediamo rispetto all'esempio di prima (ci riferiamo adesso alla figura 6.31): partiamo dallo shift

D=R0	1 1 1 1 0 1 1	1 0 1 0	V
2R0	1 1 1 1 0 1 1 -	1 1 0 0	
conf	1 0 1 0		
2R0-V*=R1	0 1 0 1 0 1 1 -		
2R1	1 0 1 0 1 1 - -		
conf	1 0 1 0		
2R1-V*=R2	0 0 0 0 1 1 - -		
2R2	0 0 0 1 1 - - -		
conf	0 0 0 0		
2R1-V*=R3	0 0 0 1 1 - - -		
2R3	0 0 1 1 - - - -		
conf	0 0 0 0		
2R1-V*=R4=R	0 0 1 1 - - - -		
			A=R Q

Figura 6.31: Esempio divisione in binario.

a sinistra, cioè $2^i R_0$. Poi facciamo confronto e sottrazione, abbiamo un valore che viene shiftato

all'inizio della prossima iterazione, perdiamo quello che stava a sinistra perché non ci interessa, abbassiamo le cifre giuste e facciamo la sottrazione. Possiamo anche identificare dal punto di vista architettonale, in un datapath, quali sono gli elementi necessari: abbiamo dividendo e divisore su un tot di bit, poi R e Q, altri due registri. In realtà, non serve un registro per ciascuno dei due. All'inizio abbiamo un dividendo che occupa tutto lo spazio (m), il primo passo è uno shift, per cui se mettiamo D in un registro perdiamo subito una posizione. Ad ogni passo facciamo uno shift e lasciamo la posizione meno significativa, sulla destra, vuota. Serve un registro da $2n$ che contiene il dividendo all'inizio e poi alla fine conterrà Q e anche R, più un registro di n bit che contenga il divisore. È chiaro che saranno necessari anche sottrattore e unità di controllo. Analizziamo questo altro esempio (6.32), applicando sempre lo stesso algoritmo: quando facciamo la sottrazione, il divisore non è contenuto nel dividendo parziale, questo vuol dire che non dobbiamo fare la sottrazione. Quando facciamo lo shift a sinistra di una posizione, perdiamo un bit

D=R0	1 0 0 0 0 1 1	1 0 0 1
2R0	1 0 0 0 0 1 1 -	0 1 1 1
conf	0 0 0 0	
2R0-V*=R1	1 0 0 0 0 1 1 -	
2R1	1 0 0 0 0 1 1 -	
conf	1 0 0 1	
2R1-V*=R2	0 1 1 1 1 1 -	
2R2	1 1 1 1 1 - -	
conf	1 0 0 1	
2R1-V*=R3	0 1 1 0 1 - - -	
2R3	1 1 0 1 - - -	
conf	1 0 0 1	
2R1-V*=R4=R	0 1 0 0 - - -	

A=R Q

Figura 6.32: Esempio divisione in binario.

significativo, questo errore è dovuto a come stiamo generando la cifra i-esima del quoziente. Stiamo ancora supponendo di poter determinare Q_i con un comparatore, mediante cui capire se il divisore è contenuto o meno nel dividendo parziale (cioè siamo ancora legati alla procedura manuale).

Un comparatore di stringhe, date due stringhe in ingresso, deve stabilire quale delle due è maggiore, per fare ciò dobbiamo confrontare tutti i bit delle due stringhe. Ci sono dei modi per velocizzare ciò perché, se abbiamo numeri naturali, basta che li confrontiamo dal bit più significativo e appena troviamo un bit in cui uno ha 1 e l'altro 0, allora quello maggiore è quello di 1, si ottimizza. Se tuttavia le stringhe sono da 32 bit e cambia solo l'ultimo bit, dovremmo confrontare cifra per cifra, con una macchina combinatoria, che richiede più di due bit in ingresso perché in ogni stadio dobbiamo tener conto anche dello stadio precedente e solo alle 32-esima iterazione ci accorgiamo che una è più grande dell'altra. Nel comparatore, si avrebbero gli stessi problemi del ripple carry adder, dobbiamo propagare le informazioni, oltre al tempo impiegato per la decisione. Mettere il comparatore, sequenziale, appesantisce la nostra macchina.

Allora non facciamo il confronto ma la sottrazione, se il risultato è positivo allora il divisore era contenuto, se è negativo invece non era contenuto. La macchina che fa la sottrazione è già presente nel datapath, inoltre se la cifra i-esima del quoziente è proprio 1, abbiamo anche già fatto la sottrazione, che comunque avremmo dovuto fare. Il problema si riscontra quando la cifra i-esima del quoziente è 0, vorrebbe dire che non dovevamo sottrarre, noi però per comparare lo abbiamo fatto, quindi dobbiamo tornare indietro. Sulla base di ciò si fondano le due macchine.

6.3.1 Modalità restoring

Quando facciamo la sottrazione, per comporre il nuovo dividendo parziale calcoliamo:

$$R_{i+1} = 2 \cdot R_i - V$$

ci accorgiamo di aver sbagliato nel calcolo della cifra i -esima del quoziente. Vuol dire che ad R_i dobbiamo sommare V . In questa modalità, **questo passaggio di restoring viene effettuato nello stesso passo in cui si effettua la sottrazione**: al passo i -esimo l'algoritmo fa sempre la sottrazione, controlla il segno, se è negativo mette Q_i a 0 e fa subito la somma. **Il sommatore viene usato 2 volte, in un singolo passo, fa prima la sottrazione e se è sbagliato fa la somma.** La durata della singola iterazione non è fissa, dobbiamo aspettare due volte l'addizionatore, che poi è la parte più critica del datapath.

6.3.2 Modalità non restoring

Se vogliamo fare qualcosa più intelligente, al passo i -esimo facciamo la sottrazione, Q_i è 0, per cui teoricamente R_{i+1} va aggiustato sommando V . Utilizziamo l'espressione di R_{i+1} per sostituirlo in quello che faremmo al passo $i+2$, iterazione successiva. L'iterazione successiva è data dal dividendo precedente meno V , andiamo a sostituire dentro ciò quello di prima, abbiamo $2(R_{i+1}+V)-V$, perché avevamo sbagliato quindi invece di R_{i+1} dobbiamo avere $(R_{i+1}+V)$. Facciamo le semplificazioni e ci accorgiamo che **non al passo i , ma in quello successivo, dobbiamo fare la somma con V . Se al passo i -esimo abbiamo generato $Q_i=0$, al passo successivo, $i+1$, invece di fare la**

$$R_i := R_i + V \quad (a)$$

$$R_{i+1} := 2R_i - V \quad (b)$$

Le due operazioni possono essere fuse:

$$i: \quad \Delta_i = R_{i+1} = 2R_i - V < 0 \Rightarrow Q_i = 0 \text{ effettuo il restoring:}$$

$$R_{i+1} = R_{i+1} + V$$

$$i+1: \quad \Delta_{i+1} = R_{i+2} = 2R_{i+1} - V = 2(R_{i+1} + V) - V = 2R_{i+1} + 2V - V = \\ = 2R_{i+1} + V$$

Figura 6.33: Calcolo dividendo parziale.

sottrazione dobbiamo fare la somma: ad ogni stadio allora guardiamo il Q generato allo stadio precedente e facciamo la somma o la sottrazione, ma mai entrambe. Il Q generato all'iterazione precedente va nella UC che darà, a seconda del suo valore, il segnale di subtract o meno, per il passo successivo. In questo modo abbiamo compattato l'operazione di restoring e facciamo sì che ad ogni operazione venga fatta una sola operazione di somma algebrica. L'architettura (figura 6.34) è molto simile a Booth, in figura non è completa, dice solo A e Q a n bit, M a n bit, però **serve un flip flop aggiuntivo in testa ad A per il segno**. A e Q sono concatenati tramite uno shift, c'è una Control Unit che dà il segnale di subtract al sommatore. Non viene mostrato il contatore per contare il numero di iterazioni, viene considerato interno all'unità di controllo. Vediamo l'algoritmo dell'unità di controllo. In A all'inizio mettiamo la prima metà

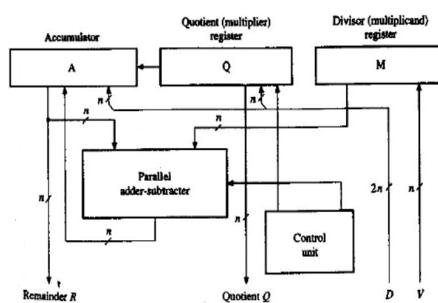


Figura 6.34: Architettura non restoring per interi con segno.

del dividendo, con uno 0 in testa, in Q mettiamo la seconda metà del dividendo, in M il divisore ed inizializziamo S , che sarebbe il flip flop per il segno, a 0. Poi facciamo sempre un left shift di

tutta la concatenazione AQ, il primo bit del registro resterà vuoto e lo riempiamo ad ogni passo: **se il bit di segno è 0 fa la sottrazione, se è 1 fa la somma.** Il bit si prende dall'iterazione precedente. Alla prima iterazione S è 0, per cui partiremo sempre con una sottrazione.

Fatto ciò settiamo Q (il bit Q(0)), che conterrà la cifra i-esima del quoziente, come not S: se il segno

```

NRDivider:      (in:INBUS; OUT:OUTBUS)
register S,A[n-1:0],M[n-1:0],Q[n-1:0],COUNT[log2n:0];
bus INBUS[n-1:0], OUTBUS[n-1:0];
COUNT:=0;S:=0;
A:=INBUS {carico la prima metà del dividendo D (0 in testa)}
Q:=INBUS {carico la seconda metà del dividendo D}
M:=INBUS; {divisore V}

LSHIFT:          S.A.Q[n-1:1]:=A.Q; {la prima volta S è 0, e dopo lo shift è ancora 0}

SUB:             if S==0 then
                  S.A:=S.A-M;
                else
                  S.A:=S.A+M;
                endif

SUM:              S.A:=S.A+M;

SETq:            Q[0]:=not S;
                  COUNT:=COUNT+1;

COUNT_TEST:      if COUNT< n then goto LSHIFT;
                  endif

CORRECTION :    if S==1 then
                  S.A:=S.A+M;
                endif

OUTPUT: OUTBUS:=Q, OUTBUS:=A;
END NRDivider;
```

Figura 6.35: Algoritmo divisore non restoring.

era negativo, S vale 1, ma questo vuol dire che il divisore non è contenuto nel dividendo parziale, per cui Qi deve essere 0 (sempre l'opposto di S). Si controlla volta per volta se siamo arrivati all'ultima iterazione, se non è così continuiamo con shift a sinistra e addizione o sottrazione a seconda dei casi. Alla fine, *anche l'ultimo passo potrebbe essere di correzione, in quel caso facciamo l'ultimo restoring finale.*

Capitolo 7

Comunicazione seriale

Esiste la necessità di fare trasmissioni seriali. È importante la convergenza tra chi invia e chi riceve i bit, perché la trasmissione, se non opportunamente sincronizzata, rischia di non far prendere il segnale al ricevitore. Il problema, infatti, non è il trasmettitore, ma il ricevitore che non sa quando inizia la trasmissione. Ci sono due tipologie di comunicazioni: **asincrona** e **sincrona**. Nell'asincrona, la sincronizzazione avviene per ogni carattere. La linea è in uno stato di riposo (attiva alta), prima di poter fare la comunicazione si abbassa per due quanti di tempo. E' chiaro che il trasmettitore debba generare una forma d'onda che sia stata definita con il ricevitore.

Per poter distinguere l'1 dallo 0, la linea sta sempre ad 1, poi vengono messi due bit di 0, eventual-



Figura 7.1: Sequenza della comunicazione seriale asincrona.

mente uno start; creata la sequenza di bit opportuna, comincia la trasmissione dell'informazione, poi ci potrebbe essere un bit di parità ed uno di stop e successivamente linea torna al valore che aveva. Il ricevitore, saputa la frequenza di questi bit che compongono il carattere, sovra-campiona la linea in modo da capire se ha avuto effettivamente uno 0 o un 1. Il sistema nel suo complesso ha un registro a scorrimento che scorre con una certa velocità ed un altro con la stessa velocità dall'altro lato, solo che in mezzo è necessario un campionatore perché per un bit inviato l'altro deve prendere più campioni.

Quali possono essere gli errori di questa comunicazione? Se il sistema fosse perfetto, potremmo non sovra-campionare la linea, ma dobbiamo perché i clock si sfasano: pur essendo sicuri che i segnali di clock dei due sistemi viaggino alla stessa frequenza o a frequenze multiple (**isocronia**), non è detto che abbiano la stessa fase. *Uno sfasamento comporta un accumulo di ritardi che causano la perdita di campioni significativi.*

Ricevuto il carattere e fatti i controlli, il carattere viene copiato in un registro, può succedere che l'entità continui a mandare altri caratteri, cioè due bit bassi dopo la linea alta invia un nuovo carattere. Se il ricevitore non riesce a prendere il bit precedente, legge il nuovo e sovrascrive il vecchio carattere con il nuovo. Se due messaggi sono copiati e nessuno li ha letti si crea un **errore di overrun**, chi legge non è così “bravo” da prendere il segnale che serve. Un altro **errore** è quello di **parità**: se, per esempio, l'informazione contiene 3 bit alti, il bit di parità deve essere 1 (perché $3+1=4$), se invece vogliamo fare il controllo di disparità deve essere messo a 0, se assume un valore

diverso ci sarà un problema sulla linea di trasmissione per cui i bit si sono corrotti (ci potrebbe essere rumore sul canale) per un quanto di tempo significativo, dato che comunque abbiamo sovraccampionato la linea. Ci potrebbe essere l'errore anche se il bit di parità è corretto, cioè quando l'errore è su due bit o se proprio il bit di parità è stato alterato. C'è anche l'**errore di framing**, dopo il campionamento presa l'informazione ci aspettiamo di prendere un valore alto (per indicare il termine della trasmissione e quindi il ritorno della linea in idle), se prendiamo il bit basso vuol dire che siamo andati oltre, cioè il clock è andato troppo veloce non avendo trovato il tappo della sequenza.

Questa comunicazione seriale descritta è quella di tipo asincrono, in cui dopo la trasmissione di ogni carattere la linea torna in stato di riposo. Nel caso di quella **sincrona**, la **trasmissione fun-**

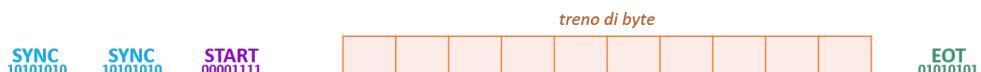


Figura 7.2: Sequenza della comunicazione seriale sincrona.

ziona con il concetto di messaggio, non di carattere, *per cui la sincronizzazione avviene ad ogni messaggio*, viene mandato prima un segnale di sync, poi un altro e poi una serie di caratteri fino a quando non si hanno nuovamente due messaggi di sync.

Quale dei due protocolli è più efficiente? Quello asincrono è più semplice perché la sincronizzazione avviene per ogni carattere, mentre nel secondo caso avviene ad ogni messaggio quindi è un po' più complesso mantenere la sincronizzazione. Se definiamo un **fattore di efficienza**, che indica i bit utili, nel caso del byte dobbiamo trasmettere 8 caratteri, ma per trasmettere 8 caratteri dobbiamo in realtà inviarne $8+2+1+1+2$ (2 pari a 0 per tenere la linea attiva bassa, 1 bit di stop, 1 di parità, altri due bit 0 per la linea bassa), quindi l'efficienza non è $8/8$, cioè non trasmettiamo 8 segnali per 8 caratteri (altrimenti l'efficienza sarebbe 1 e quindi massima), ma $8/14$. Se calcoliamo l'efficienza della trasmissione sincrona, mandiamo 100 byte, quindi $100*8$ bit, a fronte di questi mandiamo $(100+4)*8$, quindi l'efficienza è $100/104$, che è un numero molto grande; **la comunicazione sincrona è più efficiente di quella asincrona.**

La trasmissione che vediamo nell'esercizio è asincrona. Si vuole sottolineare che **il sistema di trasmissione del dato asincrono avviene quando già c'è stato il protocollo**. Questo vuol dire che, prima di iniziare la trasmissione, occorre fare l'handshaking, abbiamo un segnale in cui comunichiamo l'inizio della trasmissione, quando il ricevitore accetta la trasmissione, può effettivamente iniziare la trasmissione. In alcuni casi questi segnali potrebbero non esserci, perché *si suppone che, avendo collegate le due entità, siano disponibili a parlare*. Se questa ipotesi è vera, i due blocchi sono accesi e funzionanti, stanno in uno stato di reset e pronti a parlare.

Vediamo come funziona il sistema complessivamente. Per poter funzionare, **il sistema è caratterizzato da tre gruppi di registri, ovvero uno di controllo, uno di stato ed uno di dato, sia se esso è in trasmissione, sia se è in ricezione.**

Se la trasmissione fosse parallela potremmo prendere il dato e farlo arrivare direttamente a destra, ci sarebbe un problema di adattamento delle linee e di handshaking. Nel nostro esercizio, non ci preoccupiamo di vedere in azione il protocollo di handshaking, ma quando parliamo di due generiche entità, per cui l'ipotesi che siano attive, in stato di reset e pronte a comunicare, non è più vera, il sistema per funzionare necessita del protocollo di handshaking. Dove sono i fili del protocollo? Quest'ultimo ha l'*effetto di mettere un'informazione che va nel registro di stato*. Nel registro di controllo, invece, si deve azionare un invio di una risposta. In altre parole, **dal controllo inviamo il segnale alpha, per chiedere all'entità ricevente se vuol parlare, nel registro stato arriva beta (dal ricevitore)**, cioè l'informazione che vuole parlare. Consideriamo solo due segnali per l'handshaking perché non abbiamo voluto fare il protocollo completo, altrimenti ne avremmo dovuti mettere altri

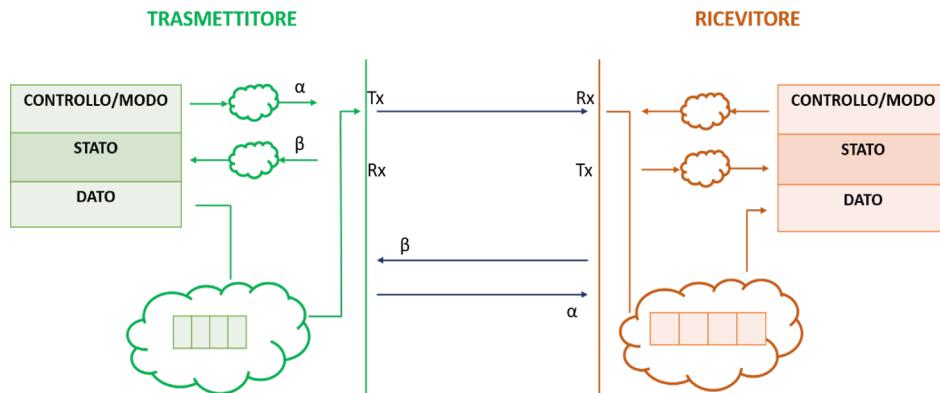


Figura 7.3: Sistema trasmettitore-ricevitore.

due per sapere se c'è la portante (visto che questo protocollo nasce inizialmente per la comunicazione con il modem).

Mettiamo quella nuvola prima di alpha e beta, perché non è detto che scrivendo nel controllo o nello stato scriviamo direttamente nei registri, ma l'operazione finale avrà effetto su essi. **Riceviamo un'informazione dallo stato e la attiviamo sul segnale di controllo.** Se la periferica è la stessa e deve funzionare sia in modalità asincrona che in modalità sincrona, servirebbe anche un registro di modo, perché dobbiamo indicare la modalità della periferica, con quanti bit di start lavora, se la parità è dispari o pari. Per poter risparmiare componenti, il registro di controllo è anche di modo, *la prima volta vediamo quell'indirizzo di memoria per il modo e poi la seconda volta per il controllo.* Se T ed R vogliono parlare tra loro, devono settare lo stesso modo e la stessa configurazione per la comunicazione (la stessa tipologia di parità, il numero di bit di start e così via).

Il processore vede il lato a sinistra, perché scrive sempre byte e legge sempre byte, ma la trasmissione è seriale, vede un solo bit alla volta. Ci sta il bit che trasmette T e il bit che riceve R. Il collegamento tra il mondo seriale delle telecomunicazioni e quello parallelo delle reti di calcolatori è dato proprio dal registro a scorrimento, che sta nei blocchi sottostanti (nuvola verde ed arancione) insieme ad altri eventuali componenti (come il campionatore per il ricevitore). Nel caso del trasmettitore, il registro a scorrimento è a caricamento parallelo, perché viene prelevato il dato dal registro dato ed uscita seriale per poter trasmettere un bit alla volta sul canale, mentre il registro a scorrimento del ricevitore è a caricamento seriale ed uscita parallela per poter posizionare i bit dati nel registro dato del ricevitore.

In questo caso siamo a **livello due della pila ISO/OSI**, con un protocollo ci preoccupiamo di realizzare una trasmissione, dobbiamo essere in grado di caricare il dato che viene trasmesso e visto dal lato a destra, se come utilizzatore a destra mettiamo il visore, vediamo un insieme di caratteri che viene prodotto e che verrà visualizzato.

7.1 Comunicazione tra dispositivi tramite interfaccia seriale: UART

La periferica **UART** prende informazioni in parallelo e le trasmette in seriale (trasmettitore), dall'altro lato il ricevitore prende le informazioni in seriale e mediante un registro a scorrimento le memorizza in parallelo. Le modalità di trasmissione possono essere di tre tipologie:

- **Simplex:** la comunicazione è unidirezionale, per cui solo il trasmettitore comunica con il ricevitore;



- **Half-Duplex:** trasmettitore e ricevitore comunicano sullo stesso canale, ma solo uno alla volta può parlare.
- **Full-Duplex:** trasmettitore e ricevitore comunicano sullo stesso canale e possono parlare contemporaneamente.

La nostra board monta la periferica UART in modalità full-duplex: in questa modalità di trasmissione, avendo la necessità di trasmettere e ricevere contemporaneamente, *le strutture dati vengono duplicate*.

Il frame dati usato per la trasmissione tramite UART è quello mostrato in figura 7.4. E' chiaro che i dispositivi devono essere d'accordo sulla struttura del pacchetto trasmesso. Nel nostro caso

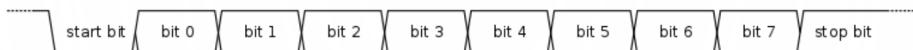


Figura 7.4: Frame dati usato per la trasmissione UART.

il pacchetto dati è caratterizzato da 10 bit, di cui uno è lo start bit, uno è lo stop bit e gli altri 8 sono i bit dati. I livelli tra start e stop devono essere diversi altrimenti non ci sarebbe alcuna transizione di stato mediante cui possiamo identificare la trasmissione di un bit diverso da quello trasmesso.

Lo stato di idle (linea inattiva) è alto perché una periferica in idle è comunque una periferica funzionante, allora per poter distinguere il caso funzionante da quello non si utilizza la linea inattiva alta; in caso contrario, se usassimo l'idle basso, il ricevitore non sarebbe in grado di capire se la linea è in idle perchè il trasmettitore non sta trasmettendo nulla oppure perchè la linea è rotta.

Siccome UART comunicanti non hanno un clock condiviso, esse si risincronizzano ad ogni carattere trasmesso. Nonostante tra i due dispositivi venga definito un **baud rate**, numero di transizioni che avvengono sulla linea (anche detto tasso di simbolo), il ricevitore lavora con una frequenza multipla (fattore 8 o 16) di quella del trasmettitore, per poter sovraccampionare la linea. Analizziamo il *funzionamento del meccanismo di ri-sincronizzazione*: il ricevitore verifica che lo start bit si abbassi, non appena si abbassa (deve essere la prima volta che la linea da idle passi a 0, perchè dopo lo start bit inizia la trasmissione dei dati), passa in stato di ricezione, ovvero campiona i primi 8 campioni e si posiziona a centro bit. Da quel momento inizia a campionare ogni 16 per poter rimanere sempre a centro bit. *Tenersi a centro bit consente al ricevitore di evitare il framing error, ovvero evitare di deviare il campionamento verso bit che non sono quelli che si aspetta di campionare*. Quindi, il **ricevitore** testa la portante sul pin di ingresso (come vedremo nel file fornito dalla Digilent prende il nome di RXD), cioè **dove fare un polling con una frequenza 16 volte più grande**. Al contrario, il trasmettitore è più semplice, l'inizio della trasmissione è sancito dall'unità di controllo, che comunica con il processore il quale abilita il segnale di scrittura. Al trasmettitore spetta il compito di generare lo start bit, nel momento in cui il buffer contiene i bit dato ed a seconda del frame dati concordato invia un certo numero di bit, nel nostro caso sono 8 bit dati. Quando la comunicazione termina il trasmettitore alza il flag **Text Buffer Empty**, per indicare che il buffer è stato svuotato, mentre il ricevitore alza il flag **Read Data Available**, cioè la disponibilità in lettura delle informazioni ricevute.

7.2 Analisi del codice

N.B.: Per lo studio del seguente argomento, si fa riferimento alla documentazione sulla periferica RS232 ed al codice sorgente [4] forniti dalla Digilent e presenti nel materiale didattico, disponibile sul sito del docente.

Passiamo adesso all'analisi del codice in maniera dettagliata, in modo da identificare la struttura del componenente UART, definire l'automa del ricevitore e del trasmettitore. Il codice è stato organizzato in un unico file che include la parte operativa e la parte di controllo, sia del trasmettitore che del ricevitore, per questo motivo ci siamo proposte di scendere nel dettaglio del file, arrivando all'identificazione dei blocchi che compongono il macroblocco **UARTcomponent**. Per poter descrivere com'è fatto il ricevitore e come il trasmettitore, partiamo dallo strato più esterno, al fine di identificare quali siano i segnali di ingresso e quelli di uscita e capire come questi possano interessare le due entità fondamentali. L'**UARTcomponent**, come si osserva dalla figura

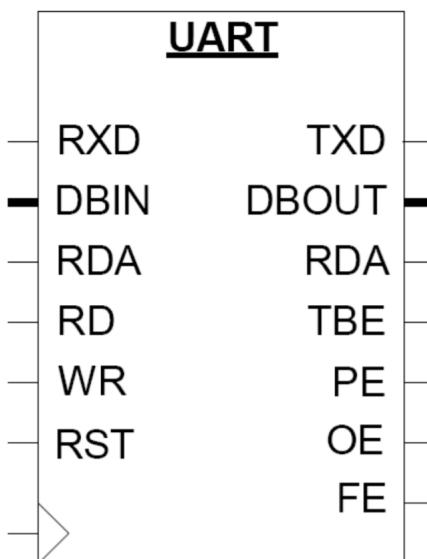


Figura 7.5: Blocco UART con pin di I/O.

7.5 è caratterizzato da un segnale di **clock in ingresso** (indispensabile la presenza di un segnale che consenta la temporizzazione della macchina).

Nota 7.1: Il canale TXD-RXD

I segnali RXD-TXD, il primo di ingresso ed il secondo di uscita, rappresentano il canale di trasmissione. Avendo introdotto il componente **UART** per la trasmissione seriale è chiaro che ci aspettiamo che questi siano definiti su un singolo bit, per di più si evidenzia il fatto che **TXD sia inizializzato ad 1**. Questo accade perché **non si vuole iniziare una ricezione spuria**: affinché la linea sia sempre in stato inattiva alta, a meno che non si voglia iniziare una trasmissione, l'uscita TXD è sempre costante e pari ad 1, in modo che il ricevitore non possa vedere un valore diverso e non possa così iniziare una ricezione non prevista.

Altri segnali di ingresso sono il reset **RST**, affinché la macchina possa partire da uno stato noto, il segnale di **WR** che serve per far partire il **trasmettitore**, i flag di uscita **RDA**, **TBE**, **PE**, **FE**, **OE**. Questi ultimi tre rappresentano proprio l'errore di parità, l'errore di framing e l'errore di overrun. Infine, si nota la presenza di un segnale di ingresso **DBIN** ad 8 bit ed un segnale di uscita **DBOUT**,

anch'esso ad 8 bit. L'UART ha **due blocchi funzionali principali** che possiamo vedere come

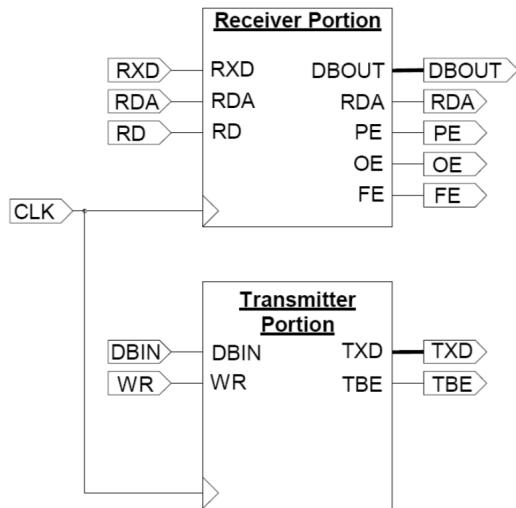


Figura 7.6: Blocchi funzionali del componente UART.

due circuiti separati, integrati all'interno di un unico componenente: *un circuito per la ricezione di informazioni seriali e un circuito per la trasmissione di informazioni seriali*. Infatti, il ricevitore riceve un byte di informazioni seriali attraverso la porta RXD di ingresso e converte queste in parallelo. Il byte convertito viene mostrato sull'uscita DBOUT. Dall'altro lato il trasmettitore prende un byte di informazioni parallele DBIN (quando si alza il bit di write) e lo trasmette in serie sulla porta TXD. Questo spiega ancora meglio perchè RXD è un pin di ingresso e TXD di uscita.

```

-- UART composta da trasmettitore e ricevitore
entity UARTcomponent is
    Generic (
        BAUD_DIVIDE_G : integer := 14; --115200 baud e f=26.6MHz
        BAUD_RATE_G : integer := 231);
    Port (
        TXD : out std_logic := '1';
        RXD : in std_logic;
        CLK : in std_logic;
        DBIN : in std_logic_vector (7 downto 0);
        DBOUT : out std_logic_vector (7 downto 0);
        RDA : inout std_logic;
        TBE : out std_logic := '1';
        RD : in std_logic;
        WR : in std_logic;
        PE : out std_logic;
        FE : out std_logic;
        OE : out std_logic;
        RST : in std_logic := '0');
end UARTcomponent;

```

Mediante il generic sono definiti il BAUD_RATE ed il BAUD_DIVIDE. Stabilito il numero di simboli da trasmettere in un secondo (il baud), ovvero 115200 (come definito nel commento), e presupponendo che la frequenza del clock in ingresso sia di 26.6 MHz, il baud rate è ottenuto come:

$$\frac{26.6 \times 10^6}{115200} = 231$$

Il baud divide è calcolato come:

$$\frac{231}{16} = 14$$

ovvero come il baud rate, diviso 16. Entrambi questi valori vengono poi convertiti in stringhe di bit mediante l'operazione di conv_std_logic_vector, in particolare il baud rate è rappresentato su 13 bit e il baud divide su 9 bit.

7.3 Ricevitore

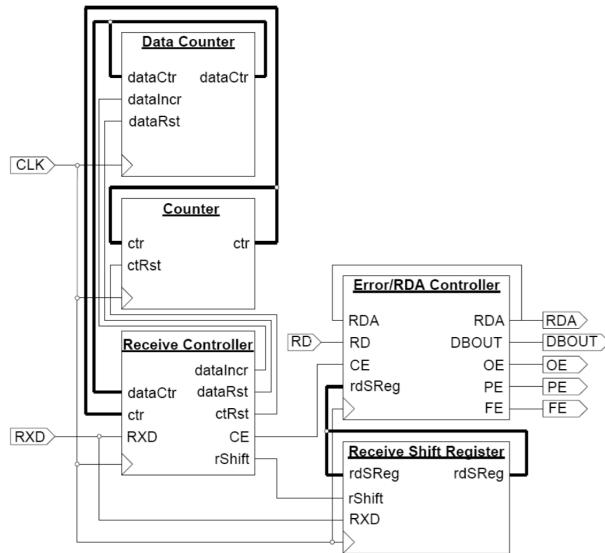


Figura 7.7: Schema a blocchi del ricevitore.

Il ricevitore UART prende dati in ingresso seriali e li converte in paralleli. Questo blocco presenta un controller dei dati seriali, due contatori che sono utilizzati per la sincronizzazione, un controller dei bit di errore ed un registro a scorrimento. Il registro a scorrimento viene utilizzato per memorizzare i dati in arrivo al pin di ingresso RXD. Lo schema a blocchi del ricevitore è quello presente in figura 7.7.

Il controller del ricevitore, che sfrutta una macchina a stati e i due contatori, è necessario per la sincronizzazione con la trasmissione, visto che i dati trasferiti sulla porta RXD arrivano ad una velocità di trasmissione specifica. Avendo già spiegato come funziona il meccanismo di sincronizzazione, risulta evidente che la macchina a stati è realizzata in modo tal da leggere il porto RXD a centro bit. In figura 7.8 rappresentato il diagramma della macchina a stati.

Il controllore del ricevitore è implementato mediante un process, sensibile al clock ed al reset. Il reset è sincrono, perché al primo fronte di salita del clock, se il reset è alto, la macchina viene portata in stato di idle (strCur rappresenta lo stato corrente), altrimenti, nel caso in cui il reset è basso, al fronte di salita del clock, avviene la transizione allo stato successivo (strNext).

```
-- controllore della state machine del ricevitore
process (CLK, RST)
begin
    if CLK = '1' and CLK'Event then
        if RST = '1' then
            strCur <= strIdle;
        else
            strCur <= strNext;
        end if;
    end if;
end process;
```

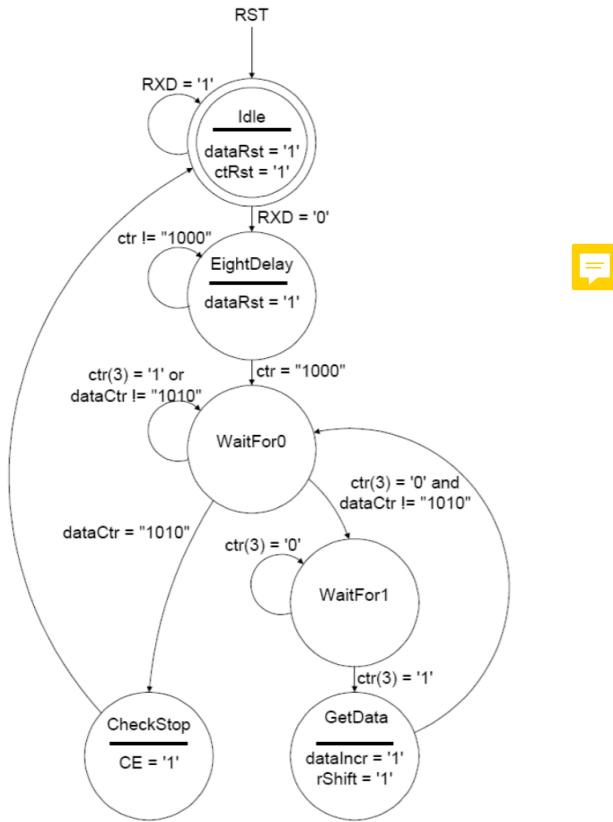


Figura 7.8: Automa a stati finiti del ricevitore.

Stato idle: strIdle

Quando non c'è una trasmissione in corso, il trasmettitore non genera lo start bit, questo vuol dire che TXD sarà tenuto alto (ecco perchè inizializzato a 1), quindi l'RXD in ingresso al ricevitore è mantenuto alto. La macchina permane in stato di idle fintanto che la porta RXD non si abbassa, cioè fino a quando non inizia la ricezione (con una precedente trasmissione) dello start bit. Quando si verifica $RXD=0'$, allora avviene la transizione allo stato EightDelay.

```

--stato di Idle del ricevitore
process (strCur, ctr, RXD, dataCtr)
begin
    case strCur is
        when strIdle =>
            dataIncr <= '0';
            rShift <= '0';
            dataRst <= '1';
            CE <= '0';
            ctRst <= '1';

            if RXD = '0' then
                strNext <= strEightDelay;
            else
                strNext <= strIdle;
            end if;
    end case;
end process;

```

Dal codice osserviamo che in questo stato il segnale di abilitazione dell'incremento del **dataCtr** (che è il conteggio del numero di bit letti) è posto a 0, così come il segnale di abilitazione di shift a destra ed anche il segnale **CE - Clock Enable**, che è l'abilitazione alla scrittura per il controller dei bit di errore, che analizzeremo in seguito. Al contrario, vengono alzati i bit di **dataRst** e **ctRst**, che sono i segnali di reset del contatore di bit letti e del divisore di frequenza.

Il divisore di frequenza del ricevitore è realizzato come segue, sfruttando un modello di astrazione comportamentale: se c'è il segnale di reset oppure il clock diviso (clkDiv) ha raggiunto la frequenza massima, ovvero la baud divide (che sarebbe la frequenza del ricevitore), viene azzerato, altrimenti incrementato

```
--divisore di frequenza del trasmettitore
process (CLK, clkDiv)
begin
    if (CLK = '1' and CLK'event) then
        if (clkDiv = baudDivide or ctRst = '1') then
            clkDiv <= "0000000000";
        else
            clkDiv <= clkDiv +1;
        end if;
    end if;
end process;
```

Stato Eight Delay: strEightDelay

Lo stato di EightDelay è presente al fine di assicurare che la porta RXD venga letta nel mezzo della trasmissione di ogni bit. Il contatore, ctr, aumenta la velocità a 16 volte più veloce della velocità della trasmissione. Quando viene raggiunto questo stato, il contatore ctr può contare fino a otto. In altre parole, questo stato ritarda la macchina per otto cicli, cioè per otto colpi di clock, affinché il ricevitore si posizioni a centro bit, così che il segnale RXD sia letto correttamente. Il contatore ctr serve a tenere traccia di questi colpi di clock, quindi fintanto che il conteggio è inferiore ad 8, la macchina continua a rimanere nello stato di EightDelay, quando il conteggio arriva ad 1000 avviene la transizione allo stato successivo di WaitFor0.

```
--stato di Eight Delay del ricevitore
when strEightDelay =>
    dataIncr <= '0';
    rShift <= '0';
    dataRst <= '1';
    CE <= '0';
    ctRst <= '0';

    if ctr(3 downto 0) = "1000" then
        strNext <= strWaitFor0;
    else
        strNext <= strEightDelay;
    end if;
```

Il contatore ctr è implementato ancora una volta con un modello di astrazione comportamentale, così come mostrato nel codice. Si sottoline che ctr è un vettore st_logic_vector caratterizzato da 4 bit, quindi specificare **ctr(3 downto 0)** non era necessario.

```
--contatore della velocita' necessaria per ricevere correttamente i dati su RXD
process (CLK)
begin
    if CLK = '1' and CLK'Event then
        if ctRst = '1' then
```

```

                ctr <= "0000";
elsif clkDiv = baudDivide then
    ctr <= ctr + 1;
else
    ctr <= ctr;
end if;
end if;
end process;

```

Questo process imposta ctr, che utilizza il clkDiv (del divisore di frequenza) per contare l'aumento alla velocità necessaria per ricevere correttamente i dati da RXD. Se il ctRst viene attivato, il contatore viene ripristinato. *Se il clk è uguale al baudDivide, allora incrementiamo una volta il ctr.*

Stato Wait for 0: strWaitFor0

Questo stato è uno **stato di ritardo**, che ritarda il ricevitore se non sono stati trasferiti ancora tutti i dati seriali. La comunicazione UART prevede la trasmissione di 10 bit, 8 di dato, 1 di parità ed 1 di stop. Se il dataCtr non è uguale a 10, ovvero 1010, vuol dire che ancora dobbiamo ricevere altri bit. In particolare, se il quarto bit di ctr è uguale ad 1, rimaniamo in questo stato. Quando il quarto bit di ctr si azzera, vuol dire che è stata raggiunta metà del ritardo (si entra in questo stato quando ctr vale 1000, cioè dopo 8 conteggi, per cui ctr si azzera dopo altri 8 conteggi, cioè dopo che sono avvenute le commutazioni 1001-1010-1011-1100-1101-1110-1111-0000) e quindi si transita nel secondo stato di ritardo che è il Wait for 1. Se invece il dataCtr è arrivato ad 1010, tutti i dati necessari sono stati acquisiti, per cui avviene la transizione allo stato di Check per verificare la presenza di errori e ripristinare la macchina di ricezione.

```
--stato di ritardo 0
when strWaitFor0 =>
    CE <= '0';
    dataRst <= '0';
    ctRst <= '0';
    dataIncr <= '0';
    rShift <= '0';

    if dataCtr = "1010" then
        strNext <= strCheckStop;
    elsif ctr(3) = '0' then
        strNext <= strWaitFor1;
    else
        strNext <= strWaitFor0;
    end if;
```

Stato Wait for 1: strWaitFor1

Entriamo in questo stato quando il quarto bit di ctr è 0. Questo stato è molto simile a quello precedente, eccetto per il fatto che aspetta che il quarto bit di ctr sia pari ad 1, cioè che siano stati raggiunti altri 8 conteggi. Quando ctr(3) è alto, allora avviene la transizione allo stato strGetData, altrimenti continua a rimanere nello stato corrente, in attesa degli 8 cicli di clock.

```
--stato di ritardo 1
when strWaitFor1 =>
    CE <= '0';
    dataRst <= '0';
    ctRst <= '0';
    dataIncr <=      '0';
```

```

    rShift <= '0';

    if ctr(3) = '0' then
        strNext <= strWaitFor1;
    else
        strNext <= strGetData;
    end if;

```

Stato Get Data: strGetData

In questo stato, vengono tenuti alti per un colpo di clock i segnali di rdShift e di dataIncr. In questo modo, il registro a scorrimento fa uno shift e il numero di bit letti viene incrementato di uno. Questo stato, in altre parole, acquisisce i dati in arrivo sul pin RXD nel registro a scorrimento rdSreg (che è stato definito all'inizio del file come vettore composto da 10 bit). Lo stato successivo sarà il Wait for 0 che avvia i due stati di ritardo (wait for 0 e wait for 1) necessari tra i turni di dati.

```

-- stato di Get Data del ricevitore
when strGetData =>
    CE <= '0';
    dataRst <= '0';
    ctrRst <= '0';
    dataIncr <= '1';
    rShift <= '1';

    strNext <= strWaitFor0;

```

Stato Check: strCheckStop

Quando sono stati acquisiti i 10 bit di dati, si arriva in questo stato, che serve ad avviare il processo che effettua il controllo degli errori. Infatti, viene alzato il flag CE. Da questo stato il ricevitore transita nuovamente in idle.

```

-- stato di Check del ricevitore
when strCheckStop =>
    dataIncr <= '0';
    rShift <= '0';
    dataRst <= '0';
    ctrRst <= '0';
    CE <= '1';
    strNext <= strIdle;

```

7.3.1 Controllo degli errori

Questo processo deve controllare i flag di errore PE, OE, FE, nonché il flag RDA. Partiamo dal vedere come sono stati definiti questi valori e poi analizziamo il processo che li gestisce.

```

-- definizione del pin TXD, registro dato trasmettitore, errore di frame, errore di parità
-- parita' e DBOUT
    frameError <= not rdSReg(9);
    parError <= not ( rdSReg(8) xor (((rdSReg(0) xor rdSReg(1)) xor
        (rdSReg(2) xor rdSReg(3))) xor ((rdSReg(4) xor rdSReg(5)) xor
        (rdSReg(6) xor rdSReg(7)))) );
    DBOUT <= rdReg;
    tfReg <= DBIN;
    TXD <= tfsReg(0);

```

```

par <= not ( ((tfReg(0) xor tfReg(1)) xor (tfReg(2) xor tfReg(3))) xor
            ((tfReg(4) xor tfReg(5)) xor (tfReg(6) xor tfReg(7))) );

```

Il bit più significativo di rdSReg indica il bit di errore di frame, per cui l'FE è legato a quel segnale. L'errore di frame indica che la UART non sta leggendo i dati al momento giusto, ovvero se il bit di stop non è 1, per cui al frameError è assegnato il negato dell'MSB del registro a scorrimento del ricevitore, in modo che *il bit FE di uscita sarà alto qualora il nono bit del registro fosse basso, ovvero il ricevitore ha preso 10 campioni, ma l'ultimo non è lo stop bit atteso.*

Si verifica l'errore di parità se il bit di parità non concorda con il numero di 1 nella porzione di dati del registro a scorrimento. Se è impostata la parità dispari, il bit di parità dovrebbe essere 1 se è presente un numero di 1 dispari nella porzione di dati del registro a scorrimento. Se, invece, la parità è pari, il bit di parità dovrebbe essere 1 se è presente un numero di 1 pari nella porzione di dati del registro. **Il nostro componenente UART è impostato sulla parità dispari**, ma può facilmente essere modificato eliminando il *not* nella definizione del *parError* e della *parità par*. La parità viene calcolata come l'inverso dei bit dati in XOR tra loro, si sottolinea che la parità viene calcolata sui bit trasmessi. L'errore di parità è il negato del bit di parità (*rdSReg(8)*) in XOR con i bit dati ricevuti. In questo modo, è possibile determinare se il bit di parità trovato in *rdSReg(8)* corrisponde ai bit di dati.

Il DBOUT è assegnato pari al *rdReg*, registro dati ad 8 bit atto a mantenere i bit ricevuti, quindi non è un registro a scorrimento, ma un elemento di memorizzazione di una stringa di 8 bit (registro dato lato ricevitore della figura 7.3). Il DBIN viene assegnato al *tfReg*, che rappresenta il registro dato del trasmettitore (quello del lato sinistro della figura 7.3). Il *tfsReg* è il registro a scorrimento del trasmettitore, usato quindi per spostare i dati trasmessi, la porta TXD è impostata pari al primo bit di *tfsReg*.

```

--process che controlla gli errori PE, FE, OE e del flag RDA
process (CLK, RST, RD, CE)
begin
    if RD = '1' or RST = '1' then
        FE <= '0';
        OE <= '0';
        RDA <= '0';
        PE <= '0';
    elsif CLK = '1' and CLK'event then
        if CE = '1' then
            FE <= frameError;
            PE <= parError;
            rdReg(7 downto 0) <= rdSReg (7 downto 0);
            if RDA = '1' then
                OE <= '1';
            else
                OE <= '0';
                RDA <= '1';
            end if;
        end if;
    end if;
end process;

```

L'ultimo errore è quello di **overrun, cioè di sovrascrittura, che si verifica quando il byte di dati ricevuto è stato scritto sul byte di dati ricevuto precedentemente ma mai letto.** Quando sono disponibili i dati in parallelo, cioè nel porto DBOUT troviamo gli 8 bit dati trasmessi e ricevuti, la porta RDA viene tenuta alta. Quando i dati vengono letti dalla porta DBOUT, il flag RDA viene abbassato. Se i dati sono stati ricevuti ed il flag RDA è ancora alto, i nuovi 8 bit

dati spostati sostituiranno i precedenti 8 bit conservati nella porta DBOUT e il bit di OE verrà alzato per indicare che si è verificato l'errore di sovrascrittura.

7.3.2 Incoming Data Counter

Questo processo controlla il conteggio dei dati per tenere traccia dei valori spostati in rdSreg. Il conteggio dataCtr viene incrementato ad ogni fronte di salita del clock quando il segnale di dataIncr, fornito dalla control unit del ricevitore, viene alzato.

```
--contatore di bit letti utilizzato dal ricevitore
process (CLK, dataRST)
begin
    if (CLK = '1' and CLK'event) then
        if dataRST = '1' then
            dataCtr <= "0000";
        elsif dataIncr = '1' then
            dataCtr <= dataCtr +1;
        end if;
    end if;
end process;
```

7.3.3 Receiving Shift Register

Questo processo controlla lo shift register del ricevitore. E' chiaro che, come negli altri casi, il livello di astrazione è comportamentale, per cui sarà l'ambiente di sintesi a dover risolvere il gap con la scheda su cui sarà sintetizzato.

Ogni volta che il segnale di rShift è alto, i dati devono essere spostati, per cui viene posto in testa al registro il bit preso in ingresso su RXD e si fa uno shift a destra di 9 bit.

```
--registro a scorrimento del ricevitore
process (CLK, rShift)
begin
    if CLK = '1' and CLK'Event then
        if rShift = '1' then
            rdSReg <= (RXD & rdSReg(9 downto 1));
        end if;
    end if;
end process;
```

7.4 Trasmettitore

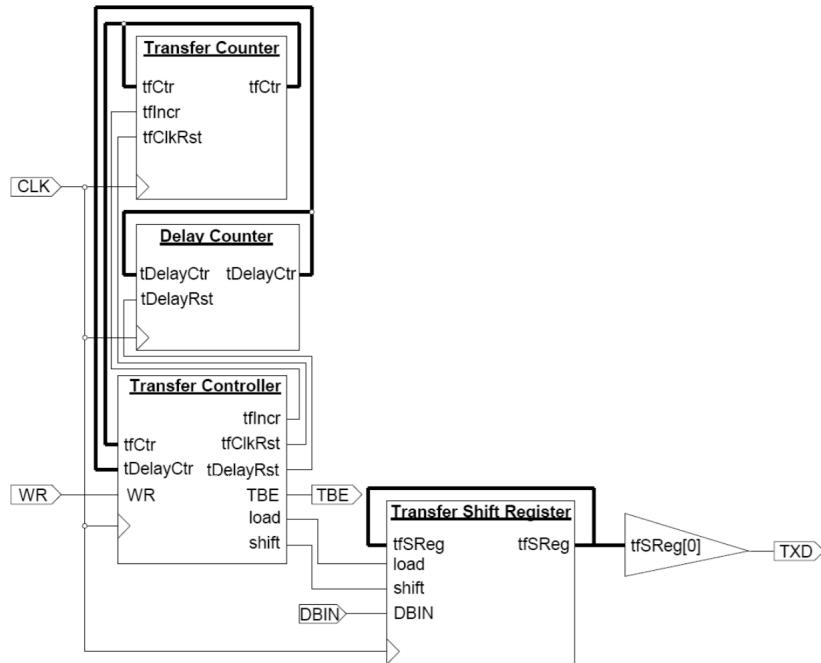


Figura 7.9: Schema a blocchi trasmettitore.

La parte del trasmettitore di UART accetta un byte di dati dalla porta DBIN e lo trasmette come dati seriali sulla porta TXD. Per trasmettere il byte memorizzato nella porta DBIN, la parte di trasferimento di UART deve contenere un controller di trasferimento, due contatori per la sincronizzazione e un registro di spostamento del trasferimento.

Il controller di trasferimento utilizza i due contatori di sincronizzazione per controllare la velocità con cui il byte di dati deve essere trasmesso attraverso la porta TXD e il numero di bit trasmessi.

```
--controller della state machine del trasmettitore
process (CLK, RST)
begin
    if (CLK = '1' and CLK'Event) then
        if RST = '1' then
            sttCur <= sttIdle;
        else
            sttCur <= sttNext;
        end if;
    end if;
end process;
```

Anche nel caso del trasmettitore, come per il ricevitore, *il controller è realizzato mediante un process con reset sincrono*, per cui al fronte di salita del clock, se c'è il segnale di reset la macchina si pone in stato di sttIdle, altrimenti si effettua la transizione da stato corrente a stato prossimo (sttNext). Il controller di trasferimento utilizza i due contatori di sincronizzazione per controllare la velocità con cui il byte di dati deve essere trasmesso attraverso la porta TXD e il numero di bit trasmessi. *Un contatore viene utilizzato per ritardare il controller di trasferimento tra le trasmissioni, mentre l'altro contatore viene utilizzato per tenere traccia di quante trasmissioni sono state inviate. La*

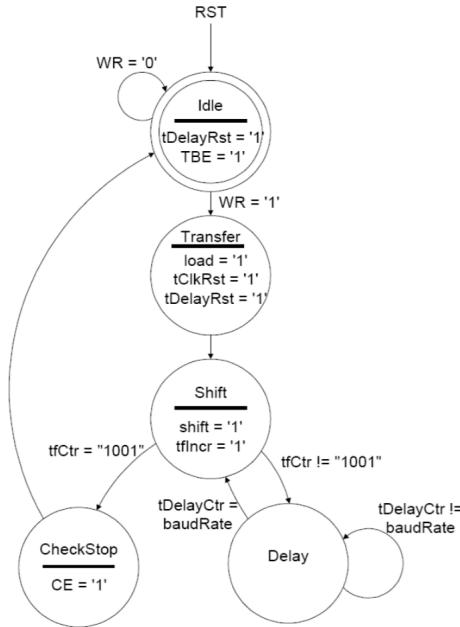


Figura 7.10: Macchina a stati finiti del trasmettitore.

porta TXD è impostata uguale al bit meno significativo del registro a scorrimento di trasferimento, consentendo la trasmissione dei dati semplicemente effettuando lo shift a destra di una posizione ogni qualvolta avviene una trasmissione.

Il diagramma di stato del controller di trasferimento è mostrato in figura 7.10.

Stato Idle: sttIdle

Questo stato rappresenta la fase di inattività e di avvio della macchina a stati del trasmettitore. Quest'ultimo permane in questo stato fino a quando il segnale di WR (write) rimane basso. Una volta che il segnale di write va alto, avviene la transizione al prossimo stato, che è l'sttTransfer.

```
-- stato di idle del trasmettitore
when sttIdle =>
    TBE <= '1';
    tClkRST <= '0';
    tfIncr <= '0';
    shift <= '0';
    load <= '0';
    tDelayRst <= '1';

    if WR = '0' then
        sttNext <= sttIdle;
    else
        sttNext <= sttTransfer;
    end if;
```

Il segnale di load e di shift sono mantenuti bassi, così come il segnale di reset tClkRST ed il tfIncr (l'utilità di questi segnali sarà spiegata nelle sezioni successive). Al contrario il segnale **TBE** (Text Buffer Empty) è **alto per indicare che il buffer di trasmissione non è attualmente in uso**. Quando la macchina di trasferimento lascia questo stato, il segnale di TBE sarà tenuto basso per indicare che il buffer di trasmissione è in uso. Infine, viene alzato il bit di tDelayRst, che rappresenta il

segnale di reset del contatore tDelayCtr.

Il divisore di frequenza del trasmettitore è realizzato mediante un process, con livello di astrazione comportamentale.

```
--transfer delay counter
process (CLK, tDelayCtr)
begin
    if (CLK = '1' and CLK'event) then
        if (tDelayCtr = baudRate or tDelayRst = '1') then
            tDelayCtr <= "00000000000000";
        else
            tDelayCtr <= tDelayCtr+1;
        end if;
    end if;
end process;
```

Questo contatore viene eseguito fino a quando il conteggio non arriva al baudRate o si alza il segnale di reset. È rappresentato su 13 bit poiché deve arrivare al valore definito di baudRate, ovvero 231. Questo contatore viene utilizzato per misurare i tempi di ritardo durante l'invio di dati sul pin TXD. Quando il contatore è uguale a baudRate o è resettato, è impostato uguale a 0.

Stato Transfer: sttTransfer

Questo stato impone i segnali load, tClkRST e tDelayRst su alti, mentre impone il segnale TBE su basso. Il segnale di load è impostato in alto per caricare il registro a scorrimento di trasferimento con i dati appropriati; i segnali tClkRST e tDelayRst vengono alzati per resettare tfCtr e tDelayCtr, cioè i due contatori di sincronizzazione (conteggio di bit trasferiti e divisore di frequenza). Il successivo stato caricato è lo stato sttDelay.

```
-- stato di transfer del trasmettitore
when sttTransfer =>
    TBE <= '0';
    shift <= '0';
    load <= '1';
    tClkRST <= '1';
    tfIncr <= '0';
    tDelayRst <= '1';

    sttNext <= sttDelay;
```

Stato Delay: sttDelay

Questo è lo stato che realizza il ritardo della macchina tra le trasmissioni. Tutti i segnali vengono tenuti bassi: se il tDelayCtr è uguale al baudRate avviene la transizione allo stato sttShift, altrimenti il trasmettitore continua a rimanere in questo stato.

```
--stato di delay del trasmettitore
when sttDelay =>
    TBE <= '0';
    shift <= '0';
    load <= '0';
    tClkRst <= '0';
    tfIncr <= '0';
    tDelayRst <= '0';
```

```

if tDelayCtr = baudRate then
    sttNext <= sttShift;
else
    sttNext <= sttDelay;
end if;

```

Stato Shift: sttShift

In questo stato, i segnali di abilitazione shift e tfIncr vengono alzati, mentre tutti gli altri sono tenuti bassi. Inoltre, viene controllato il tfCtr, per vedere quanti bit sono stati trasmessi. Tenendo il segnale di shift alto, si effettua uno shift a destra di un bit ad ogni colpo di clock, ed ogni volta il tfCtr, conteggio di bit trasmessi, viene incrementato di uno, grazie al segnale di tfIncr abilitato. Fintanto che tfCtr non arriva a 1010, cioè a 10, non sono stati trasmessi tutti i bit, quindi la macchina ritorna allo stato di Delay. Se invece, il tfCtr arriva a 10, viene caricato lo stato finale che è sttWaitWrite.

```

-- stato di shift del trasmettitore
when sttShift =>
    TBE <= '0';
    shift <= '1';
    load <= '0';
    tfIncr <= '1';
    tClkRST <= '0';
    tDelayRst <= '0';

    if tfCtr = "1010" then
        sttNext <= sttWaitWrite;
    else
        sttNext <= sttDelay;
    end if;

```

Stato di Write: sttWaitWrite

Questo stato si occupa di controllare che il segnale di WR inizialmente attivato dalla macchina sia stato riportato a 0. Senza questo stato, il segnale di scrittura sarebbe tenuto alto per lungo tempo, comportando così trasmissioni multiple. Quando il segnale di write si abbassa, viene caricato lo stato di idle che riporta il trasmettitore allo stato di partenza.

```

--stato di write del transfer
when sttWaitWrite =>
    TBE <= '0';
    shift <= '0';
    load <= '0';
    tClkRst <= '0';
    tfIncr <= '0';
    tDelayRst <= '0';

    if WR = '1' then
        sttNext <= sttWaitWrite;
    else
        sttNext <= sttIdle;
    end if;

```

7.4.1 Transfer Data Counter

Il segnale tClkRST rappresenta il segnale di reset del tfCtr, ovvero il **transfer counter** che serve per tenere traccia del numero di bit trasmessi.

```
-- contatore di bit trasmessi
process (CLK, tClkRST)
begin
    if (CLK = '1' and CLK'event) then
        if tClkRST = '1' then
            tfCtr <= "0000";
        elsif tfIncr = '1' then
            tfCtr <= tfCtr +1;
        end if;
    end if;
end process;
```

Questo processo incrementa tfCtr ogni volta che si verifica il fronte di salita del clock e che il segnale di tfIncr è alto (per questo motivo nello stato di idle viene posto a zero dato che non è necessario alcun conteggio). Se il segnale di tClkRST è alto, il conteggio viene ripristinato a 0.

7.4.2 Transfer Shift Register

I segnali di load e shift riguardano il registro a scorrimento del trasmettitore.

```
--controller shift register trasmettitore
process (load, shift, CLK, tfSReg)
begin
    if CLK = '1' and CLK'Event then
        if load = '1' then
            tfSReg (10 downto 0) <= ('1' & par &
                                         tfReg(7 downto 0) &'0');
        elsif shift = '1' then
            tfSReg (10 downto 0) <= ('1' & tfSReg(10 downto 1));
        end if;
    end if;
end process;
```

Questo processo, infatti, utilizza i segnali load, shift e clk per controllare il registro di shift (tfSReg, registro a scorrimento di 11 bit inizializzati ad 1, in modo che, seppur iniziasse una trasmissione, la linea è sempre in idle). Una volta che il segnale di load è pari ad 1, tfSReg ottiene prende un 1, il bit di parità, i bit di dati trovati all'interno di tfReg (registro, inteso come elemento di memorizzazione, degli 8 bit dati) ed un bit 0, questa operazione equivale a prelevare i bit dati inseriti per esempio dal processore nel registro dato della figura 7.3 aggiungendo opportunamente bit di stop, bit di parità e lo start bit. In questo modo, il registro a scorrimento può essere utilizzato per spostare in serie i dati da trasferire, ogni volta che il segnale di shift è 1.

Capitolo 8

Dispositivi programmabili e FPGA

8.1 Programmable Logic Device (PLD)

Un dispositivo PLD (Programmable Logic Device) è un componente elettronico usato per costruire circuiti digitali riconfigurabili, è di particolare interesse perché c'è il concetto di **programmabilità**. Diversamente dalla porta logica, **un PLD non ha una funzione definita in fonderia** per cui, prima di essere usato, necessita di una fase di configurazione. Questa è la fase che cambia tra gli ambienti di sviluppo ISE e Vivado. Il vantaggio risiede in tre aspetti cruciali:

- il costo di fonderia è abbattuto, infatti i PLD hanno un costo di pochi dollari (la scheda che abbiamo ha un costo limitato, è complessa perché è un ambiente di sviluppo, però un semplice PLD costerebbe ancora meno);
- il dispositivo può essere configurato per prototipazione;
- la funzione logica implementata può cambiare se le specifiche mutano (così come avviene nel software).

Possiamo riprogrammare perché non abbiamo investito per costruire l'ASIC, per il circuito. Se dobbiamo riprogrammarlo, **dobbiamo aggiungere qualcosa che consenta di cancellare e riscrivere quanto fatto**, il vantaggio è che l'hardware rimane sempre lo stesso. Un po' come i processori cambiano il comportamento con il programma, cioè sono programmabili, questi PLD cambiano il comportamento con una configurazione programmabile, se poi la programmazione impone che possiamo mettere a bordo un processore o una memoria, diventano dispositivi programmabili in cui abbiamo messo un oggetto programmabile con un programma programmabile, come accade nei sistemi embedded. Vediamo i vari modi di fare un PLD. Un primo modo è utilizzare una **memoria ROM**. Le ROM sono memorie universali che possono implementare funzioni multi-uscita mappando direttamente tabelle di verità, ad esempio ROM con m input (linee indirizzo) ed n output (linee dato). La memoria contiene m parole, ciascuna di n bit. Teoricamente sono disponibili 2^m possibili funzioni booleane, la struttura delle ROM permette però di definirne massimo n . Il vantaggio della ROM è che non richiedono la minimizzazione, contengono l'intera tabella di verità, gli ingressi delle funzioni sono indirizzi e le uscite vengono memorizzate in ogni riga. Sono dispositivi lenti perché ogni configurazione del sistema richiede di usare tutte le celle e tutti i circuiti per la parte di decodifica, cioè per ogni configurazione degli ingressi bisogna ripetere il ciclo di accesso.

Un'altra possibilità è il **PLA** (*Programmable Logic Array*) che è un dispositivo pensato per

creare funzioni combinatorie come somma di prodotti: sono disponibili **due piani di porte logiche programmabili**, cioè ci sono batterie di porte AND con batterie più piccole di porte OR, per poter bruciare la configurazione che serve in termini di forma canonica. **Se riusciamo a fare delle semplificazioni è utile**, in quanto le porte e gli addendi delle porte sono in numero limitato. In figura 8.1 vediamo un PLA con n ingressi ed m uscite. Ciascuna porta AND ha $\frac{p}{n}$ linee di ingresso e produce p uscite. Le porte OR dispongono di $\frac{m}{p}$ ingressi. In uscita abbiamo il segnale di uscita e la sua versione negata. **L'ambiente di sviluppo deve prendere ciò che abbiamo scritto in VHDL e tradurlo per un sistema programmabile che va a bruciare determinate porte.**

A seconda di come è fatto l'hardware, il ciclo di sviluppo in VHDL è in grado di decidere cosa bruciare e cosa no. Inoltre, **se la funzione è più grande deve essere partizionata e quindi si devono creare i collegamenti tra le parti partizionate.** Il compilatore VHDL è infatti molto complesso, bisognerebbe conoscere la tecnologia ed il linguaggio e fare la traduzione dal linguaggio verso la tecnologia, se varia la tecnologia cambia tutto.

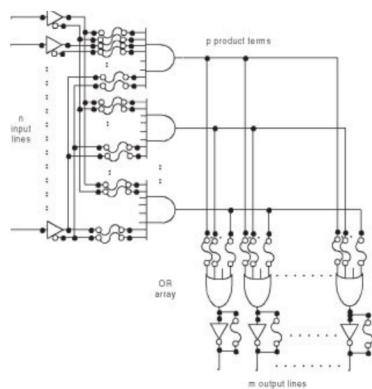


Figura 8.1: Schema del PLA

Il PAL (Programmable Array Logic) è la versione ridotta del PLA, perché è strutturalmente uguale ma senza possibilità di programmare il piano delle OR. Nel PLA tutto è programmabile, nel PAL ci sono delle connessioni fisse. Possiamo non utilizzare tutti gli addendi delle OR, per esempio, ma non possiamo collegare i blocchi come vogliamo. I contatti del resto sono tra le cose più difficili da realizzare, motivo per cui questa soluzione è più semplice.

I GAL (Generic Array Logic) sono i primi dispositivi a permettere la riprogrammazione, sono dei PAL riprogrammabili più volte. Il GAL è un dispositivo in cui la programmazione è fatta mediante elementi di memoria, quindi la modalità di programmazione degli elementi varia. PLA, PAL e GAL integrano poche porte logiche, poche centinaia se li compariamo con dispositivi ASIC, e la loro architettura è poco scalabile. La programmazione di questi dispositivi è per di più complicata, sono richieste macchine apposite che programmino il circuito con opportune sovratensioni. Se prendiamo un PAL, PLA o GAL dobbiamo bruciare dei collegamenti, cioè dobbiamo usare sistemi di configurazione che non sono sempre semplici da disporre, mentre noi vorremmo usare per la configurazione un dispositivo la porta USB, un dispositivo generica. La scheda con cui lavoriamo ha una porta USB, per cui ha un protocollo che consente di ricevere dei dati e mandarli da qualche parte: la scheda all'interno ha un'intelligenza, perché deve prendere un file ed in relazione a questo deve costruire un sistema.

Il CPLD (Complex PLD) interconnettono in un solo chip più dispositivi GAL. La matrice di interconnessione (*switch matrix*) è essa stessa programmabile. Mentre prima avevamo un gate array o un PAL, quindi o riuscivamo a realizzare quello che dovevamo con uno di essi oppure li interconnettavamo, a mano, i sistemi CPLD hanno delle isole di PAL o di PLA, che sono

configurabili tramite crossbar: hanno dei nodi all'interno e all'interno di questi nodi hanno sistemi di collegamento. Ci sono blocchi logici e tra loro c'è una matrice di switch che consente il collegamento tra i vari blocchi (come mostra la figura 8.2). Questo sistema ha un maggior livello di reversibilità rispetto ai primi, che usavano solo il blocco logico.

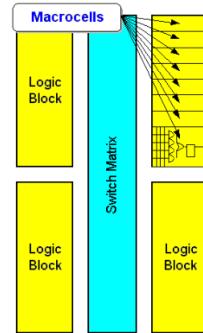


Figura 8.2: Schema del CPLD

8.2 FPGA

Gli FPGA sono **dispositivi programmabili dall'utente e non in fonderia**. I dispositivi hanno sia la logica programmabile, sia la logica circuitale, sia hardware speciale. La parte programmabile è composta da moltissime celle, contenenti 4 funzioni:

- la **Look Up Table** (LUT) è una tabella che dato l'ingresso da l'uscita (simile alla tabella di verità);
- un **multiplexer** perché fa convogliare più fili in un solo punto, se servono più uscite in un solo punto è importante;
- la **logica di propagazione del riporto**;
- **flip flop di tipo D**, che è quello che quindi si usa nella sintesi perché lo troviamo già sintetizzato.

Questi 4 elementi bastano per implementare ogni tipo di sistema perché questo è **un insieme funzionalmente completo**, è la base che usano i compilatori per tradurre tutto. Con la tabella di verità e il flip flop D possiamo fare ogni macchina sequenziale, con una certa tempificazione. Con il mux possiamo convogliare più segnali in un solo punto, possiamo limitare il numero di segnali che escono dalla scheda, dal blocco. Con la propagazione del riporto possiamo lavorare sugli addizionatori, da cui possiamo fare moltiplicatori, divisori, sottrattori e così via. Rendiamo più efficiente l'addizione, l'operazione base del sistema, solo se ci serve, altrimenti non la usiamo proprio ma comunque l'abbiamo. È una macchina che ci permette di fare velocemente i calcoli se servono, possiamo fare la tabella di verità se serve. Rispetto a com'è fatta la LUT dipende dalla tecnologia dell'FPGA, potrebbe essere fatta con la ROM, con PLA, ecc. Teoricamente può essere una tabella che implementa porte logiche, riconfigurabili o ROM, quasi sempre si fa con le ROM. La parte di hardware speciale contiene circuiti integrati di largo impiego, come gestori del clock, circuiti aritmetici o memorie veloci. **L'FPGA compone una funzionalità a partire da celle di base semplici, grazie a meccanismi di interconnessione tra le celle.** Interconnettendo le celle riesce a realizzare meccanismi di complessità maggiore. La figura mostra lo schema di una cella elementare dell'FPGA, la **SLICEL**. La LUT ha un numero di bit in ingresso ed uscita che è fissato e dipende dalla scheda che prendiamo.

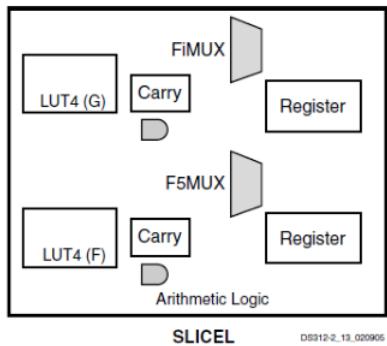


Figura 8.3: Cella elementare di un FPGA

Quando vogliamo costruire una macchina più complessa, abbiamo un problema, prendiamo la cella SLICEL e se la LUT è di 4 e vogliamo fare una tabella da 8, una sola cella non è sufficiente. **L'FPGA consente di programmare la LUT ma anche di portare tutti questi moduli in un sistema di crossbar, cioè di unioni ancora programmabili** in cui facciamo entrare i risultati delle LUT intermedie, per avere il risultato della macchina più complessa che vogliamo realizzare. Non realizziamo tutte le tabelle in modo diretto ma dobbiamo fare il place & route, dobbiamo mettere la tabella nel circuito. L'ambiente di sviluppo compila rispetto all'FPGA che ha sotto, scriviamo in VHDL, l'ambiente di sviluppo sapendo qual è l'FPGA che sviluppiamo farà l'azione di place & route, di decomposizione, rispetto all'architettura. L'allocazione dei componenti nel caso dell'FPGA sarà fatta dal sistema. In alcuni casi dobbiamo montare Xilinx con un divers back-end, perché a seconda della scheda che abbiamo la parte terminale che fa la traduzione sul silicio deve essere differente. **A bordo della scheda si deve avere un sistema per generare un clock**, infatti c'è un VCM che genera il clock e consente di fare derivazioni del clock. La scheda poi presenta elementi combinatori, registri, multiplexer, circuiti per fare il riporto e il clock. **La scheda non ha segnali di ingresso, devono essere generati**, possiamo generare i segnali di ingresso con un generatore di segnale oppure con interruttori e tasti. Il tasto mette un valore ad 1 e poi ritorna 0, l'interruttore è 1 o 0 a seconda di cosa ci serve. Per simulare questi sistemi o li colleghiamo tra loro oppure mettiamo manualmente gli ingressi tramite gli interruttori, verrà settata la LUT e si andrà avanti. Il sistema può essere difficile, magari si può provocare o meno un'oscillazione, perché dipende da come il compilatore produce la LUT: se lo fa con porte logiche queste non sono uguali e si generano ritardi, per cui il sistema non è perfettamente simmetrico; se invece è fatto con ROM, questa ha un tempo di accesso sempre uguale, è costante, quindi il sistema è sempre simmetrico e di conseguenza può oscillare. **È importante sapere come è fatta la LUT, se introduce dei ritardi aleatori oppure, essendo un sistema simmetrico, introduce ritardi costanti.**

L'alternativa principe agli FPGA sono gli ASIC, che sono dispositivi programmabili in fabbrica. **Gli FPGA sono tipicamente utilizzati perché abbassano i "non recurring engineering cost"**, cioè abbassano tutti i costi legati al fatto che il sistema va progettato, testato ed implementato. Si comprano degli oggetti FPGA che hanno un costo non troppo elevato, facciamo il prototipo del sistema e quando si deve produrre su larga scala si passa agli ASIC. In alcuni casi, ci sono applicazioni settoriali specifiche in cui vengono usati proprio gli FPGA.

Si possono avere anche componenti molto più complessi nel chip dell'FPGA, per esempio ci sono intere memorie, interi blocchi moltiplicatori, addirittura in alcuni casi piccoli processori embedded. Un'altra caratteristica è che tipicamente hanno una complessa distribuzione del clock, possono esserci più sorgenti di clock ed una complessa rete di interconnessione. Un chip che sembra piccolo è enorme dal punto di vista di spazio di interconnessione: c'è bisogno di portare i segnali di temporizzazione in modo che siano quanto più precisi in tutti i punti del circuito. Per fare questo si adottano tecniche specifiche che sono implementate in questi dispositivi. Quello che si cerca di fare

è ridurre lo skew ed il jitter: lo *skew* è quel fenomeno che fa sì che lo stesso segnale di clock sia distribuito in più punti e arrivi in punti diversi come temporizzazione, per cui *parti diverse del circuito vengono tempificate in maniera diversa*; il *jitter* è il fenomeno tipico di segnali periodici perché soltanto nel mondo ideale un segnale è perfettamente periodico, *nella realtà il periodo non è così preciso*, servono tecniche per ripristinare i segnali di tempificazione. Noi vediamo le schede FPGA prodotte da Xilinx, che non è l'unico produttore, ma è uno dei principali insieme ad Altera, acquistata dalla Intel. Questi produttori rilasciano i propri ambienti di sviluppo gratuiti, ci sono versioni che nel nostro caso si chiamano web pack, in cui c'è una serie di funzionalità che possiamo utilizzare.

8.2.1 Famiglia Spartan 3E

I componenti principali di un FPGA Spartan3E sono cinque:

- il logic block che in Xilinx si chiamano **CLB** (*Configurable Logic Blocks*), i quali contengono le LUT che sono componenti logici fondamentali all'interno dell'FPGA;
- **blocchi di input/output**, che servono per interconnettere tutto ciò che sta nell'FPGA con i possibili input/output dell'utente, cioè tutto ciò che stare fuori all'FPGA;
- **blocchi di RAM**, cioè dei banchi di memoria;
- **blocchi moltiplicatori**, che sono complessi, motivo per cui conviene averne alcuni implementati in hardware, dato che entrano in gioco in molte applicazioni;
- componenti per la gestione efficiente dei segnali di clock, ovvero i **digital clock manager**

In figura 8.4 possiamo vedere l'architettura di tali dispositivi. I blocchi di I/O si trovano tutti intorno all'FPGA, all'interno c'è una matrice di CLB, per righe e colonne. Abbiamo le RAM messe in colonne, con sopra e sotto blocchi di CLB. Possiamo avere o 1 o 2 colonne di RAM, a seconda del device. I blocchi moltiplicatori sono sempre affiancati al blocco di RAM perché sono alimentati dall'uscita della RAM. Poi abbiamo i blocchi per la gestione del clock, in numero vario a seconda del chip, almeno 2 (uno nella parte top e uno nella parte bottom) e massimo 8. **C'è una fitta rete di interconnessione perché tutti i componenti devono comunicare tra loro in modo efficiente.**

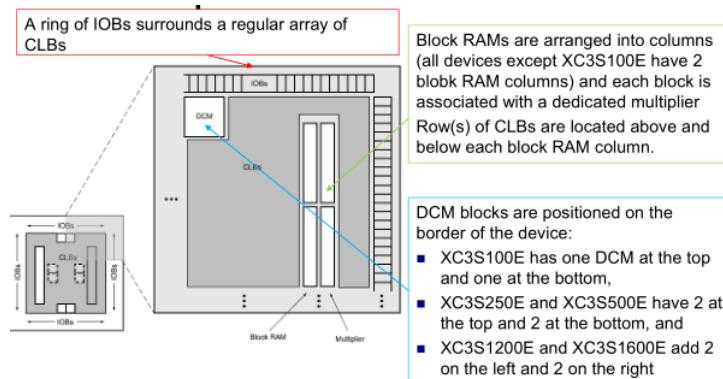


Figura 8.4: Architettura di un FPGA

Vediamo più nello specifico i block RAM, sugli FPGA abbiamo due grossi tipi di sistemi di memorizzazione, i block RAM sono proprio blocchi di memoria, poi abbiamo la distributed RAM, altri piccoli registri, latch e altro, per memorizzare dati parziali, ma non sono veri e proprio blocchi di RAM. I block RAM possono essere di vario numero e dimensioni e sono blocchi di 18 Kbit per

porta, in colonna, ***sono dual port***. Noi siamo abituati a fare un solo accesso alla volta, o in lettura o in scrittura, questi possono funzionare in dual port: ***possono fare contemporaneamente lettura e scrittura dal blocco***, sono divisi in due porti identici, a e b, possiamo leggere e scrivere dallo stesso blocco e anche trasferire un dato dal porto a al b. Sono configurabili quindi possiamo decidere noi se fare i blocchi dual port o meno. Dopo i block RAM ci sono i moltiplicatori, 18x18, un bit per ogni entry della RAM.

I CLB sono disposti in matrice ma ogni CLB è fatto da 4 pezzi che sono detti ***slice***, ogni CLB ha 4 slice (come mostra la figura 8.5). L'unica cosa che può fuorviare è che si indicano le colonne con la x e le righe con la y. ***Le slice contengono le LUT***. Un blocco ha 4 slice, ognuna delle quali ha due LUT. Una LUT dal punto di vista funzionale consente di fare una funzione logica combinatoriale. Una funzione combinatoriale si può fare con le memorie, con i mux: ***la LUT è un generatore di funzioni combinatorie basata su memoria***, in particolare nella Spartan3 la LUT ha 4 ingressi, questi è come se fossero gli indirizzi di 16 locazioni diverse, ciascuna che contiene il valore di quella funzione da 4 ingressi. ***Quando programmiamo la LUT la facciamo diventare una certa funzione di 4 variabili. La minimizzazione ha un po' meno senso allora perché queste tabelle hanno tutti i valori delle RAM, dovremmo riuscire a perdere proprio una variabile, altrimenti non abbiamo vantaggi.*** Abbiamo tanti pezzi piccoli e connettibili perché una funzione booleana si può partizionare, lo fa il compilatore, dunque l'ambiente di sviluppo.

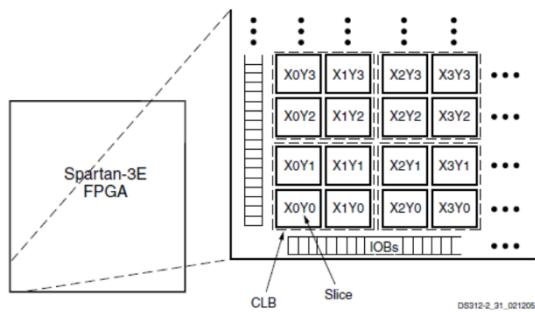


Figura 8.5: CLB

Le slice sono di 2 tipi, di tipo L o N. Le ***slice di tipo L*** hanno le LUT che fanno funzioni combinatorie. Le ***slice di tipo M*** fanno lo stesso ma anche *operazioni legate alla parte di memorizzazione*, consentono di realizzare anche degli ulteriori pezzetti di memoria e shift register, oltre a componenti combinatori. Ciò perché in architetture complesse oltre alla parte combinatoria serve anche quella sequenziale, per non complicare l'architettura di volta in volta usiamo la memoria dei blocchi. ***Le slice L, facendo solo cose combinatorie, sono più veloci, più performanti***, nel caso della famiglia FPGA spartan 3 ne abbiamo in ugual numero. ***Le slice di tipo M hanno anche shift register e RAM***.

Nella nomenclatura del datasheet si distingue la slice in *parte top e bottom* (figura 8.6), in entrambe troviamo un registro, poi un carry con una porta and, che è la logica che consente di fare somme con la logica intelligente di propagazione del riporto. Oltre a ciò abbiamo un mux che nella *parte bottom* si chiama *f5 mux*, nella parte fi mux, dove i vuol dire che il pedice non è sempre lo stesso. Dunque, ogni slice ha due metà, top e bottom, ogni slice ha 4 input LUT function generator (2 LUT) ha due elementi di storage (uno top e uno bottom), i multiplexer e poi la logica di carry. ***In più la slice M ha gli shift register a 16 bit e i distributed RAM block***. Questi componenti hanno lo stesso ruolo che hanno i codici del processore rispetto alla compilazione: qualcuno è sicuro che tutte le figure di programmazione che abbiamo scritto si possono poi compilare sull'hardware che abbiamo. Come quando abbiamo un programma C e viene tradotto in istruzioni di linguaggio macchina (assembler), così ***esiste un compilatore che vedendo l'architettura sotto è in***

grado di riconoscere le figure di programmazione VHDL e di portarle sull'architettura. Si è creata la rete combinatoria per realizzare le macchine combinatorie, servono le macchine sequenziali per fare quelle sequenziali, servono moltiplicatori e addizionatori perché il compilatore deve tentare di fare anche un po' di sintesi comportamentale, se scriviamo allora la moltiplicazione, il compilatore la riconosce e la va a mappare, laddove invece è più complicato deve fare più cose (l'automa, la codifica, ecc.). **Se aiutiamo il compilatore facendo noi dei pezzetti piccoli, che si collegano tra loro, il compilatore è avvantaggiato; se invece facciamo un programma enorme che deve andare sul silicio, il sistema è svantaggiato.** Per questo un sistema a blocchi avvantaggia, perché è più facile da portare sul silicio. Tanta modularità serve per arrivare a cose semplici che sappiamo essere implementabili. Più facciamo invece dei processi e più saranno macchine sequenziali su cui non abbiamo controllo, i blocchi sono assegnati secondo la logica del compilatore. L'algoritmo codifica un problema con un linguaggio che dev'essere interpretato/compilato da una macchina, **maggior è la distanza tra il linguaggio e la macchina più difficile sarà la sintesi, potrebbe anche non essere fattibile.** Per questo spezziamo automi grandi in piccoli, facciamo una parte di controllo e una operativa: nella parte operativa mettiamo tutte le operazioni (modello dataflow), nel controllo invece mettiamo solo un automa, generiamo solo i segnali di abilitazione.

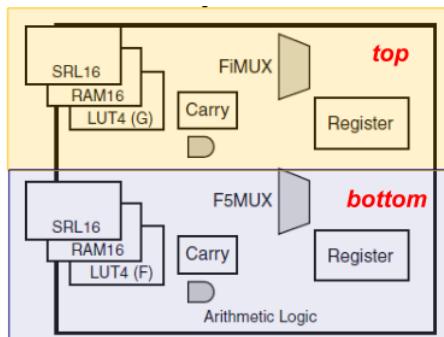


Figura 8.6: Struttura della Slice

I mux servono perché se avessimo solo le LUT con queste potremmo fare una qualsiasi funzione di 4 variabili ma se ne vogliamo di più possiamo usare i mux. L'F5 ogni volta che c'è collega gli ingressi di due LUT, manda gli stessi ingressi alle due LUT e utilizza un segnale di selezione, in tutto così possiamo fare una funzione di 5 variabili. Se vogliamo usarlo proprio come mux, l'architettura sfrutta ogni LUT per realizzare un piccolo mux 2:1 e poi ne combina due per farne uno 4:1. **Ogni mux ci consente di fare poi funzioni più grandi,** l'f5 ci consente di fare anche alcuni funzioni di 9 ingressi, quelle in cui i 4 bit nell'una e nell'altra LUT non sono indipendenti tra loro. Con F5 possiamo aumentare il set di funzioni rappresentabili. F6 prende l'uscita di 2 F5, ci permette di farli ancora più grandi. Per fare ciò dobbiamo attraversare più slice, perdiamo di prestazioni ma facciamo funzioni più grandi.

C'è anche l'**aritmetic logic** sulle slice, infatti abbiamo visto nei riquadri precedenti il blocco carry con una porta AND. Tuttavia nella realtà l'architettura implementata è quella che vediamo in figura 8.7, dove evidenziamo le parti che riusciamo a riconoscere. Abbiamo visto il ripple carry ma esiste anche il carry lookahead, in cui si faceva il discorso sulla condizione di propagazione. Se dobbiamo sommare due bit e sono entrambi alti, siamo sicuri di aver generato il riporto perché $1+1=0$ con riporto di 1; se invece abbiamo 2 bit di cui uno è alto e l'altro è basso, generiamo il riporto solo se avevamo 1 in ingresso, questa è la condizione di propagazione. Se guardiamo in una slice, nella parte bottom, con la LUT realizziamo la XOR: il multiplexer cerchiato in verde indica che o mettiamo fisso 1, quindi facciamo sempre propagare, oppure vediamo se davvero propagare oppure no. Immaginiamo che all'uscita della F-LUT ci sia 1, quindi c'è da propagare un riporto,

in uscita dal multiplexer cerchiato rosso esce il CIN di ingresso che arriva dal basso, altrimenti se la porta AND dà uscita alta, per cui $a_0 * b_0$ genera una condizione di propagazione, esce dal primo multiplexer più grande ed esce direttamente dal mux rosso. **Stiamo realizzando una piccola struttura carry lookahead sulla singola cifra.** Le due LUT all'interno di una slice sanno fare la somma tra due stringhe di due bit. Se volessimo realizzare il sommatore, in maniera comportamentale non sappiamo cosa succede, mentre in modo strutturale possiamo avvicinarci a quello che c'è sotto, addirittura possiamo adoperare delle primitive per usare proprio le risorse delle slice ed il datasheet ci indica come farlo. La cosa interessante è che **nella slice c'è una linea di carry già predisposta che propaga il riporto nella stessa colonna.** Osserviamo che finché rimaniamo nella stessa colonna dell'FPGA, la linea di carry va diretta, mentre se dobbiamo spostarci in un'altra colonna di CLB ci saranno problemi di interconnessione. Se facciamo considerazioni su come variano le prestazioni dei sommatori al variare degli ingressi, mentre per un certo periodo abbiamo un andamento lineare, ad un certo punto avremo un andamento diverso dovuto all'impatto dell'interconnessione, che, nonostante sia efficiente, ha comunque un peso.

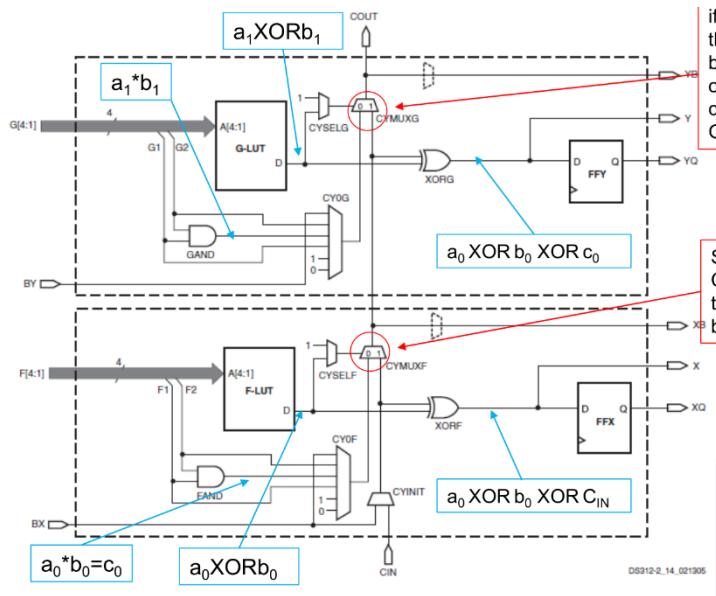


Figura 8.7: Aritmetic Logic

Le SLICEM hanno una parte di memorizzazione di RAM e shift register: si possono programmare le LUT per funzionare invece che come generatore di funzioni di 4 variabili, proprio come delle memorie laddove i 4 ingressi sono gli indirizzi e nelle LUT ci sono i valori. Servono più bit per memorizzare i dati. Possiamo fare lo stesso per uno shift register: abbiamo 16 possibili locazioni, le possiamo vedere come 16 bit in sequenza all'interno di uno shift register e possiamo programmare la LUT per fare lo shift ogni volta.

I **DCM** servono per aggiustare skew e jitter. **Eliminano lo skew, basandosi sulla tecnica di delay locked loop**, prendono una frequenza in ingresso e fanno degli aggiustamenti di fase per far sì che il segnale in uscita abbia stessa fase e frequenza di quello in ingresso. Possiamo anche sintetizzare nuove frequenze, noi abbiamo un oscillatore al quarzo con una sua velocità, se ci serve una velocità più lenta o veloce, sfruttiamo ciò. Il DCM consente di fare anche moltiplicazione, divisione di frequenza e phase shifting.

Osservazione: Il quarzo, a seconda di come viene tagliato, una volta alimentato, messo in un circuito elettronico, fa generare una differenza di potenziale che crea un'onda quadra. La purezza

del quarzo, il taglio e il circuito elettronico, rendono il circuito più o meno stabile, dunque c'è un errore nell'oscillazione. C'è poi *l'errore di skew, significa che la frequenza non è proprio rettangolare ma tende ad allargarsi per gli effetti parassiti presenti sulle linee di comunicazione, più aumenta la frequenza e più le onde tendono ad allargarsi*: il sistema deve compensare sulle linee di trasmissione tutte le cose che tenderebbero a fare allargare dal punto di vista elettrico il segnale, che diventa non quadrato ma smussato, se lo diventa troppo diventa complesso discriminare 0 da 1. *Tenta quindi di riportarlo il più possibile con dei fronti che varino in periodi piccoli, in modo da poter essere riconosciuti come 0 o 1 in periodi piccoli.* Il quarzo sta dentro, senza la batteria il circuito elettronico non oscilla, *il quarzo nel circuito elettronico produce un'onda quadra perché il quarto alimentato produce una differenza di potenziale*, un segnale oscillante che se vogliamo sia preciso costa. Il quarzo ha una sua precisione e in base a questo dividiamo il clock altrimenti il sistema non funziona.

La parte di interconnessione è complessa, **abbiamo una switch matrix per ogni componente logico**, per ogni CLB, per ogni I/O, per ogni DCM e anche per ogni componente moltiplicatore. Questi blocchi sono collegati tra loro tramite le *interconnect resources*, dei fili particolari, con strutture particolari. *Ad esempio, le long lines* sono a blocchi di 24 fili e collegano **ogni 6 tiles insieme**, sono fatte sia in orizzontale che in verticale e consentono di arrivare lontano all'interno dell'FPGA. *Sono molto buone per trasportare i segnali di clock*, lavorando ad alta frequenza, quindi laddove non ci siano linee dedicate, sono finite le linee per i clock, i segnali di temporizzazione si possono portare su queste lines. *Altri tipi di linee sono hex, double, direct connections*, che collegano 3:3, 2:2, tutti i CLB nelle vicinanze. Questo va programmato quindi, quando facciamo il nostro design, accediamo alla parte di sintesi, non solo dobbiamo dire in ogni latch cosa deve andare ma anche come mettere tutte queste interconnessioni insieme.

Per programmare un FPGA dobbiamo avere il *bit stream*, vuol dire che, dato l'FPGA, seppur abbiamo implementato nel design una porta AND, **dobbiamo programmare tutti gli elementi di memoria dell'FPGA**, quindi il bit stream avrà sempre la stessa dimensione fissato il tipo di device. Per la programmazione possiamo fare due cose. La prima è **caricare il bit stream in una memoria flash esterna**, all'accensione o dopo un reset il device si auto-configura dalla flash, la cosa positiva è che è salvato lì e l'FPGA si comporta come il componente che abbiamo progettato. Altrimenti, dobbiamo programmare il dispositivo ogni volta con un sistema diverso. Il protocollo utilizzato per la programmazione è il **protocollo JTAG**, utilizzato per il testing. È un protocollo seriale, in maniera seriale diamo i bit della programmazione dell'FPGA, tramite la **boundary scan chain**. Diamo tutti gli input ai vari sistemi che manterranno i dati, in maniera volatile, per cui al successivo spegnimento della scheda il bit stream caricato va perso e va ricaricato.

Per la famiglia Artix 7 l'architettura è fondamentalmente la stessa, solo che le LUT sono un poco più evolute, perché invece di essere a 4 ingressi sono a 6 ingressi; mentre prima avevamo due registri per slice, nell'Artix 7 abbiamo 8 FF; le slice di tipo M non sono più in numero uguale rispetto a quelle L, sono tipicamente di meno per motivi di prestazioni; ci sono anche sistemi di connessione più evoluti; ci sono slice fatte apposta per DSP, processamento dei segnali. La tecnologia di realizzazione è più evoluta, c'è una grande differenza perché nello stesso spazio possiamo fare più cose.

Osservazione: Quando vogliamo fare la sintesi, la distanza tra com'è fatta la piattaforma e quello che vogliamo fare dice se il sistema è sintetizzabile o meno. Il PAL ci permette di avere delle porte semplici, abbiamo una maggiore programmazione della parte or/and nelle due differenti tecnologie. Fa una tabella di verità, implementa una forma canonica di tipo P. Se possiamo retro-azionare una di queste porte possiamo fare i FF. Possiamo programmare l'interconnessione tra le porte logiche. Se vogliamo fare una rom è facile, se vogliamo fare un FF con PAL e PLA è complicato, dobbiamo fare and/or retroazionare, non è facile farne il place route. Finché facciamo automi semplici la tecnologia viene bene, quando diventa più complicato non ci aiuta tanto. Il PLA brucia dei contatti, i CPLA invece sono programmabili, hanno dei sistemi divisi in due livelli. Abbiamo dei componenti

base, in numero limitato, si devono interconnettere con altri componenti base. Abbiamo una serie di matrici per connettere tra loro isole di componenti. Queste isole sono composte da LUT, in grado di implementare una funzione combinatoria, poi addizionatore a propagazione di riporti (che serve negli addizionatori, nei moltiplicatori, ecc.). poi abbiamo dei registri, è più vantaggioso. Se dobbiamo fare il modello RT (transfer register) il PLA non è immediato, l'FPGA invece sì. La cosa migliore sarebbe l'ASIC ma è più complesso. Non possiamo fare un PLA perché la distanza tra quello che dobbiamo fare e quello che ci dà il PLA è troppo grande. Oggi PLA e PAL si possono usare ma si usano per lo più gli FPGA, per una classe di applicazioni. Le prestazioni di cose già definite sono più veloci, questo chiarisce perché ci rivolgiamo a un FPGA. Il vantaggio di PLA e PAL rispetto a FPGA è che hanno una struttura più nota, la testabilità è maggiore. La minimizzazione può servire in alcuni casi, se riusciamo a usare un numero di porti minori può essere vantaggioso. In generale, supponiamo di avere una forma canonica di tipo P di 4 variabili, ha 16 possibili addizioni, normalmente non ci sono 16 possibili ingressi alle porte ma in numero limitato. Se abbiamo ad esempio 9 mintermini facciamo la funzione di 7 e poi il complemento. Oppure se ne vogliamo fare una di 9 e ne abbiamo 8 dobbiamo comporre tra di loro le funzioni. Se riusciamo a minimizzare invece entriamo negli 8 fili e quindi non abbiamo questo problema.

N.B: Si consiglia la lettura dell'appendice D per un approfondimento riguardo ai dispositivi logici programmabili.

8.3 Protocollo JTAG

Gli FPGA sono largamente impiegati grazie alla facilità con cui è possibile programmarli. Tipicamente un pc, tramite un opportuno protocollo, configura i dispositivi. Non serve un hardware speciale o un protocollo, il protocollo di comunicazione è JTAG: veloce, versatile e presente in tutti i circuiti digitali. La scheda che abbiamo e il programma dell'ambiente di sviluppo si collegano tramite il protocollo JTAG che consente alla scheda di parlare con il pc e ricevere la configurazione. JTAG viene sfruttato per configurare gli elementi di memoria nel PLD per ottenere il comportamento desiderato. JTAG però non nasce con questo intento ma per il testing di circuiti integrati digitali. È un protocollo nato con un'altra finalità, viene poi applicato a questo sistema per poter configurare il circuito. È un protocollo seriale che permette di gestire, tramite un'apposita architettura la *boundary scan chain*.

Se pensiamo a un sistema retroazionato, abbiamo una difficoltà notevole nel testarlo: essendoci la retroazione, mettiamo un segnale e solo per alcuni segnali siamo in grado di far variare i valori nei registri, perché se lo stato rimane lo stesso non abbiamo la possibilità di vedere che è cambiato. Mettiamo un segnale e vediamo l'uscita ma se l'uscita è sempre la stessa e lo stato non cambia non possiamo testare niente. Per testare serve un numero di prove molto elevato, alcune sono inutili, altre utili. Le tecniche di scan aprono la retroazione, separano i banchi di registri dalla rete combinatorie, consentono di testare separatamente la rete combinatorie e i registri, se funzionano entrambi l'insieme funziona, evitando di avere la retroazione. Dato che normalmente i contatti ci sono, il protocollo JTAG si usa per levare le retroazioni e consentire così il test del circuito, separatamente. Questo è il motivo per cui questo protocollo è stato adattato per riconfigurare i sistemi. All'interno dei sistemi non c'è solo il pezzo visto ma molte più cose, per questo è complicato. La diffusione di questa tecnica (*boundary scan*) è dovuta ai vantaggi che offre rispetto alle altre tecniche adoperabili: richiede pochi pin di interfacciamento, è generale, è compatibili con le tecnologie produttive, ha una gestione semplice e permette di raggiungere ogni componente sul chip. Se abbiamo un sistema che deve lavorare normalmente, interconnettiamo la macchina così come deve essere interconnessa, se la vogliamo provare nel suo funzionamento inseriamo il multiplexer, che, invece di mandare tutti i registri dove devono andare, crea una grande catena di registri consentendo di testarli. Supponiamo di avere una macchina con un flip flop in retroazione (FF) e poi di avere un'altra macchina,

con cui si interconnette, come mostra la figura 8.8. Sono due macchine uguali. Se abbiamo un funzionamento normale vuol dire che l'uscita della macchina combinatoria viene retroazionata ed entra nel registro, se aggiungiamo il multiplexer, invece, l'ingresso può provenire da due strade, dalla macchina combinatoria o dall'oggetto successivo. Se arriva il segnale normale prendiamo il valore nel registro, se arriva il segnale di *scan* prendiamo il valore dall'elemento che può essere il segnale che proviene da un multiplexer che sta prima. Se mettiamo la retroazione, otteniamo il sistema retroazionato, senza invece è un sistema aperto. **Tramite il multiplexer possiamo realizzare il sistema aperto.**

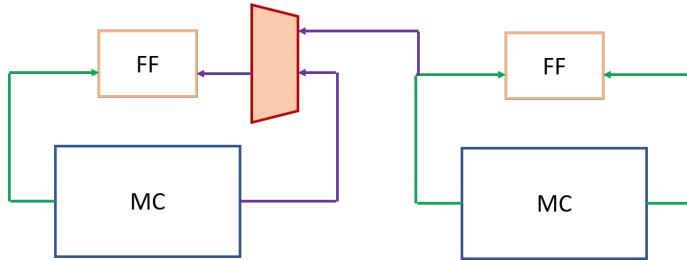


Figura 8.8: Schema retroazionato con multiplexer

Se non avessimo il multiplexer, prenderemmo il segnale sempre in senso antiorario, prenderemmo sempre la linea del corretto funzionamento, ottenendo una macchina chiusa con retroazione. Per pilotare il FF dobbiamo pilotare la rete che pilota il FF, che è più complesso perché per molti ingressi il FF non cambia il valore. Supponiamo di stare in q_0 (000), poi andiamo in q_1 (001) e poi in q_2 (010). Supponiamo di voler testare il primo flip flop (il bit più significativo), finché rimaniamo in q_0 il FF sta sempre a 0, non cambia il valore, se lo vogliamo testare non vediamo nulla, per cui se si rompe la scrittura non possiamo testarlo. Se viceversa andiamo da q_0 a q_1 abbiamo cambiato stato, ma quel FF sta sempre a 0, non lo riusciamo a vedere. Se deve andare da q_0 a q_2 , con questa codifica, il FF di peso più significativo sta sempre a 0. Abbiamo dato tre ingressi ed il FF non è mai variato, non riusciamo a testarlo, perché se diamo l'ingresso e lo stato cambia il FF rimane sempre allo stesso valore. *Il nostro problema è vedere se funziona o meno, cioè se da 0 scrive 1 e da 1 scrive 0.* Se è sempre presente la retroazione non risolviamo il problema, dovremmo mettere per ognuna di queste macchine molti segnali per testarli, ma per testarlo in realtà servirebbero due valori, cioè se una volta scrive 0 ed una volta scrive 1, ma per scrivere 0/1 dovremmo vedere il funzionamento complessivo di tutta la macchina. L'idea allora è mettere un oggetto in mezzo che, nel normale funzionamento, chiude la retroazione, perché la macchina normalmente deve essere retroazionata, durante il test invece apre la retroazione, prende un segnale da fuori. **La macchina che chiude o apre la retroazione è un multiplexer perché se non c'è il segnale di scan funziona come se il filo fosse retro-azionato, mentre se c'è il segnale di scan prende da un elemento esterno il valore.** È un grande vantaggio perché siamo riusciti ad aprire il sistema però *abbiamo perso un po' in velocità, in quanto abbiamo messo un'altra macchina, ma se non aggiungiamo quella macchina il sistema non è testabile*, per cui va per forza aggiunta.

Se facciamo passare un segnale che diamo in ingresso al registro di destra, in ingresso al registro di sinistra, abbiamo fatto un registro a scorrimento perché abbiamo connesso i FF tra loro: **invece di testare uno solo FF, possiamo testare un insieme di flip flop, mandandogli in ingresso una sequenza, in serie, e vediamo se i FF la recepiscono o meno.** Questo consente di creare delle catene tra gli elementi di memoria, che è importante perché **abbiamo diminuito il numero dei pin che escono per il test**, perché ne esce uno solo che va al primo, al secondo, terzo e così via e vede tutti i registri in che configurazioni sono. Questa tecnica è basata su un multiplexer e consente di aprire o chiudere i circuiti con un numero limitato di pin. Il vantaggio è che il MUX apre la configurazione e il segnale può entrare. Questa tecnica è diventata buona per fare la configurazione, perché decide di pilotare un multiplexer che interconnette o meno delle parti tra loro, è poi un po' più complesso perché va realizzata la LUT.

Se vogliamo testare i registri dobbiamo capire se riusciamo a scrivere 0 o 1. Se portiamo tutti i fili fuori, riusciamo a fare il test, ma dovremmo portare un filo fuori per ogni registro, nell'esempio in figura 8.9 sono 4. Per poter fare il test con i 5 fili, dato che il registro R può essere alimentato o dalla macchina o dall'esterno, dobbiamo inserire il multiplexer che presi due fili in ingresso ne fa uscire uno. Avremmo risolto il problema di come pilotare i registri ed essere sicuri che ci sia 0 o 1, ma se facciamo così *non abbiamo risolto il problema del numero di fili*. **Per ridurre il numero di fili, quando il sistema si apre interconnettiamo tutti questi registri tra loro, realizzando un registro a scorrimento**, il cui vantaggio è che ha un punto di ingresso, per cui se mettiamo un uno in ingresso e il registro a scorrimento è fatto da 4 elementi in 4 passi riusciamo a posizionare l'uno in ognuno di essi. *Ovviamente impieghiamo più tempo*, perché invece di dare 5 segnali dobbiamo aspettare 4 quanti di tempo affinché il segnale si propaghi dappertutto, d'altro canto diminuiamo il numero di pin, altrimenti non sapremmo proprio realizzarlo.

Osservazione: il test spesso viene effettuato quando la macchina è *fuori linea*, in tal caso allora perdere del tempo in più non è un grande problema. Questo non è sempre vero, perché se abbiamo macchine che fanno sistemi ferroviari e le vogliamo testare mentre lavorano, durante il tempo in cui le testiamo non possono operare, eseguire algoritmi per cui se il test è molto lungo l'utilità della macchina diventa nulla. **Dobbiamo capire quanto dura un test rispetto all'utilizzo della macchina**, se è fuori linea non c'è problema, se è in linea c'è un problema.

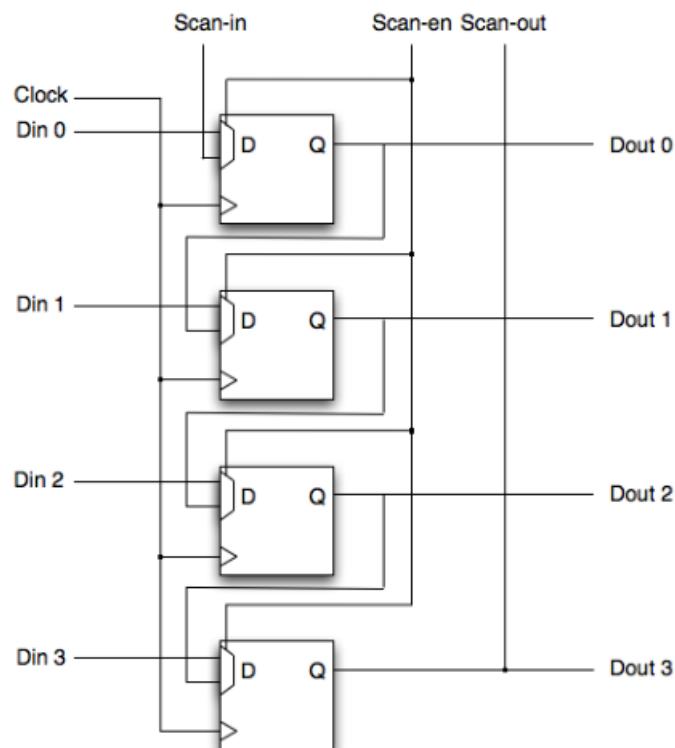


Figura 8.9: Boundary Scan Chain

Abbiamo creato questa rete di interconnessione perché i circuiti si possano riconfigurare tra loro mediante protocolli fatti con bit che arrivano in serie, i quali chiudono ed aprono i registri. Il file che arriva seriale consente di chiudere ed aprire questi registri, **JTAG è il protocollo che manda la serie di bit al registro a scorrimento, i cui elementi aprono o chiudono le porte**. A seconda di quello che aprono o chiudono, configurano il circuito. Come configurano il

circuito non ci interessa perché dipende da come è fatto il compilatore tecnologico di quella scheda.

Cerchiamo di capire perché **possiamo usare il boundary scan sia per il testing sia per la configurazione**. Se dobbiamo scrivere un 1 che apre una porta, comunque 1 dobbiamo scrivere. *Se siamo in fase di test dobbiamo anche leggere, cioè controllare che, avendo scritto 1 leggiamo effettivamente 1, mentre se siamo in configurazione dobbiamo solo scrivere 1*, perché l'1 è il punto di partenza da cui si aziona il sistema. Scrivere 1 vuol dire configurare il circuito, il circuito lo abbiamo, dobbiamo solo scegliere cosa aprire e chiudere. Per aprire e chiudere in modo permanente non possiamo fare la and e la or, perché se lo facessimo con le porte logiche, quando va via il segnale andrebbe via anche l'alimentazione. Per questo c'è anche il registro, dato che ogni cosa che configuriamo deve avere anche un elemento che rimanga stabile. *Il problema della configurazione è scrivere un valore in un registro*. Osserviamo la figura 8.10, esce un filo, Dout, se è 1 configura la macchina in un modo, altrimenti se è 0 in un altro. Se dobbiamo fare lo scan, il Dout lo dobbiamo poi mettere in ingresso a qualcosa, per controllarlo, se invece non dobbiamo, basta modificarlo. Ecco perché parlare di test o di configurazione è grossomodo la stessa cosa, perché parlare di test vuol dire vedere cosa c'è scritto in un registro ma comunque scrivere in un registro, perché vogliamo avere un effetto, l'effetto va sulla configurazione mentre il valore viene dall'altro lato. Alcuni registri (quelli che servono a switchare) non si cambiano mai, altri si possono testare e ricambiare, sono due insiemi differenti.

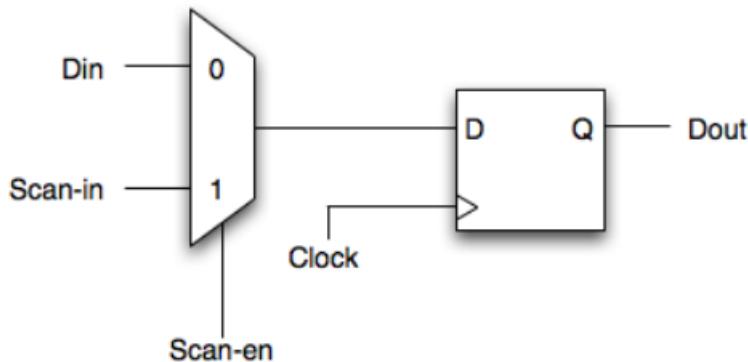


Figura 8.10: Boundary Scan Chain

N.B: Si consiglia la lettura dell'appendice E per un approfondimento circa il protocollo JTAG.

8.4 Timing Analysis

ISE ha un tool interno, che *prende in ingresso il design vhdl ed eventuali constraint*, forniti da noi, ed *effettua la sintesi* che consiste in tre passi. Il primo è il **parsing**, cioè il check sintattico; la parte di **hdl synthesis**, in cui si cercano di individuare delle macro, in altre parole *il tool cerca di individuare pattern conosciuti per poi arrivare a sintetizzare con dei blocchi che ha effettivamente a disposizione sulla logica target*. Va ad individuare così delle **macrofunzionalità**, tuttavia non siamo ancora a livello della logica. Nell'ultima fase, **low level optimization**, si prendono le macro e si cerca di fare ancora un'altra ottimizzazione, per esempio eliminando pezzi che non servono. ISE dà la possibilità di configurare il **design goal**, cioè l'obiettivo che vogliamo nel momento in cui lanciamo la sintesi. I goal principali sono l'**ottimizzazione di tempo**, l'**ottimizzazione di area** o la **power reduction** per realizzare un circuito che consumi il meno possibile. *Di default il goal di ottimizzazione è la velocità*, è possibile in ISE settare questi parametri e quindi cambiare l'optimization goal. Nei goal si possono definire strategie precise, quindi modificare delle proprietà, parametri per la parte di sintesi, definito un certo goal. La strategia predefinita di ISE, che utilizza

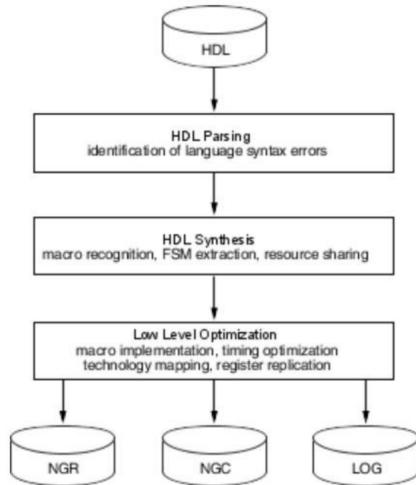


Figura 8.11: Processo di sintesi in ISE.

quando tipicamente noi lanciamo la sintesi, si chiama **balanced**, il cui goal è sempre la speed, ma cerca di bilanciare il goal di ottenere un design molto veloce con il fatto che il processo di sintesi sia oneroso. Effettua un bilanciamento fra il goal e l'effort che deve mettere il processo di sintesi. Si può cambiare strategia, per esempio si può privilegiare il runtime (vogliamo anche impiegare molto tempo per la sintesi però vogliamo un design quanto più possibile ottimizzato) oppure possiamo ottimizzare il power per cui l'obiettivo è la riduzione di potenza o ancora sul runtime, quindi vogliamo minimizzare il runtime per la sintesi.

Si possono definire dei **constraint**, quelli che mettiamo nell'ucf riguardano il mapping tra i segnali del design ed i pad dell'FPGA, che poi nel nostro caso si traducono con i sistemi di input/output della board, ma in realtà questi sono collegati uno alla volta con i pin esterni dell'FPGA. Nello stesso ucf, possiamo definire constraint specifici di tempificazione.

In realtà, si possono definire i constraint in due file diversi, uno è l'ucf e l'altro è il **file .xcf** che ha la stessa sintassi, si scrive allo stesso modo, ma, se aggiunto, *può essere preso in considerazione durante la sintesi*. Se inseriamo i parametri di tempo nell'ucf, la sintesi non li guarda, quindi l'analisi temporale comincia dopo la sintesi, cioè nella fase di implementazione. **E' possibile specificare requisiti di tempificazione da prendere in considerazione già durante la sintesi;** se la sintesi arriva solo a definire le macro, quindi non va sulla logica di base, *il risultato della tempificazione non è preciso perché non abbiamo mappato gli oggetti sulle slice dell'FPGA e non abbiamo configurato la rete di interconnessione*, quindi non siamo in grado di calcolare i tempi veri, ma possiamo fare già delle operazioni che vanno verso l'operazione di ottimizzazione dei tempi. Nel nostro caso non le facciamo, a livello industriale possono aver senso.

Dopo la fase di sintesi, c'è quella di **implementazione**, che comprende a sua volta altri passi: **translate**, che mette insieme i risultati della sintesi con i constraint dati creando un file che va in input agli step successivi. C'è poi la sotto-fase di **map**, la quale decide che ad una determinata macro deve corrispondere una precisa LUT o un componente dell'FPGA, quindi *in questa fase i componenti che sono realmente all'interno dell'FPGA vengono mappati con le macro-teoriche di alto livello*. L'FPGA ha tanti device, messi in forma di matrice, quindi, dopo la fase di map, sappiamo che una precisa logica del nostro progetto viene messa in una determinata slice, ma non sappiamo ancora se utilizziamo la slice, ad esempio, della seconda riga quarta colonna. In alcuni casi settando opportune proprietà possiamo fare anche il **placement** (decidiamo che è una slice,

ma anche che è la slice numero [2,3]). Una cosa che non facciamo nel mapping è il routing, cioè definire la matrice delle interconnessioni. *Già dopo il mapping possiamo fare l'analisi di una temporizzazione della macchina*, anche se non siamo ancora vicini alla verità, proprio non c'è la parte di routing, che potrebbe essere pesante (la rete di interconnessione a bordo dell'FPGA è complessa, fatta per ottimizzare percorsi sia di componenti lontani che componenti vicini). Successivamente c'è il **place&route**, che effettua il placement delle risorse e ne definisce le interconnessioni tenendo conto dei timing constraint eventualmente specificati. L'ultimo passo è la **generazione del file di programmazione**, il bit stream.

Quando generiamo i report sui tempi, a seconda della fase in cui li leggiamo hanno senso diverso. Quelli del file ucf sono i **constraint lock**, i quali indicano un certo input/output del design a quale pin corrisponde, poi ci sono *constraint per il tempo*. Fra questi ultimi, che possono essere applicati sia a livello globale di tutto il design ma anche a parti di esso (possiamo individuare dei moduli, delle reti), il primo più importante è il constraint period, scritto come:

“NET nome_della_porta PERIOD = valore”

dove “valore” è il valore che desideriamo abbia il periodo del clock. Generalmente, generiamo un design, che è sincrono, a cui diamo il clock della scheda, tuttavia ci chiediamo cosa succeda internamente al circuito che stiamo creando. *Qual è la vera tempificazione del circuito?*

Se abbiamo requisiti molto stringenti, abbiamo bisogno di essere sicuri che i requisiti temporali siano soddisfatti, per esempio vogliamo che tutto il design sia clockato con un clock di 5ns, scriviamo il constraint con period ed il tool farà di tutto affinché sia rispettato, se non riesce lo comunica. Questo vuol dire che, se non diamo il constraint, il tool fa del suo meglio, però è probabile che otteniamo un valore peggiore.

Nota 8.1: Il segnale di clock

Il clock arriva sui componenti della scheda mediante una linea di trasmissione. La linea di trasmissione risponde all'equazione di telegrafisti, più lunga è la linea e più capacità e resistenze abbiamo, quindi l'onda quadra del clock si allarga sempre di più, questo effetto di allargamento si chiama **skew**. Se vogliamo essere sicuro che abbia una certa forma, dobbiamo tentare di capire se all'interno dei percorsi che facciamo fare al segnale riusciamo a limitare la lunghezza dei collegamenti, perché collegamenti più corti consentono di avere un effetto di skew minore. Il clock che prendiamo noi dalla scheda può oscillare ad una certa frequenza, ma se vogliamo mandare il clock all'ultimo componente di una catena di componenti, che deve essere sincronizzata dallo stesso clock, è molto probabile che l'ultimo componente avrà un clock allargato con questo effetto di skew. **Con questi constraint stiamo dicendo al tool di tentare di non allargare troppo questo fronte di clock**, se non ci riesce lo comunica, se ci riesce vuol dire che riusciamo a fare un place&route di tutte queste attività. Più piccola è la scheda, meno probabilità ha di riuscirci perché deve passare tra più blocchi e più blocchi significano attraversamenti e quindi che lo skew aumenta a dismisura. Nell'FPGA possiamo provare, nell'ASIC questa linea di trasmissione può essere disegnata in modo opportuno, ecco ancora la differenza tra il dominio dell'ingegneria elettronica e quella informatica.

Il constraint di PERIOD si applica solo ai design che hanno elementi sequenziali sincroni all'interno, se facciamo una macchina combinatoria e proviamo a scrivere un constraint del genere nell'ucf otteniamo qualcosa che non ci aspettiamo. Il constraint PERIOD si usa per design sincroni e si riferisce a tutti i percorsi tra due flip-flop. Gli elementi sincroni di base sull'FPGA sono dei FF, perché sull'FPGA ci sono slice, di tipo M e di tipo L, al cui interno hanno anche la logica sequenziale.

Il tool consente di inserire dei constraint nei percorsi combinatori, come il percorso pad-to-pad che è il constraint FROMTO, che si può definire a livello globale (per cui tutti i percorsi pad to pad devono rispettare quel certo vincolo) oppure possiamo stabilire che il percorso tra a e b deve essere di un certo tipo. Inoltre, possiamo guardare i percorsi che vanno da un pad al primo elemento sequenziale che incontrano, oppure dall'ultimo sequenziale fino ad un pad. Dopo la fase di map, possiamo già fare un'analisi che non è però accurata, per cui conviene farla direttamente dopo la PAR (place&route). Per poter fare la timing analysis dobbiamo configurare opportunamente il tool.

8.5 Timing Analysis: CLA puramente combinatorio

Partiamo dall'analisi dei tempi del circuito carry look ahead, macchina combinatoria per la somma di stringhe di 8 bit. Nel menù di sinistra, dei processi da fare, dopo il map e dopo il place e route

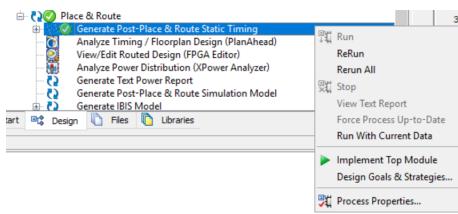


Figura 8.12: Process properties.

c'è la fase di generate post map static timing e generete post place&route static timing. Clicchiamo su questo processo con tasto destro e poi selezioniamo "Process Properties". Da qui, si apre una finestra che consente di selezionare le proprietà del processo selezionato (abbiamo considerato direttamente il post PAR).

Poiché non abbiamo constraint nel design, *dobbiamo fare in modo che nel report generato di de-*

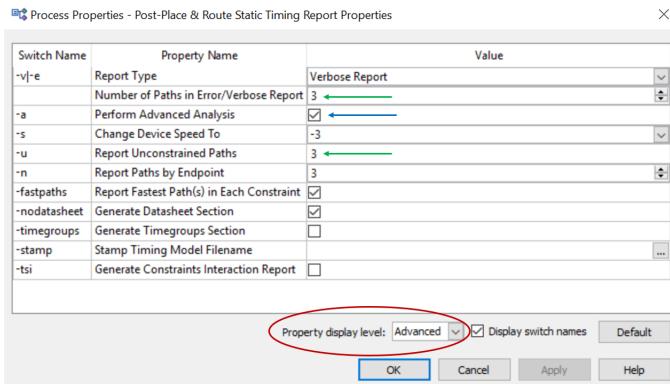


Figura 8.13: Process properties.

fault il tool vada ad inserire il report sulla tempificazione dei path unconstrained, cioè i path su cui non abbiamo definito constraint, dato che se non lo specifichiamo noi non li genera.

Selezioniamo il property display level come advanced (cerchiato in rosso in figura 8.13), nel campo report unconstrained path inseriamo un numero positivo che indica quanti unconstrained path vogliamo vedere nel report (nel nostro caso pari a 3). Decidiamo di avere un report verboso (selezionato in "Report Type"), di modo che dia tutte le informazioni possibili; poi, troviamo il "number of paths in error verbos report", che settiamo pari a 3.

Il sistema ordina i path in base al tempo e sono dati da tutte le possibili combinazioni ingresso/uscita, noi possiamo decidere quali vedere (con l'unconstrained path) e mettiamo anche il report path raggruppati per end point (per pin), è un modo più semplice di visualizzazione. Per settare queste cose dobbiamo spuntare “performance advanced analysis”. Poi facciamo apply, ok e facciamo girare di nuovo la timing analysis. Un file .twx viene generato, e aprendolo da ISE è possibile

```
=====
Timing constraint: Default path analysis
125 paths analyzed, 9 endpoints analyzed, 0 failing endpoints
0 timing errors detected. (0 setup errors, 0 hold errors)
Maximum delay is 8.051ns.
=====
```

Figura 8.14: TA del Carry Look Ahead.

leggere i risultati del report. In particolare, in figura 8.14, otteniamo il maximum delay, che è sicuramente più grande di quello che otterremmo in’analisi post map, dato che considera anche i tempi di routing. E’ un tempo attendibile che tiene conto anche dei percorsi fino ai pad. Non possiamo dire che il sommatore impiega 8.05 ns, ma è il sommatore più tutto ciò che serve per arrivare agli estremi dell’FPGA.

8.6 Timing Analysis: CLA con registri

Per poter valutare il sommatore, possiamo realizzare un *top module* più grande che lo contenga, mettendo un registro a monte ed uno a valle, a questi due registri diamo uno stesso clock e reset. *Tutti gli input che vanno verso la macchina diventano del registro di ingresso, tutti gli output che uscivano dalla macchina diventano gli output del registro di uscita.* Se aggiungiamo i due registri

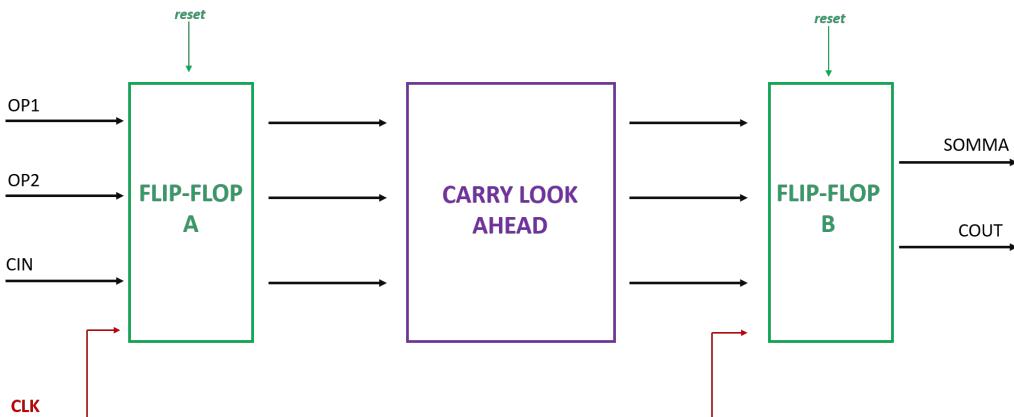


Figura 8.15: Schema CLA con registri.

ISE capisce che sono tali e li trasforma in FF; quando chiediamo di calcolare il tempo lo misura come tempo tra due elementi sincroni, cioè il tempo che ci vuole in tutti i path tra due elementi FF, ne prende il peggiore di tutti ed il worst-case è il tempo che impiega il sommatore.

A questo punto, realizzato il top module come mostrato in figura 8.15, passiamo alla definizione dei timing constraint. Clicchiamo sul top module, user constraint ed infine su *create timing constraint* (figura 8.16). Si apre, poi, una finestra, come mostrato in figura 8.17, in particolare definiamo un timing constraint sul segnale di clock (quindi abbiamo selezionato il clock domains nel menù sulla sinistra).

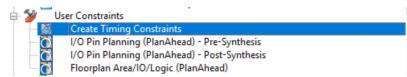


Figura 8.16: Timing constraint.

Fatto ciò, si genera il file .xcf in cui è definito il constraint come segue:

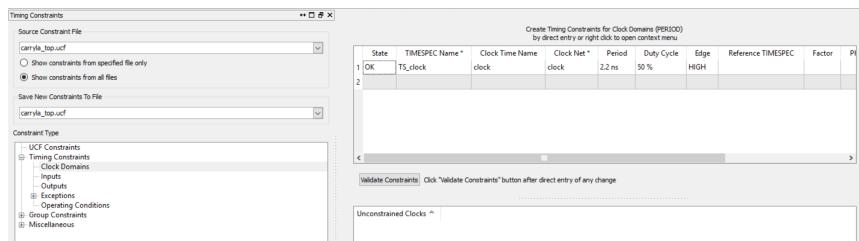


Figura 8.17: Timing constraint.

NET “clock” TNM.NET = clock; TIMESPEC TS_clock = PERIOD “clock” 2.2 ns

Eseguiamo nuovamente tutti i passi come visto precedentemente, ISE genera il file .twx, che aperto sempre sull’ambiente, ci permette di leggere i risultati del report, presenti in figura 8.18. Si osserva che il constraint è stato rispettato, abbiamo chiesto un period di 2.2 ns ed la timing analysis riesce a rispettarlo, addirittura fornendo un minimum period di 1.9 ns, più piccolo di quanto richiesto. Osserviamo infine *la presenza del net “clock_BUFGP”*: il clock, dovendo andare in vari punti del

```
=====
Timing constraint: Default period analysis for net "clock_BUFGP"
121 paths analyzed, 43 endpoints analyzed, 0 failing endpoints
0 timing errors detected. (0 setup errors, 0 hold errors)
Minimum period is 1.994ns.
=====
```

Figura 8.18: Timing Analysis CLA con registri.

design, viene bufferizzato internamente, quindi quello che si usa davvero è il buffer e può essere bufferizzato in vari modi.

Capitolo 9

Il processore

Nella seguente sezione, vedremo come è possibile progettare un processore, ovvero una *macchina programmabile*, che possiamo considerare come una *macchina dedicata ad eseguire un programma*. Rispetto ai sistemi studiati, il progetto di un processore risulta molto complesso, per cui spesso si ricorre a una logica differente: la logica microprogrammata.

9.1 La logica microprogrammata

Se dobbiamo realizzare un sistema complesso, conviene suddividerlo in due unità: unità di controllo e unità operativa, come visto nel paragrafo 1.7. Una volta progettata l'unità operativa, abbiamo due modi per realizzarne l'unità di controllo:

- **Logica cablata.** Progettiamo l'unità di controllo in termini di porte logiche.
- **Logica microprogrammata.** L'unità di controllo si costituisce di una micro-ROM e di una logica che la controlli. In tale ROM sono mantenute le microistruzioni necessarie allo svolgimenti di ciascun codice operativo, dunque i segnali necessari all'unità operativa per eseguire ciascuna istruzione.

Dal punto di vista dell'unità operativa è indifferente, in entrambi i casi l'unità di controllo provvederà a generare i segnali di abilitazione verso l'UO, ciò che cambia è solo come questo accade. Infatti, nel caso di logica cablata avremo una macchina sequenziale o una macchina combinatoria, realizzata tramite porte logiche, che a fronte di determinati input (e stati nel caso di macchina sequenziale) produrrà in output i segnali di abilitazione, *calcolandoli* istante per istante; nel caso di logica microprogrammata, invece, l'unità di controllo andrà a prelevare la sequenza dei segnali di abilitazione *memorizzati nella micro-ROM*, in corrispondenza del codice operativo da eseguire. *Nella logica cablata tutto ciò che accade è definito dall'architettura dell'unità di controllo, invece nella logica microprogrammata dal contenuto della micro-ROM.* Ciò significa che laddove fosse necessario effettuare alcune modifiche alla logica di controllo, nel caso cablato bisognerebbe riprogettare l'unità da capo, nel caso microprogrammato invece sarà sufficiente andare a modificare le sequenze memorizzate nella ROM, dunque il secondo modello risulta essere *maggiormente versatile*.

Vogliamo adesso realizzare un processore, *una macchina dedicata a svolgere istruzioni*, appartenenti a un determinato set. Il sistema allora si complica ulteriormente in quanto non esegue sempre la stessa operazione ma operazioni differenti, guidate dalle istruzioni. Anche in questo caso l'unità di controllo dovrà generare una sequenza, **la difficoltà consiste nel dover identificare, tramite la particolare istruzione, la sequenza di abilitazione da dare all'unità operativa.** Per realizzare l'istruzione in logica cablata l'unità di controllo calcola di volta in volta i

segnali da dare all'unità operativa, passo dopo passo; nel caso di logica microprogrammata invece l'istruzione deve identificare una particolare porzione della memoria, nel quale è contenuta una sequenza che da il via a tante microfasi. Se il processore è di tipo **RISC (Reduced Instruction Set Computer)** è più facile riuscire a realizzarlo anche in logica cablata, che per altro risulta essere più efficiente. Se invece il processore è di tipo **CISC (Complex Instruction Set Computer)**, avremo un maggior numero di istruzioni e ciò complica molto il progetto in logica cablata. Dovremmo avere tanti stati quante sono le possibili operazioni e inoltre, se ipotizziamo di avere molti modi di indirizzamento, i vari stati si complicano ulteriormente.

Supponiamo di voler realizzare il processore in logica microprogrammata, dev'essere presente un generatore degli indirizzi di partenza, ovvero l'indirizzo della micro-ROM a cui si accede per eseguire una data operazione, la memoria di controllo, ovvero la micro-ROM stessa, e un **Program Counter della micro-ROM (MPC)**. Le *istruzioni ISA* sono le istruzioni di interfaccia con il programmatore, a ciascuna di esse corrisponde una **micropcedura**, costituita da un blocco di **microistruzioni**, memorizzate nella micromemoria. Per eseguire un codice operativo eseguiamo una micropcedura, una sequenza di microistruzioni contenuta nella memoria, la difficoltà dunque consiste nel riuscire a *determinare, dato un codice operativo, l'indirizzo di memoria da cui partire per operare con le microistruzioni*. In definitiva, c'è un gap semantico tra l'istruzione e la parte operativa che la deve realizzare. Per ogni istruzione di alto livello allora determiniamo una serie di istruzioni di più basso livello che la macchina è in grado di eseguire.

Un altro problema è come **ottimizzare la memoria**. Dato che spesso istruzioni differenti condividono parte delle sequenze da eseguire, per non duplicare tali informazioni occorre organizzare la memoria in modo da individuare e scrivere un'unica volta le microistruzioni comuni. Una prima tecnica prevede di includere in ciascuna microistruzione un **campo Next-Address** che indichi la localizzazione della successiva microistruzione da eseguire. Ovviamente, nel caso di un salto, dunque nel caso di un'esecuzione non sequenziale, occorre anche capire in che modo modificare tale indirizzo così da ottenere il nuovo valore corretto. Aggiungere il campo Next-Address vuol dire aumentare la lunghezza di ciascuna Control Word, dunque aumentare ancora la dimensione della memoria. Per ridurne la lunghezza possiamo codificare la Control Word, individuando blocchi di bit mutuamente esclusivi. Tale operazione permette di avere una memoria più piccola ma comporta l'utilizzo di un hardware aggiuntivo, un decoder, per decodificare l'informazione. Si passa allora da un'**architettura orizzontale** (senza codifica, dunque con memorie più grandi) a un'**architettura verticale** (con un maggior grado di codifica e con dell'hardware aggiuntivo per risolverlo). In alternativa, possiamo utilizzare la tecnica del **bit OR-ing: le parti comuni di istruzioni differenti vengono codificate in modo che la distanza sia di un solo bit**. Supponiamo di star eseguendo la microistruzione memorizzata all'indirizzo 169, la successiva istruzione si trova memorizzata con tecnica di indirizzamento immediato alla locazione 170, all'indirizzo 171 invece con tecnica di indirizzamento indiretto viene memorizzata un'altra microistruzione della stessa istruzione. In questo modo, procediamo normalmente nella sequenza fino a 169, poi, se si verifica una condizione che specifica la modalità di indirizzamento indiretto, allora si utilizza una OR per cambiare il bit meno significativo dell'indirizzo ad 1. Fino 169 procediamo normalmente e in un caso continuiamo in 170, nell'altro caso saltiamo da 171 a un'altra sequenza di istruzioni. Notiamo che con questa tecnica il bit di salto non deriva da un registro di stato ma dalla codifica stessa dell'istruzione.

9.2 Esempi di processore

In questa sezione, analizzeremo esempi di processore mostrati a lezione, evidenziando contrasti e analogie, e cercando di capire in quali casi è conveniente adottare la logica cablata e in quali la logica microprogrammata.

9.2.1 Processore a 5 blocchi

Supponiamo che il processore lavori come una macchina a 5 stati, nel primo preleva l'istruzione (PI), nel secondo decodifica l'istruzione (ID), nel terzo esegue (EX), nel quarto memorizza i risultati in memoria (MEM), infine nell'ultimo scrive nei registri interni, dunque fa il write-back (WB), come mostra la figura 9.1.

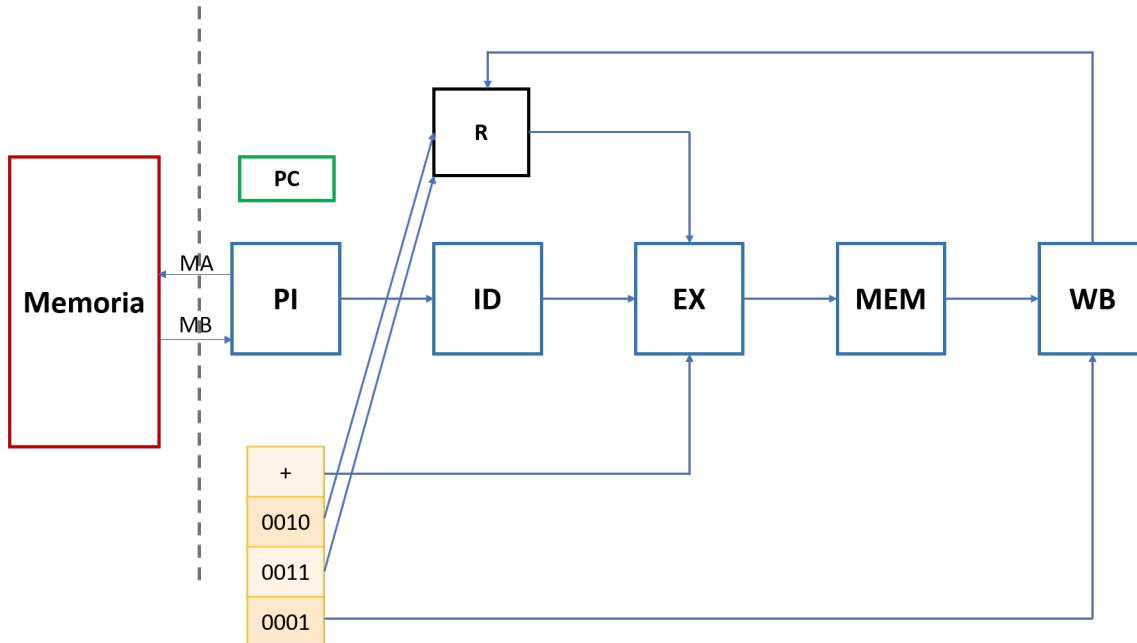


Figura 9.1: Schema del processore a 5 blocchi.

Supponiamo inoltre che al fronte di salita del clock venga prelevata la seguente istruzione:

$$R1=R2+R3$$

Il prelievo avviene al primo passo, poi viene decodificata (dunque si capisce il tipo di operazione e vengono prelevati gli operandi) e le informazioni ricavate passano allo stadio di esecuzione. Nel blocco MEM in questo caso non va fatta alcuna operazione, il risultato infine sarà scritto in R1 nel blocco di WB. Ogni stadio impiega un tempo T per produrre il risultato, **in totale in 5T, all'interno dello stesso colpo di clock, il sistema riesce a effettuare l'operazione.** Sebbene non sia stato fatto niente durante la fase di MEM, ne abbiamo comunque considerato il tempo perchè dimensioniamo il clock sul percorso più lungo. Supponiamo invece di voler realizzare, con lo stesso sistema, **l'istruzione che segue prelevata al fronte di salita del clock:**

$$R1=R2+R3+var$$

dove il valore *var* risiede in memoria. Non sono più sufficienti 5 quanti di tempo perchè bisognerà leggere prima *var* dalla memoria, occorre allora porre delle limitazioni. Per svolgere l'istruzione allora dovremo prima leggere *var* dalla memoria e poi effettuare la somma, in due operazioni differenti. In figura 9.2 è mostrata la codifica della prima istruzione vista. Il primo elemento indica l'operazione da effettuare, ovvero l'addizione, supponendo che sia codificata su 4 bit, mediante una rete di codifica capiremo quale operazione effettuare. I 4 bit successivi codificano il primo operando (R2), poi il secondo (R3) e infine l'informazione necessaria al WB per memorizzare il risultato (R1). Ciò rappresenta la **parola di controllo** che agisce sull'unità operativa in modo da effettuare l'operazione desiderata. Realizzare quest'architettura in **logica cablata** risulta semplice. L'unità

0110 (+)	0010 (R2)	0011 (R3)	0001 (R1)
----------	-----------	-----------	-----------

Figura 9.2: Struttura della parola di controllo.

di controllo essenzialmente deve solo prendere l'istruzione al colpo di clock, i valori della parola di controllo poi saranno passati ai vari blocchi che realizzeranno i propri compiti. La logica è cablata e la macchina è ancora combinatoria. Notiamo inoltre che è presente una decomposizione funzionale dei blocchi: il primo preleva, il secondo decodifica, il terzo esegue e così via. Possiamo allora ulteriormente separare i blocchi interponendo tra di essi dei registri, in tal modo otteniamo un'architettura pipelined, adesso le istruzioni elaborate sono 5, una in fase di prelievo, una in fase di decodifica, e così via.

Nota 9.1: Pipelines

Una delle architetture più importanti è l'architettura pipelined, in cui più unità di elaborazione lavorano insieme, per raggiungere un obiettivo. Lo schema è quello mostrato in figura 9.3, c'è una parte combinatoria e una di memorizzazione del risultato della parte combinatoria. Per garantire che la pipeline funzioni, il sistema lavora con un sistema di clock, che funge da sincronismo.

Un esempio di tale struttura è il processore, nell'architettura classica del modello di Von Neumann le fasi di fetch, decodifica ed esecuzione sono eseguite in successione, l'una dopo l'altra. Con l'architettura pipelined le diverse fasi, invece, sono svolte contemporaneamente dalle varie unità, poste in cascata e interconnesse da registri. Dunque, mentre un'unità esegue l'istruzione i , l'unità precedente procede con la decodifica dell'istruzione $i+1$, l'unità prima ancora preleva l'istruzione $i+2$, e così via. Non è attiva una sola macchina alla volta ma più macchine, che lavorano in parallelo. Il tempo di attraversamento è sempre invariato, perchè il numero di unità non cambia, ma migliora notevolmente la produttività. Se supponiamo che ogni blocco impieghi un tempo T per svolgere il proprio compito, un processore sequenziale impiega $3T$ per terminare le operazioni, anche nell'architettura pipelined il tempo è $3T$ ma la differenza è che ogni T otteniamo un risultato, a meno della prima iterazione. Per avere la stessa produttività allora possiamo lavorare a frequenze più basse, questo è un vantaggio. Per il corretto funzionamento i tempi di esecuzioni delle singole fasi devono essere il più possibile uguali, altrimenti tutte le unità devono lavorare alla velocità della macchina più lenta. Oltre a ciò, va sempre garantita anche l'alimentazione. In conclusione, la pipeline si comporta come una sorta di "catena di montaggio", invece di avere un singolo pezzo (*sistema a singola fase*) che esegue tutto il lavoro, quindi a una velocità elevata per raggiungere una determinata produttività, abbiamo più pezzi (*sistema a più fasi*) che lavorano contemporaneamente, quindi ciascuno a una velocità più bassa. Siamo passati allora da uno svolgimento temporale delle attività a uno svolgimento spaziale, fortemente sincrono. Rispetto a un sistema a singola fase probabilmente consuma di più, data la presenza di più macchine che lavorano sempre contemporaneamente, ma si preferisce comunque questa architettura perchè permette anche di facilitare la rete di controllo.

Se in quest'architettura volessimo aggiungere delle operazioni, come ad esempio la moltiplicazione, sarebbe facile, bisognerebbe modificare solo l'unità EX. Diventa invece più difficile gestire i salti, quando PI prende un'istruzione la successiva non è più in sequenza, il PC va allora aggiornato: il PC adesso non è utilizzato solo da PI ma anche da qualcosa che ne modifichi opportunamente il valore in caso di salto. In assenza di salti è semplice, siamo sicuri di dover prendere sempre le

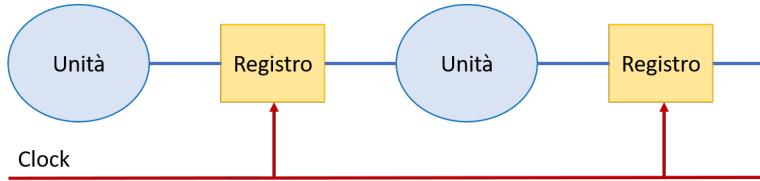


Figura 9.3: Architettura pipelined.

istruzioni in sequenza e quindi lo schema pipelined funziona.

9.2.2 Architettura a registri generali

In figura 9.4 troviamo lo schema di un'architettura a registri generali. La macchina presenta un insieme di registri, connessi tramite un bus. Quest'ultimo in realtà è un multiplexer/demultiplexer, in quanto uno dei registri può caricare il valore sul bus, quindi in questo caso il bus si comporta come un multiplexer, e uno dei registri può anche leggere il valore del bus, in tal caso il bus si comporta come un demultiplexer.

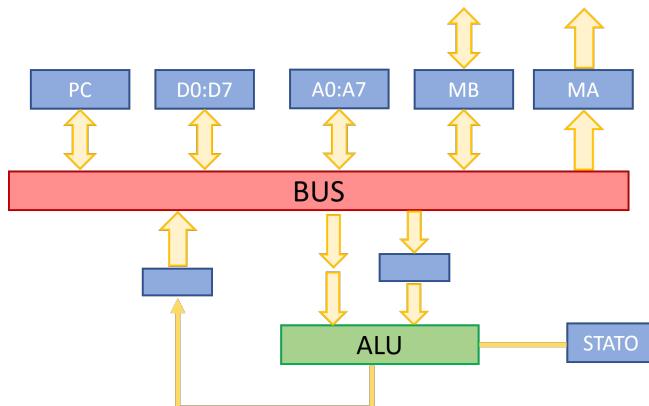


Figura 9.4: Architettura a registri generali.

Per mettere i registri in contatto è necessario allora un certo tempo, *sul fronte di salita del clock caricchiamo il multiplexer, sul fronte di discesa il demultiplexer consente di prendere il dato*. Dobbiamo allora gestire una frequenza tale da consentire di abilitare correttamente le operazioni, fornendo il tempo necessario alla parte combinatoria (mux/demux) per reagire. Il bus può essere realizzato tramite un solo componente se utilizziamo la **logica tristate**, uno stesso filo realizza sia il multiplexer sia il demultiplexer. *Utilizzando gli FPGA, tuttavia, non possiamo ricorrere a tale logica, siamo costretti a utilizzare porte AND/OR, dunque a realizzare multiplexer e demultiplexer.*



Figura 9.5: Logica tristate.

Anche in quest'architettura serve un PC che indica, passo dopo passo, l'istruzione da prelevare in memoria. Serviranno poi registri per contenere i dati (D0:D7, come nel Motorola 68000), questi

possono o meno servire per puntare alla memoria con modi di indirizzamento indiretto, in tal caso il modello è versatile e possiamo usare ogni dato per mettere un valore o un indirizzo. In caso contrario, avremo dei registri dedicati a contenere gli indirizzi (A0:A7). Serve inoltre un'ALU, che può prendere un valore dal bus ma in generale necessita di due operandi, per cui tipicamente si serve di un proprio registro di appoggio. Inoltre, prima di mettere il dato prodotto sul bus lo può passare a un registro. Infine, serve un *registro di stato* in cui memorizzare alcuni risultato ottenuti da operazioni. Per la comunicazione con la memoria servono i registri di **Memory Address** e **Memory Buffer**, in particolare il primo è solo di uscita per gli indirizzi, il secondo permette di leggere e scrivere dati (ingresso e uscita). Per copiare il contenuto del registro D0 in D1 in questo datapath, in linguaggio assembler possiamo scrivere:

```
MOVE.W D0,D1
```

Sebbene sia stato utilizzato il linguaggio assembler, l'istruzione è ancora di alto livello, per poterla eseguire si dovranno effettuare molte micro-operazioni, che il programmatore non conosce, coordinate dalla rete di controllo. In particolare, per prima cosa dobbiamo prendere il PC e copiarlo nel MA: sul bus va prima abilitato PC nel multiplexer e poi il MA sul demultiplexer. **In questo passaggio sarà necessario avere un protocollo che ci permetta di dialogare opportunamente con la memoria.** La seconda fase consiste nel ricevere il dato richiesto nel MB, dove non possiamo mantenerlo altrimenti non sarebbe più possibile leggere dalla memoria. La terza operazione allora è copiare il contenuto del MB nell'**IR (Instruction Register)**. Solo dopo queste 3 operazioni può iniziare l'istruzione vera e propria, nella fase 4 prendiamo D0 e lo copiamo su T (registro di appoggio), nella fase 5 infine copiamo il contenuto di T in D1, tramite il bus. Di seguito riassumiamo le operazioni eseguite:

1. PC→MA
2. MB
3. MB→IR
4. D0→T
5. T→D1

Notiamo che **in realtà la seconda non è una vera e propria operazioni ma è un problema di temporizzazione**: prima poniamo l'indirizzo nel MA e poi riceviamo il dato nell'MB. Per fare la prima operazione dobbiamo opportunamente comandare il bus, ovvero selezionare PC sul multiplexer e MA sul demultiplexer. Passiamo allora al bus la sequenza di controllo mostrata in figura 9.6. Tale sequenza deve durare un tempo tale da prendere prima il valore dal PC e poi metterlo sull'MA, l'intervallo del clock deve garantire che il valore sia messo e prelevato. **Poiché il clock del sistema è unico, dobbiamo essere sicuri che tra un clock ed un altro riusciamo a fare tutte le microoperazioni che dobbiamo fare, che impiegheranno più o meno tempo a seconda della loro complessità.** Se dobbiamo fare un'addizione, dobbiamo capire cosa succede nelle varie fasi del clock. Per poter fare delle operazioni dobbiamo poter comandare l'ALU, ci sono sequenze per comandare ALU, il PC e così via, che sono generate dall'unità di controllo, quella schematizzata in figura 9.4 è invece l'unità operativa.

Se decidiamo di lavorare in logica cablata anzichè microprogrammata, cambia solo l'unità di controllo. Se invece vogliamo aumentare il parallelismo tra i registri, cambia il datapath ma non l'unità di controllo. Se pensiamo alle possibili istruzioni, le fasi 1,2 e 3 sono pressocchè uguali, quello che cambia è la particolare esecuzione, dunque le fasi 4 e 5. In realtà, **la prima parte è uguale se presupponiamo che le istruzioni siano a lunghezza fissa**, se sono a lunghezza variabile invece potrebbe capitare che preso il primo pezzo ne serva un altro, le operazioni cambiano. Osserviamo che **nel modello a 5 blocchi le istruzioni devono avere lunghezza fissa** perché effettuato il prelievo si passa al blocco successivo, **se fossero a lunghezza variabile il passaggio da un blocco al successivo non impiegherebbe lo stesso tempo.** Se avessimo più di un bus, il

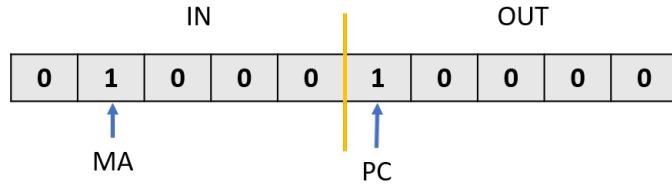


Figura 9.6: Esempio di controllo del bus.

sistema risulterebbe sicuramente più veloce ed efficiente ma la sequenza di controllo risulterebbe più complicata, servirebbero più fili e più contatti.

Infine, abbiamo studiato il processore MIC-1, la cui analisi è proposta nella seguente sezione.

9.3 Il processore MIC-1

N.B.: Per lo studio del seguente argomento, si fa riferimento al libro originale di A. S. Tanenbaum [2] e alla dispensa [3], presente in appendice F.

Il processore MIC-1 nasce come interprete hardware del bytecode, generato da un programma Java. Il nome, infatti, è IJVM dove la I indica che la macchina tratta solo numeri interi. Il nostro intento non è utilizzare tale processore nel suo ruolo nativo ma per analizzarne la **microarchitettura**. Il livello di microarchitettura ha il compito di implementare le istruzioni ISA (“*Instruction Set Architecture*”). In particolare, il processore MIC-1 presenta un’architettura a stack e una logica **micropogrammata**. Nelle seguenti sezioni approfondiremo nel dettaglio tali aspetti.

9.3.1 Datapath

La parte operativa del processore di Tanenbaum è mostrata in figura 9.7. Sono presenti 10 registri e due bus tramite cui essi possono scambiare dati. Dato che l'intento della macchina non è il solo trasferimento dei dati, è presente anche un'ALU, che consente di fare operazioni logico/aritmetiche di base, mostrate in tabella 9.8. Al di sotto dell'ALU troviamo anche uno shift register, utile in alcune operazioni matematiche, quali moltiplicazioni e divisioni binarie.

I registri necessari alla comunicazione con la memoria sono il *MAR* (*Memory Address Register*), l'*MDR* (*Memory Data Register*), il *PC* (*Program Counter*) e l'*MBR* (*Memory Byte Register*), notiamo che si tratta di 2 coppie di registri, una coppia è dedicata alla lettura delle istruzioni, l'altra coppia invece alla lettura e scrittura dei dati. Risulta allora evidente che **area dati e area istruzioni, in questo modello, sono separate**. In particolare, MAR e MDR sono registri destinati ai dati, ciò si evince dal fatto che il registro MDR ha una freccia verso la memoria che va in entrambi i sensi (in quanto i dati possono essere letti e scritti); PC e MBR sono utilizzati per le istruzioni, infatti l'MBR ha solo una freccia entrante, in quanto le istruzioni si possono solo leggere. La memoria del processore è una memoria costituita da parole di 8 bit, univocamente identificate da indirizzi di 32 bit, dunque possiamo indirizzare, all'interno di tale memoria, 2^{32} parole. In realtà l'indirizzamento della memoria tramite il PC e tramite il MAR è leggermente differente, ponendo l'indirizzo di un byte nel PC, al successivo colpo di clock troveremo disponibile sull'MBR il byte indicato; invece, ponendo l'indirizzo nel MAR, al successivo colpo di clock troveremo nell'MDR i 4 byte corrispondenti alla word 2 (dunque dal byte 8 al byte 11), in quanto l'MDR è di 32 bit. Per ottenere tale risultato il MAR effettua uno *shift equivalente*, sposta tutti i bit avanti, inserendo nei due bit meno significativi “00”. Tale *shift equivalente* a una moltiplicazione per 4 e ci permette di passare dall'indirizzo del byte 2 all'indirizzo della word 2. Dunque, per prelevare i dati, rappresentati su 32 bit, è sufficiente un accesso in memoria, invece per prelevare le istruzioni,

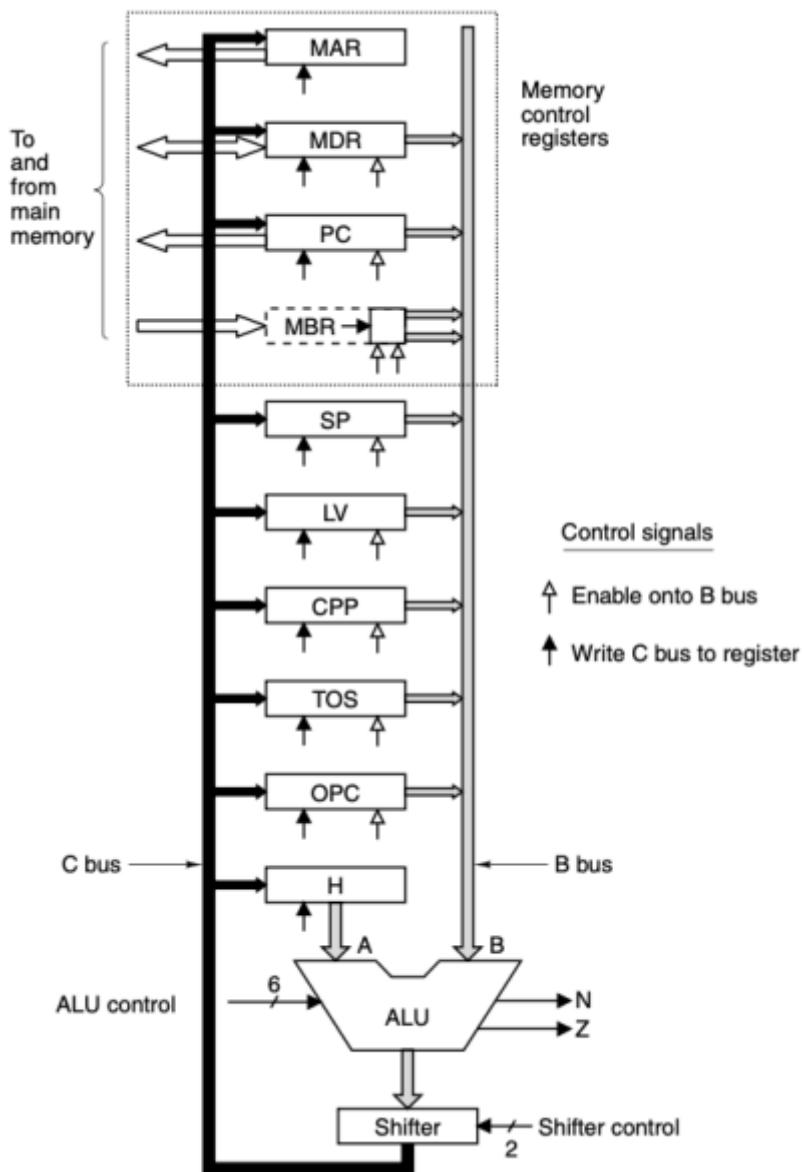


Figura 9.7: Unità operativa del processore MIC-1.

che possono avere una lunghezza variabile in multipli del byte, può essere necessario più di un accesso in memoria. L'*escamotage* utilizzato per il registro MAR è legato al fatto che nelle reali implementazioni della macchina è presente un'unica memoria, orientata al byte. In definitiva, dunque, la coppia MAR/MDR è utilizzata per leggere e scrivere parole di dati del livello ISA, mentre la combinazione PC/MBR è utilizzata per leggere il programma eseguibile del livello ISA.

Nota 9.2: Protocollo con la memoria

Per quanto concerne l'interfacciamento del processore con la memoria, Tanenbaum immagina di comunicare con una sorta di cache perfetta, che a ogni colpo di clock è in grado di fornire il dato richiesto. Il protocollo con la memoria in fase di lettura è semplice, al colpo di clock si pone l'indirizzo sul MAR (o sul PC nel caso di istruzioni) e al colpo di clock successivo esso sarà presente sull'MDR (o sull'MBR nel caso di istruzioni), senza necessità di alzare un segnale di lettura. Per effettuare la scrittura, invece, al colpo di clock dev'essere alto il segnale di *Write Enable (WE)*, dev'essere disponibile l'indirizzo sul MAR e il dato da scrivere nell'MDR. La figura 9.9 mostra tale funzionamento.

F₀	F₁	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	\bar{A}
1	0	1	1	0	0	\bar{B}
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B - A
1	1	0	1	1	0	B - 1
1	1	1	0	1	1	-A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	-1

Figura 9.8: Segnali di controllo dell'ALU e relative funzioni.

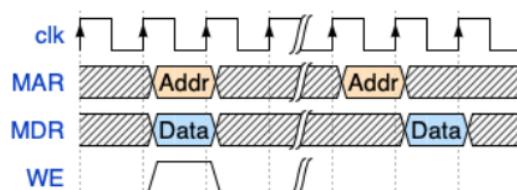


Figura 9.9: Protocollo con la memoria.

I registri rimanenti sono il *registro H (registro di Holding)*, che permette di contenere uno degli operandi dell'ALU, i registri CPP, LV, OPC e i registri necessari alla gestione dello stack, ovvero il registro *TOS (Top of Stack)* e *SP (Stack Pointer)* che ci permettono di accedere alla testa dello stack, rispettivamente all'indirizzo e al valore. Il *registro LV* serve per puntare alle variabili di appoggio che utilizza la macchina Java, *CPP* è un puntatore a valori costanti, anch'esso utilizzato per semplificare la gestione della macchina virtuale, e *OPC* permette di appoggiare qualcosa di utile nella micro-istruzione.

Per quanto riguarda la **comunicazione tra i registri**, il bus B permette di caricare su di esso il valore di un registro, mentre il bus C permette di scrivere su un registro il contenuto presente su di esso. Dunque, **i due bus rappresentano rispettivamente un multiplexer e un demultiplexer**. Se vogliamo scrivere il contenuto del registro X_i sul registro X_j dobbiamo selezionare con il multiplexer X_i e sul demultiplexer X_j . I due bus però, come notiamo dalla figura 9.7, sono collegati tramite l'ALU, ci aspettiamo quindi che una delle possibili funzioni dell'ALU sia far passare l'operando B inalterato, ed è quello che accade in corrispondenza della seconda riga della tabella in figura 9.8. Tale scelta è motivata dal fatto che quando dimensioniamo il clock dobbiamo considerare il percorso più lungo, ovvero quello in cui un dato per passare da B a C entra nell'ALU che effettua un'operazione e richiede anche uno shift. Lo svolgimento di tale operazione necessita allora di 4 tempi (mostrati in figura 9.10): il tempo Δw è necessario per prendere l'istruzione dalla memoria di controllo, contenente i bit che pilotano gli elementi del datapath; il tempo Δx per selezionare il registro e portarne il contenuto sul bus B, Δy per far lavorare l'ALU ed eventualmente lo shift register (notiamo infatti che è il tempo più lungo), e infine Δz affinché i valori diventino stabili e dunque prelevabili dal bus C. Tipicamente, viene lasciato un piccolo lasso di tempo aggiuntivo per garantire che tutto avvenga correttamente, solo al termine di ciò i segnali risulteranno effettivamente stabili sul bus C. Dunque, dato che non si può ipotizzare di avere un clock a lunghezza variabile, e per altro predicibile in quanto dovrebbe avere una determinata lunghezza a seconda dell'istruzione considerata, si preferisce semplificare il circuito, imponendo tale passaggio per l'ALU.

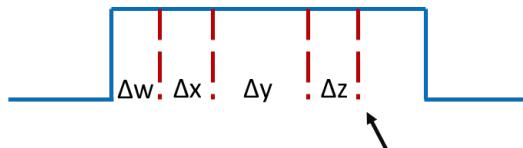


Figura 9.10: Diagramma temporale di un ciclo del datapath.

Osserviamo che **questo sistema è sincrono**, si muove su due colpi di clock senza dover aspettare che l'ALU comunichi di aver terminato le proprie operazioni. Inoltre, è un **modello transfer register**, in cui i dati viaggiano tra i registri tramite due bus e vengono modificati unicamente dall'ALU.

Per controllare il datapath del MIC-1 occorrono **29 segnali** (figura 9.11):

- 9 segnali per effettuare la selezione del bus C;
- 9 segnali per effettuare la selezione del bus B;
- 8 segnali per la gestione dell'ALU (in particolare 6 per l'ALU e 2 per lo shift register);
- 2 segnale di read/write per specificare se si vuole scrivere o leggere dalla memoria tramite la coppia MAR/MDR;
- 1 segnale per indicare il memory fetch del valore presente in PC/MBR.

I valori di tali segnali determinano le operazioni da eseguire durante un ciclo del processore. Notiamo che i registri presenti nel disegno sono 10, eppure servono 9 segnali per controllare i bus B e C. Ciò è dovuto al fatto che non tutte le combinazioni sono effettivamente significative.

9.3.2 Unità di controllo

L'unità di controllo del processore MIC-1 è realizzata in logica microprogrammata. È quindi presente un generatore degli indirizzi di partenza, ovvero l'indirizzo della micro-ROM a cui si accede

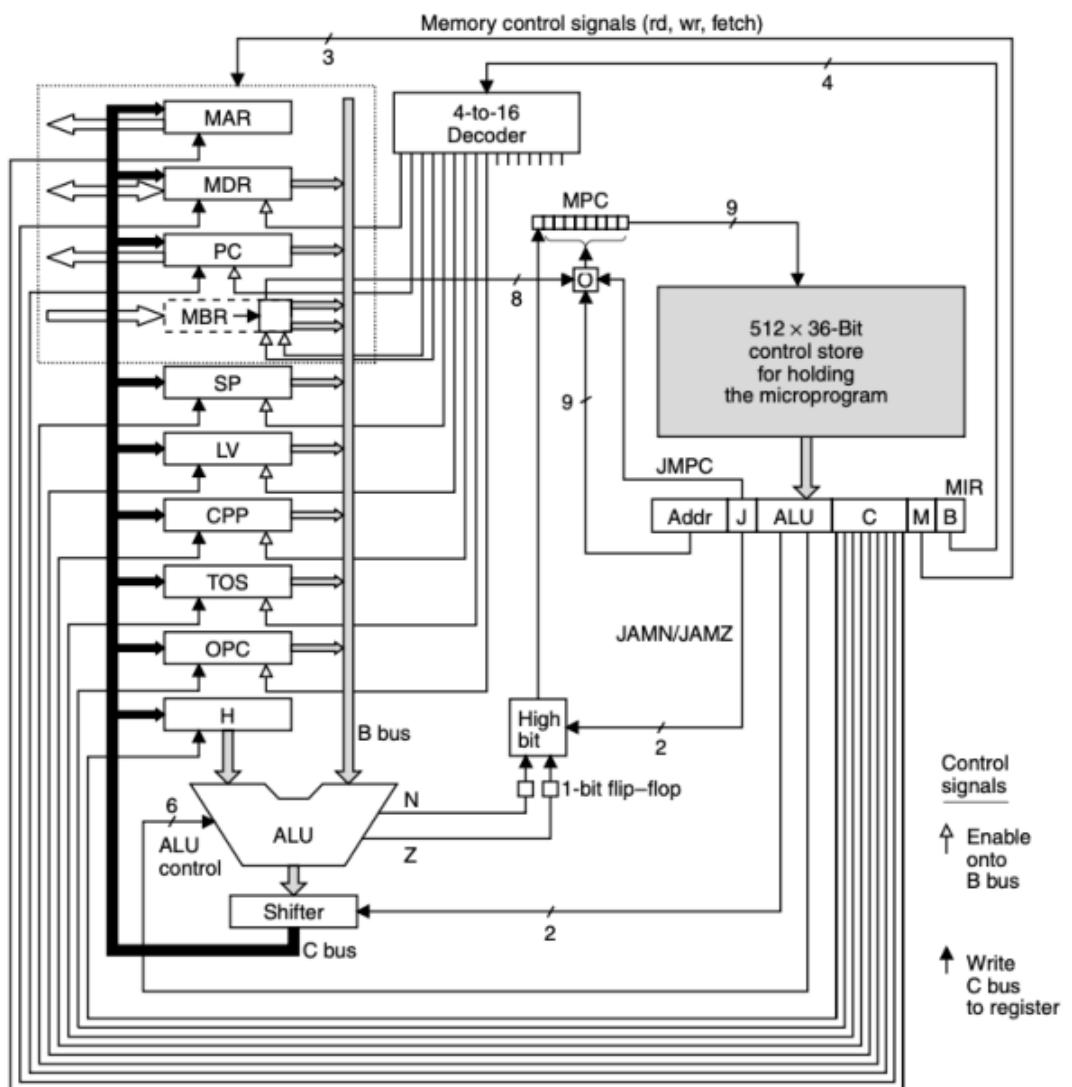


Figura 9.11: Schema completo del MIC-1.

per eseguire una data operazione, la memoria di controllo, ovvero la micro-ROM stessa, e un **Program Counter della micro-ROM (MPC)**. Le *istruzioni ISA* sono le istruzioni di interfaccia con il programmatore, a ciascuna di esse corrisponde una *microprocedura*, costituita da un blocco di *microistruzioni*, memorizzate nella micromemoria. Per eseguire un codice operativo eseguiamo una microprocedura, una sequenza di microistruzioni contenuta nella memoria, la difficoltà dunque consiste nel riuscire a determinare, dato un codice operativo, l'indirizzo di memoria da cui partire per operare con le microistruzioni.

Un altro problema è come **ottimizzare la memoria**. Dato che spesso istruzioni differenti condividono parte delle sequenze da eseguire, per non duplicare tali informazioni occorre organizzare la memoria in modo da individuare e scrivere un'unica volta le microistruzioni comuni. La tecnica adottata nel processore MIC-1 è di includere in ciascuna microistruzione un **campo Next-Address** che indichi la localizzazione della successiva microistruzione da eseguire. Ovviamente, nel caso di un salto, dunque nel caso di un'esecuzione non sequenziale, occorre anche capire in che modo modificare tale indirizzo così da ottenere il nuovo valore corretto. Aggiungere il campo Next-Address

vuol dire aumentare la lunghezza di ciascuna Control Word, dunque aumentare ancora la dimensione della memoria. Per ridurne la lunghezza possiamo codificare la Control Word, individuando blocchi di bit mutuamente esclusivi. Tale operazione permette di avere una memoria più piccola ma comporta l'utilizzo di un hardware aggiuntivo, un decoder, per decodificare l'informazione. Si passa allora da un'**architettura orizzontale** (senza codifica, dunque con memorie più grandi) a un'**architettura verticale** (con un maggior grado di codifica e con dell'hardware aggiuntivo per risolverlo).

L'unità di controllo deve generare la sequenza dei segnali di controllo necessari a pilotare l'unità operativa, ad ogni ciclo di clock. Nel caso del processore MIC-1 sono necessari 29 segnali, essi saranno prodotti dall'unità di controllo. Tuttavia, questi bit servono a determinare il percorso dei dati per un singolo ciclo di clock, viene dunque aggiunta una parte necessaria a determinare cosa effettuare al ciclo successivo: i campi *Addr* e *JAM*. Il campo *Addr* (a 9 bit) è necessario a individuare la microistruzione da eseguire al successivo ciclo di clock, il campo *JAM* (su 3 bit) serve a gestire le operazioni di salto. La struttura della Control Word è mostrata in figura 9.12.

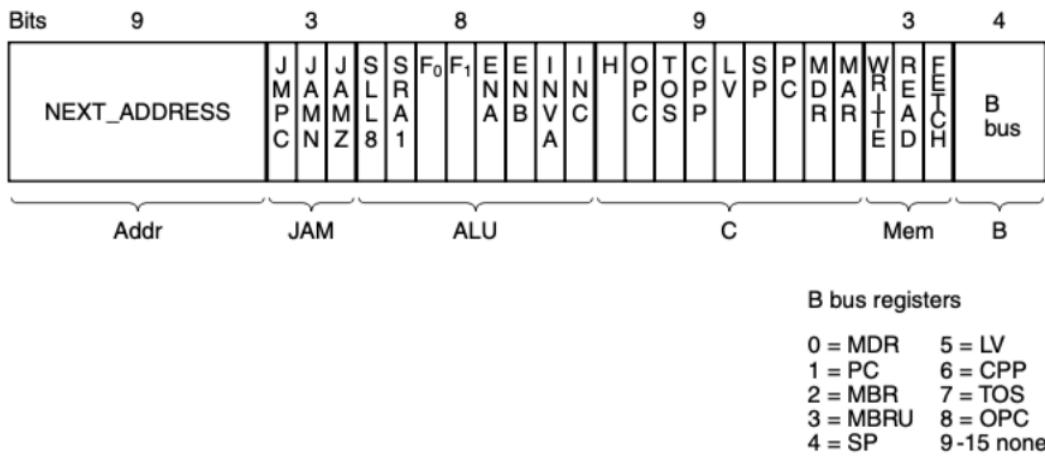


Figura 9.12: Struttura della Control Word.

Notiamo che la Control Word non è di 41 bit, come ci aspetteremmo, ma di 36 bit, ciò è dovuto al fatto che il segnale di controllo relativo al bus B viene codificato, dunque espresso su 4 bit anzichè 9. Tale codifica implica l'aggiunta di un decoder necessario a far arrivare l'informazione su 9 bit al bus B, col vantaggio però di non dover far viaggiare 5 fili aggiuntivi in tutta la memoria ma solo a partire da un certo punto in poi. Se avessimo codificato C invece di B avremmo avuto il vantaggio di ridurre i tempi necessari, infatti mentre il decodificatore davanti al bus B ne aumenta il tempo di risposta, davanti al bus C ciò non accade in quanto è sempre l'ultimo ad attivarsi (dopo il bus B, l'ALU e lo shift register), dunque durante tale tempo di attesa possiamo decodificare senza problemi il valore da passare al bus C. Tuttavia, il motivo per cui si sceglie di codificare l'informazione di B e non di C è che il primo bus rappresenta un multiplexer, i cui segnali di selezione devono essere mutuamente esclusivi, dunque sono codificabili; il secondo invece rappresenta un demultiplexer, i cui segnali di selezione non sono necessariamente mutuamente esclusivi. Codificare il controllo di C richiederebbe allora di individuare gruppi di bit mutuamente esclusivi, codificarli e poi decodificarli con un hardware maggiormente complesso. Con la sola codifica del bus B il modello di Tanenbaum risulta essere abbastanza orizzontale, anche se non totalmente.

Il campo Next-Address indica l'indirizzo della prossima micro-operazione. In particolare, l'indirizzo della prima micro-operazione da eseguire per un determinato codice operativo è indicato dal codice operativo stesso, il quale è codificato con l'indirizzo della parola (microistruzione) della

micro-ROM da cui partire. Le successive microistruzioni saranno calcolate a partire da questa. Come ogni memoria (spesso indicata come **memoria di controllo** per non confonderla con la memoria principale), anche questa prevede la presenza di un registro MA (Memory Address) e MB (Memory Buffer) per il prelievo delle istruzioni. Analogamente a come la memoria principale memorizza le istruzioni ISA, la memoria di controllo memorizza invece le microistruzioni, nel nostro caso è una memoria da 512 word da 36 bit. Inoltre, in questo caso l'indirizzo della successiva microistruzione è contenuto all'interno della microistruzione stessa, a cui si passerà al termine della microistruzione eseguita, a meno di salti. Come anticipato, anche la memoria di controllo necessita di un PC, chiamato **MPC (Micro Program Counter)**, invece nel registro **MIR (Micro Instruction Register)** è memorizzata la microistruzione in esecuzione, i cui bit determinano i segnali di controllo che guidano il datapath, come mostra la figura 9.11. Il tempo δW definito nella tempificazione del clock (figura 9.10) è proprio il tempo necessario affinché la parola puntata da MPC venga trasferita nel MIR. Se le operazioni che dobbiamo effettuare sono in sequenza, il campo Next-Address indicherà la successiva istruzione da eseguire. Dunque, tale campo fornisce lo stato prossimo, grazie al quale non è necessaria la presenza di un contatore, per incrementare di volta in volta il PC, perché per prendere l'istruzione successiva sarà sufficiente utilizzare il valore contenuto nel Next-Address. Quando invece è necessario effettuare un salto, il valore del Next-Address viene opportunamente modificato, prima di essere inserito nel PC. Per fare ciò si utilizzano i bit del campo JAM e i bit N e Z provenienti dall'ALU (e salvati in due flip flop per non avere problemi di instabilità). A seconda di tali valori, nel registro MPC copieremo il Next_Address, diversamente alterato:

- Se il campo JAM vale 000, allora il Next_Address viene copiato senza modifiche sull'MPC;
- Se JAMN è alto, si calcola l'OR tra il bit più significativo e il flag N, se ne pone il risultato nel bit più significativo di MPC;
- Se JAMZ è alto, si calcola l'OR con il flag Z e si pone il risultato nel bit più significativo di MPC;
- Se JAMN e JAMZ sono entrambi alti, si calcola l'OR rispetto a entrambi i flag;
- Se è alto il bit JMPC si effettua l'OR tra gli 8 bit di MBR e gli 8 bit meno significativi di next_address e il risultato viene posto in MPC.

In definitiva, il bit alto del next address viene modificato dal blocco *high bit* in figura 9.11, secondo la seguente funzione booleana:

$$F = (JAMZ \text{ and } Z) \text{ or } (JAMN \text{ and } N) \text{ or } \text{NEXT_ADDRES}(8)$$

Tramite queste tecniche, è possibile realizzare eventuali diramazioni, necessarie in caso di salti.

Capitolo 10

La memoria

Vogliamo realizzare un componente di tipo memoria. Ogni elemento di memoria, per sua natura, ha un **chip select (CS)** che dice se funziona o meno, serve poi un segnale di **$!R/W$** , read o write (il segnale di write e tipicamente read è il negato), un **dato**, che si può scrivere o leggere, con un certo numero di bit. Per quanto riguarda il dato, serve un filo entrante (**Din**) e uno uscente (**Dout**), se la logica invece fosse tristate ne basterebbe uno solo. In realtà, parliamo in generale di un filo ma *se il dato ha più di un bit, avremo più di un filo*. Essendo una memoria, serve una **linea di indirizzo**, se indichiamo con N il numero di registri, le linee di indirizzo saranno $\log_2(N)$. Consideriamo come sistema di base una macchina realizzata in questo modo, con 4 registri, dunque 2 linee di indirizzo, un bit di ingresso e un bit di uscita, come mostra la figura 10.1.

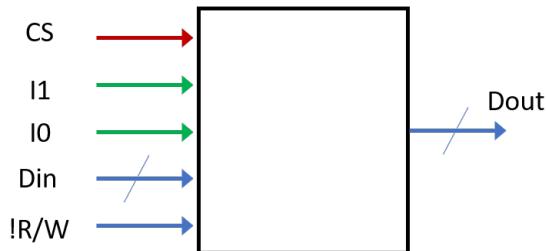


Figura 10.1: Schema di un componente di memoria.

Supponiamo di voler costruire una memoria di dimensioni maggiori, utilizzando questo oggetto base, disponibile ad esempio in libreria. Vogliamo realizzare una macchina che ha lo stesso comportamento ma su un numero di fili maggiori, ad esempio 16 celle e parallelismo 1 bit. L'idea è semplice se ragioniamo in modo strutturato, dobbiamo usare la macchina vista per farne una più grande. In ingresso alla “scatola” grande ci sarà sempre un chip select (CS), un dato che esce e uno che entra, il segnale $!R/W$, perché deve sempre scrivere/leggere, ma le linee di indirizzo non sono più 2, saranno 4 ($\log_2(4)$). In questo schema servono 4 oggetti piccoli, messi all'interno, ci saranno delle celle di memoria che avranno comunque in ingresso due bit di indirizzo, probabilmente i meno significativi, perché il primo blocco memorizzerà i dati 0000-0011, il secondo 0100-0111, il terzo 1000-1011 e il quarto 1100-1111. Questa attività è analoga a quanto visto con i multiplexer (paragrafo 2.3.3), la memoria si divide in modo coerente tenendo conto dei pesi più e meno significativi. **Mandiamo I1 e I0 (bit meno significativi dell'indirizzo) a tutti i componenti, che selezionano la cella all'interno del blocco interno, invece, i due più significativi (I2 e I3) selezionano il blocco**, vanno in una macchina che è un decoder (da 2 fili ne produce 4) e in uscita otteniamo i chip select dei blocchi interni (cs0, cs1, cs2 e cs3), in modo da abilitare il blocco opportuno (figura 10.2).

È una macchina modulare, man mano riusciamo a costruire dei sistemi sempre maggiori in

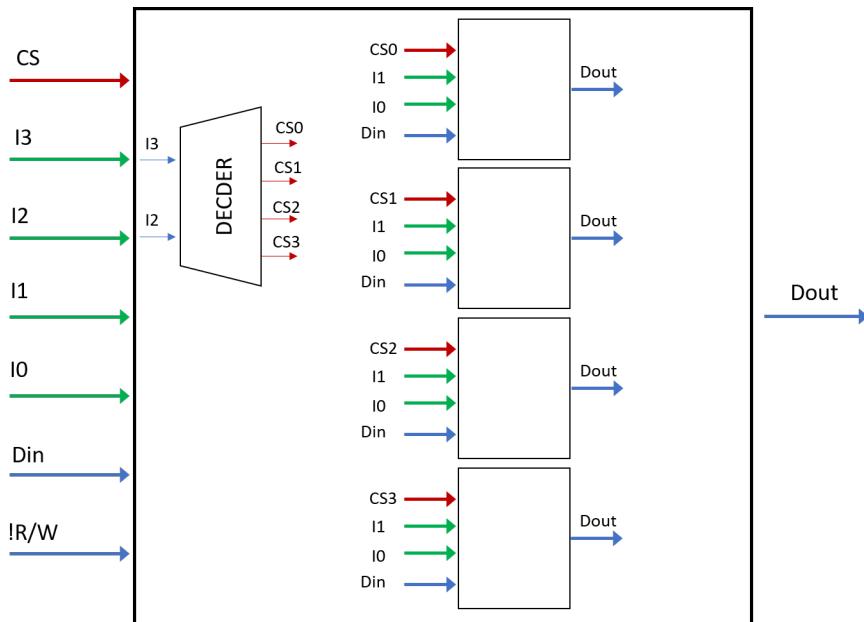


Figura 10.2: Schema strutturale della memoria a 16 celle.

quanto questa memoria si basa sul concetto di indirizzo: *la decomposizione parte dall'indirizzo*. Il problema sono Din e Dout della macchina grande, *da dove entrano e da dove escono?* Dipende da com'è fatto il modulo di memoria, **se è tristate**, per cui se non lo abilitiamo non funziona, allora **possiamo mandare Din a tutti** (in caso d'iscrittura) e **prendere le uscite in parallelo da tutti** (in fase di lettura), l'unico blocchetto interno che funziona è quello che da il dato, gli altri sono "spenti". **Se invece la logica non è tristate**, Din può andare nel blocco 0,1,2 o 3, può andare in 4 possibili registri di uscita, analogamente il dato che esce può essere preso da 4 possibili registri di uscita: **servono un demultiplexer e un multiplexer e le linee di selezione sono le stesse che selezionano il blocco** (i chip select ottenuti dal decoder). Nel demultiplexer mettiamo W nel multiplexer R, in questo modo il primo consente di scrivere quando arriva il valore e l'altro consente di leggere (figura 10.3).

Il blocco piccolo, contenuto all'interno, interno funziona con lo stesso criterio di quello grande: abbiamo 4 registri (R0, R1, R2 e R3), dobbiamo selezionare su quale dei registri si scrive/legge il dato. **Con le due linee di indirizzo in ingresso abilitiamo il registro di interesse tramite un decoder**, da cui escono i 4 valori. **Anche qui è necessaria la coppia multiplexer/demultiplexer**, si deve poter leggere e scrivere, nel demultiplexer allora entra un valore e ne escono 4, nel multiplexer invece ne entrano 4 e ne esce 1. Il segnale che arriva al multiplexer è di read, al demultiplexer è di write. Manca solo il CS che in effetti mettiamo nel decoder (figura 10.4), così senza CS il sistema non funziona, non viene selezionata nessuna cella, altrimenti si seleziona una cella e tutto funziona. Questo è lo schema interno, molto simile allo schema del blocco grande, fermo restando che **gli indirizzi hanno senso diverso**.

Nel passaggio dal blocco piccolo al sistema grande abbiamo aumentato il numero di registri. Se invece volessimo aumentare il **parallelismo**, ad esempio da 1 bit a 2 bit, se **ogni blocco memorizza un bit, per memorizzarne due ne serve una coppia**, ciascuno dei blocchetti visti in figura 10.2, sarà affiancato da un blocco identico (come mostra il blocco tratteggiato in figura 10.5). Ogni coppia lavora con lo stesso CS, così quando selezioniamo il primo bit a sinistra, selezioniamo anche il primo bit a destra. *Il decodificatore può essere uno*, perché seleziona lo stesso elemento della cella, invece **i multiplexer/demultiplexer devono essere 2, o N quanti sono i bit** che devono entrare/uscire, perché ogni bit entra ed esce da un solo blocco. Diventa

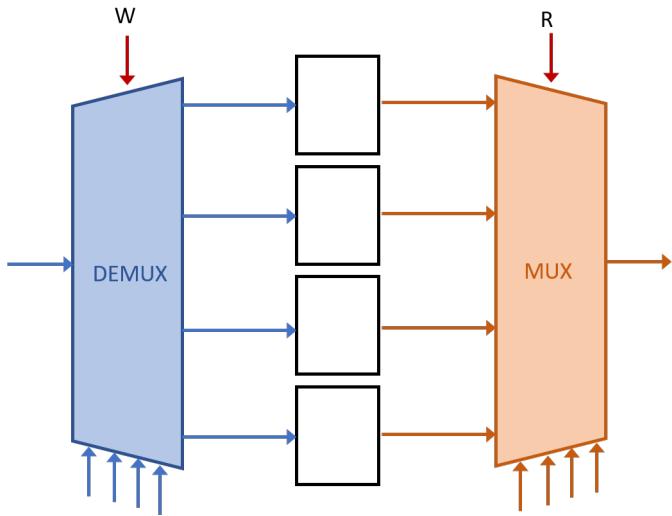


Figura 10.3: Demultiplexer e multiplexer utilizzati in lettura e scrittura.

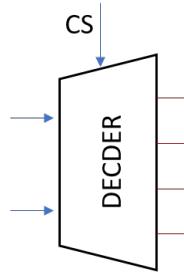


Figura 10.4: Decoder utilizzato nel blocco interno.

allora un sistema in cui, partendo da un modulo di 4 celle e con parallelismo di un bit, utilizzando 8 componenti abbiamo realizzato un modulo di 16 celle con parallelismo 2 bit.

Se invece realizzassimo direttamente il blocco piccolo in modo da fornire due bit invece di 1, all'interno del chip ci saranno due banchi di registri per cui da un chip usciranno direttamente due fili. Stiamo dicendo che la modifica non viene fatta all'interno del blocco grande ma in quello piccolo. I componenti ora li mettiamo all'interno dello stesso integrato. L'architettura con cui usiamo più decodificatori è quella vista prima, modulare, perché se mettiamo un altro blocco accanto, ogni blocco ha un suo decodificatore interno e in più c'è quello esterno. Usando invece il chip che già fornisce due bit serve un solo decodificatore, quello che seleziona i banchi. Multiplexer e demultiplexer sono sempre due, perché il problema che c'è è lo stesso di prima, ma possiamo salvare l'utilità di avere un decodificatore in meno, perdendo un grado di flessibilità.

Le memorie sono il trionfo di quanto visto con il multiplexer, la selezione degli indirizzi per fare cose più grandi da cose più piccole. Le memorie che ragionano con questo criterio **necessitano di un protocollo**, diamo prima il dato, poi l'indirizzo, poi il CS. Il CS lo diamo alla fine perché se non si assestano i segnali quando diamo il CS i segnali non sono ancora stabili.

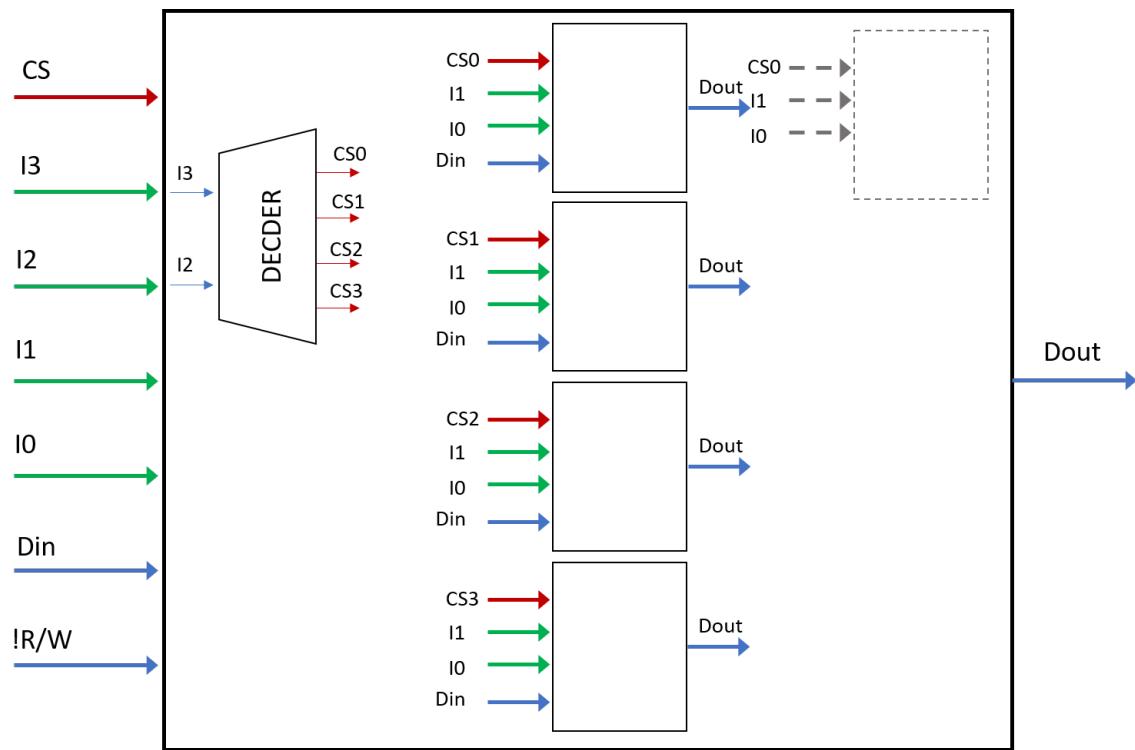


Figura 10.5: Schema con parallelismo a 2 bit.

10.1 Memorie associative

Le memorie fatte col modello appena illustrato sono **memorie statiche**: *non perdono il dato, se alimentate*. Queste sono in combutta con altri tipi di memoria, le **memorie dinamiche**, che invece *perdono il dato*. Per non perdere il dato in un sistema di memoria dinamica, con una certa frequenza dobbiamo leggere e scrivere il dato dalla memoria, serve una memoria più complicata. In certi momenti la memoria non funziona perché dobbiamo fare un *refresh*, ovvero delle operazioni di copia. Le memorie si fanno anche dinamiche perché **quelle statiche sono più ingombranti**. Se riuscissimo a fare delle memorie statiche grandi quanto vogliamo, con gli indirizzi, non ci sarebbero problemi, sarebbero veloci, non perderebbero il dato. Tuttavia, non possono essere troppo grandi, altrimenti l'ingombro è maggiore. *La cache è una memoria statica*, infatti dev'essere di piccole dimensioni. Vogliamo vedere le **memorie associative**, noi siamo abituati a vedere una memoria che da un indirizzo e il corrispondente dato è sicuramente presente. Vediamo invece delle memorie basate sul concetto di chiave, *se la chiave è in memoria allora il dato c'è, se la chiave non è in memoria allora il dato non c'è*. È un problema di memorie che sono necessariamente piccole, non possiamo mettere tutto quello che vogliamo ma chi cerca qualcosa ci da un indirizzo. È ovvio che, se abbiamo una memoria piccola (A), può darsi che non abbiamo la cosa nella memoria piccola mentre nella memoria grande (B) sì. Facciamo una copia dei dati della memoria B, che magari sono 16, ne portiamo in A 4, un numero più limitato e quando arriva l'indirizzo dobbiamo capire se c'è, e nel caso lo diamo, o non c'è, e nel caso bisogna prenderlo nella memoria grande.

Consideriamo il sistema con tutte e 16 le celle di memoria, realizzato come visto in figura 10.2, dunque il dato sarà sicuramente all'interno di essa; dall'altro lato invece abbiamo un sistema con solo 4 celle (come mostra la figura 10.6), non è detto abbia tutti i dati di quello precedente ma è sicuramente più veloce. Gli diamo 4 fili che **non rappresentano un indirizzo perché non è**

detto che quella cella sia al suo interno, rappresentano la *chiave*. Se ad esempio diamo in

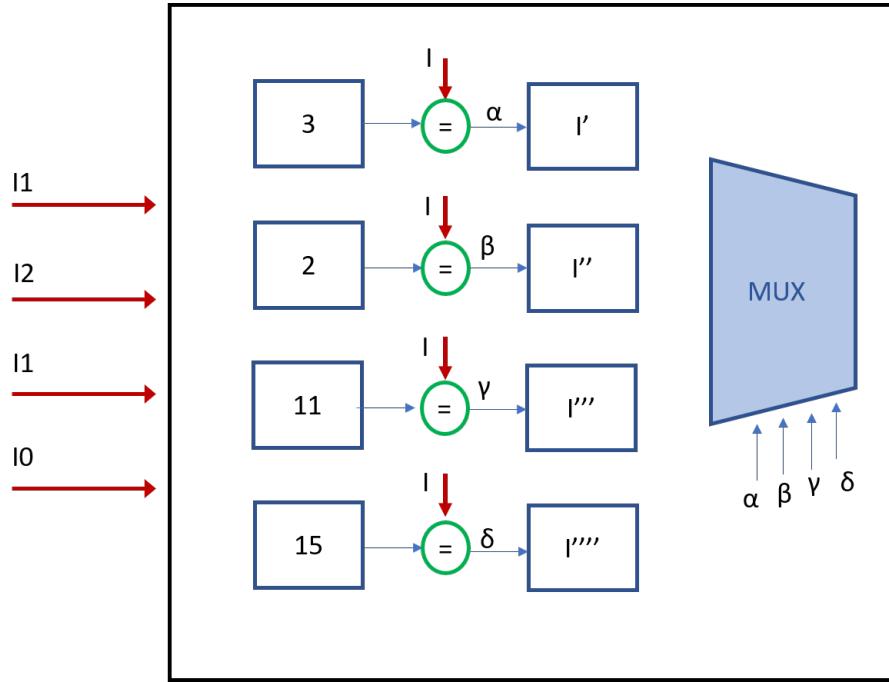


Figura 10.6: Schema di una memoria associativa.

ingresso 1111, nel primo blocco sicuramente la cella c'è, invece nel nuovo blocco non è detto che ci sia il registro con l'informazione 15, ma se c'è la prendiamo. *C'è anche un problema di coerenza tra i due blocchi* ma per il momento non ci interessa. Vogliamo capire come funziona una memoria che non lavora con gli indirizzi ma con le chiavi. Concettualmente ciò che deve fare è “*se (if) hai il dato allora (then) lo dai, altrimenti (else) non lo dai*”. In questa memoria sono memorizzate 4 chiavi e 4 informazioni, **ogni informazione è associata a una chiave**, invece nel primo blocco non c'era la chiave perché con l'indirizzo prendevamo direttamente l'informazione. Le chiavi memorizzate sono ad esempio 3, 2, 11 e 15, l'ordine non è importante, abbiamo solo la cella 3, 2, 11 e 15, con le relative informazioni. Se andiamo in 3 con il primo blocco prendiamo I' , se ci andiamo con questo nuovo blocco lo prendiamo prima. **La macchina su cui si basa** il modello in figura 10.6 è **il comparatore**: mettiamo l'indirizzo I in ingresso e questo va in un comparatore, o meglio in 4 comparatori, ognuno compara rispetto alle chiavi memorizzate. I risultati dei comparatore sono 4, che chiamiamo α, β, γ e δ , **vanno in ingresso anche al multiplexer**, perché se c'è 3 dobbiamo prendere I' , se c'è 2 I'' e così via. Arriva l'informazione, il comparatore seleziona quale dei 4 c'è, se mettiamo 15, l'unica uscita alta è quella dell'ultimo comparatore, si abilita delta ed esce I'''' , che abbiamo, se mettiamo 11 di alza γ , esce I''' . L'aspetto negativo è che se a un certo punto mettiamo 14, non c'è in nessuna chiave, il comparatore da tutti 0, il multiplexer non seleziona nessun valore. Ciò significa che *il sistema non ha al suo interno un valore e lo deve prendere dalla memoria grande*, dal primo blocco, che invece sicuramente avrà il valore. **Per capire se il valore c'è o meno facciamo una OR tra α, β, γ e δ** , se sono tutti 0 vuol dire che non c'è il dato: abbiamo avuto un ***cache miss*** (figura 10.7).

Questo sistema funziona bene perché non funziona con la logica dell'indirizzamento ma del comparatore. La prima problematica è che **il comparatore è difficile da realizzare**, noi dobbiamo mettere un comparatore per ogni chiave per cui è complicato. **Dal punto di vista dell'occupazione, ogni informazione memorizza un indirizzo, per un solo bit di dato servono 4 bit di indirizzo, per cui occupa molto**. Per lavorare bene non basta solo la chiave, magari

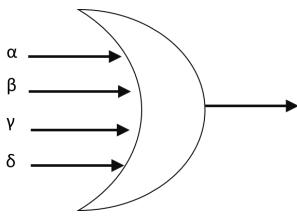


Figura 10.7: Circuito per calcolare la condizione di cache miss.

servono dei blocchi, come le cache che non lavorano in un modo puramente associativo come quello visto. Teoricamente, i sistemi di memoria sono di due tipi differenti, si possono montare ed esistono **memorie ad indirizzamento lineare** (tutto è in memoria, come visto nel primo esempio) o associativo, basate sui comparatori per cui quando diamo un indirizzo questo viene comparato con la cella e se c'è il valore si abilita il valore del registro in uscita, altrimenti non si abilita. L'abbiamo usata in sistemi operativi perché quando utilizziamo la memoria virtuale, essa ci dà una chiave e ci dà lo spiazzamento, se non abbiamo la chiave c'è un *page fault*: **la MMU è una memoria associativa**.

Parte II

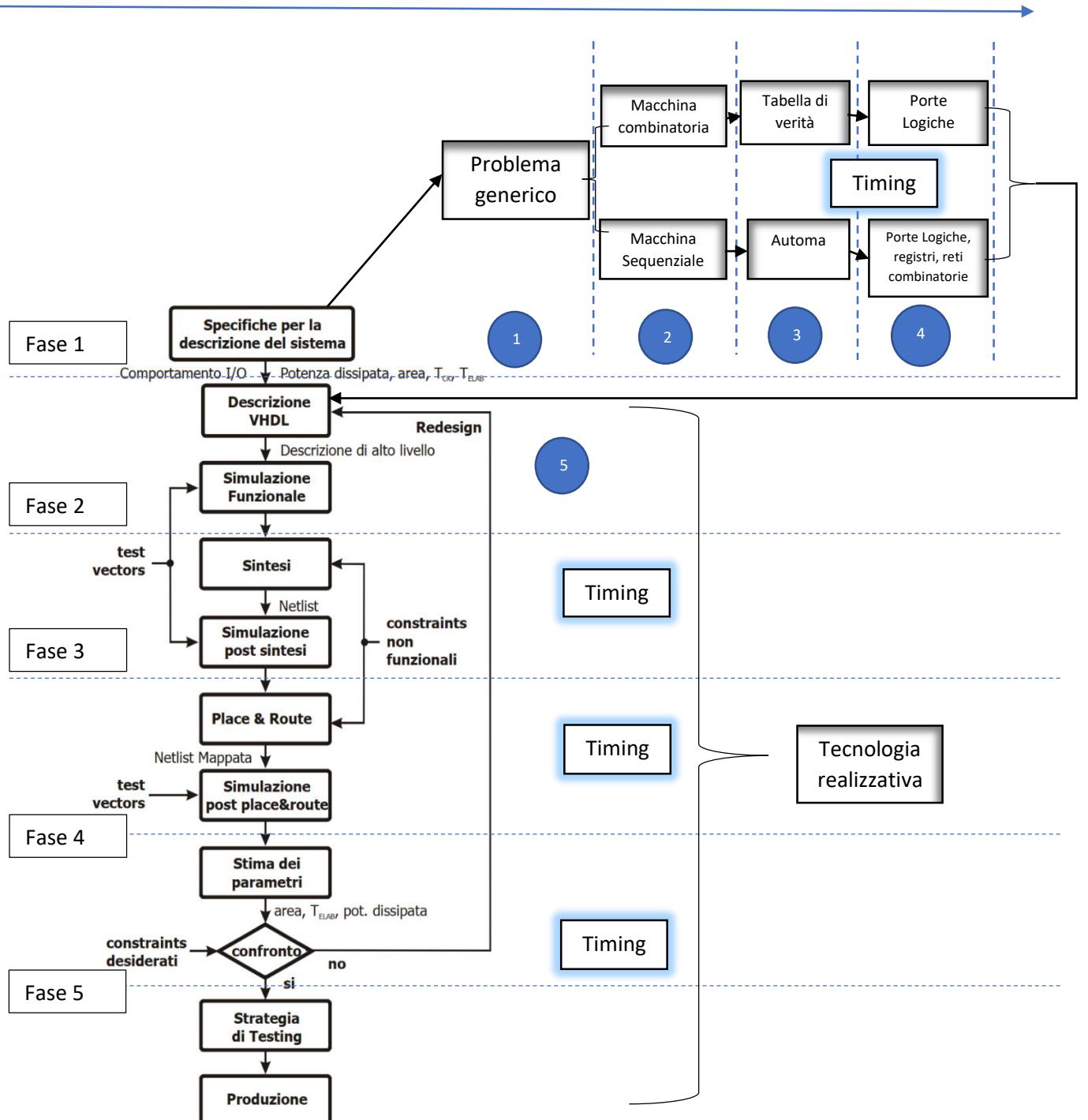
Appendici

Appendice A

Workflow dal problema alla sua soluzione

(*Andrea Abbate, Laura Cimmino, Adriano De Simone, Grazia Napolitano*)

Fasi di progettazione di un sistema



Fase 1:

1.1

Nella progettazione di un sistema digitale, si parte dalla specifica del problema in termini funzionali (comportamento ingresso-uscita, interfaccia con l'esterno, etc.) e non (throughput, area del circuito, frequenza del clock, dissipazione di potenza, etc.).

Per **problema generico** si intende un problema di qualsivoglia natura che sia calcolabile, dato un numero finito di ingressi discreti. Si dice che un problema è **calcolabile** se ammette una soluzione reale, mediante un algoritmo, in un tempo finito.

Un problema generico può essere risolto seguendo due approcci differenti:

- utilizzando macchine programmabili, le quali vengono costruite con particolari tecniche che tengono conto dell'obiettivo per cui vengono realizzate;
- utilizzando macchine dedicate, costruite ex novo a partire da componenti digitali; per questo motivo, esse possono essere o completamente dedicate o composte da una parte programmabile e da una parte dedicata.

Definizione: Macchine Programmabili e Dedicated

Macchine Programmabili: si tratta di macchine di tipo General Purpose, ovvero dispositivi, strumenti e meccanismi caratterizzati da una certa versatilità, adatti ad impieghi differenti e non specializzate per particolari esigenze.

I sistemi General Purpose sono basati su processori e su dispositivi periferici, che hanno un set di istruzioni base, le quali possono essere sequenziate per implementare un qualsiasi algoritmo; essi possono raggiungere elevate frequenze di clock (2-3 GHz), dissipano molta potenza, hanno un costo variabile in funzione della complessità e prevedono tempi di progettazione dell'hardware relativamente brevi. [1]

Le macchine programmabili possono costituire sottoparti di macchine dedicate.

Macchine Dedicated: si tratta di macchine costruite, ex novo o da macchine General Purpose, per eseguire unico e specifico compito. A parità di tecnologia, esse hanno prestazioni molto più elevate rispetto alle macchine general purpose; inoltre possono raggiungere elevate frequenze di clock, con costi di produzione bassi ma costi di NRE (Non Recurring Engineering Costs) elevatissimi. Prevedono infine tempi di progettazione dell'hardware molto più lunghi. [2]

La macchina dedicata può avere una componente programmabile.

Un particolare esempio di macchina dedicata è il processore (nella fattispecie, sono macchine generate perché dedicate ad un linguaggio, quindi realizzano poi una macchina programmabile) che utilizza il modello Register Transfer, descritto da un insieme di micro-operazioni, in cui ciascuna istruzione in un processore permette la gestione dell'hardware, in particolare definisce il trasferimento di dati fra registri.

1.2

La realizzazione di una macchina dedicata può essere effettuata in due modi distinti, progettando macchine combinatorie o sequenziali.

Definizione: Macchina combinatoria e sequenziale

Macchine Combinatorie: si tratta di macchine nelle quali il valore delle uscite ad un determinato istante dipende unicamente dal valore degli ingressi in quello stesso istante e, pertanto, non hanno necessità di memorizzare lo stato.

La specifica formale di una macchina combinatoria è la **tabella di verità**.

La tabella di verità è una funzione definita in maniera tabellare per cui alla variabile dipendente sono associate tutte le possibili combinazioni delle n variabili indipendenti.

Per ottimizzazione di una funzione si intende la sua trasformazione, attraverso passi successivi, con lo scopo di ottenere un'espressione equivalente ma migliore rispetto ad una data metrica di valutazione (area occupata, tempo necessario a produrre un dato risultato, potenza o energia assorbita ecc.). [3]

Si indica con $\mathbf{U} = \mathbf{F}(\mathbf{I})$ la funzione che descrive questo comportamento caratterizzante.

N.B. si è assunto per tale notazione che \mathbf{U} fosse l'uscita ed \mathbf{I} l'ingresso. \mathbf{F} rappresenta la funzione che lega ingresso ed uscita ed è differente da macchina a macchina.

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

Esempio Tabella di Verità

“Macchine Sequenziali: si tratta di macchine nelle quali il valore delle uscite in un determinato istante dipende non solo dal valore degli ingressi ma anche da quello degli stati (inteso come una condizione in cui si trova il sistema, dipendente dagli ingressi precedenti e dallo storico degli stati visitati) nello stesso istante; tengono conto dell’evolvere degli stati e sono pertanto provviste di memoria.

Il formalismo che descrive una macchina sequenziale è il diagramma a stati finiti, in cui l'evoluzione dello stato è dettata da un evento; se quest'ultimo è prodotto da una sorgente indipendente di sincronizzazione, la macchina è denominata **sincrona**, mentre nel caso in cui non lo sia, la macchina è definita **asincrona**.” [4]

“Quindi, nei circuiti sequenziali il valore delle uscite in un dato istante dipende sia dal valore degli ingressi in quello stesso istante sia dagli ingressi presentati al circuito nei tempi precedenti - quindi il fattore tempo compare fra quelli che determinano il comportamento del circuito. Una stessa configurazione di ingresso applicata in due istanti di tempo successivi può produrre due valori d'uscita differenti, se la “storia” (la successione degli ingressi) è diversa. Un circuito sequenziale ha memoria degli eventi passati e, quindi, richiede degli elementi in grado di conservare informazioni.” [5]

In particolare, una macchina sequenziale sincrona ha in ingresso un'abilitazione i cui fronti di salita o discesa indicano una sequenza di istanti o degli intervalli di tempo nei quali hanno luogo le transizioni di stato.

Un particolare segnale di abilitazione è il clock, che è un segnale periodico.

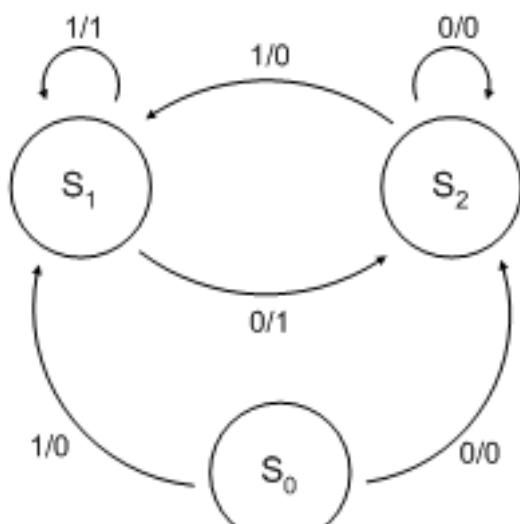


Figure-Diagramma/automa a stati finiti

Differentemente da quanto visto per le macchine combinatorie, in base alla relazione che sussiste nel rapporto tra ingresso, stato ed uscita, le macchine sequenziali possono essere ulteriormente classificate in:

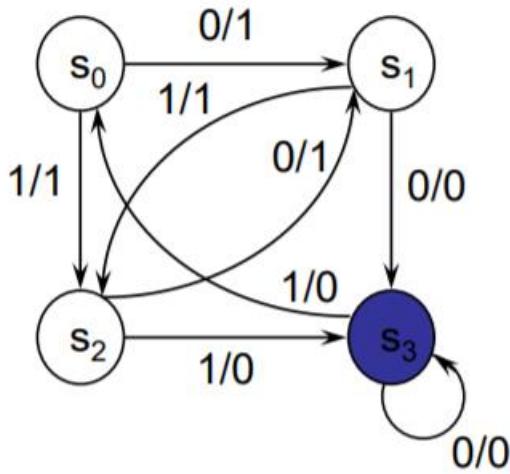
- **Macchine di Mealy:** il valore delle uscite in un istante t è funzione sia del valore degli ingressi che di quello degli stati nello stesso istante t $U=F(I,S)$;
- **Macchine di Moore:** il valore delle uscite in un istante t è funzione unicamente del valore degli stati $U=F(S)$.

NB: come già fatto prima, anche qui, si considera che U sia l'uscita, I sia l'ingresso ed S sia lo stato. F è la funzione che lega ingresso, stato ed uscita ed è differente da macchina a macchina.

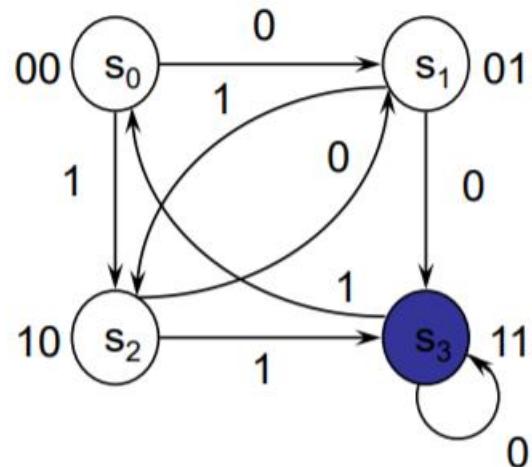
Una macchina sequenziale può essere descritta da un ulteriore formalismo che è la **Tabella degli Stati**, dove i , u e q (ingresso, uscita e stato) sono codificati in binario.

- Gli indici di colonna sono i simboli di ingresso $i \in I$
- Gli indici di riga sono i simboli di stato $q \in Q$ che indicano lo stato presente

- Gli elementi sono: – Macchine di Mealy: La coppia { q' , u }:
- $q' = t(i, q)$ è il simbolo stato prossimo
- $u = w(i, q)$ è il simbolo di uscita – Macchine di Moore;
- Nelle macchine di Moore i simboli d'uscita sono associati allo stato presente. [6]



Macchina di Mealy



Macchina di Moore

1.3

Per poter utilizzare i sistemi visti al passo 3 si necessita di descriverli attraverso un formalismo matematico che consenta di giungere ad una ottimizzazione della funzione d'uscita della macchina. In tal senso, le macchine combinatorie possono essere gestite ricorrendo alle tabelle di verità (ottimizzabili con i metodi di minimizzazione di Karnaugh e Quine-McCluskey), o ricorrendo a tool software come SIS o ESPRESSO) mentre quelle sequenziali vengono gestite ricorrendo agli automi. La minimizzazione è un processo ottimale per ridurre il numero di componenti (porte logiche) con cui procederemo alla costruzione del circuito e, quindi, ci consente di ammortizzare costi e tempi di sviluppo ed esecuzione.

NB: Non tutte le reti combinatorie sono ottimizzabili e, come visto, non vi è un criterio unico, inoltre se ottimizzate, sono una descrizione ottimale su più fronti del progetto della rete ma, tuttavia, se si ha, ad esempio, a disposizione una ROM, l'ottimizzazione non è più un passo fondamentale da effettuare e può essere pertanto evitato. Rimane un passo fondamentale nella progettazione ASIC, perché ottimizzando la funzione migliorano tutti gli aspetti tecnologici alla base della costituzione del sistema (area, potenza, velocità, surriscaldamento, ...), a valle della messa su silicio in fonderia.

Ottimizzando una funzione combinatoria, si possono introdurre problematiche di alee o di sintetizzazione che complicano la fisica realizzabilità. Da buoni ingegneri, bisogna sapere attuare un trade-off tra le cose. Le alee e le risoluzioni vengono affrontate di seguito.

1.4

A valle della sintetizzazione ed ottimizzazione della funzione di uscita della macchina di nostro interesse (operata mediante i metodi di cui al passo 4), c'è da tenere conto di tempi e tecnologie di sviluppo. Questi

due fattori entrano in gioco mantenendo una forte interazione tra loro, ragion per cui è quasi impossibile parlare di gestione dei ritardi temporali e controllo dei casi indesiderati, senza riflettere sulla realizzazione fisica dei circuiti.

Le macchine combinatorie prevedono una progettazione ed implementazione delle funzioni logiche (ottenute al passo 4) mediante una rete di porte logiche (AND, OR, NOT o di insiemi funzionali come la NAND e la sua duale, la NOR), in cui potrebbero manifestarsi **ALEE**.

Definizione: Aleee

Si chiamano Aleee (o hazard) quei fenomeni per i quali le uscite, anche se solo per brevi intervalli di tempo, assumono dei valori imprevisti. Sono classificate in:

- Aleee **Transitorie**: le uscite della rete assumono valori diversi da quelli progettati soltanto nel transitorio conseguente alle variazioni degli ingressi;
- Aleee **di Regime** per macchine sequenziali: l'uscita a regime assume un valore diverso da quello progettato.

ALEA TRANSITORIA (combinatoria)	SOLUZIONE
Multipla: Variazione simultanea di due o più variabili di ingresso.	Eliminare ingressi non adiacenti
Per impulsi concomitanti: Presenza di due o più impulsi.	Evitare impulsi concomitanti [9]
Statica: Variazione temporanea dell'uscita che dovrebbe rimanere costante.	Aggiungendo gli implicanti ridondanti si "coprono" le variazioni che determinano l'alea
Dinamica: Oscillazione temporanea dell'uscita.	Si eliminano eliminando le aleee statiche nelle sottoreti componenti

[7]

Le aleee che si verificano nelle macchine sequenziali sono le aleee di regime; tuttavia, se si risolvono le aleee transitorie nella parte combinatoria della rete sequenziale si avranno meno probabilità di insorgenza di aleee di regime. I problemi inerenti prettamente alla parte di memorizzazione della macchina vengono risolti utilizzando macchine asincrone a sincronizzazione esterna.

Non necessariamente si avranno problematiche d'uscita ma, onde evitare casi indesiderati, c'è bisogno di un'opportuna tempificazione dei segnali.

Nel caso delle macchine sequenziali, come visibile in info-grafica, alla progettazione e realizzazione della rete di porte logiche si aggiungono complessità e problematiche dovute a progetto e realizzazione di registri e reti combinatorie. Dunque, anche qui si necessita di un opportuno meccanismo di tempificazione.

Definizione: Clock

Come accennato precedentemente, uno dei principali meccanismi di tempificazione e/o di abilitazione è ottenuto utilizzando un segnale di clock.

Il clock è un segnale periodico, definito da una propria frequenza $1/T$.

Può assumere soltanto due valori (clock alto o basso) e la transizione tra un valore e l'altro definisce un fronte di salita (rising edge, transizione 0 \rightarrow 1) o un fronte di discesa (falling edge, transizione 1 \rightarrow 0).

In un sistema digitale sincrono il segnale di clock viene utilizzato per determinare i cambiamenti di stato negli elementi di memoria. [8]

Definizione: Flip Flop

Per la memorizzazione degli stati, si vede necessaria la scelta e/o progettazione/posizionamento, come già in elenco, di flip flop ad hoc.

La tempificazione è risolta osservando e scegliendo a seconda delle esigenze il comportamento di 3 tipologie di flip flop:

1. **Macchine Latch:** cattura l'ingresso/dato per tutto l'intervallo in cui il segnale di abilitazione è attivo, ovvero ogni variazione è memorizzata;
2. **Macchine Edge Triggered:** Un flip-flop edge-triggered è sensibile ad una variazione del segnale di abilitazione A e non durante tutto il periodo in cui A è attivo,
Edge-triggered sul fronte di salita (ETs): sensibile al fronte basso-alto (0 -> 1) del segnale A;
Edge-triggered sul fronte di discesa (ETd): sensibile al fronte alto-basso (1 -> 0) del segnale A;
3. **Master-Slave:** Un flip-flop master slave è un flip-flop in cui 2 latch sono collegati in serie. Il segnale di abilitazione (che può essere il clock) è negato per uno di essi. Di conseguenza il secondo flip flop reagisce in risposta ad un cambio di stato del primo. Infatti, sul fronte attivo del segnale di abilitazione, il primo flip-flop è abilitato e di conseguenza è sensibile all'ingresso. Il secondo, invece, resta disabilitato avendo in ingresso il l'abilitazione negata. Viceversa, sull'altro fronte, il primo flip-flop è disabilitato, mentre il secondo è abilitato, riportando in uscita ciò che era stato memorizzato dal primo flip-flop. [9]

1.5

Come già accennato precedentemente, per poter risolvere un problema ottenendo i risultati migliori, è possibile utilizzare l'approccio “divide et impera”, ovvero si decompone ricorsivamente il problema originario in problemi di dimensioni più piccole.

La decomposizione implica concetti importanti quali la gerarchizzazione (suddivisione di un sistema in componenti più semplici), l'individuazione dei moduli che compongono un determinato livello di astrazione, e il collegamento tra questi ultimi. Di conseguenza, acquista maggiore importanza la presenza di moduli preesistenti, le “ librerie”, che permettono il riuso dei blocchi logici del sistema, aumentandone la produttività.

Dopo lo studio delle metodologie risolutive finora mostrato ci si trova, quindi, a dover procedere verso la progettazione fisica, per la quale si ricorre all'utilizzo di circuiti integrati.

Un circuito integrato è un dispositivo elettronico (“chip”), che contiene al suo interno un insieme di celle logiche opportunamente interconnesse in modo da realizzare la funzionalità desiderata.

Ogni cella è composta da una combinazione di elementi logici di base: flip flop, porte AND e porte NOT. [10]

I seguenti sono due tipi differenti di circuiti:

- **FPGA** (Field Programmable Gate Array): sono dispositivi rivoluzionari, diffusi in una manciata di anni. Assieme ai CPLD sono i PLD (Programmable Logic Device) che dominano il mercato. Il dispositivo è programmabile solo dal produttore (e non più in fonderia). I dispositivi contengono

sia logica programmabile, sia circuiti hardware speciali. La parte programmabile è composta da moltissime celle, contenenti 4 funzioni:

- LUT (look-up table);
- Multiplexer;
- Logica di propagazione del riporto (FA);
- Flip Flop D.

La parte di hardware speciale contiene circuiti integrati di largo impiego, come gestori del clock, circuiti aritmetici, memorie veloci, etc.

L'FPGA compone una funzionalità a partire da celle di base semplici grazie a meccanismi di interconnessione tra le celle.

Gli FPGA, e recentemente anche i CPLD, sono largamente impiegati grazie alla facilità con cui è possibile programmarli. Tipicamente è un PC, che tramite un opportuno protocollo, configura i dispositivi programmabili. [11]

- **ASIC** (Application Specific Integrated Circuit): è un circuito integrato creato appositamente, una volta per tutte durante la fabbricazione, per risolvere un'applicazione di calcolo ben precisa (special purpose).

La specificità della progettazione, focalizzata sulla risoluzione di un unico problema, consente di raggiungere delle prestazioni in termini di velocità di processazione e consumo elettrico difficilmente ottenibili con l'uso di soluzioni più generiche (general purpose).

Lo sviluppo di questi circuiti è, però, molto costoso e per questo motivo sono utilizzati in campi in cui possono essere usati in maniera massiccia (alti volumi di mercato), come l'elettronica di consumo, mentre per usi su scala più limitata vengono preferite le tecnologie riprogrammabili (FPGA). Uno specifico settore in cui si sono fatti strada i processori ASIC è il mondo bitcoin, grazie alle elevate prestazioni raggiungibili con questo tipo di tecnologia. [12]

La differenza tra i due approcci sta tutta nel grado di libertà e va valutata in base a quanto mostrato dalla seguente tabella:

CRITERI VALUTATIVI	ASIC	FPGA
1. Livello di Integrità	Si	No
2. Velocità del circuito	Si	No
3. Temperatura (consumo)	Si	No
4. Costo	No	Si

ASIC, come si evince, risulta meno performante in termini di costo poiché le schede ASIC non prevedono alcuna forma pre-configurata sul silicio, quindi, realizzarle "ad hoc" comporta necessariamente un sovrapprezzo dovuto anche ai maggiori tempi di realizzazione.

Uno strumento per la progettazione e l'implementazione di sottosistemi logici altamente funzionali è il **VHDL** (Very High Speed Integrated Circuit Hardware Description Language), con cui dal livello di astrazione più esterno si può scendere gerarchicamente in dettaglio fino alle porte logiche.

Il VHDL è un linguaggio descrittivo, largamente diffuso a livello mondiale (standard IEEE), utilizzato per la caratterizzazione di sistemi complessi: non descrive un algoritmo, ma porte, segnali e sottosistemi fisici.

Non è, quindi, un linguaggio compilato, ma è un linguaggio concorrente di descrizione dell'hardware.

Ciò è necessario poiché è l'hardware stesso ad essere concorrente, in quanto si alimenta tutto il sistema e non una sua parte, e i sistemi fisici lavorano in contemporanea.

- Sincronia e clock

Un segnale dato da un clock viene utilizzato per **sincronizzare** il funzionamento di tutte le sottoparti del sistema; esso viene inviato agli ingressi di tutti i dispositivi come, per esempio, i flip flop, e in questo modo viene creata una macchina sincrona.

La forma d'onda preferibile per un segnale di clock è una forma periodica, poiché permette di controllare in maniera semplice l'intervallo di tempo T tra due operazioni; essa è generabile tramite un oscillatore periodico ad alta stabilità, tramite un flip flop di un contatore che agisca come un divisore di frequenza etc.

Un circuito (o macchina) sincrono(a) è un circuito logico caratterizzato da un unico segnale periodico di clock per tutti i flip flop, ovvero con una frequenza di ripetizione del segnale di clock perfettamente uguale su ogni flip flop, soggetto tuttavia a inevitabili sfasamenti dovuti ai ritardi di propagazione.

Due o più macchine sincrone che sono fra loro asincrone (in quanto usano segnali di clock distinti, quindi due oscillatori di diversa frequenza) compongono un circuito asincrono: in esso la fase tra i diversi clock asincroni è in genere continuamente variabile.

Il segnale di clock non può e non deve avere “glitch”, ovvero non può avere impulsi fisici brevi ed improvvisi, causati da errori non prevedibili (non voluti logicamente): questi possono essere causati, ad esempio, da interferenze elettromagnetiche esterne sulla linea, o commutazioni non simultanee di livelli logici.

Se un ingresso sincrono subisce una transizione o un glitch durante il setup time (il tempo minimo per cui deve rimanere stabile l'input prima del fronte di clock) o hold time (il tempo minimo per cui deve rimanere stabile l'input dopo il fronte di clock) c'è una probabilità non nulla che il flip flop carichi in uscita il valore sbagliato (upset – cambio di stato). Quindi, in una macchina logica reale (FPGA o ASIC), il valore caricato in uscita su ciascun flip flop dipende dalla struttura logica, ma anche dai tempi di propagazione dei segnali. Bisogna quindi tenere conto delle temporizzazioni e delle loro possibili variazioni.

I **tempi di propagazione** sono i ritardi tra la transazione in un punto fisico di una linea di un circuito logico e la transazione logicamente conseguente in un altro punto fisico.

I segnali usati per pilotare gli ingressi sincroni possono avere glitch, ammesso che questi non avvengano durante il setup o hold time.

I tempi di propagazione dipendono dalla tecnologia del dispositivo, ovvero dalle caratteristiche fisiche delle celle logiche, e dalla struttura logico/fisica realizzata, ovvero dalle interconnessioni fra celle, dal numero di porte logiche etc.; essi variano in base a tolleranze di fabbricazione (ogni dispositivo ha caratteristiche che fluttuano attorno a specifiche nominali), all’aging del dispositivo (il suo stato di invecchiamento), o parametri operativi, quali la tensione di alimentazione e la temperatura interna dei semiconduttori (“junction temperature”, la quale è sempre maggiore della temperatura ambiente).

- Simulazione

A differenza di un programma compilato, per cui è necessaria l'esecuzione per la verifica di correttezza, per il VHDL è necessario simulare il sistema, utilizzando parametri e condizioni quanto più vicini possibile al caso reale.

Nella definizione di un sistema simulato, si avrà, comunque, un vantaggio rispetto al sistema reale: la possibilità di misurare qualsiasi grandezza fisica e stato del sistema, in quanto, essendo appunto simulato, possono essere valutati tutti i segnali; infatti le misurazioni su un sistema reale possono essere effettuate attraverso l'analisi dell'uscita a valle di ingressi noti, ma, laddove fosse necessario, misurare anche stati intermedi del sistema diventa tutt'altro che banale.

Ciò significa che, se nel sistema è presente un qualsivoglia errore, esso potrà essere scoperto solo tramite l'uscita, che sarà ovviamente sbagliata, e non sarà possibile sapere esattamente dove questo è stato commesso.

Fase 2:

Descrizione del problema in VHDL e, quindi, simulazione funzionale per verificare la correttezza del suo comportamento (e verificare la congruenza con le specifiche). In caso di comportamento errato, si ripete la fase precedente.

Fase 3:

Raffinamento della simulazione tramite sintesi, che consiste nella trasformazione di una descrizione ad alto livello in una di livello inferiore e che produce come risultato una **netlist** (file che descrive i collegamenti tra i componenti del circuito), la quale consente di utilizzare una tecnologia implementativa: arrivati a questo punto, viene fatta un'ulteriore simulazione per verificare l'esattezza della sintesi.

N.B Nel caso di sintesi errata o incongruente alle specifiche, si riparte dalla fase 2.

Fase 4: Place & Route

Allocazione effettiva delle porte sul silicio; questa fase viene gestita diversamente per ASIC e FPGA:

- Caso ASIC:

Si può decidere dove piazzare le porte, avendo a disposizione una piastra di silicio vergine.

- Caso FPGA:

È possibile scegliere soltanto come collegare i componenti già presenti nel circuito.

Fase 5:

Viene fatta un'ulteriore simulazione per verificare nuovamente la correttezza del sistema, utilizzando i parametri effettivi del circuito: se i risultati sono accettabili, sulla base delle specifiche iniziali, si può procedere con l'implementazione e la produzione.

In caso contrario si riparte dalla fase 1, e si procede con un redesign.

Per simulare, quindi, bisogna avere una buona descrizione fisica del sistema e un avanzamento del tempo logico opportuno, che può essere a divisione di tempo fisso (passo di quantizzazione), oppure gestito ad eventi (il tempo avanza al verificarsi di ogni evento).

N.B. Nella fattispecie, VHDL è ad eventi.

Il VHDL provvede unicamente alla descrizione, alla definizione dei moduli e alla simulazione.

GHDL è un interprete open source di VHDL, attraverso il quale è possibile interpretare i comandi descrittivi dell'hardware che, però, non genera la netlist.

Xilinx (uno dei principali produttori di dispositivi logici programmabili) è un ambiente di sviluppo integrato (IDE) per VHDL e supporta anche il livello tecnologico.

La descrizione di un sistema digitale (*n*dr con VHDL) può essere realizzata a diversi livelli di astrazione:

- **Behavioral:** viene specificata la funzione ingresso-uscita del componente, ma non come questo è implementato;
- **Structural:** il sistema viene specificato in termini di sottocomponenti;
- **Data-flow:** descrizione del sistema in termini di flusso (concorrente) dei dati e dei segnali di ingresso e controllo.

Esistono anche altri livelli di astrazione intermedi:

- **RTL** (Register Transfer Level): consente una descrizione del sistema in termini di registri e macchine combinatorie (tutte le reti che vengono utilizzate sono in realtà combinatorie e temporificate mediante l'uso dei registri, che si comportano da buffer);
- **Gate Level:** descrive la struttura del sistema in termini di porte logiche. [13]

Ai fini di una maggiore comprensione di questo compendio, si vuole, qui, fornire una distinzione tra i seguenti termini:

- **Simulabile:** si dice di un sistema per cui è possibile sviluppare un modello matematico della realtà che consente di valutare e prevedere la dinamica di una serie di eventi o processi susseguenti all'imposizione di certe condizioni da parte di analisti o utenti;
- **Sintetizzabile:** si dice di un sistema simulabile di cui è possibile descrivere e/o realizzare le componenti che ne delineano la struttura fisica, ovvero l'individuazione dei componenti e delle interconnessioni necessarie per realizzarlo seguendo la preassegnata specifica funzionale.

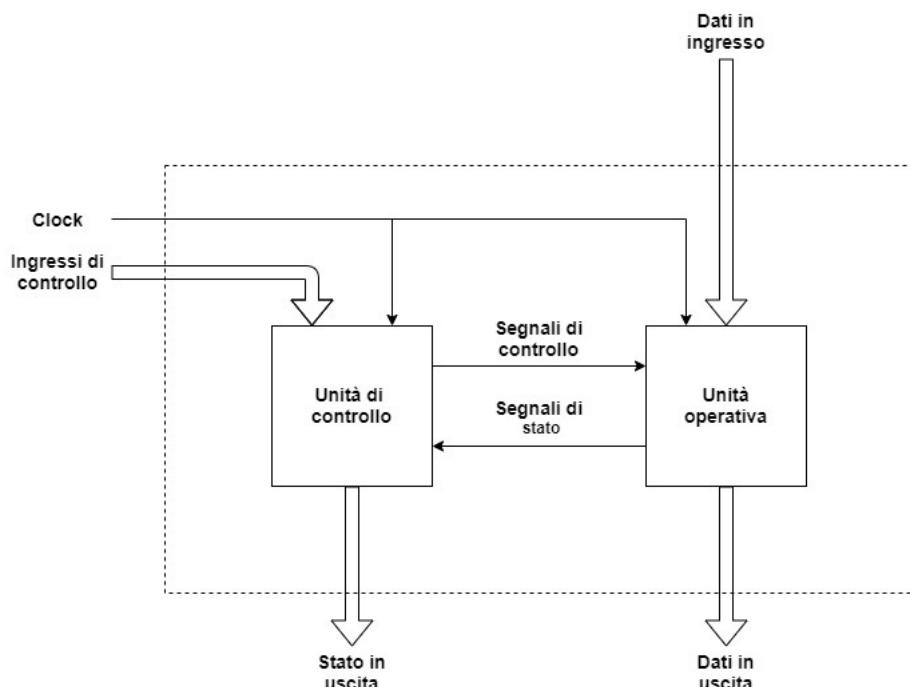
Appendice B

Sistemi complessi: analisi e sviluppo

(*Stefano Mercogliano, Daniele Ottaviano, Francesco Vitale*)

1 Decomposizione funzionale di sistemi complessi

Quando la complessità del sistema risulta aumentare, e vede l'interconnessione di più componenti secondo esigenze che vengono specificate dall'analisi dei requisiti di sistema, risulta conveniente adottare un modello che consente di dominare la complessità delle risposte che il sistema deve dare a certi input. Lo schema che durante il corso abbiamo seguito è il seguente:



Decomposizione funzionale di un sistema complesso

L'interfaccia, pertanto, del sistema, è costituita da:

- Un insieme di ingressi di controllo *esterni*;
- Un insieme di dati in ingresso;
- Un ingresso necessario a tempificare le operazioni da svolgere nel contesto del sistema;

- Uno stato in uscita che fornisce informazioni riguardo al sistema;
- Un insieme di dati in uscita prodotti dal sistema.

Già dalla prima fase di sviluppo, dunque, ragionare sull’interfaccia e su quali siano ingressi e uscite è l’approccio migliore per un progetto efficace.

Un appunto da fare, prima di arrivare a esporre in dettaglio i due blocchi, è la motivazione che si cela dietro questa trattazione. In teoria, se si dispongono di *tutte le combinazioni possibili di input* in ingresso al sistema, sarebbe ‘possibile’ modellare la risposta del sistema a ognuno di questi input. Tuttavia, già per un discreto numero di bit in ingresso, le corrispondenze da determinare sarebbero troppe per poter essere sviluppate a mano, ma anche eventualmente da memorizzare sul supporto informatico. È proprio per questo che il riferimento si rivelerà essere particolarmente utile nella progettazione dei sistemi di elaborazione.

1.1 Unità operativa

Il primo dei due blocchi che verrà trattato riguarda l’unità operativa.

Questo blocco incapsula la parte inherente all’elaborazione dei dati ottenuti in ingresso sulla base degli *ingressi di controllo* forniti dalla unità di controllo, e risponde fornendo uno stato che sarà utile alla unità di controllo per scandire le prossime operazioni da comandare al blocco di elaborazione. Al termine delle elaborazioni, in uscita verrà rilevato il prodotto di queste ultime.

Importanti direttive da sottolineare per la progettazione di questo blocco riguardano:

- L’astrazione dal comando e la tempificazione delle operazioni: l’unità operativa dovrà essere funzionale alle elaborazioni che le sono richieste di svolgere. In che modo le operazioni si succederanno, o quando inizieranno, o quando termineranno, sono problematiche inerenti al *controllo* della unità operativa;
- È importante fare in modo di *non adattare l’interfaccia del blocco cercando di anticipare il controllo delle operazioni*, ma limitarsi a prevedere quali sono gli ingressi utili ad abilitare eventuali registri, consentire il caricamento dei dati, e quali sono le uscite di stato che il blocco dovrà fornire.

- L'individuazione dei componenti elementari e le loro interconnessioni utili a realizzare il comportamento desiderato a fronte degli input forniti al blocco.

Constatare che *controllo* e *fase elaborativa* sono due cose da tenere separate, non significa che l'unità operativa deve essere puramente combinatoria: l'evoluzione della macchina avverrà sempre in sincronia con un ingresso di temporizzazione (che nello schema è nominato *clock*). Ciò significa che la fase elaborativa può essere il prodotto di più operazioni eseguite in sequenza, ma che non necessitano, per poter essere attuate, di un controllo.

Il controllo, è bene ribadirlo anche in questa sede, è utile a scandire l'inizio, *quali operazioni effettuare*, condizioni di terminazione, ma è del tutto svincolato da *come* le operazioni verranno svolte.

1.2 Unità di controllo

Il blocco a cui è delegato il controllo della parte operativa è costituito dall'unità di controllo.

Non sussiste unità di controllo senza unità operativa; se non è stata dapprima sviluppata la parte da *controllare*, non ha senso sviluppare il controllo di tale parte. È proprio per questo che, nella precedente sezione, si è messa enfasi sul disaccoppiamento dell'interfaccia del blocco operativo da quello di controllo.

Come verrà approfondito successivamente, l'unità di controllo può essere sviluppata seguendo approcci tra loro differenti, che sono tra loro tuttavia accomunati dall'obiettivo cardine del controllo, che riguarda:

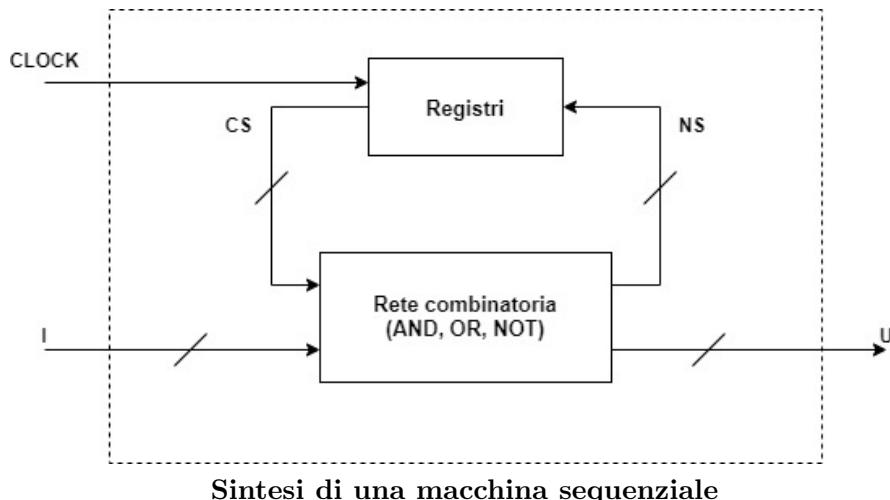
- La lettura di segnali di controllo esterni e la risposta *controllata*, da inoltrare al blocco operativo, utile soprattutto a determinare il reset della macchina o l'inizializzazione della struttura;
- La decisione da effettuare in risposta allo stato della macchina o al numero di operazioni elementari svolte: ciò determina, anche in questo caso, la risposta *controllata* che verrà inoltrata all'unità operativa;
- La notifica dello stato totale della macchina, che potrà essere utilizzata da blocchi che *inglobano* il sistema sviluppato per la progettazione di sistemi *più complessi*.

Ancora, è utile anche in questa sede pungualizzare che non è detto che la unità di controllo sia sequenziale, e che quindi abbia una memoria. L'unità di controllo potrebbe, in effetti, essere anche puramente combinatoriale: un semplice esempio vede l'unità di controllo ridotta a implementare una logica di priorità tra più segnali provenienti da diverse sorgenti.

1.2.1 Logica di controllo

Come si è già accennato nella precedente sezione, l'unità di controllo, anche in virtù del fatto che in un progetto è sviluppata a valle della unità operativa, può essere sviluppata adottando logiche tra loro differenti ma che assolvono allo stesso compito. Vengono denominate *logica cablata* e *logica micropogrammata*:

- La logica cablata è quella che prevede la sintesi della unità di controllo come un'interconnessione di porte logiche atte a realizzare la parte combinatoria di un circuito a cui può essere eventualmente apposto un insieme di registri per incapsulare la memoria di un sistema sequenziale. Una sintesi del genere risponde, pertanto, al classico modello dei sistemi sequenziali (a meno, come già detto, del banco di registri), che deriva proprio dalla **progettazione di un automa**.

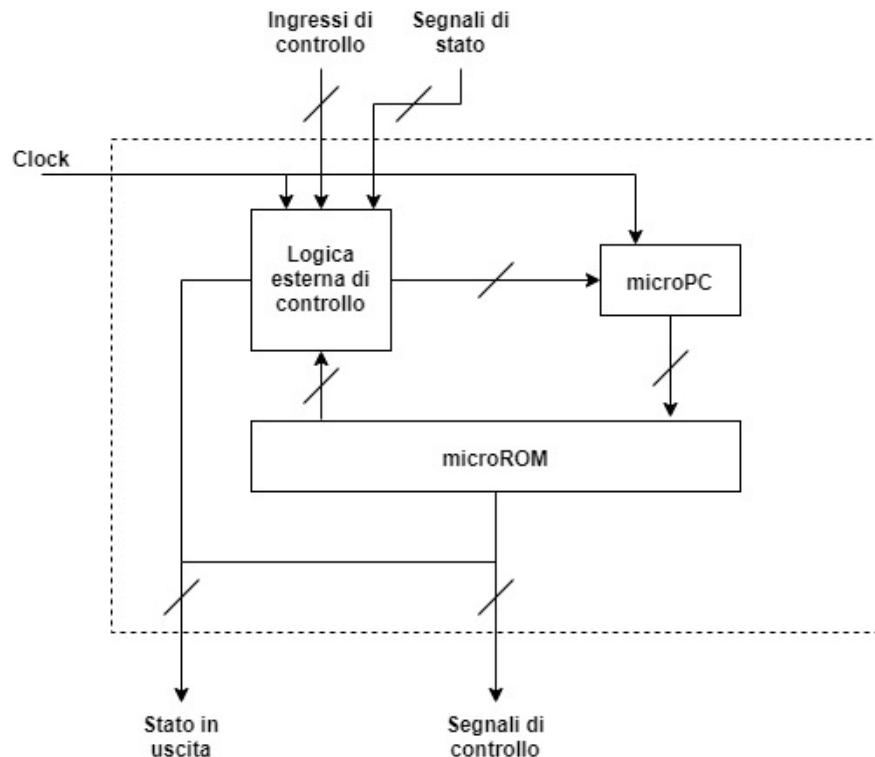


Un sistema del genere è potenzialmente *veloce*, ma decisamente *poco*

flessibile: basta constatare che se la logica di controllo dovesse, per qualche motivo, venire estesa, l'automa stesso dovrebbe essere rimodelato, e pertanto la rete del tutto rivoluzionata.

- La logica microprogrammata segue un approccio diverso dalla sintesi diretta di un automa. Poichè a ogni stato coincidono certi segnali di controllo inoltrati in uscita, di cui alcuni di questi non dipendono da alcuna particolare condizione, la possibilità di poter modellare, in risposta a certi stati, dei segnali *costanti* che vengono inoltrati in uscita, porta alla considerazione di un modello diverso da quello già presentato, ossia un modello in un certo senso *programmabile*, poichè l'aggiunta di uno stato, o di un'uscita, comporterebbe semplicemente l'estensione di una struttura con l'aggiunta di nuove costanti da segnalare in uscita. Secondo quanto detto, questa logica necessita di un componente che possa essere *programmabile*, e che consenti, quindi, di poter manipolare i dati contenuti nella struttura in maniera flessibile, indipendente dal particolare controllo da attuare, o dai segnali da inoltrare; un componente di tale natura è proprio la ROM.

Non verrà, in questa sede, esposta in dettaglio la struttura di una ROM, bensì sinteticamente verrà presentata (in astratto, in concreto successivamente) la struttura di una rete di controllo realizzata mediante logica microprogrammata:



Come già anticipato, si ha una (micro)ROM nel sistema di controllo, che però è coadiuvata da altri due blocchi, che come si vede sono nominati *logica esterna di controllo* e *microPC*:

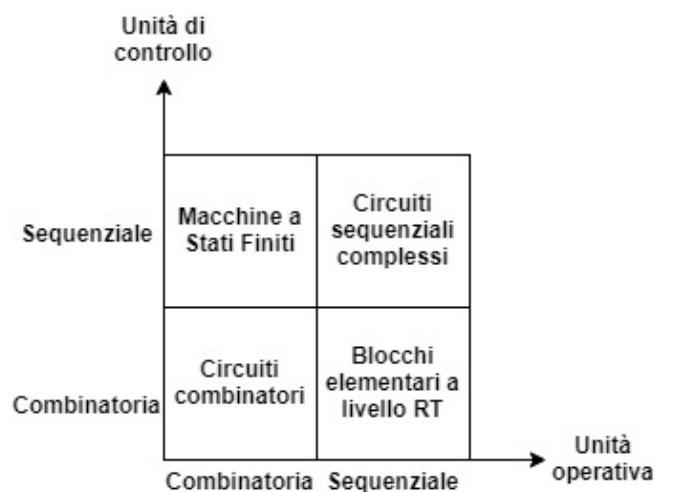
- La logica esterna di controllo è necessaria poichè oltre ad alcune uscite costanti che possono essere mandate in uscita *non filtrate*, sono presenti altri segnali ricavabili dalla ROM il cui valore è determinabile solo a fronte della particolare condizione che il sistema di controllo rileva. In funzione, dunque, degli ingressi di controllo e dei segnali di stato, la logica esterna di controllo determinerà il valore dei rimanenti segnali di controllo cui sottostanno a delle specifiche condizioni, e in più aiuterà a determinare qual è lo stato

del sistema totale in uscita;

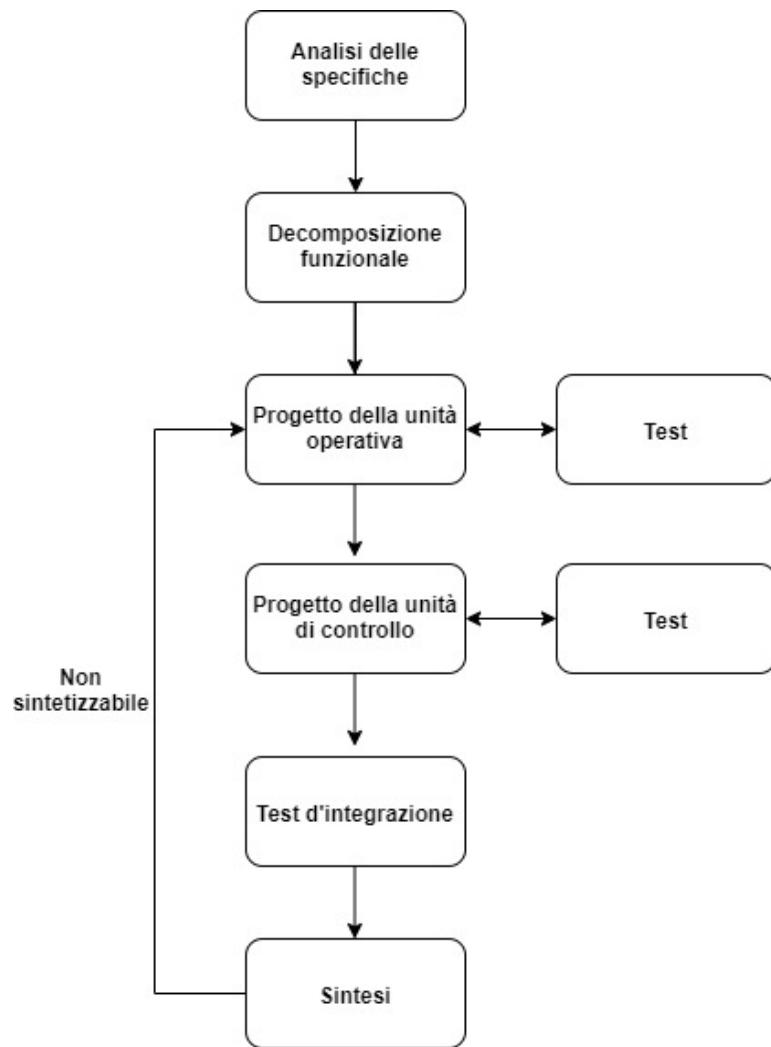
- Una ROM è una memoria indirizzabile: il (micro)PC è funzionale a determinare qual è l'indirizzo della prossima *parola* da trarre dalla ROM. Ciò che è interessante notare è che la stessa ROM concorre a determinare qual è l'indirizzo della prossima parola da trarre: tuttavia questa cosa rientra tra le informazioni che vanno modulate in base alle condizioni di contorno. Tutto ciò significa che per determinare il prossimo indirizzo, la logica esterna di controllo sarà determinante.

1.3 Classificazione di sistemi

Una volta che si è astratto il processo di decomposizione funzionale del sistema, progettando unità operativa e unità di controllo, è possibile classificare il sistema stesso nel contesto di una delle tante possibili tassonomie dei sistemi. Quella che in questo testo si prende in esame è la tassonomia che classifica i sistemi in base al comportamento delle uscite a fronte degli input. I comportamenti, in particolare, possono essere due, e sono, come già noto, *combinatorio* e *sequenziale*: in base al comportamento di unità di controllo e unità operativa, il seguente schema, che è ripreso da [1], esplicita quel che già in realtà è stato ribadito più volte, ossia che tutti i comportamenti per unità operativa e unità di controllo sono ammessi, indipendentemente dalla semantica che questi blocchi simboleggiano:



2 Ciclo di sviluppo di un sistema complesso



2.1 Analisi delle specifiche

Analizzare le specifiche (autoprodotte o fornite) è il primo passo necessario per poter passare alla fase di progettazione del sistema. In questa fase, compreso il problema, è opportuno scandire il comportamento che il sistema dovrà assumere a fronte degli input forniti: prima ancora di determinare come scomporre le funzioni, è importante comprendere quali sono gli input possibili, come la macchina deve operare in risposta agli input (di cui si conosce la natura), da che punto la macchina comincia a operare, e in che condizioni deve terminare la propria esecuzione.

In questa fase è anche molto utile scandire esplicitamente quali sono i passi algoritmici che implementano il comportamento desiderato: questa componente sarà di grande aiuto sia nell'individuare i componenti della unità operativa, sia nel comprendere quali sono le condizioni sugli input che la macchina dovrà prendere carico nell'esecuzione.

Un'analisi comprensiva dovrebbe anche valutare i vincoli associati alla tecnologia in possesso, per poterne poi prendere carico durante la fase di progettazione, ma ciò esula dagli obiettivi di questo testo di riferimento.

2.2 Decomposizione funzionale

La fase di decomposizione funzionale calca esattamente ciò che è stato espresso nella prima parte di questo documento, e attua il principio ingegneristico di *separazione degli interessi*. Una volta compreso il problema, stabiliti gli input possibili, e uno pseudocodice che rappresenta i passi algoritmici che la macchina dovrà eseguire, la decomposizione funzionale sarà utile a determinare quali sono i problemi presentati dalle specifiche e quale blocco sarà preposto a risolverli: da qui segue, naturalmente, la progettazione di unità operativa e unità di controllo.

2.3 Progettazione della unità operativa

Dell'unità operativa si è già ampiamente discusso: in questa sede, ci si limita a ribadire che la progettazione di questa unità è propedeutica alla progettazione della unità di controllo, e viene approfondito (pur sempre in astratto) il contenuto della parte operativa.

2.3.1 Componenti e interconnessioni

Internamente, la parte operativa possiede un insieme di *componenti* che sono tra loro interconnessi: i componenti possono avere qualsiasi comportamento, a patto che servano unicamente a realizzare gli obiettivi della unità operativa, ossia l'elaborazione degli input ottenuti in ingresso. La fase di decomposizione funzionale è stata determinante: stabiliti quali sono i componenti notevoli (multiplexer, registri a scorrimento, macchine aritmetiche), e quali invece vanno implementati ad-hoc (reti combinatorie o sequenziali speciali), è sufficiente stabilire per ognuno di loro le interfacce che andranno a collegarsi all'interfaccia più esterna del blocco operativo. Una volta stabiliti componenti, interconnessioni, e input necessari a soddisfare l'elaborazione delle informazioni, si aprirà la strada alla progettazione della unità di controllo, che fissata l'interfaccia della unità operativa, dovrà essere sviluppata in funzione di quest'ultima.

2.4 Unità di controllo

Agganciandosi a quanto prodotto nella fase di progettazione della unità operativa, si dispone di un'interfaccia da controllare, ma anche della possibilità di poter disporre di uscite di stato (provenienti dal blocco operativo), ingressi di controllo esterni, e un riferimento temporale per la sincronizzazione delle operazioni.

2.4.1 Tempificazione delle operazioni e modellazione

Partendo da quanto si è concluso nella decomposizione funzionale, nel caso il controllo sia sequenziale, si può modellare il comportamento della unità di controllo tramite un diagramma degli stati (se la macchina è sequenziale) oppure procedere alla realizzazione di una rete puramente combinatoria.

In ogni caso, non è solamente sufficiente comprendere come sequenzializzare le operazioni, ma è necessario (pena il mancato funzionamento del circuito risultante) prendere in considerazione il riferimento temporale a disposizione, che è assolutamente **determinante**. La tempificazione del controllo stabilisce *quando* il controllo dovrà avvenire, che è uno dei fattori più sensibili nella progettazione di un sistema complesso. Nel caso, infatti, la tempificazione del controllo sia inadeguato, si rischierebbe di non avere il

comportamento atteso; ad esempio, potrebbe succedere che la condizione di terminazione sia valutata *prima* che l'unità di elaborazione possa aver effettuato l'ultima operazione, oppure potrebbe succedere che lo stato tratto dall'esterno sia *ricavato al momento sbagliato* (e quindi, di conseguenza, venga fatta la scelta sbagliata).

Le soluzioni ai problemi di tempificazione non sono determinabili a priori, ma alcune linee guida possono essere utili nella corretta tempificazione delle operazioni:

- Dare la possibilità al circuito operativo di evolvere correttamente prima di trarne lo stato oppure valutare un altro ingresso di controllo;
- Lavorare su più fronti del segnale di tempificazione (clock): può fare la differenza far effettuare un conteggio sul fronte di discesa piuttosto che di salita a un contatore;
- Prendere in considerazione i vincoli tecnologici per un opportuno dimensionamento del segnale di tempificazione.

2.5 Test e composizione

È importante, durante il ciclo di sviluppo del sistema, fare in modo di testare opportunamente tutti i moduli che si vanno a realizzare. Il mancato test di componenti può portare a comportamenti inattesi da parte del sistema integrato: quando componenti fallaci sono già stati interconnessi tra loro, è molto più difficile comprendere quale sia il problema del sistema.

Uno sviluppo ideale, pertanto, prevede il testing dei singoli componenti, il testing dell'unità operativa composta, il testing dell'unità di controllo e infine il testing del sistema integrato.

2.6 Sintesi

Al termine di tutto, constatato dalla simulazione che il comportamento rispetti le specifiche, si può tentare di sintetizzare il circuito. Nello sfortunato caso quest'ultimo non sia sintetizzabile, bisogna procedere da capo a progettare l'unità operativa.

3 Progetto di un sistema complesso: prodotto scalare

Per poter chiarificare i concetti precedentemente esposti abbiamo deciso di mostrare un esempio pratico dell'attuazione del nostro approccio ingegneristico alla risoluzione di un problema. Nella fattispecie ci siamo concentrati in uno degli esercizi a scelta proposti in aula, ovvero il prodotto scalare tra due vettori di numeri interi.

Affronteremo dunque problematiche tipiche alla progettazione e allo sviluppo di una macchina digitale e cercheremo di fornire una falsariga sul come approcciare alla realizzazione delle suddette macchine.

3.1 Analisi delle specifiche e decomposizione funzionale

La prima cosa da fare, come già detto in precedenza, è elaborare le specifiche relative ad un progetto. Tale passo preliminare è fondamentale in qualsiasi progetto ingegneristico, sia esso in ambito software, che hardware. Nello specifico del nostro prodotto scalare, ipotizziamo di avere una traccia di questo tipo:

"Si vuole realizzare una macchina che possa svolgere un prodotto scalare tra due vettori di numeri interi di dimensione M, con ogni numero rappresentato su un totale di N bit."

Dovendo scegliere i parametri M ed N, abbiamo stabilito arbitrariamente di fare un prodotto scalare tra 2 vettori di 3 elementi, ognuno dei quali rappresentabili su un totale di 8 bit. La scelta del byte come intervallo di rappresentazione è stata dettata dalla necessità di riutilizzare un componente da noi precedentemente progettato: il moltiplicatore di Booth per operandi a 8 bit. Il prodotto scalare, da un punto di vista matematico, può essere implementato come segue:

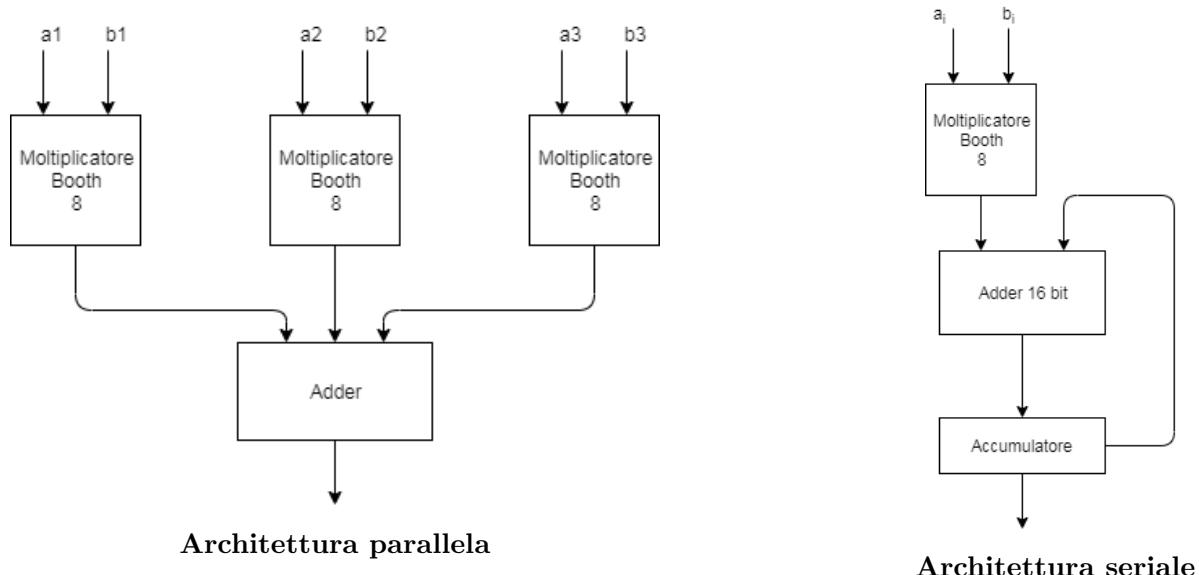
$$A \cdot B = |A| \cdot |B| \cos(a)$$

Naturalmente una implementazione del genere avrebbe richiesto l'identificazione della norma dei singoli vettori, e successivamente il prodotto per una sinusoida, dovendo dunque conoscere anche l'angolo compreso tra i due. Ci siamo

allora affidati ad una seconda definizione, decisamente più attuabile:

$$A \cdot B = \sum_{i=1}^M a_i \cdot b_i$$

Posto questo come punto di partenza, la nostra macchina dovrà essere in grado di realizzare M addizioni ed M prodotti a N bit. Abbiamo quindi ipotizzato due architetture di concetto preliminari, una di tipo parallelo, che aveva come downside principale l'incremento di area, e l'utilizzo di più componenti, ma con performance migliori, e una struttura di tipo seriale, con ottimizzazione in termini di area e componenti, ma meno performante in termini di velocità: Abbiamo infine scelto la strada "seriale", in maniera tale



da poter riutilizzare tutti componenti che già possedevamo, e poter adottare un modello architettonico simile ad uno già visto durante il corso (operazione modulo M).

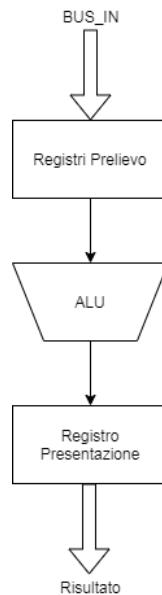
È bene notare come l'adesione ad un modello precedentemente utilizzato sia una base fondamentale dalla quale partire. In prima battuta è infatti cruciale stabilire una struttura che possa essere funzionale, a partire dalla quale poter implementare eventuali migliorie.

Abbiamo così stabilito quali debbano essere le funzionalità della nostra macchina, e quali componenti potremmo dover andare ad utilizzare. Volendo

entrare più nel dettaglio, possiamo provare ad abbozzare una prima architettura per la nostra macchina, componendola di:

1. Registri di prelievo dati;
2. ALU;
3. Registro di presentazione dati.

L'idea dalla quale passeremo a sviluppare il nostro progetto prevede di avere dei registri per prelevare gli M vettori di N bit. Le informazioni contenute in questi registri saranno poi elaborate da una ALU; tale termine viene utilizzato in maniera impropria, in quanto realizzerà unicamente l'operazione di moltiplicazione e somma, tuttavia per poter dare al nostro schema una struttura quanto più generica possibile, abbiamo optato per questa nomenclatura. Vedremo di seguito l'implementazione dell'unità operativa, e nella fattispecie, l'implementazione per i registri di prelievo e dell'ALU.



Architettura concettuale dell'unità operativa

Alla fine dell'operazione siamo interessati a mantenere il risultato appena ottenuto in un registro, attualmente alla presentazione.

3.2 Progettazione della unità operativa

Abbiamo identificato a questo punto i componenti di cui avremo bisogno. In questa sezione andremo dunque ad esplodere tali componenti e a mostrare come, a partire da macchine notevoli sequenziali e non, siamo stati in grado di implementarli.

Per mantenere una certa semplicità espositiva non entreremo nel dettaglio del codice VHDL descrittivo dei vari componenti, ma lo allegheremo esternamente alla documentazione.

3.2.1 Unità di prelievo e di presentazione

Prima problematica fondamentale era quella relativa al prelievo dei valori di ingresso. Infatti prevedendo una architettura seriale, e non parallela, si rendeva necessaria l'introduzione di un elemento di memorizzazione che potesse "salvare" le i -esime componenti dei vettori. Abbiamo supposto di poter prelevare coppie omologhe di tali componenti, appartenenti ai vettori X e Y in ingressi.

Tale prelievo verrà effettuato un numero totale di M volte, che nel nostro caso è 3. questo ci porterà ad avere 3 fasi di inserimento, successivamente le quali potrà seguire il comportamento della macchina.

Ragionevolmente, dato che ogni intero sarà registrato su N bit, avremo bisogno di M registri di N bit. E allora avremo una struttura di memorizzazione di dimensione 8×3 , ovvero 24 celle di memorizzazione.

Abbiamo quindi deciso di utilizzare 3 registri a caricamento parallelo su 8 bit, abilitati su fronte di salita, e li abbiamo combinati strutturalmente tra loro per creare una struttura più complessa denominata Vector. Così definito un Vector è allora una matrice $N \times M$, in grado di mantenere informazioni relative solo ad un vettore in ingresso. Dovendo noi gestire 2 vettori per poterli moltiplicare tra loro, l'unità di prelievo prevederà 2 registri Vector.

Al fine di comprendere meglio ciò che è stato fatto, vale la pena soffermarsi un attimo in più su questa struttura appena presentata. In primo luogo è possibile notare come l'impostazione per questo "registro" sia molto simile a quella di una memoria.

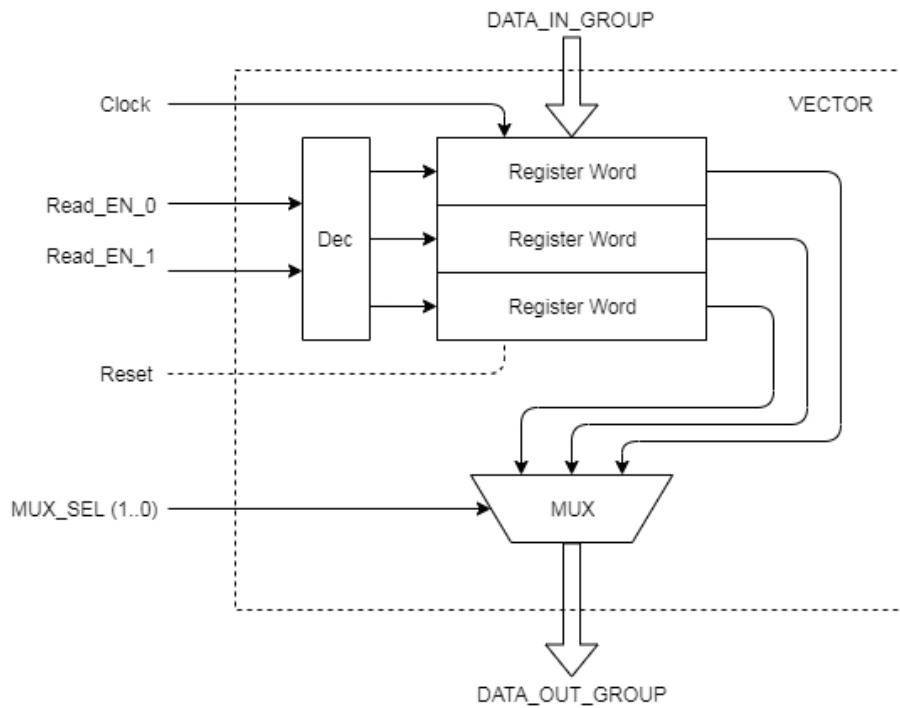
Ogni registro può essere infatti visto come una parola di 1 byte di una

memoria, composta, nel nostro caso, da 3 parole. è allora necessario poter indirizzare la memoria, in maniera tale da selezionare correttamente una parola in lettura/scrittura.

Abbiamo allora previsto, come interfaccia per il Vector, un segnale di abilitazione alla lettura `READ_EN`, che passando in un decodificatore, selezionerà la giusta parola.

Inoltre, data sempre l'architettura seriale, abbiamo dovuto prevedere in uscita un multiplexer, che potesse selezionare, al variare di un apposito segnale di ingresso `MUX_SEL`, realizzato su $\lceil \log_2 M \rceil$ bit, la corretta word da passare in uscita verso l'ALU.

Possiamo di seguito presentare l'architettura del Vector:



3.2.2 Unità di elaborazione : ALU

Una volta letti i dati in ingresso, si passa alla fase elaborativa, durante la quale un segnale di selezione sceglierà opportunamente il dato da mandare

dal vector verso l'ALU. Vedremo a breve come questo segnale di selezione viene effettivamente generato.

In primo luogo la nostra unità logico-aritmetica dovrà essere in grado di realizzare un prodotto ed una moltiplicazione. A tal fine abbiamo deciso di riutilizzare due architetture già viste in precedenza durante il corso : il moltiplicatore di Booth e l'addizionatore a propagazione ad onda a 16 bit.

Ciò non è naturalmente sufficiente, in quanto, effettuata la prima moltiplicazione e addizione, dovremo ancora effettuarne altre $M-1$. L'idea è quindi quella di mantenere il risultato ad ogni iterazione, ed incrementarlo del valore della nuova moltiplicazione.

Da qui nasce l'idea di utilizzare un registro di accumulazione, che possa mantenere il dato relativo alla precedente operazione di somma ad ogni iterazione. Tempificato opportunamente tale registro, è possibile avere garanzia della consistenza dei dati che esso mantiene. Dopo un numero totale di M iterazioni, il nostro registro conterrà il valore finale dell'operazione, che potrà essere inoltrato al registro di presentazione.

Teniamo presente che il moltiplicatore che abbiamo utilizzato, quello di Booth, darà in uscita un valore di **STOP**, simbolo della fine dell'operazione. Questo ci porta ovviamente a dover introdurre una problematica classica nello sviluppo di sistemi digitali, ovvero i vincoli derivati dall'utilizzo di una macchina componente.

Naturalmente, terminata una operazione aritmetica, il nostro accumulatore dovrà essere abilitato, e per far ciò utilizziamo sia il segnale di stop del moltiplicatore, che un segnale esterno di abilitazione **AB_ACC** : in questa maniera siamo sicuri di prelevare il dato consistente nel momento giusto. (NB: Il clock deve essere dimensionato in maniera tale che alla fine di ogni iterazione anche l'addizionatore abbia avuto tempo di commutare).

Il risultato del registro accumulatore viene poi prelevato e mandato come secondo addendo dell'addizionatore.

Abbiamo tuttavia la necessità di tener traccia anche del numero di moltiplicazioni che sono state realizzate. Sappiamo, che dovremo realizzare un numero pari ad M operazioni, ragion per cui avremo bisogno di un contatore modulo M . Questo contatore ha una duplice funzione:

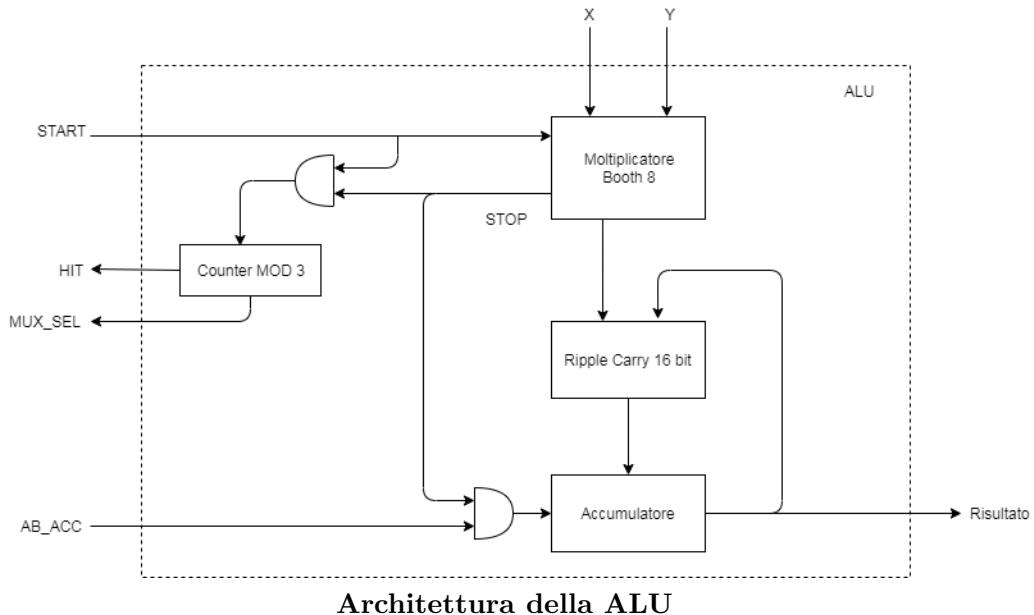
1. Notificare il sistema del completamento dell'operazione, in maniera tale

da farlo tornare in una fase di stasi;

2. Abilitare la selezione del corretto valore in ingresso all'ALU.

Infatti il valore del conteggio viene prelevato istante per istante, e viene usato come segnale di abilitazione per i vector. In questa maniera prima che inizi l'operazione, il contatore avrà valore "00", e quindi non verrà selezionato nulla. Iniziata l'operazione, dopo un colpo di clock, il nostro contatore registrerà il valore "01", e selezionerà di conseguenza il primo registro dal vector. e così via fino a completamento dell'operazione.

Il segnale di conteggio di questo contatore sarà una AND tra il segnale di start che darà il via all'operazione, e il segnale di STOP. In questa maniera, fintanto che il moltiplicatore è in funzione, e all'atto della conclusione della i -esima moltiplicazione, incrementerò il contatore, e preleverò l'ingresso $i+1$ -esimo.



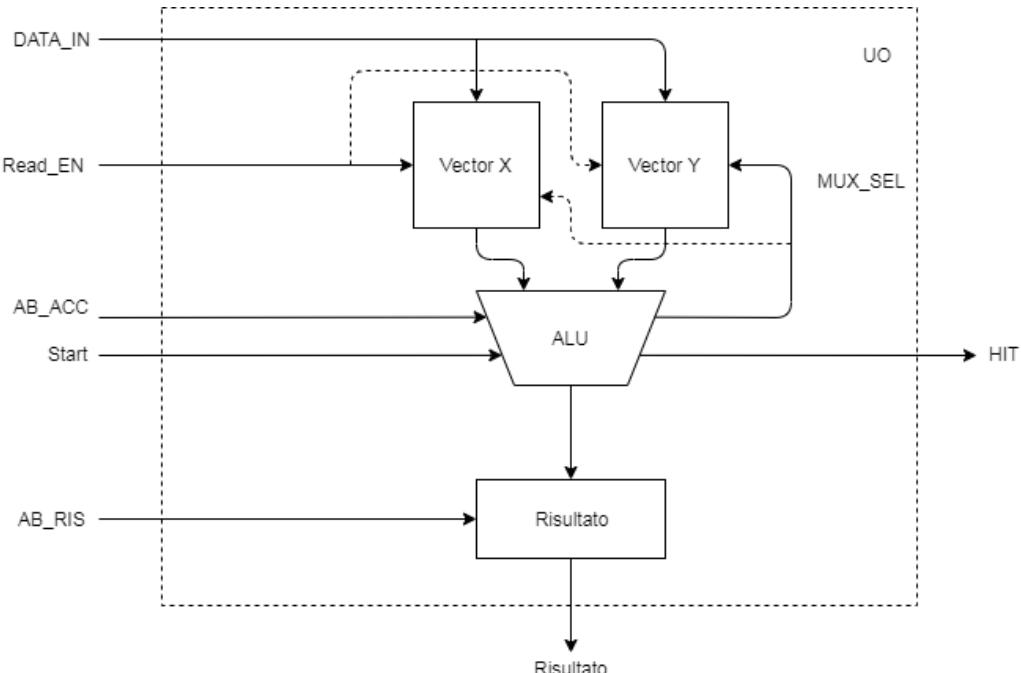
3.2.3 Presentazione dell'unità operativa

A questo punto non ci resta altro da fare che collegare i componenti appena descritti per realizzare l'unità operativa. A tal proposito dobbiamo porre in

uscita la nostra unità di presentazione, ovvero un registro atto a contenere il risultato. Questo registro non sarà differente dall'accumulatore o dai registri utilizzati per l'implementazione del vector, e verrà abilitato da un segnale di AB_RIS.

Quando l'ALU termina la moltiplicazione, il registro risultato viene abilitato al fine di prelevare il dato dall'ALU, per poi essere mandato in uscita dall'unità operativa.

NB: per semplicità omettiamo Clock e Reset dagli schemi grafici, nonostante questi siano presenti e fondamentali per il corretto funzionamento della macchina.



Architettura dell'unità operativa

3.3 Progettazione della unità di controllo

Per quanto ne concerne l'unità di controllo, avevamo la necessità di controllare diversi segnali sia per la gestione dei registri Vector, che per quella dell'ALU.

Ogni volta che si va a realizzare una unità di controllo, bisogna tenere ben presente cosa stiamo andando a controllare, ovvero di cosa ha effettivamente bisogno l'unità operativa. Come già detto nel paragrafo 2, abbiamo due possibili scelte, una cablata e una microprogrammata. Abbiamo deciso di partire da una logica cablata, rappresentabile mediante un grafo degli stati e realizzabile tramite un automa a stati finiti. In secondo luogo presenteremo come sia possibile tradurre tale struttura in una microprogrammata.

è buona prassi ipotizzare come primo stadio, quello di **IDLE**, durante il quale la macchina è in quiescenza, fino all'innescarsi di un determinato evento. Questo evento può essere di qualsiasi tipo: input esterni, come bottoni, oppure input provenienti da altre macchine.

Nel nostro caso specifico abbiamo deciso di utilizzare un **bottone**.

Poichè avevamo necessità di pulire tutti i registri prima di prelevare nuovi ingressi e dunque procedere con le operazioni successive, abbiamo scelto come primo stato della macchina, a seguito dell'**IDLE**, uno stato di reset. In questo stato abilitiamo dunque il segnale di reset, e all'atto dell'asserramento del segnale di **READ_IN**, passiamo allo stato di inizializzazione.

Gli stati d'ingresso saranno 3; preleveremo infatti ad ogni segnale di read i-esimo, il valore i-esimo di entrambi i vettori. Ci siamo rifatti anche in questo caso alla struttura dell'automa del modulo M vista durante il corso. Infatti dati N ingressi sul nostro bus, abbiamo realizzato N stati sequenziali.

I tre stati sono molto simili, e non fanno altro che cambiare il segnale di selezione **READ_OUT**, che ricordiamo, era necessario a selezionare la word nel vector da abilitare in caricamento. A seguito dell'ultimo segnale di **READ**, dallo stato **IN3**, passiamo allo stato operativo.

Nello stato **OP** viene abilitato il moltiplicatore nell'ALU mediante il segnale di **START**, e il registro accumulatore tramite un segnale di **AB_ACC**. Infine, non appena il segnale di **FINE** viene asserito, viene abilitato il registro risultato, e si ritorna nello stato di idle; l'operazione è terminata.

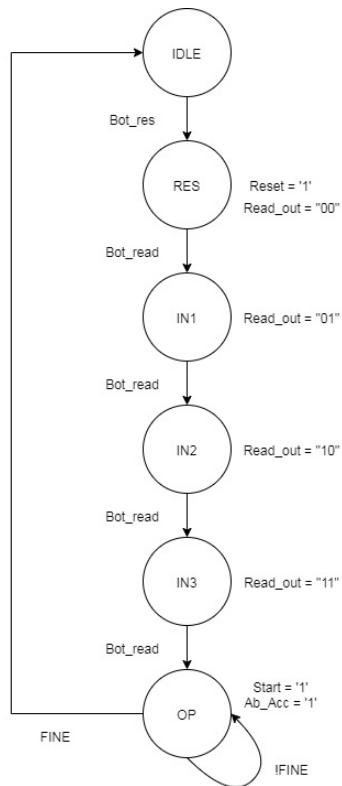
Il sistema riesce tramite questo sistema ad andare in convergenza, e a predisporci, in caso, ad una nuova operazione a partire dallo stato di idle.

È bene notare che arrivati a questo punto della progettazione sia indispensabile realizzare un testing completo sia per quanto ne concerne l'unità operativa che l'unità di controllo. Un consiglio che possiamo dare, come detto anche in precedenza, è quello di realizzare testbench ad hoc per entrambe le

unità, e se funzionanti, collegarle tra loro, completando il progetto.

È chiaro, che affinchè il testing possa essere facilitato, dobbiamo esser certi che i componenti utilizzati a monte della progettazione siano a loro volta funzionanti. Ciò può esser garantito dall'utilizzo di componenti di libreria, oppure da elementi già precedentemente costruiti e testati da voi o da altri. Ed è così che la modularità di un progetto ingegneristico e il rigore nel realizzarlo vanno a semplificare e velocizzare l'intero processo di produzione.

Notiamo come ad ogni stato debba corrispondere una variazione dei segnali di controllo che la suddetta unità vada a generare. È infatti proprio lo stato dell'automa, che evole in base ai segnali iniettati dall'unità operativa o dall'esterno, a determinare il controllo per il corretto funzionamento dell'intera macchina.



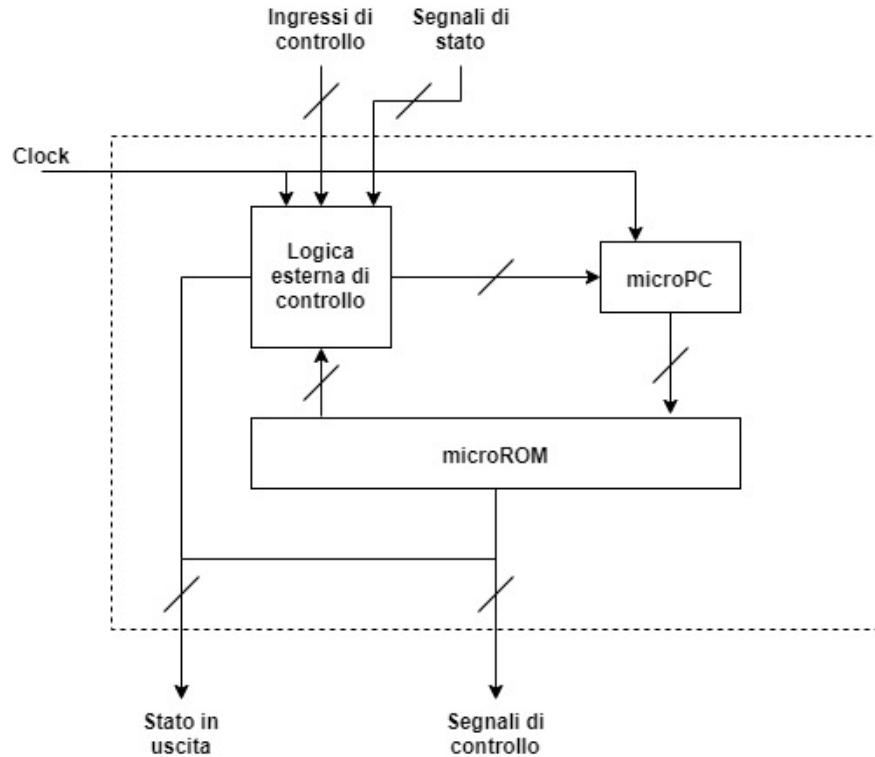
Automa derivato dalla progettazione

I segnali che verranno dunque generati dall'unità di controllo saranno:

- START: segnale per dare il via all'operazione;
- RESET: segnale per resettare i componenti dell'architettura;
- ACC: segnale di abilitazione per il registro dell'ALU "accumulatore";
- RES: segnale di abilitazione per il registro di presentazione;
- READ-OUT : segnale di abilitazione alla lettura dei registri (2 bit).

3.3.1 Traduzione in logica microprogrammata

Per concludere la trattazione di questo capitolo, vogliamo presentare la traduzione da una logica cablata ad una microprogrammata. Ai fini di una trattazione più chiara, riproponiamo il modello generale da noi realizzato, già presentato nel primo capitolo: Tale struttura prevede di avere una microROM, collegata



ad una logica di controllo esterna. Tale logica, in linea di principio minimale,

deve contenere eventuali gestioni di segnali non solo dall'esterno, ma anche da parte della microROM. infatti, la microROM deve poter implementare una logica eventuale di salto condizionato, che gli permetta di accedere ad un indirizzo differente da quello previsto all'interno della sua parola.

Entriamo dunque nel merito della ROM stessa. Una ROM è un tipo di memoria a sola lettura, strutturata come un'insieme di WORD. ogni WORD avrà una dimensione che definirà il parallelismo dell'architettura di memoria, mentre il numero di WORD mi identificherà l'indirizzamento della nostra ROM.

Dunque le parole della memoria verranno inizializzate a dei valori di default, che verranno prelevati in uscita. Tali valori corrispondono di fatto ai segnali di abilitazione che verranno inoltrati all'unità operativa. Una WORD tuttavia si deve anche comporre di un campo `Next_Address`, o se vogliamo, lo stato successivo nella nostra macchina. Tale indirizzo verrà inoltrato alla logica esterna di controllo, che a sua volta lo reindirizzerà verso il program counter.

Il registro PC ha come fine quello di selezionare la WORD corretta in base all'esigenza. Naturalmente, se il segnale next-address passa inalterato in PC, allora la microROM leggerà come istruzione successiva quella effettivamente contenuta all'interno della parola precedentemente letta.

Può però capitare che la logica di controllo alteri il segnale di next-address, realizzando di fatto un salto condizionato. Notiamo inoltre che in casi più complessi sarebbe potuto esser necessario specificare anche un codice operativo come entry della ROM, ma avendo nel nostro scenario solo una istruzione possibile, ovvero il prodotto scalare, ci siamo limitati ad utilizzare una word che preveda solo un record per i dati e uno per l'indirizzo successivo.

Fatto questo preambolo, siamo partiti dal definire la struttura della nostra ROM a partire dall'automa in logica cablata che avevamo precedentemente realizzato.

Nella struttura di controllo cablata avevamo realizzato 6 stati, ognuno dei quali gestiva in uscita un totale di 4 segnali da 1 bit ciascuno, e un segnale da 2 bit, relativo al vector da abilitare in lettura. Possiamo quindi iniziare a stabilire la struttura della nostra generica WORD.

Indubbiamente la nostra WORD dovrà prevedere:

- 1 bit per il segnale di start per l'ALU;
- 1 bit per il segnale di reset della UO;
- 1 bit per l'abilitazione del registro accumulatore;
- 1 bit per l'abilitazione del registro di presentazione del risultato;
- 2 bit per la selezione del vector in lettura.

Possiamo inoltre realizzare una associazione uno ad uno tra gli stati del nostro automa originale, e l'indirizzabilità della nostra microROM. A 6 stati corrisponderanno infatti 6 parole, codificabili su un totale di 3 bit. Ma allora, possiamo aggiungere il campo PC-next che indicherà il prossimo indirizzo al quale riferirci, a partire dalla parola selezionata.

La struttura finale per una generica WORD sarà del tipo: A questo punto,

PC- Next	Start	Reset	Acc	Res	READ-OUT
----------	-------	-------	-----	-----	----------

Parola della microROM

fissata la struttura della parola, e il numero di `control_word` necessarie alla costruzione della nostra ROM, dobbiamo procedere con la programmazione della stessa. Programmare la microROM significa semplicemente andare a fissare, per ogni stato/parola, il valore dei segnali di uscita e del `PC_next`.

Riferendoci alla struttura dell'automa vista nella sezione relativa all'unità di controllo, presentiamo lo schema grafico per la nostra implementazione: In termini di linguaggio descrittivo, abbiamo deciso di definire 6 costanti di

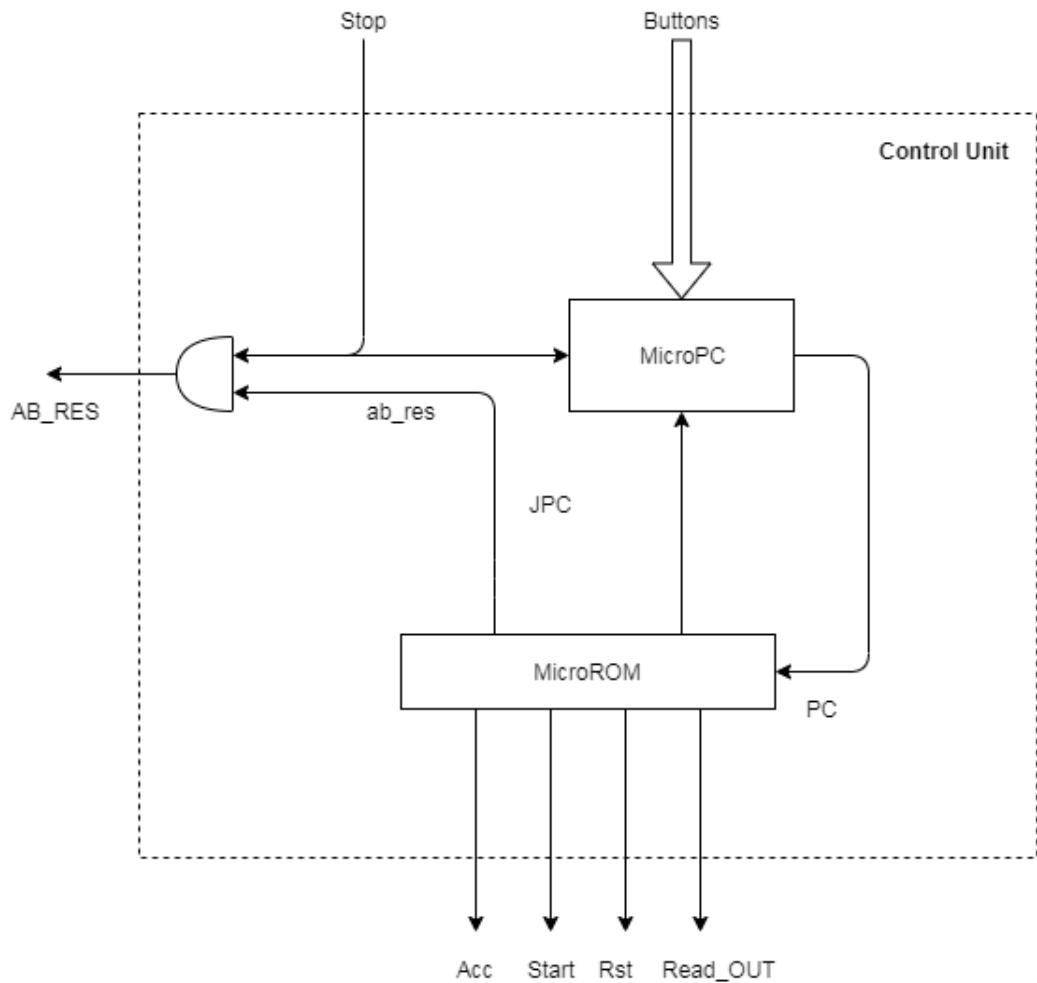
	PC- Next		Start	Reset	Acc	Res	READ-OUT	
IDLE	0	0	0	0	0	0	0	0
Reset	0	0	1	0	1	0	0	0
IN1	0	1	0	0	0	0	0	1
IN2	0	1	1	0	0	0	0	0
IN3	1	0	0	0	0	0	1	1
OP	1	0	1	1	0	1	1	0

microROM

tipo `control_word`, con le quali andremo poi ad inizializzare la struttura, definita come array di `control_word`, una volta istanziata.

Bisogna ora capire come procedere con lo sviluppo dell'unità di controllo. Presa come riferimento la WORD definita in precedenza, dobbiamo scegliere quali saranno i segnali generati dalla ROM ad inviare direttamente all'unità operativa, e quali invece dovranno essere gestiti da una unità interna.

Sicuramente il segnale `PC_next` rimarrà interno alla struttura, e verrà iniettato all'interno di una logica di "gestione salti", che genererà l'indirizzo corretto per il microPC. Come detto in precedenza, la rom eseguirà un "fetch" al colpo di clock successivo, andando a selezionare la corretta parola da eseguire.



Logica microprogrammata derivata dalla progettazione

Internamente al codice abbiamo deciso di inglobare la logica di controllo interna e il PC in un unico process/registro. Questo process realizza un registro abilitato sul fronte di salita del clock, che valuterà i due ingressi, ovvero i bottoni e il JPC in ingresso. Tale JPC altro non è che il Next address mandato in uscita dalla microROM, relativo alla parola attualmente selezionata.

Infatti, seguendo lo schema dell'automa di prima, se siamo nello stato di idle, corrispondente alla parola di indirizzo "000", vi rimarremo fintanto che il pulsante di reset non sarà premuto. In maniera analoga è ripetuto il ragionamento per le altre parole. Presentiamo, per chiarezza espositiva, una porzione di codice:

```

if (CLOCK' event and CLOCK = '1') THEN
    IF (RESET_IN = '1' AND jpc_next_int = "000") THEN
        pc_next_int <= "001";
    ELSIF (READ_IN_1 = '1' AND jpc_next_int = "001") THEN
        pc_next_int <= "010";
    ELSIF (READ_IN_2 = '1' AND jpc_next_int = "010") THEN
        pc_next_int <= "011";
    ELSIF (READ_IN_3 = '1' AND jpc_next_int = "011") THEN
        pc_next_int <= "100";
    ELSIF (READ_IN_2 = '1' AND jpc_next_int = "100") THEN
        pc_next_int <= "101";
    ELSIF (FINITO = '1' AND jpc_next_int = "101") THEN
        pc_next_int <= "000";
    ELSE
        pc_next_int <= jpc_next_int;
    END IF;
END IF;

```

Possiamo notare che questi else if vanno a realizzare un tipo di salto condizionato, mentre è solo la condizione di "default", rappresentata dal solo else, ad indicare l'assenza di salto. Infatti in questo caso, verrà direttamente inviato come prossimo indirizzo quello specificato dalla word.

NB: nel nostro particolare caso, ogni campo **next_address** punta a se stesso (esempio: **IN3** punta all'indirizzo "100", che è lo stesso indirizzo che mi permette di accedere ad **IN3**). Ciò può essere fuorviante ed è dovuto al fatto che la nostra macchina evolve solo in relazione a degli eventi, e che dunque possono esser gestiti solo da una logica esterna. Nel caso avessimo avuto uno stato dal quale transitare a seguito di un colpo di clock, allora il campo next address relativo a quella parola avrebbe riportato l'indirizzo della parola relativa allo stato successivo.

Possiamo vedere che mentre i segnali di **Read_out**, **Start**, **AB_ACC** e **reset** possono essere inviati all'unità operativa senza ulteriori controlli, non possi-

amo dire lo stesso del segnale AB_RES. tale segnale infatti sarà sempre alto entrati nello stato di OP, o alternativamente la WORD "101". Naturalmente per non presentare un risultato inconsistente dobbiamo aspettare che l'operazione termini, ovvero che il segnale di stop arrivi. Una semplice AND tra STOP e AB_RES mi permetterà di abilitare correttamente la lettura del risultato. Inoltre notiamo come il segnale di stop entri anche nella struttura del microPC, poichè determinante per calcolare l'indirizzo "000" a partire da quello "101" al termine dell'operazione.

Appendice C

Flip Flop Master-Slave ideale

(*Marco Barletta, Riccardo Corvi, Andrea Marchetta, Giuseppe Ruggiero*)

Flip Flop Master-Slave Ideale

A cura di:

Marco Barletta
Riccardo Corvi
Andrea Marchetta
Giuseppe Ruggiero

21 ottobre 2019

1 Master-Slave

In questi appunti ci proponiamo di descrivere il funzionamento del flip flop Master-Slave (asincrono in termini di automa, sincrono in termini di ingressi). La caratteristica del M-S ideale è quella di campionare il valore dell'ingresso D sul fronte di salita del clock e di presentarlo in uscita sul fronte di discesa, trattando il segnale di clock come se fosse un impulso ideale. Questo permette di superare i problemi legati al tempo di hold, tipici dei flip-flop edge-triggered (i.e. quando campiono sul fronte di discesa devo essere sicuro di mantenere in ingresso il segnale giusto fino alla discesa del segnale di clock, oppure, quando campiono sul fronte di salita, dato che commuta con l'abilitazione ancora alta potrei avere delle inconsistenze a valle). Per realizzare un flip-flop con questo comportamento bisognerebbe crearne l'automa a stati finiti, minimizzare gli stati, realizzare le mappe di Karnaugh e sintetizzare quindi la macchina.

Ci sono più modi per approssimare il master-slave ideale: negli appunti del prof. Mazzeo e della prof.ssa Sami si fa uso di due latch in cascata. Il comportamento complessivo non è quello ideale poiché il flip-flop master è trasparente per tutto il periodo in cui l'abilitazione (clock) è alta. Tuttavia, è un'approssimazione molto semplice da realizzare (in figura è mostrato una implementazione con latch di tipo RS).

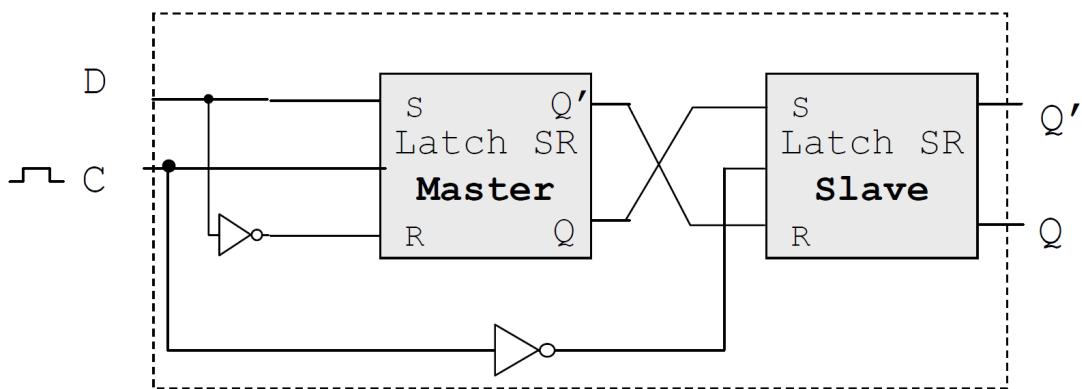


Figura 1: Master-Slave con due latch RS in cascata

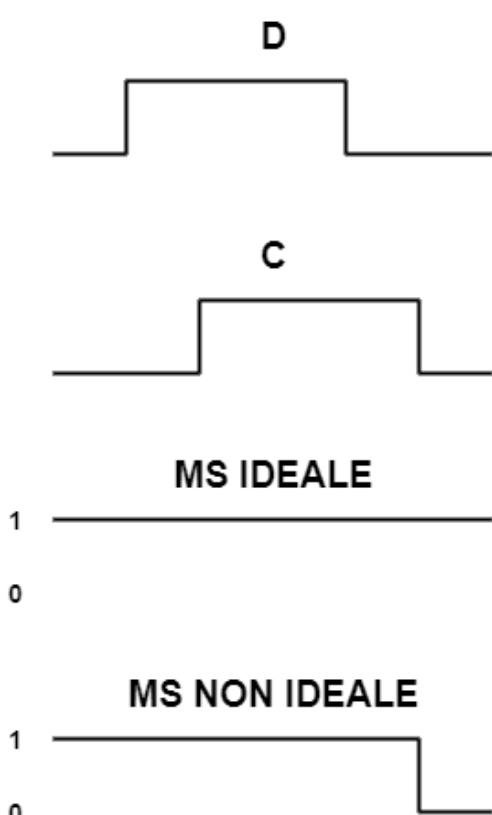


Figura 2: Esempio di temporificazione

Tale macchina, come accennato in precedenza, ha una relazione ingresso-stato a livello, proprio perché il latch è trasparente per tutto il periodo di abilitazione del clock. La differenza con la macchina ideale si può notare in un esempio che mostriamo nella figura a lato. Nell'immagine sono stati tracciati il valore del dato D e del segnale di controllo C nel tempo con il corrispettivo valore di uscita dei MS ideali e non, ci accorgiamo che le uscite sono diverse. Supponendo di partire con uscita pari ad 1, il MS ideale, che campiona il dato sul fronte di salita e lo presenta sul fronte di discesa, manterrà l'uscita ad 1 in quanto, sul fronte di salita del clock, D era pari ad uno. Diversamente, nel caso non ideale, il latch master inseguirà l'ingresso D per tutta la durata del clock, motivo per cui, al momento della sua abilitazione, il latch slave vedrà in ingresso 0 che riporterà in uscita.

Una approssimazione più complessa ma più corretta del flip flop master slave si ottiene ponendo in cascata due flip flop edge-triggered che lavorano su fronti opposti. Il primo campiona sul fronte di salita, il secondo sul fronte di discesa. Il comportamento risultante è quello del MS ideale, ciononostante la macchina non ne è una realizzazione esatta, in quanto un flip-flop edge-triggered D può avere 4 stati, per cui una cascata di due flip flop D può complessivamente avere 4×4 stati (di cui molti saranno irraggiungibili), mentre la macchina ideale ne ha solo 6.

Nell'immagine si vede una possibile realizzazione di un flip flop master slave a mezzo di due flip flop D, il primo rising edge-triggered ed il secondo falling edge-triggered.

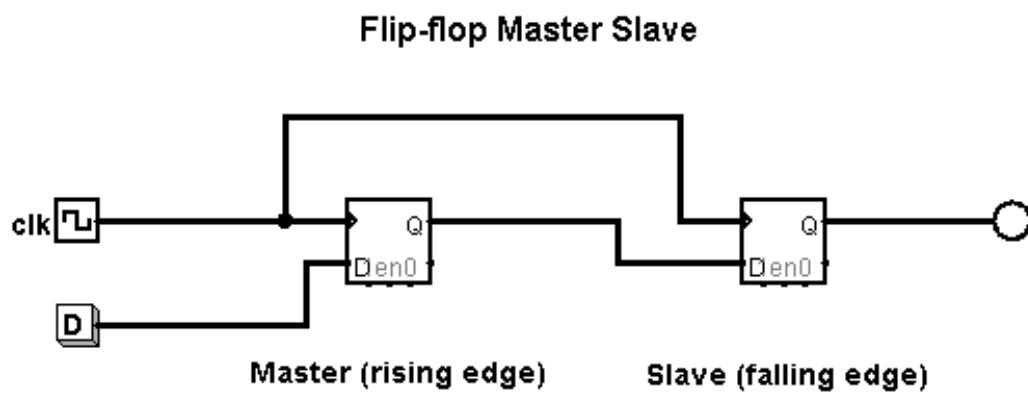


Figura 3: Schema a blocchi di un Master-Slave a comportamento ideale.

Per capirne meglio il funzionamento però esplodiamo la rappresentazione dei flip flop fino al livello di porta logica ottenendo la figura 5. Evidenziamo che la struttura, per campionare sul fronte di salita e presentare sul fronte di discesa del segnale di abilitazione, deve essere realizzata con un flip-flop edge triggered a porte NAND per il master e uno a porte NOR per lo slave (tali flip-flop sono ideali realizzati come descritti dal prof. Mazzeo a partire da pag. 25, e pertanto sono composti da 6 porte logiche). Il funzionamento è il seguente: quando il clock si alza, il master campiona il dato D in ingresso e lo presenta in uscita dopo un certo ritardo δ . Il flip flop slave riceve in ingresso il dato campionato dal master ma, avendo il segnale di clock alto, non campiona ancora. Essendo il flip-flop slave costruito con porte NOR eventuali

Flip-Flop Master-Slave Ideale

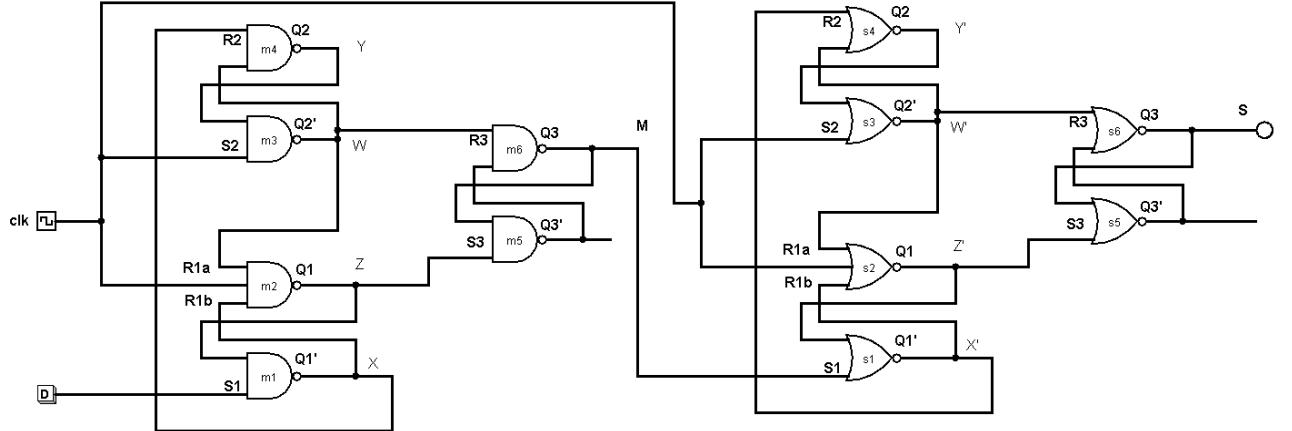


Figura 4: Diagramma delle porte logiche.

variazioni vengono mostrate solo sui fronti di discesa. Infine, quando il clock si abbassa lo slave campiona l'ingresso inviato dal master e presenta il segnale in uscita. In conclusione, mostriamo l'automa a stati finiti del flip-flop master-slave ideale, i cui stati sono da intendersi come di seguito: gli stati **q0** e **q1** indicano gli stati in cui presenta rispettivamente 0 e 1; gli stati **q00** e **q11** sono gli stati in cui, all'alzarsi del segnale di abilitazione, campiona lo stesso dato che sta già presentando; gli stati **q01** e **q10** sono gli stati in cui è stato campionato un valore diverso da quello che sta correntemente presentando. Tuttavia, l'uscita è rimasta invariata in quanto, non avendo ancora commutato il clock, il nuovo valore non è ancora stato presentato.

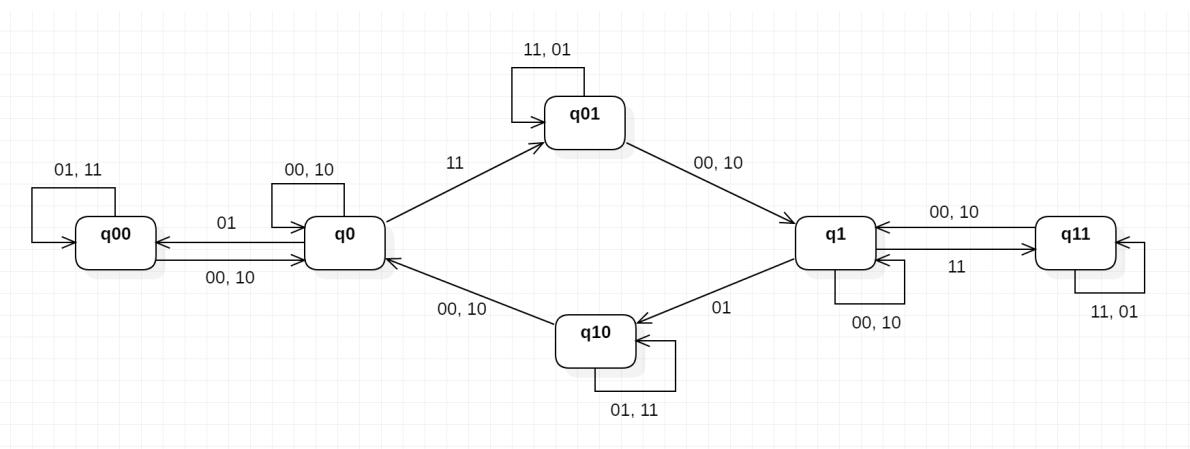


Figura 5: Automa a stati finiti del Master-Slave ideale.

Appendice D

Overview dei dispositivi logici programmabili

(Di Pace Vincenzo, Esposito Ciro, Fusiello Fabiano)

PLD

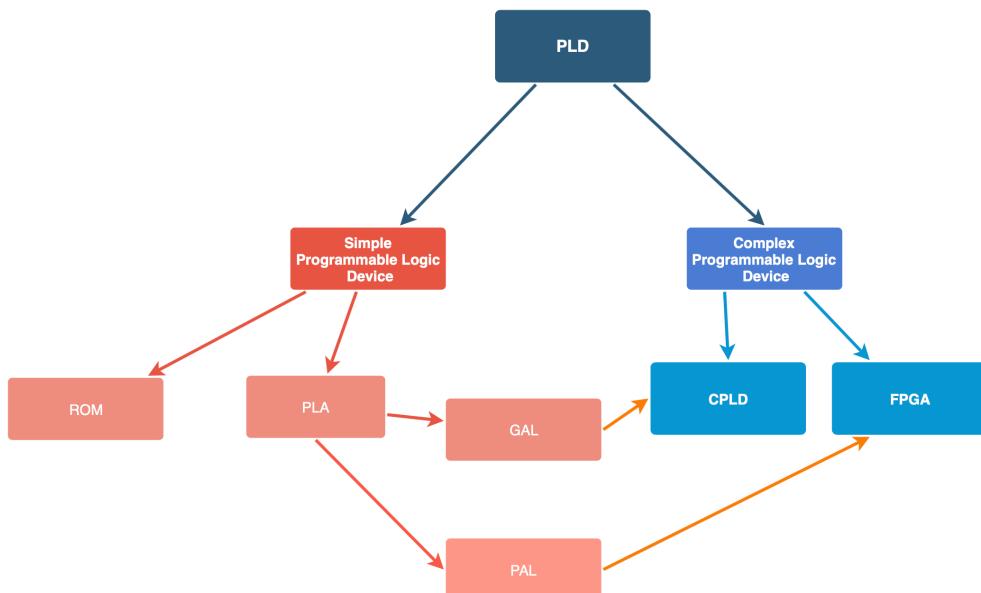
I dispositivi digitali giocano un ruolo chiave nel progetto dei dispositivi digitali. Essi sono chips general purpose che possono essere configurati per un'ampia gamma di applicazioni. Il primo tipo di dispositivo programmabile ad essere stato largamente utilizzato è stata la PROM (Programmable Read Only Memory). Un circuito logico può essere implementato utilizzando le linee di indirizzamento come ingressi del circuito, e le uscite sono definite dai bit immagazzinati. Con questa strategia può essere implementata qualunque tabella delle verità. Sono disponibili due versioni delle PROM, quelle che possono essere programmate solo dal produttore, e quelle che possono essere programmate dall'utente finale. Il primo tipo è chiamato mask-programmable mentre il secondo è chiamato field-programmable. Questi ultimi offrono diversi vantaggi:

- i dispositivi field-programmable sono meno costosi a bassi volumi di produzione rispetto a quelli mask-programmable perché sono componenti standard.
- i dispositivi field programmable possono essere programmati immediatamente, in pochi secondi, mentre i dispositivi mask-programmable devono essere fabbricati dalla fonderia.

Sebbene le PROM sono una valida alternativa per realizzare semplici circuiti logici, è chiaro che la struttura di una PROM è più adatta all'implementazione di memorie. Un altro tipo di dispositivo programmabile progettato specificatamente per l'implementazione di circuiti logici è chiamato PLD (Programmable Logic Device).

Un dispositivo logico programmabile (in lingua inglese Programmable Logic Device, abbreviato spesso in PLD), nell'elettronica digitale, è un circuito integrato programmabile largamente utilizzato nei circuiti digitali. A differenza di una porta logica, che implementa una funzione logica predefinita e non modificabile, un PLD, al momento della fabbricazione, non è configurato per svolgere una determinata funzione logica. Prima di poter utilizzare un PLD in un circuito, è necessario programmarlo.

I PLD sono generalmente divisi in due categorie: i Simple Programmable Logic Device e i Complex Programmable Logic Device, a seconda che abbiano un numero di pin rispettivamente maggiore o minore di 48. Per i dispositivi più complessi, tuttavia, si usa il termine Field programmable gate array. Infine, seguendo l'evoluzione storica dell'elettronica digitale, i dispositivi elettronici digitali riprogrammabili più evoluti sono i microprocessori dei comuni computer general purpose.



Un dispositivo logico programmabile (PLD) è un componente elettronico usato per costruire circuiti digitali riconfigurabili. Diversamente dalla porta logica, che lavora secondo una funzione fissa e non alterabile, un PLD non ha una funzione definita in fonderia. Prima di essere utilizzato il PLD ha bisogno di una fase di programmazione, cioè di configurazione.

Il vantaggio risiede in tre aspetti cruciali:

- Il costo di fonderia è abbattuto: avviare la produzione di una nuova serie di circuiti integrati ha un costo di setup di 106~7\$; i PLD hanno un costo di pochi \$;
- Il dispositivo può essere configurato per prototipazione;
- La funzione logica implementata può cambiare se le specifiche mutano (così come avviene nel software).

Una PLD è formata tipicamente da un array di porte AND connesse ad un array di porte OR. Per essere implementato in una PLD un circuito logico deve essere rappresentato nella forma di una somma di prodotti.

ROM

Le ROM sono memorie universali che possono implementare funzioni multi uscita mappando direttamente tabelle di verità.

- E.g. ROM con m input (linee indirizzo) ed n output (linee dato). La memoria contiene m parole, ciascuna di n bit. Teoricamente sono disponibili 2^m possibili funzioni booleane. La struttura della ROM permette però di definirne massimo n.

Non è richiesta minimizzazione: la ROM contiene l'intera tabella di verità!

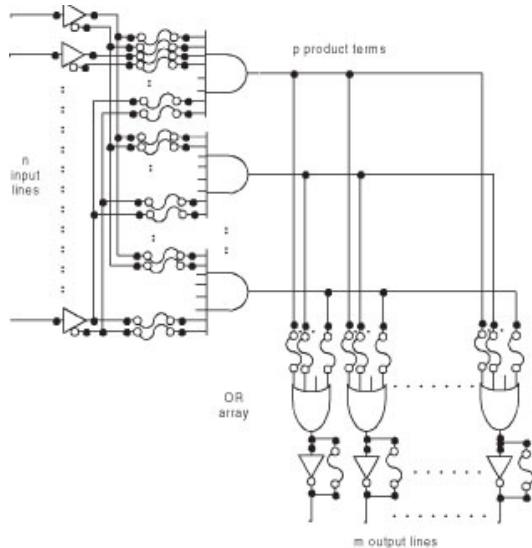
Gli ingressi delle funzioni booleane sono gli indirizzi e le uscite vengono memorizzate in ogni riga. Sono dispositivi lenti poiché per ogni configurazione degli ingressi bisogna ripetere il ciclo di accesso.

L'architettura delle memorie ROM è una matrice in cui ad ogni intersezione di riga e colonna vi è una cella di memoria, e può essere schematizzata come un decodificatore di indirizzo e un codificatore, realizzato a matrice di diodi o transistor, che possono essere sia a giunzione bipolare che ad effetto di campo. Ad ogni cella della matrice corrisponde una locazione di memoria in cui viene scritto il bit in fase di costruzione.

La lettura avviene tramite l'invio delle linee di indirizzo al decodificatore: esso ha il ruolo di attivare una riga della matrice del codificatore, la quale restituisce l'informazione contenuta nelle memorie allocate nelle intersezioni della riga selezionata e le colonne.

PAL e PLA

La versione base di una PLD è chiamata PAL (Programmable Array Logic), abbreviazione di Programmable Logic Array (in italiano matrice logica programmabile), che nell'elettronica digitale, indica un dispositivo logico programmabile usato per implementare circuiti logici combinatori.



PLA ad n ingressi ed m uscite. Ciascuna porta AND ha p/n linee di ingresso e produce p uscite. Le porte OR dispongono di m/p ingressi.

Una PAL è formata da un piano di AND programmabile e da un piano di OR fisso. Le uscite delle porte OR possono essere in alcuni casi registrate mediante dei flip flop. Le PAL sono dispositivi di tipo field-programmabile.

Una versione più flessibile delle PAL è la PLA (Programmable Logic Array). Anche la PLA è formata da un piano di AND e una di OR, ma in questo caso le connessioni su entrambi i piani sono programmabili.

DIFFERENZE ED ESEMPI PLA E PAL

Iniziamo con l'introdurre delle differenze tra i due dispositivi circuitali, soprattutto sul piano strutturale. Una PLA (Programmable Logic Array) offre la possibilità di personalizzare i collegamenti di entrambe le linee di porte logiche (piano AND e piano OR) e ha una struttura più complessa rispetto a quella di una PAL (Programmable Array Logic). In un dispositivo PAL, invece, solo il piano AND (la linea delle porte AND) risulta programmabile, mentre quella OR è fissa.

	PAL – Programmable Array Logic	PLA – Programmable Logic Array
Struttura	Piano AND configurabile, piano OR fisso e non modificabile	Piano AND e OR entrambi programmabili
Costi e realizzabilità	Dispositivo a basso costo e di semplice realizzabilità	Architettura elaborata e più cara rispetto a quella di un PAL
Numero totale di funzioni	L'utente può configurare solo le AND con numero limitato di funzioni	Offre un numero esteso di funzioni da programmare
Prestazioni del dispositivo	Le linee fisse delle porte OR garantiscono elevate prestazioni	Le performances sono rallentate per via di una rete di link AND-OR

Regole di base per la costruzione di un circuito PAL/PLA Ricordiamo, in via preliminare, l'approccio da utilizzare per realizzare i collegamenti di rete all'interno di un dispositivo di tipo PAL o PLA:

1. **Gli ingressi devono essere deterministici:** per ciascun valore di ingresso generico X_i (il pedice indica il numero dell'ingresso), deve essere trasmesso il valore di X oppure il suo complementare, vale a dire che non è possibile trasmettere all'interno di una rete PAL/PLA

l'ingresso e il suo complementare (devono essere mutuamente esclusivi), altrimenti avremmo un'alea in prossimità di quell'ingresso.

2. **I piani AND e OR devono avere porte dello stesso tipo:** ovvero per ciascun piano le logic gates dovranno essere le stesse in termini di ingressi/uscite. Da ora in poi per convenzione verranno chiamate ANDx e ORx le porte con un prefissato "x" numero di ingressi.
3. **Le funzioni vengono collocate in uscita dal piano OR:** esse sintetizzano la configurazione finale di un generico circuito PAL/PLA, in termini analitici.

REALIZZAZIONE LOGICA DI UN DISPOSITIVO PAL E PLA – ESEMPIO E SPIEGAZIONI

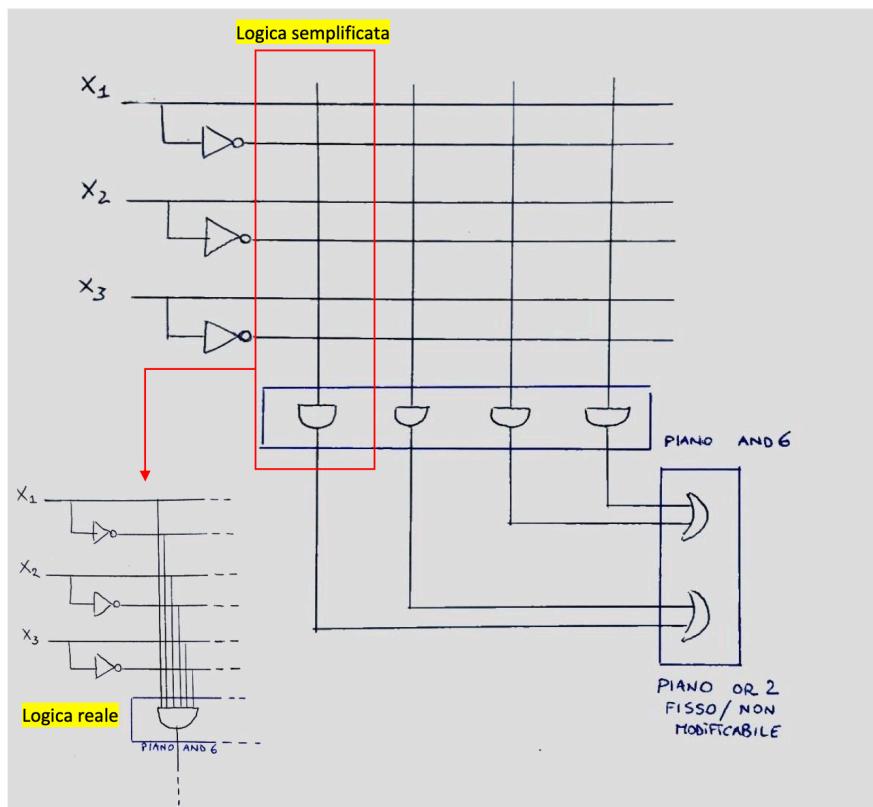
Attraverso un esempio saremo in grado di comprendere meglio la logica di funzionamento di un dispositivo PAL, e come vengono realizzate le funzioni con esso.

L'esempio sottostante indica una realizzazione di una PAL generica con:

- **3 ingressi**
- **2 funzioni (in uscita)**
- **4 porte AND e 2 porte OR.**

Come si può notare, la rete di interconnessioni all'interno di tale circuito dipende esattamente solo da questi parametri di ingresso/uscita, e dalla composizione dei piani AND/OR.

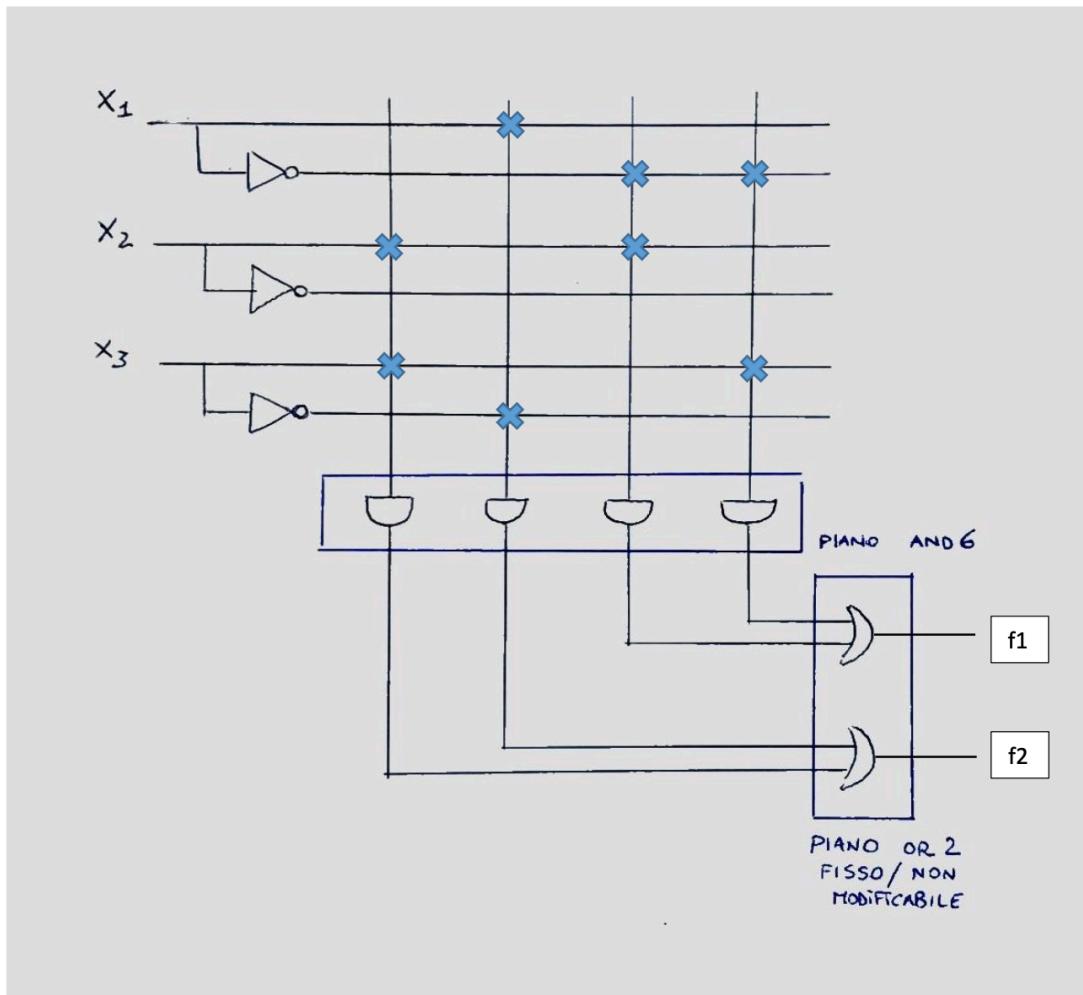
N.B. La rete in figura è ideale e ha il compito di esplicitare il meccanismo di funzionamento, e per semplicità di realizzazione vengono omesse le linee di ingresso per ciascuna porta dei piani AND e OR (sostituite da un'unica linea fittizia, in quanto tutte assumeranno lo stesso valore). Ovviamente la versione reale deve comprendere tali collegamenti, come indicato da una immagine in allegato.



REALIZZAZIONE LOGICA DI UN DISPOSITIVO PAL – PROGRAMMABLE ARRAY LOGIC

Utilizziamo lo stesso esempio per la realizzazione di un dispositivo PAL e possiamo sollecitarlo con alcuni segnali in ingresso. Vorremmo calcolare le due funzioni in uscita della soluzione circuitale indicata sotto forma di esempio.

Utilizziamo una crocetta per ogni segnale di ingresso applicato e a partire da esse andremo ad ottenere le funzioni in uscita.



Calcolo delle funzioni in uscita:

Per f_1 si ha la OR tra l'AND di x_2 e della versione negata di x_1 e l'AND tra x_3 e la versione negata di x_1 , anche per f_2 si ha l'analogo ragionamento, pertanto si può scrivere quanto segue:

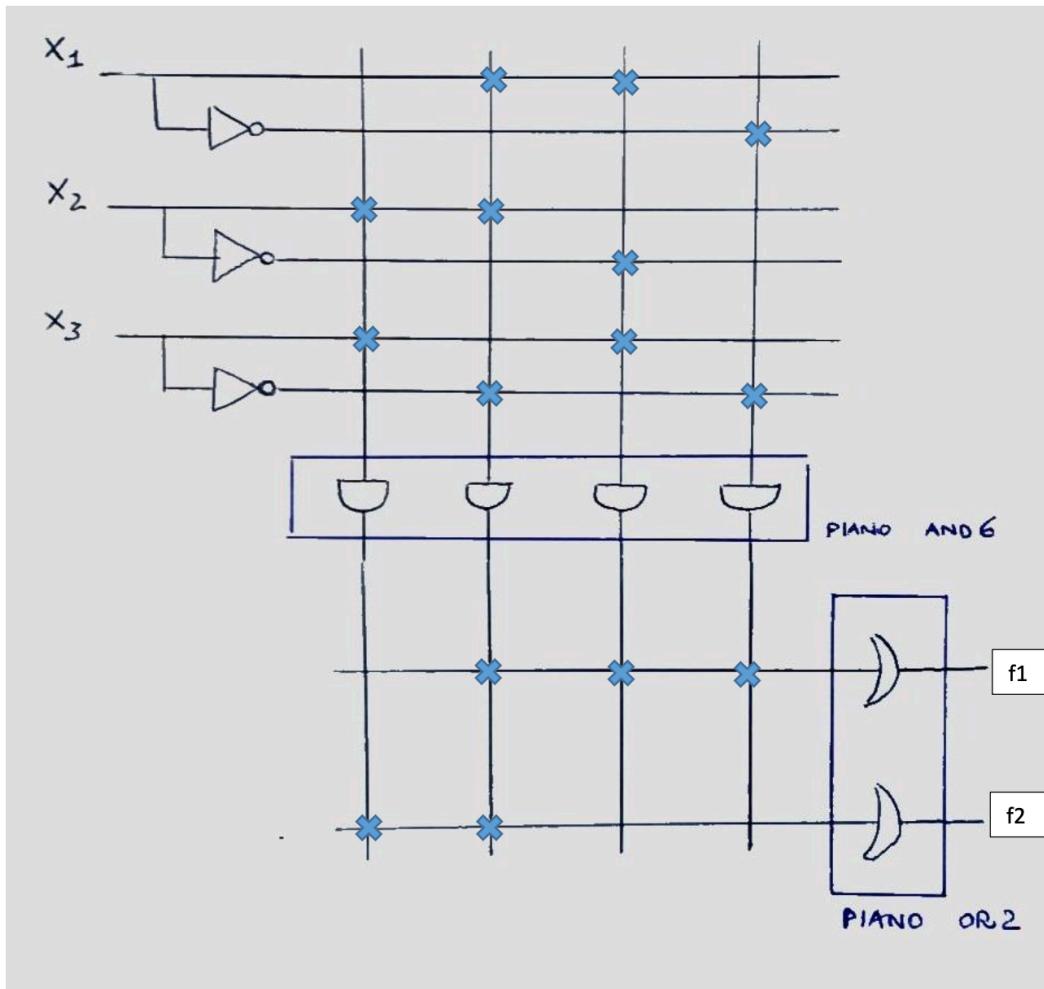
$$f_1 = \overline{x_1}x_2 + \overline{x_1}x_3$$

$$f_2 = x_2x_3 + x_1\overline{x_3}$$

REALIZZAZIONE LOGICA DI UN DISPOSITIVO PLA – PROGRAMMABLE LOGIC ARRAY

In questo caso, invece, dovremmo programmare anche la parte situata sul piano OR, in questo modo vengono selezionati solo i collegamenti attivi. Invece in una PAL vengono presi in considerazione tutti gli ingressi al piano OR.

È possibile, comunque, riutilizzare lo stesso esempio, leggermente modificato a causa della programmabilità del piano OR.



Calcolo delle funzioni in uscita:

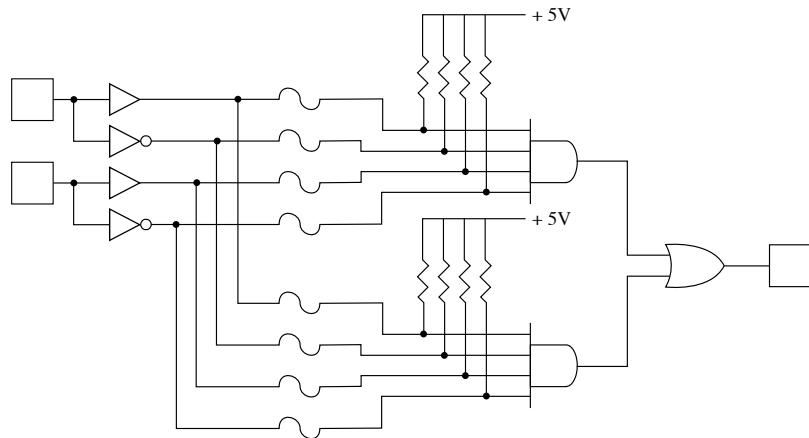
Per f_1 vengono selezionate solo le ultime tre linee mentre la prima viene ignorata, mentre per f_2 si utilizzano per l'OR solo le prime due linee. Pertanto si può scrivere quanto segue:

$$f_1 = x_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 x_3 + \bar{x}_1 x_3$$

$$f_2 = x_2 x_3 + x_1 x_2 \bar{x}_3$$

GAL

GAL (Generic Array Logic): primi dispositivi a permetterne la riprogrammazione (PAL riprogrammabili più volte). La programmazione del dispositivo infatti è mediata da elementi di memoria (e.g. FAMOS, celle SRAM, etc.)



Simplified programmable logic device

Gli elementi programmabili connettono sia gli ingressi che i loro complementari alle porte AND, collegate a loro volta ad una porta OR.

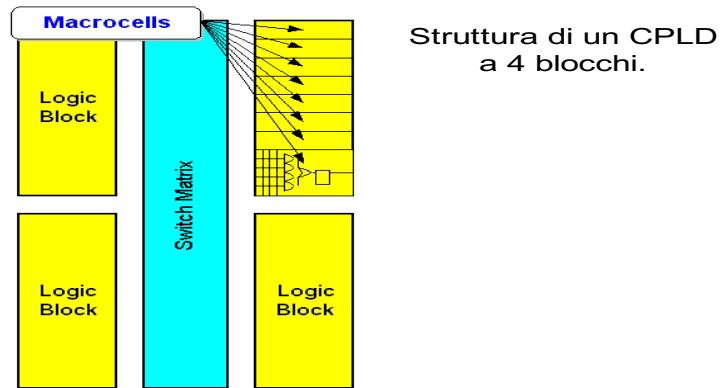
La struttura di una GAL è derivata da quella delle PLA, più precisamente si tratta di PAL con l'aggiunta di dispositivi programmabili in uscita, detti output logic macrocell (OLMC).

La GAL è costituita da un piano logico programmabile, che consiste in una matrice di porte AND, programmabili tramite celle di memoria Programmable Read Only Memory, collegata ad una serie di OR non programmabile. Le porte OR consentono quindi di guidare gli ingressi alle macrocelle, che costituiscono la logica di output del dispositivo. Ogni uscita può avere fino ad otto porte AND, delle quali una è destinata alla gestione di un buffer bidirezionale di uscita. La tipologia di OLMC dei primi dispositivi, caratterizzati da 10 entrate ed 8 uscite, era fissata dal costruttore.

La rigidità dell'apparato di output portò la AMD, nel giugno del 1983, ad introdurre il dispositivo 22V10, dotato di 24 pin e 10 macrocelle in uscita, ognuna delle quali può essere configurata dall'utilizzatore. Tale dispositivo rimpiazzò quasi tutti i PAL esistenti allora.

CPLD

Un Complex Programmable Logic Device (spesso abbreviato con la sigla inglese CPLD), nell'elettronica digitale, è un dispositivo programmable logic device, programmabile e cancellabile, con un numero di pin maggiore di 48. I CPLD sono un'evoluzione delle GAL: un CPLD si può infatti considerare come integrazione di più GAL all'interno di un singolo chip.



Struttura di un CPLD a 4 blocchi.

I tipi di PLD descritti permettono l'implementazione di circuiti logici ad alta velocità. Tuttavia la loro semplice struttura è anche il loro principale difetto. Essi possono implementare solo piccoli circuiti logici che contengono un modesto numero di termini prodotto a causa del numero limitato di connessioni disponibili.

Le CPLD (Complex PLD) estendono il concetto di PLD ad un livello di integrazione più elevato per migliorare le prestazioni del sistema. Invece di costruire PLD più grandi, con più ingressi, termini prodotto e macrocelle, una CPLD contiene diversi blocchi logici, ognuno simile ad una piccola PLD. I blocchi logici comunicano l'uno con l'altro utilizzando segnali indirizzati mediante una rete di interconnessioni programmabile.

CPLD interconnette, in un solo chip, più dispositivi GAL. La matrice di interconnessione (switch matrix) è essa stessa programmabile.

I CPLD si possono definire dispositivi "sea of gate" poiché realizzano, in modo massivo, solo somme di prodotti

La programmazione permette al CPLD di simulare un generico circuito digitale di complessità non elevata. A differenza delle FPGA le CPLD mantengono la programmazione anche quando non sono alimentate perché contengono delle memorie non volatili. I CPLD vengono usati per applicazioni particolari dove sono richieste alte velocità o bassi costi o funzionalità di glue logic ovvero di interfacciamento tra due dispositivi complessi.

Alcuni tipi di CPLD si programmano usando il PAL programmer, ma questo metodo diventa poco pratico quando si devono collegare componenti con centinaia di pin. Un metodo molto più efficiente consiste nel saldare i dispositivi su un circuito stampato e quindi inviare loro, mediante un PC, un flusso di dati che, opportunamente decodificati dai circuiti interni dei CPLD, conferiscono agli stessi la configurazione necessaria a realizzare le funzioni logiche desiderate.

Ciascun produttore ha un proprio nome che identifica questa modalità di programmazione. Per esempio, la Lattice Semiconductor lo chiama "in-system programming". È in corso al riguardo un progetto di standardizzazione da parte del JTAG (Joint Test action Group).

FPGA

FPGA: sono dispositivi rivoluzionari, diffusisi in una manciata di anni. Assieme ai CPLD sono i PLD che dominano il mercato. È l'acronimo di Field- Programmable Gate Array, a significare che il dispositivo è programmabile solo dall'utente (e non più in fonderia).

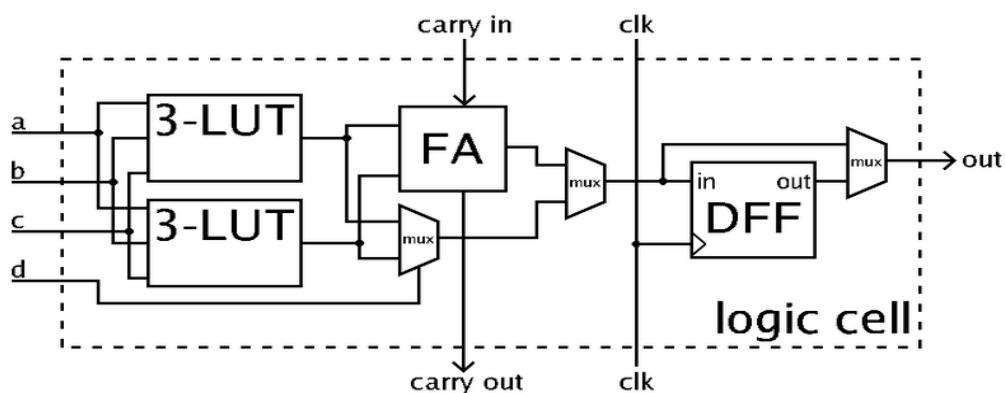
Le FPGA furono introdotte per la prima volta nel 1985 dalla Xilinx. Da allora molti differenti tipi di FPGA sono stati sviluppati da numerose altre compagnie: Actel, Altera, Plessey, Plus, Advanced Micro Devices (AMD), QuickLogic, Alotronix, Concurrent Logic, e molte altre.

I dispositivi contengono sia logica programmabile, sia circuiti hardware speciali.

La parte programmabile è composta da moltissime celle, contenenti 4 funzioni:

- LUT (look-up table);
- Multiplexer;
- Logica di propagazione del riporto (FA);
- Flip Flop D.

Un'FPGA (Field Programmable Gate Array) è un array di celle logiche che possono essere interconnesse in un qualunque modo. Come nelle PAL le connessioni tra elementi sono programmabili dall'utente.



Cella base generica all'interno di dispositivi FPGA.

Negli anni 80/90 vi fu l'introduzione di dispositivi logici ad elevata scala di integrazione di FPGA (e CPLD), programmabili sul campo mediante linguaggi HDL



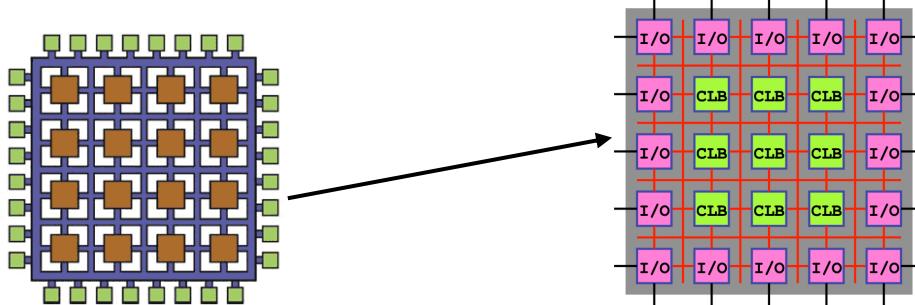
FPGA Altera



FPGA Xilinx

STRUTTURA FPGA

Una FPGA consiste in un insieme di Configurable Logic Blocks (CLB) che possono essere connessi tra loro. La funzione dei singoli CLB e delle connessioni viene impostata dal progettista mediante programmazione (“sul campo”). Tale programmazione può essere ripetuta più volte (teoricamente “infinite”).



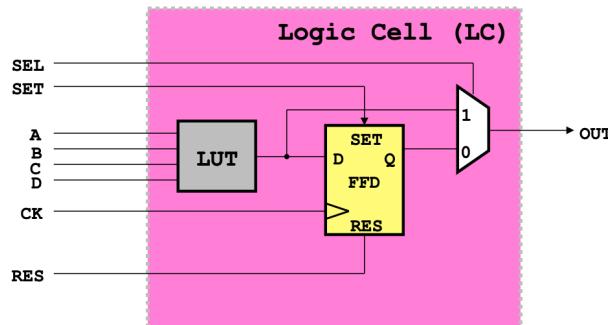
- Numero di CLB molto elevato (migliaia)
- Generalmente è anche disponibile della RAM interna - Block RAM (centinaia di KByte)
- Alcuni blocchi sono dedicati all'I/O
- Disponibili (decine) di sommatori, moltiplicatori

Un’FPGA è composta da un array bidimensionale di blocchi logici che possono essere connessi tramite una rete di interconnessioni. Nelle interconnessioni sono presenti degli switch programmabili che servono per connettere i blocchi logici con i fili della rete di interconnessioni, o segmenti di filo gli uni con gli altri.

Esistono diversi produttori di FPGA e differenti tecnologie. Sostanzialmente FPGA di produttori diversi si differenziano per due aspetti principali:

- Fusibili
- Memorie flash - Memorie SRAM

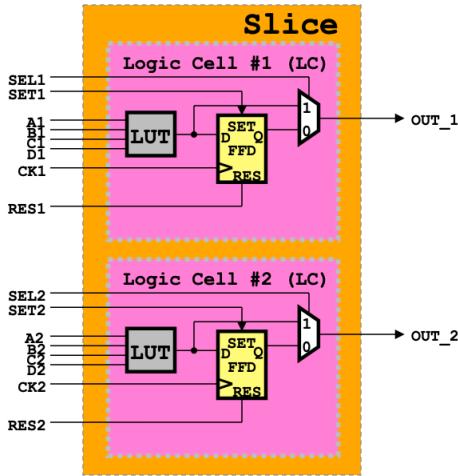
Un’analisi dettagliata dei blocchi logici configurabili (CLB) di una tipica FPGA esula dagli obiettivi di questo corso. Tuttavia, è molto interessante capire come tali blocchi sono organizzati.



Il blocco LUT (Look-up-table) non è altro che una rete combinatoria programmabile (vedi pg successiva).

Il blocco LUT, può essere anche riprogrammato per agire come uno shift-register o un una memoria (distributed RAM)

Tipicamente le Logic Cells sono raggruppate in slices. Ogni slice contiene alcune (2, 4,...) Logic Cells.



Infine, i blocchi logici configurabili (CLB) sono ottenuti raggruppando alcuni slices (2,4,...) opportunamente connessi tra loro (non sono indicati i dettagli delle possibili connessioni nelle figure).

Le FPGA sono normalmente configurate per realizzare reti sincrone (e.g. mediante approccio diretto e per implementare algoritmi).

Per questo motivo è necessario prevedere (almeno) un segnale di clock da inviare alle RSS (Reti Sequentziali Sincrone) che compongono il progetto su FPGA.

Spesso, nei progetti più complessi esistono più domini di clock, ovvero diversi moduli della logica su FPGA utilizzano clock diversi (sia come frequenza sia come duty-cycle).

Nascono quindi delle problematiche inerenti al passaggio di informazioni tra diversi domini di clock.

Le FPGA sono state sviluppate per andare incontro alle esigenze del mercato, come ad esempio:

- Le prestazioni, mettono in grado i sistemi real time di operare a frequenze sempre più elevate.
- Densità e capacità, mettono in grado di aumentare l'integrazione, di integrare sempre più componenti su un chip (system on chip), e utilizzare tutte le porte disponibili sull'FPGA, fornendo così una soluzione efficiente anche dal punto di vista dei costi.
- Facilità di utilizzo, mettono in grado i progettisti di portare sul mercato il loro prodotto velocemente.

Gli FPGA, e recentemente anche i CPLD, sono largamente impiegati grazie alla facilità con cui è possibile programmarli. Tipicamente è un PC, che tramite un opportuno protocollo, configura i dispositivi programmabili.

Il protocollo di comunicazione è JTAG: veloce, versatile e presente in tutti i circuiti digitali.

La tecnologia JTAG viene sfruttata per configurare gli elementi di memoria all'interno del PLD onde ottenere il comportamento desiderato, questo però non nasce per tali scopi, ma per il testing di circuiti integrati digitali.

In sostanza è un protocollo seriale che permette di gestire, tramite un'apposita architettura, la boundary scan.

FPGA DELLA FAMIGLIA SPARTAN-3E

Il circuito contiene i seguenti elementi programmabili:

- Configurable logic blocks (CLB): costituiscono la principale risorsa logica per l'implementazione di circuiti sincroni sequenziali e puramente combinatoriali. Ciascun CLB contiene 4 slice e può emulare qualsivoglia circuito integrato sincrono. Tutti i CLB sono identici!
- Slices: è una struttura contenente 2 LUT (Look-up table);
- Le LUT possono essere usate per realizzare memorie RAM e shift-register. All'interno della slice sono altresì presenti multiplexer e un addizionatore;
- Input/Output Blocks: sono componenti analogiche che controllano il flusso dei dati tra i pin di I/O e i componenti logici all'interno dell'FPGA. Ciascun blocco supporta flussi di dati bidirezionali con comportamento tri-state. Supportano una vasta gamma di segnali standard e sono equipaggiati da registri veloci;
- Digital Clock Manager: forniscono non solo un meccanismo di calibrazione del clock (deskew), ma anche una soluzione puramente digitale per gestire la moltiplicazione, divisione, shifting in fase e distribuzione del clock. Sono costituiti da un anello ad aggancio di ritardo (DLL) e da logica programmabile (range: ~300MHz/5MHz);
- Block RAM: forniscono spazio di memoria volatile utile per molti design hardware. Ciascuna memoria può essere configurata in dual port, per un totale di spazio disponibile pari a 18 Kb;
- Multiplier Blocks: sono componenti aritmetici molto veloci che effettuano l'operazione di moltiplicazione. Ciascun moltiplicatore dispone di due operandi in ingresso da 18 bit ciascuno.

BOUNDARY SCAN

Il boundary scan, noto anche come JTAG, fu proposto come soluzione innovativa per far fronte all'aumento vertiginoso delle densità dei componenti sulle schede elettroniche e al diffondersi di nuovi tipi di contenitore, come il BGA, che rendevano estremamente difficoltoso l'accesso ai punti di misura necessari per eseguire il collaudo.

Il grande vantaggio della tecnologia boundary scan è dato dal fatto che dedicando un numero molto limitato di piedini, tipicamente quattro, e una piccola area di silicio all'interno dei componenti digitali per inserire le logiche di controllo destinate alle funzionalità di collaudo, risulta possibile creare un sistema di verifica adatto a rilevare un gran numero di potenziali guasti sulla scheda, ottenendo un elevato grado di copertura anche quando la maggior parte dei componenti non è raggiungibile per contatto da parte di un tester.

La diffusione di questa tecnica è dovuta ai vantaggi che offre rispetto alle altre tecniche adoperabili:

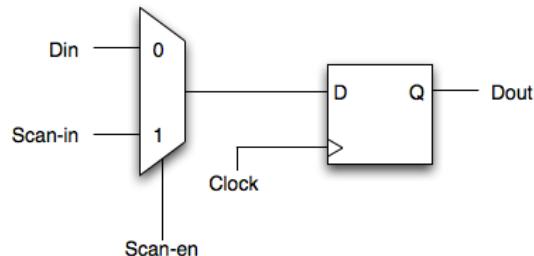
- Richiede pochi pin di interfacciamento;
- È una tecnica generale, che può essere applicata a qualsiasi design;
- È compatibile con le tecnologie produttive;
- Ha una gestione abbastanza semplice;
- Permette di raggiungere ogni componente di memoria on-chip;

Sostanzialmente è una lunga catena di flip-flop del design, intervallata da multiplexer. Il ruolo del mux è quello di lasciare il design intatto quando il sistema è in regime di funzionamento, oppure di

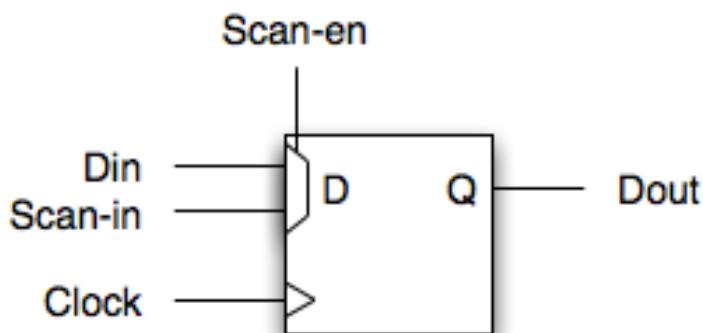
realizzare una grande catena di flip-flop interconnessi a formare un grande shift-register se il sistema è in fase di test.

La scan chain ha un solo punto di ingresso, un punto di uscita, uno di abilitazione ed un segnale per la temporizzazione esterna.

Se più chip in un sistema sono dotati dello stesso meccanismo è possibile collegare tutti loro in una scan chain unica ancora più grande.



Scan-en è il segnale di modo: se è uno seleziona il segnale di servizio come dato, altrimenti quello originale.



Sinteticamente lo si può indicare in questa forma

La boundary scan chain si ottiene collegando l'uscita di ciascun flip-flop con l'ingresso scan-in del successivo. Così si ottiene una lunga catena di registri connessi come in uno shift register.

Se il segnale Scan-en è abilitato, allora tutti i multiplexer selezioneranno come dato quello in uscita dal flip-flop precedente. In questo modo ad ogni colpo di clock i bit avanzano a partire dal segnale Scan-in fino a quello di Scan-out in modo sequenziale.

Se il segnale Scan-en è disabilitato, allora tutti i multiplexer selezioneranno come dato quello utente. In questo modo il sistema non risente della presenza della scan-chain, se non per un ritardo ulteriore pari al tempo di propagazione del componente multiplexer.

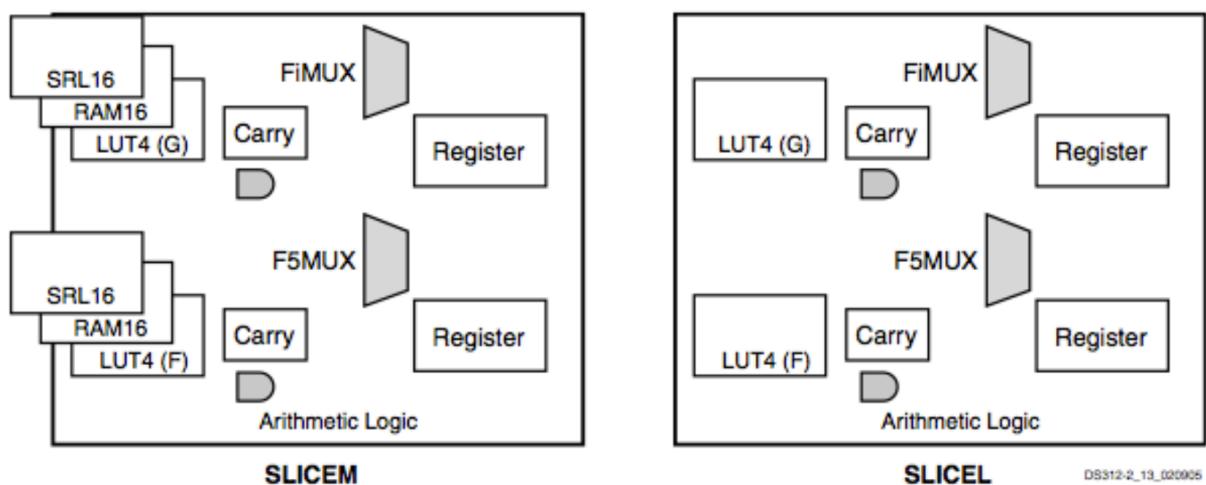
ES:

Contatore mod16: il circuito è composto da 4 registri di memoria per mantenere il conteggio. Un caso di test prevede di effettuare un conteggio a partire da i e verificare che sia $i+1$:

- preparare il flusso di bit da configurare con lo stato corretto
- abilitare scan-en

- fornire in ingresso allo scan-in il vettore di stato fornendo impulsi di scan, per un totale di 4 impulsi
- disabilitare scan-en
- fornire un colpo di clock per far evolvere il contatore (fornire un conteggio)
 - riabilitare scan-en
 - leggere dal pin scan-out il vettore di stato (4 colpi di clock).
 - Se è disponibile un altro test si può fornire un nuovo flusso di bit mentre si scarica quello precedente

SLICEM E SLICEL



È l'unità base programmabile per la famiglia di FPGA Xilinx. Tutte le slice sono raggruppate in coppie e ciascuna coppia è organizzata in colonne con una catena indipendente di carry.

- La coppia posta a sinistra supporta sia funzioni logiche che funzioni di memoria (SLICEM).
- La coppia posizionata a destra invece è capace solo di fornire funzioni combinatorie (SLICEL).

La doppia disposizione consente di risparmiare moltissima area su silicio, e quindi del costo finale del dispositivo. In effetti è stato verificato che se entrambe le coppie di slice fossero dotate di elementi di memoria, le prestazioni dell'FPGA risulterebbero degradate molto in termini di ritardo e potenza dissipata.

L'FPGA così costruito sa emulare “male” una memoria.

Una slice (indipendentemente dal tipo) include:

- 2 generatori di funzione a 4 input (LUT)
- 2 elementi di memoria
- 2 multiplexer
- Logica aritmetica del riporto (FA): XOR e AND

La SLICEM supporta in più due funzioni:

- 2 blocchi di RAM 16x1
- 2 shift register da 16 bit

La combinazione di una LUT e di un elemento di Memoria (Flip Flop D) è detto logic cell. L'aggiunta del multiplexer e del riporto rende la cella logica molto più versatile di qualsiasi dispositivo programmabile. Infatti ciascuna slice è equivalente a 2,25 celle logiche.

FONTI

- <https://www.docenti.unina.it/webdocenti-be/allegati/materiale-didattico/34118425>
- http://vision.deis.unibo.it/~smatt/DIDATTICA/Sistemi_EMBEDDED_M/PDF/03%20-%20FPGA.pdf
- https://it.wikipedia.org/wiki/Programmable_Logic_Array
- https://it.wikipedia.org/wiki/Generic_Array_Logic
- https://it.wikipedia.org/wiki/Complex_Programmable_Logic_Device
- https://it.wikipedia.org/wiki/Complex_Programmable_Logic_Device
- https://en.wikipedia.org/wiki/Boundary_scan

AUTORI

- Di Pace Vincenzo
- Esposito Ciro
- Fusiello Fabiano

Appendice E

JTAG

(Barbaraci Mariarosaria, Improtta Cristina)

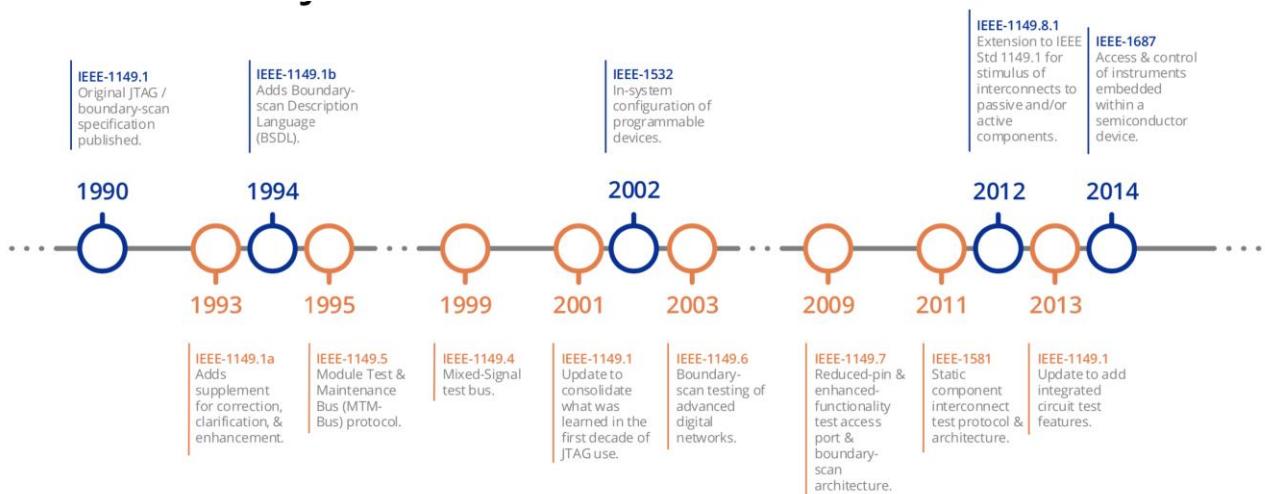
JTAG

A cura di Barbaraci Mariarosaria e Imrota Cristina

Introduzione

Il protocollo JTAG, il cui acronimo sta per “Joint Test Action Group”, è uno standard industriale finalizzato alla verifica e al testing dei circuiti integrati dopo la loro produzione; comunemente definito “boundary-scan”, esso fu sviluppato dall’ Institute of Electrical and Electronics Engineers (IEEE) in seguito alla rapida evoluzione dei componenti e alla crescente necessità di ridurre gli spazi occupati e quindi le dimensioni delle schede. L’enorme sviluppo e diversificazione dei prodotti immessi sul mercato dell’elettronica hanno reso sempre più indispensabile l’utilizzo di un testing accurato e ponderato, anche a costo di sacrificare risorse in termini di spazio: si parla in questo contesto di Design For Testability (DFT), cioè la tecnica dell'affrontare a monte il problema del testing e montare on-chip le strutture per effettuarlo.

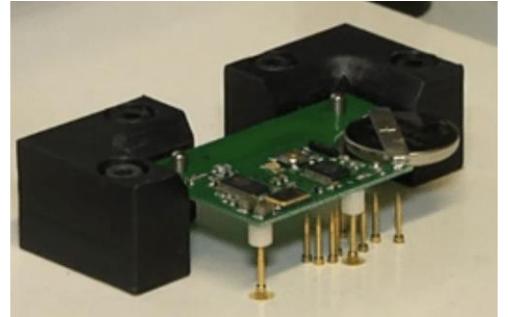
Ne venne avviato lo sviluppo negli anni ’80, per poi essere standardizzato solo nel 1990; nei successivi anni ’93 e ’94, vennero rilasciate le prime versioni correttive ed esemplificative della specifica originale. L’inizio degli anni 2000 vede la scoperta di numerose nuove applicazioni per il protocollo e la conseguente estensione delle sue capacità, tanto che la versione pubblicata nel 2009 costituisce ancora oggi lo standard per molti microcontrollori.



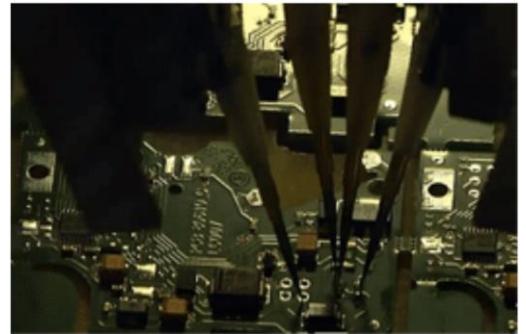
In-circuit testing

Oltre all’adozione del JTAG, l’in circuit test è diviso principalmente in due categorie: l’in circuit test a letto d’aghi (ICT) e il test a sonde mobili o flying probe test (FPT).

L'in circuit test a letto d'aghi adopera il testing elettrico per isolare ogni componente della scheda e verificarne il corretto funzionamento in relazione con gli altri e rispetto al progetto originale. L'accesso ai componenti viene effettuato con l'aiuto di un letto d'aghi specifico per ciascun tipo di scheda che prende il nome di fixture; le fixture, tuttavia, sono spesso complesse e dal costo piuttosto elevato. Inoltre, certe configurazioni circuitali non rendono facilmente possibile l'individuazione dei difetti e la localizzazione dei guasti con questo tipo di tecnica, la quale si rende utile soltanto nel caso di bassi mix di codici e alti quantitativi di schede prodotte.

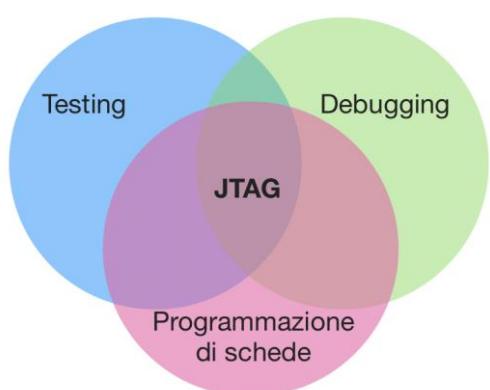


L'in circuit test a sonde mobili usa una tecnologia simile alla precedente, ma invece di usare un'interfaccia di contatto fisso e personalizzata ad hoc come la fixture, utilizza una serie di sonde mobili che entrano in contatto con il circuito stampato. A causa del movimento fisico delle sonde, i tempi dei test associati all'in circuit test a FPT risultano più lunghi rispetto a quelli del test a letto d'aghi, ma si mostra conveniente nel caso di produzioni molto variabili e con un basso volume.



Il vantaggio del JTAG rispetto a queste metodologie è che l'unico hardware richiesto per la verifica è un controller JTAG, mentre le tecniche ICT e FPT richiedono apparecchiature di test specializzate che non sempre risultano disponibili o facilmente reperibili.

Possibili utilizzi del protocollo



Gli utilizzi di questo protocollo, comunque, non si limitano soltanto al testing, ma spaziano anche verso la programmazione delle schede FPGA/CPLD e verso il debugging e l'emulazione del software.

In merito al debugging dei sistemi embedded, JTAG rappresenta uno strumento fondamentale per accedere a sotto-blocchi di circuiti integrati, i quali potrebbero non avere dei meccanismi dedicati alla risoluzione degli errori; infatti, un cosiddetto "JTAG adapter", un emulatore in-circuit, sfrutta il protocollo per accedere ai moduli di debug della CPU sin dalle prime istruzioni dopo il suo reset, rendendo possibile agli sviluppatori la correzione di eventuali malfunzionamenti sia a livello di istruzioni macchina, sia mediante linguaggio di alto livello.

circuit, sfrutta il protocollo per accedere ai moduli di debug della CPU sin dalle prime istruzioni dopo il suo reset, rendendo possibile agli sviluppatori la correzione di eventuali malfunzionamenti sia a livello di istruzioni macchina, sia mediante linguaggio di alto livello.

È inoltre possibile per gli sviluppatori FPGA, mediante JTAG, la produzione di veri e propri tool di debugging volti alla ricerca di errori in altri componenti della scheda, come ad esempio nei registri, che sarebbero invece impossibili da trovare con le sole operazioni di boundary scan.

Ancora, JTAG permette di trasferire una certa quantità di dati all'interno degli elementi di memoria presenti sulla scheda grazie al JTAG port; mediante questo port, in più, è possibile monitorare alcune tipologie di dati del dispositivo, come la temperatura, il voltaggio, il valore di corrente ecc. Si può adoperare questo protocollo di comunicazione quindi per scrivere software su memorie flash attraverso bus dati, come nel caso della CPU, o vere e proprie interfacce JTAG.

Per quanto riguarda il testing, si adotta la tecnica del boundary scan: si include all'interno del design della scheda una catena di elementi di memoria, flip-flop, intervallati da multiplexer, che abbia un solo punto di ingresso, di uscita, di abilitazione e di sincronizzazione esterna. Lo scopo è quello di essere in grado di controllare il funzionamento del dispositivo grazie all'inserimento di segnali in maniera arbitraria. Queste celle possono quindi operare in due modalità, proprio grazie all'aggiunta dei multiplexer: nella modalità funzionale, la loro presenza viene praticamente ignorata e la scheda funziona normalmente; nella modalità test invece è possibile isolare i singoli componenti dai pin esterni, così da verificarne il comportamento in risposta a determinati stimoli.

Questo approccio fa sì che non sia necessaria alcuna particolare configurazione del componente sotto testing e riduce in maniera significativa l'accesso fisico richiesto per l'analisi del dispositivo; in più, necessita di pochi pin di interfacciamento ed è versatile dal punto di vista dell'applicabilità e della compatibilità.

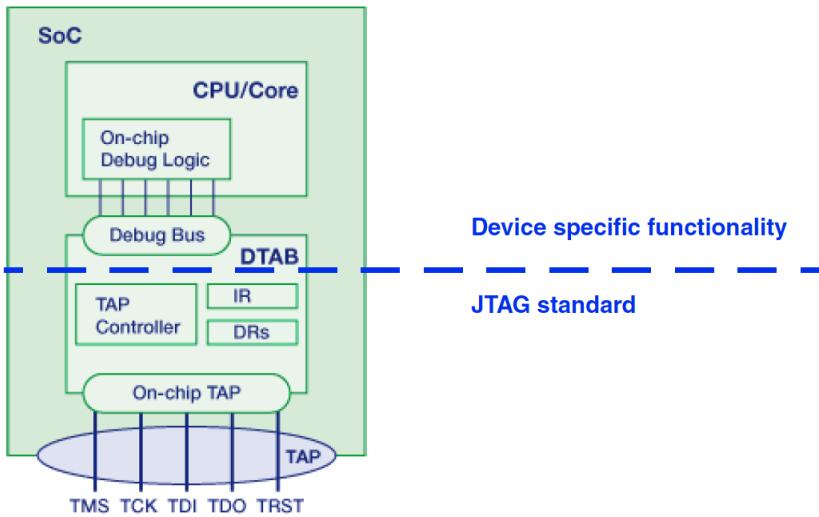
Testing

JTAG è un protocollo seriale di comunicazione che definisce un'architettura standard per il TAP (Test Access Port) e il Boundary Scan utilizzati per il test di un componente PCB (Printed Circuit Board).

Le funzionalità solitamente offerte dal JTAG sono:

- **Debug Access**, il quale è utilizzato dai tool di debug per accedere alle risorse e funzionalità di un chip al fine di renderle accessibili e modificabili.
- **Boundary Scan**, utilizzato, solitamente, da tool per il test dell'hardware.

Ogni componente che vuole rispettare lo standard JTAG avrà una configurazione del genere:



Il JTAG può essere implementato in un dispositivo grazie ad un modulo logico DTAB; il Debug and Test Access Block (DTAB) è implementato su un chip target come un dispositivo “passivo” e non si attiva finché non riceve delle richieste.

Per rispettare lo standard IEEE 1149.1, tale blocco deve essere costituito da:

- Il TAP (Test Access Port), il quale consiste di 4 segnali, più un quinto opzionale di reset, ed è l’interfaccia primaria per pilotare i test e accedere alla logica delle schede.
- Il TAP controller, il quale è definito da una macchina a 16 stati che, tramite i segnali di ingresso, controlla le diverse azioni possibili.
- Un Instruction Register (IR).
- Dei Data Register (DRs).

Un Debug Bus è utilizzato, infine, per la comunicazione con la logica di debug presente sul chip e può essere implementato in modo diverso a seconda del componente da testare e dalla Debug logic sul chip.

JTAG Interface

L’ interfaccia fisica del JTAG, o test access port (TAP) consiste in quattro segnali obbligatori e un segnale opzionale di reset asincrono. La tabella sottostante riassume i segnali di TAP.

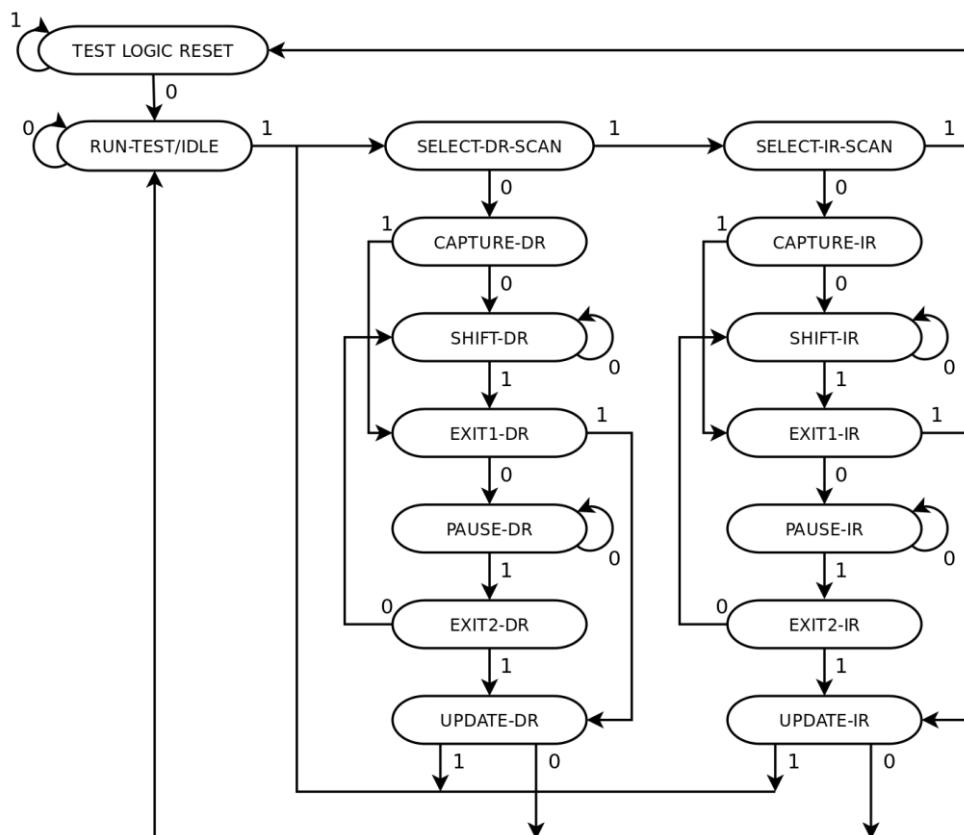
Abbreviazione	Segnale	Descrizione
TCK	Test Clock	Sincronizza le operazioni interne della macchina a stati.
TMS	Test Mode Select	Campiona sul fronte di salita del clock in modo da determinare lo stato successivo
TDI	Test Data In	Rappresenta i dati trasferiti nel dispositivo di test o nella logica programmabile. È campionato sul fronte di salita del clock quando la macchina a stati interna è nello stato opportuno.

Abbreviazione	Segnale	Descrizione
TDO	Test Data Out	Rappresenta i dati trasferiti fuori dal dispositivo di test o dalla logica programmata e è valido sul fronte di discesa del clock quando la macchina a stati interna è nello stato opportuno.
TRST	Test Reset	Un pin opzionale che, quando attivo, può riportare ad uno stato di reset il TAP controller.

Molte interfacce TAP impiegano segnali aggiuntivi rispetto a quelli richiesti dallo standard. Per esempio, le applicazioni di debugging on-chip possono includere dei segnali di terminazione e ripristino, mentre le applicazioni di in-system-programming possono incrementare la velocità di programmazione sfruttando pin aggiuntivi per operazioni critiche (time-critical), come far commutare (toggling) il segnale di abilitazione per la scrittura o fare una verifica ciclica (polling) dei segnali di pronto/occupato.

TAP Controller

Il TAP controller è definito dallo standard IEEE, e utilizza, come detto in precedenza, una macchina a stati finiti a 16 stati.



Le transizioni sono determinate dallo stato del TSM e dal fronte di salita del TCK.

Due percorsi analoghi attraverso la FSM sono utilizzati per catturare e/o aggiornare i dati attraverso una scansione dei registri IR o il DRs.

Si vogliono citare solo alcuni degli stati dell'automa che hanno una particolare rilevanza:

- **Test Logic Reset**, il quale riporta l'instruction register al suo valore di default (IDCODE o BYPASS). Tale stato può essere raggiunto da qualsiasi altro ponendo per 5 volte TMS ad '1'.
- **Run Test/Idle** e **Select DR-Scan** sono utilizzati da molti debugger come stati di attesa.
- **Shift-IR**, è lo stato in cui lo strumento di debug può inserire un'istruzione nel IR; tale istruzione sarà attivata una volta che si raggiungerà lo stato Update-IR.
- **Shift-DR**, è lo stato in cui lo strumento di debug preleva o immette dati nei DR seconda dell'istruzione caricata.

Istruzioni dell'IR

Le funzionalità offerte dal DTAB sono accessibili tramite le istruzioni nell'IR. Caricando un'istruzione, il relativo Data Register è selezionato e fornisce o accetta dati in base all'istruzione selezionata.

Alcune delle istruzioni sono obbligatorie e definite dallo standard IEEE, ma la maggior parte di esse sono implementate liberamente dal costruttore del dispositivo e sono specifiche per esso.

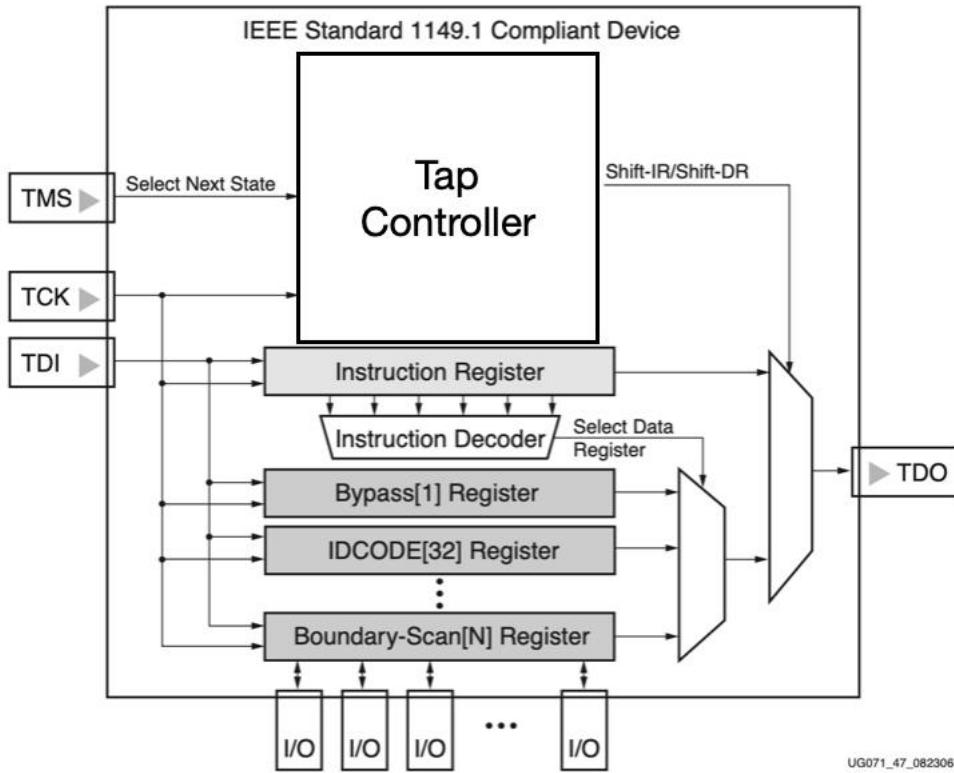
Si riportano le istruzioni necessarie definite dallo standard affinché un dispositivo sia conforme ad esso:

- **BYPASS**, viene utilizzata per collegare più dispositivi in daisy-chain in configurazioni multi-core.
- **EXTEST**, causa il collegamento di TDI e TDO al Boundary Scan Register(BSR); lo stato dei pin del dispositivo viene campionato nello stato di Capture-DR e nuovi valori vengono inseriti nel BSR nello stato di Shift-DR; questi nuovi valori saranno applicati ai pin del dispositivo nello stato di Update-DR.
- **SAMPLE/PRELOAD**, tale istruzione collega TDI e TDO al BSR, ma il dispositivo lavora secondo il suo normale funzionamento. In questo stato è possibile campionare i valori in ingresso o in uscita dai pin, ma anche prevaricare dei dati di test nel BSR prima che sia caricata un'istruzione di EXTEST.

Oltre a queste istruzioni obbligatorie lo standard ne definisce altre, e le istruzioni tipicamente implementate e disponibili sono le seguenti:

- **IDCODE**, identifica il dispositivo.
- **INTEST**, collega sempre TDI e TDO al BSR, ma lo collega ai segnali della logica interna al dispositivo.

Tipica architettura JTAG



Il figura si vuole riportare, a questo punto, dopo una descrizione dettagliata dei suoi componenti, uno schematico di una tipica architettura che realizza lo standard JTAG. Nell'architettura raffigurata si può identificare una unità operativa e un'unità di controllo, il TAP Controller.

Oltre gli elementi già approfonditi, si può notare che si hanno 3 tipi di Data Register primari:

- **Boundary Scan Register**, BSR, il quale è utilizzato per effettuare un test dei pin di I/O.
- **Bypass Register**, costituito da un singolo bit e che permette un veloce transito del dato in ingresso al TDI, subito in uscita verso il TDO.
- **IDCodes**, registro che contiene l'ID code e numero di revisione del dispositivo. In aggiunta a questi, c'è la possibilità da parte dell'utente di configurare dei registri (USER1,USER2).

È ben visibile, in figura, come i pin di I/O siano collegati al BSR. Il Boundary Scan Register è costituito da un insieme di celle elementari che prendono il nome di "Boundary scan cells" e sono proprio queste celle a rendere possibile il test dei pin di un dispositivo senza un diretto accesso fisico.

Boundary-Scan Test

Le celle boundary-scan sono collegate in modo da formare uno shift-register, il quale è acceduto tramite la porta seriale TDI (Test Data Input) e TDO (Test Data Output). Durante il normale funzionamento di un dispositivo queste celle sono invisibili,

mentre nella modalità di test permettono di iniettare/leggere dei valori su/da i pin di I/O oppure, nella modalità di ‘internal’ permettono di interfacciarsi con i valori sulla core-logic del dispositivo.

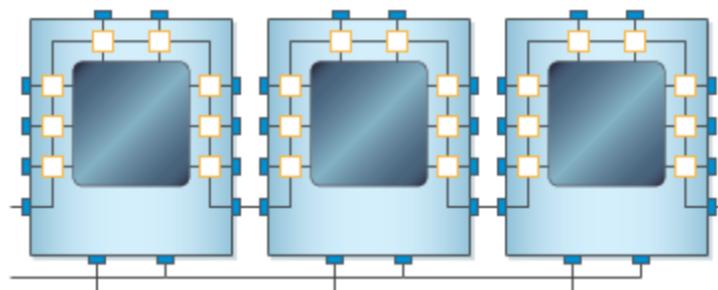
BSDL Files

Il file Boundary-Scan Description Language (BSDL) è utilizzato per descrivere il comportamento del boundary scan e le capacità di un determinato componente.

Scan Chain

Spesso dispositivi d’interesse sono composti da più componenti JTAG, per cui questi vengono interconnessi in daisy-chain (collegamento in serie tra più impedenze) all’interno del dispositivo e controllati simultaneamente.

Il test Boundary-scan può utilizzare un primo componente per pilotare i segnali che saranno rilevati da un secondo componente e così verificare la continuità dei segnali



da pin a pin.

I dispositivi possono essere configurati nella modalità di BYPASS per accorciare la lunghezza totale delle catene e ridurre il tempo di test.

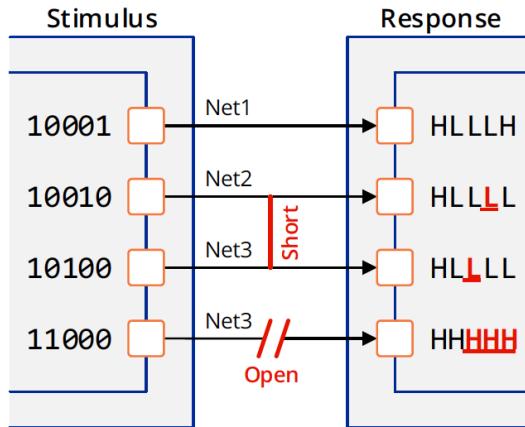
Nei progetti più complicati si potrebbero utilizzare dei circuiti addizionali o bridge JTAG dedicati per configurare in maniera selettiva delle catene.

Connection Testing

Il connection testing controlla le connessioni dal dispositivo verso gli altri componenti del circuito e verifica che tali connessioni rispettino le specifiche di progetto. Tali connessioni possono presentare 4 tipi di errori:

- Cortocircuito
- Circuito aperto
- Stuck-at ('1','0','X')
- Pull-resistor fault

Il connection test è uno strumento inestimabile per la validazione in un processo di produzione. Per meglio spiegarne il funzionamento si presenta un esempio.



Due dispositivi conformi al JTAG sono collegati come in figura. Il primo dispositivo pilota, tramite 4 suoi output, gli input della seconda rete. Nel caso illustrato si hanno due errori: il primo è un cortocircuito tra Net2 e Net3, il secondo è un circuito aperto sull'ultima Net. Assumiamo, inoltre, che il cortocircuito implica la realizzazione di una AND tra Net2 e Net3, mentre il circuito aperto si comporta come una condizione stuck-at-1.

Per rilevare ed isolare i difetti, si inseriscono il pattern scelti nel BSR del primo dispositivo e si

utilizzano come input per la seconda rete; dopodiché i valori catturati dal BSR del secondo dispositivo vengono prelevati e confrontati con i valori attesi. Nell'esempio i risultati sottolineati in rosso sono quelli non conformi a quelli attesi, di conseguenza il tester classifica quei collegamenti come difettosi.

Appendice F

Il processore MIC-1

(*Prof. Nicola Mazzocca, Ing. Alberto Moriconi*)

1 Il Livello Microarchitetturale

Il processore Mic-1 è un utile esempio didattico, presentato in [1] per due scopi principali:

1. mostrare come, usando elementi logici di base come quelli già studiati nel corso, sia possibile realizzare una microarchitettura che implementi un semplice set di istruzioni;
2. mostrare come anche la realizzazione di un sistema apparentemente complesso, come un processore, si riduca in realtà alla progettazione di un'unità operativa e di un'unità di controllo, e del modo in cui devono comunicare.

Il set di istruzioni implementato dal Mic-1 è un sottoinsieme di quello della Java Virtual Machine, denominato *IJVM* in quanto opera unicamente sugli interi.

Una particolarità di questo processore è quella di non disporre di registri generali; la sua architettura è infatti di tipo *a stack*: le sue istruzioni aritmetiche e logiche non hanno operandi esplicativi, ma li prelevano da una struttura *last-in-first-out* allocata nella memoria principale, in cui devono essere posti in precedenza.

Per iniziare a comprendere come opera un processore di questo tipo, consideriamo ad esempio l'esecuzione dell'operazione di somma e assegnazione

```
a1 = a2 + a3;
```

che è tradotta in linguaggio assembly IJVM come

```
ILOAD a2  
ILOAD a3  
IADD  
ISTORE a1
```

Lo stato dello stack durante l'esecuzione evolve come in fig. 1.1; inizialmente, le locazioni di memoria *a2* e *a3* contengono gli operandi, mentre la locazione *a1* è non inizializzata; dunque:

- (a) la prima istruzione *ILOAD* legge il contenuto della locazione di memoria *a2* e ne esegue il push su stack;
- (b) la seconda istruzione *ILOAD* legge il contenuto della locazione di memoria *a3* e ne esegue il push su stack;
- (c) l'istruzione *IADD* esegue il pop dei due elementi in cima allo stack e il push della loro somma;

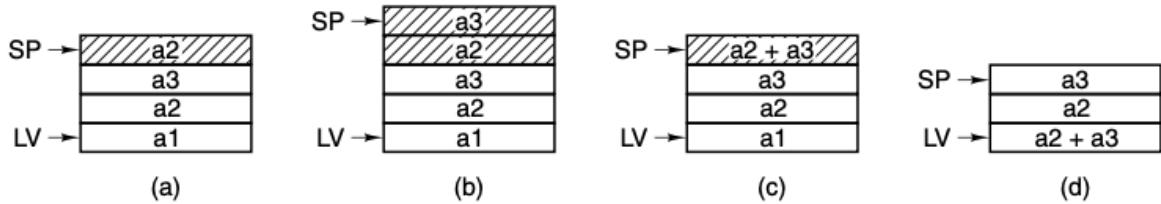


Figura 1.1: Evoluzione dello stack durante le operazioni di somma e assegnazione

- (d) l'istruzione `ISTORE` esegue il pop della somma e la scrive in memoria alla locazione `a1`.

Supponiamo che il processore si trovi al passo (c) e debba eseguire l'istruzione `IADD`; per farlo deve, in prima analisi:

1. eseguire un primo accesso in memoria, per prelevare il primo operando;
2. eseguire un secondo accesso in memoria, per prelevare il secondo operando;
3. sommare i due operandi;
4. eseguire un terzo accesso in memoria, per scrivere il risultato.

Per fare questo è necessario controllare gli accessi in memoria, l'ALU, ed eseguire delle operazioni di *book-keeping* (e.g. aggiornare il puntatore alla testa dello stack, che è modificato dalle operazioni di push e pop).

L'implementazione del processore Mic-1 che presentiamo è detta in *logica microprogrammata*: ciascuna istruzione IJVM è implementata come una sequenza di *microistruzioni* (detta talvolta *microprecedura*); tali sequenze compongono il *microprogramma*.

Nota 1.1

Il microprogramma è tipicamente memorizzato in una ROM interna al processore.

Per capire come è strutturata una microistruzione e come è possibile realizzare un microprogramma che implementi un set di istruzioni, è necessario introdurre anzitutto l'unità operativa del processore.

1.1 L'Unità Operativa

L'unità operativa del processore che intendiamo progettare comprende l'ALU, i suoi ingressi, e le sue uscite (tra cui i registri che si interfacciano con la memoria), ed è mostrata in fig. 1.2.

I registri hanno dimensione di 32 bit e non sono accessibili al programmatore, ma solo al microprogramma.

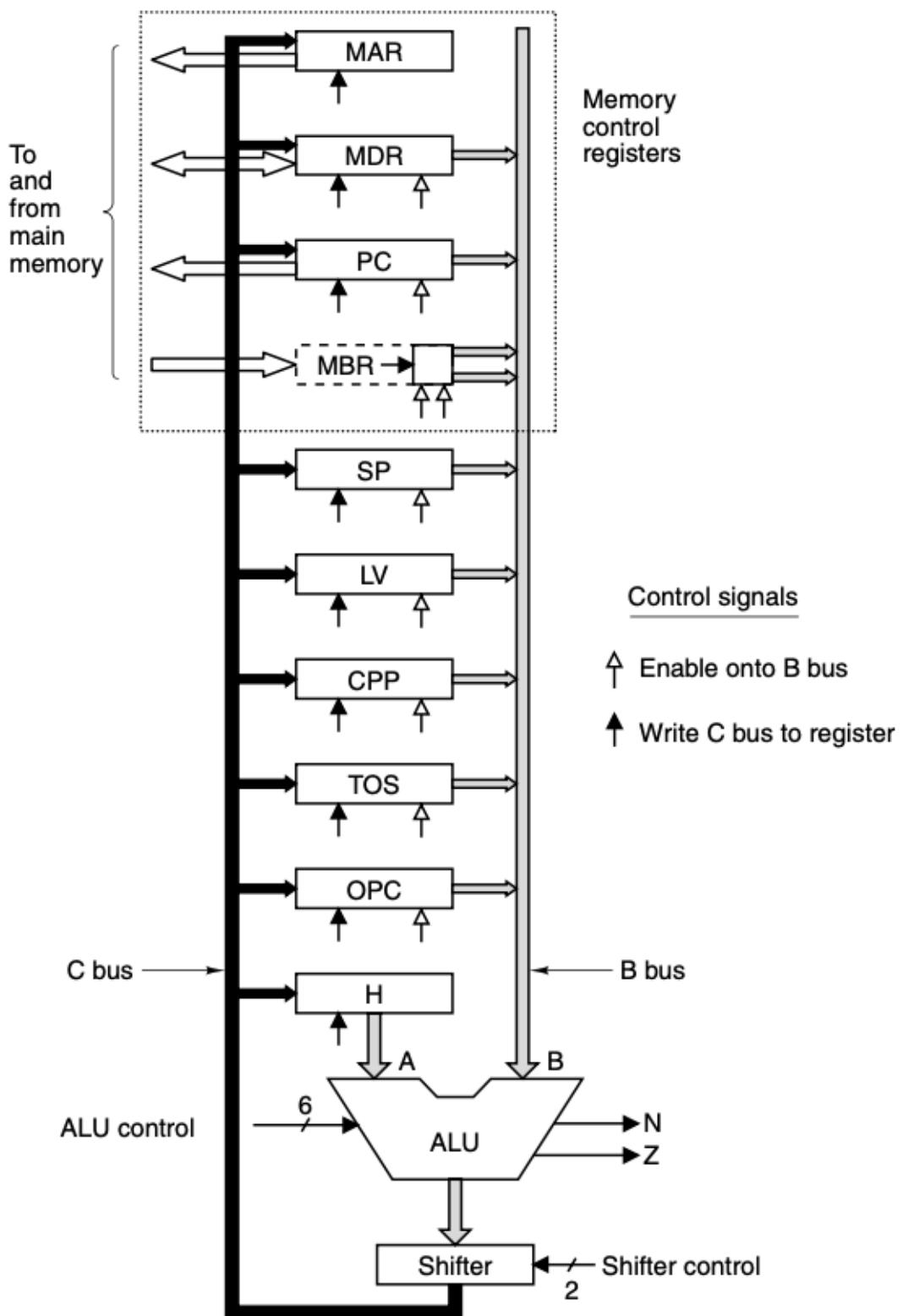


Figura 1.2: Parte operativa del processore Mic-1

Nota 1.2

Quando diciamo che i registri non sono accessibili al programmatore intendiamo che essi non fanno parte del *modello di programmazione*, i.e. non vengono utilizzati esplicitamente come operandi delle istruzioni. Vengono invece utilizzati dal micropogramma per implementare le istruzioni stesse.

L'unità operativa dispone di due bus, indicati con B e C, collegati rispettivamente al secondo ingresso e all'uscita dell'ALU; il primo ingresso dell'ALU è invece collegato esclusivamente al registro H (*holding*).

Con alcune eccezioni (i cui motivi diverranno chiari a breve), i registri dispongono di una coppia di segnali di controllo che permettono:

- di abilitare il collegamento del registro al bus B, rendendolo effettivamente l'operando B dell'ALU;
- di abilitare la scrittura sul registro del risultato fornito dall'ALU sul bus C.

Solo un registro può essere collegato al bus B in un determinato istante, mentre il risultato dell'ALU (i.e. il dato sul bus C) può essere scritto su più registri se necessario.

Nota 1.3

Il collegamento dei registri al bus B può essere implementato usando porte tri-state o un multiplexer; nel nostro caso, useremo un multiplexer in quanto gli FPGA di cui facciamo uso dispongono di buffer tri-state esclusivamente ai pad di IO.

Alcuni registri sono cablati in modo da poter essere usati solo per uno scopo specifico:

- i registri dell'interfaccia con la memoria:
 - MAR - memory address register;
 - MDR - memory data register;
 - PC - program counter;
 - MBR - memory byte register;
- il registro che mantiene il primo operando dell'ALU:
 - H - holding.

Gli altri registri sono funzionalmente equivalenti, ed i loro nomi sono assegnati sulla base dell'uso che se ne fa nel micropogramma:

- SP - stack pointer;
- LV - local variables;

- CPP - constant pool pointer;
- TOS - top of stack;
- OPC - scratch register.

1.1.1 L'Unità Aritmetico Logica

L'ALU del processore Mic-1 ha due ingressi, che denominiamo A e B; l'ingresso A è collegato esclusivamente al registro H, mentre l'ingresso B è collegato al bus omonimo, che può essere guidato dalle nove sorgenti indicate con una freccia grigia in fig. 1.2; le sue funzioni sono controllate mediante sei linee di controllo, che consideriamo asserite se poste a 1:

- F0 e F1 determinano l'operazione da eseguire;
- ENA e ENB abilitano singolarmente gli ingressi;
- INVA inverte l'ingresso A;
- INC incrementa di 1 il risultato.

Non tutte le combinazioni di questi segnali sono rilevanti; alcune delle più importanti per implementare il set di istruzioni IJVM sono riportate in tab. 1.3.

A questi segnali di controllo se ne aggiungono due ulteriori, utilizzati per eseguire lo shift del risultato dell'ALU prima che esso venga posto sul bus C:

- SLL8 (*Shift Left Logical 8 bit*) esegue lo shift a sinistra di 8 bit, ponendo i bit meno significativi a 0;
- SRA1 (*Shift Right Arithmetic 1 bit*) esegue lo shift a destra di 1 bit, con estensione del segno (i.e. il bit più significativo non cambia).

Oltre a quella relativa al risultato, l'ALU dispone di due uscite *flag* ulteriori:

- il flag N è posto a 1 se il risultato è negativo;
- il flag Z è posto a 1 se il risultato è 0.

1.1.2 L'Interfaccia con la Memoria

Il processore Mic-1 può comunicare con la memoria mediante due interfacce:

- MAR e MDR controllano un'interfaccia word-addressable a 32 bit in lettura e scrittura;
- PC e MBR controllano un'interfaccia byte-addressable a 8 bit in sola lettura.

F₀	F₁	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	\bar{A}
1	0	1	1	0	0	\bar{B}
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B - A
1	1	0	1	1	0	B - 1
1	1	1	0	1	1	-A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	-1

Figura 1.3: Segnali di controllo ALU e relative funzioni

Nota 1.4

Come vedremo in seguito, la prima interfaccia è utilizzata per accedere ai dati su cui operano le istruzioni IJVM, rappresentati su 32 bit; la seconda, invece, per prelevare le istruzioni IJVM dalla memoria, che possono avere una lunghezza variabile in multipli di byte (e richiedono dunque più accessi successivi in memoria). Gli indirizzi sono comunque rappresentati su 32 bit per entrambe le interfacce.

Le due interfacce funzionano in modo simile in lettura: l'indirizzo da cui si vuole leggere deve essere posto nel registro MAR (PC); al successivo fronte di salita del clock, il dato è disponibile nel registro MDR (MBR). Indirizzano però in modo diverso la memoria: scrivendo 0x00000002 nel registro PC e abilitando una lettura, il byte all'indirizzo 0x00000002 (i.e. il byte di indirizzo 2) viene letto e posto negli 8 bit meno significativi del registro MBR; scrivendo lo stesso indirizzo nel registro MAR e abilitando una lettura, i 4 byte 0x8-0x11 (i.e. la word di indirizzo 2) vengono letti e posti nel registro MDR.

L'interfaccia MAR/MDR dispone inoltre di un segnale di *write enable* (WE) che, se asserito al fronte di salita di clock, indica alla memoria di scrivere il dato contenuto in MDR all'indirizzo contenuto in MAR.

Il protocollo per l'interfaccia a 32 bit è mostrato in fig. 1.4.

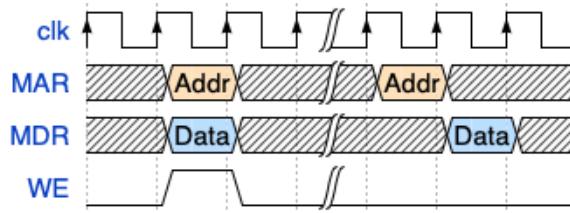


Figura 1.4: Protocollo di scrittura e lettura sull’interfaccia MAR/MDR

1.1.3 Implementazione VHDL dell’Unità Operativa

L’implementazione dell’unità operativa richiede di codificare in VHDL tre parti principali:

- i registri;
- i bus;
- l’ALU

Nota 1.5

Per il momento lasceremo in sospeso la struttura dell’entity relativa all’unità operativa, in quanto per capire quali segnali vengono esposti e perché è utile aver prima compreso anche il funzionamento dell’unità di controllo.

Implementazione dei Registri

I registri dell’unità operativa sono implementati mediante un singolo process, che ne gestisce la logica di reset e di scrittura mediante il bus C:

datopath.vhd

```
-- Registers
signal sp_reg      : reg_data_type;
signal lv_reg      : reg_data_type;
signal cpp_reg     : reg_data_type;

[ ... ]

-- Processor registers
reg_proc : process(clk) is
begin
  if rising_edge(clk) then
```

```

if reset = '1' then
    sp_reg  <= x"00000101";
    lv_reg  <= x"00000100";
    cpp_reg <= x"00000080";

    [...]

else
    if c_to_reg_control(c_ctrl_mar) = '1' then
        mar_reg <= c_bus;
    end if;
    if c_to_reg_control(c_ctrl_pc) = '1' then
        pc_reg <= c_bus;
    end if;
    if c_to_reg_control(c_ctrl_sp) = '1' then
        sp_reg <= c_bus;
    end if;

    [...]

    end if;
end if;
end process reg_proc;

```

Il tipo `reg_data_type` è introdotto come abbreviazione per un `std_logic_vector` di 32 bit (31:0); è definito, insieme ad altri tipi utili per l'implementazione del processore, nel file `common_defs.vhd`:

```

common_defs.vhd

-- Data widths
--! Processor register data width
constant reg_data_width      : positive := 32;

[...]

-- Subtypes
--! Processor register data
subtype reg_data_type is std_logic_vector(reg_data_width
→ - 1 downto 0);

```

Il segnale `c_to_reg_control` è di tipo `c_ctrl_type`, anch'esso definito nel file `common_defs.vhd`, ed è indicizzato mediante una serie di costanti relative agli indici dei singoli segnali di attivazione:

```
common_defs.vhd

--! C bus control width
constant c_ctrl_width          : positive := 9;

[...]

--! C bus control type
subtype c_ctrl_type is std_logic_vector(c_ctrl_width - 1
                                         & downto 0);

[...]

--! C control MAR bit
constant c_ctrl_mar           : natural := 0;
--! C control MDR bit
constant c_ctrl_mdr           : natural := 1;
--! C control PC bit
constant c_ctrl_pc             : natural := 2;
--! C control SP bit
constant c_ctrl_sp             : natural := 3;

[...]
```

Implementazione dell'ALU

In VHDL, l'`entity declaration` per l'ALU che abbiamo descritto può essere scritta come:

```
alu.vhd

entity alu is
  port (
    --! ALU control
    control      : in  alu_ctrl_type;
    --! ALU operand A
    operand_a    : in  reg_data_type;
    --! ALU operand B
```

```

    operand_b      : in  reg_data_type;
    --! ALU result
    sh_result     : out reg_data_type;
    --! Negative flag
    negative_flag : out std_logic;
    --! Zero flag
    zero_flag     : out std_logic
  );
end entity alu;

```

L'implementazione realizzata è puramente behavioral.

Implementazione dei Bus

In prima analisi, i bus sono implementati come dei signal VHDL:

datapath.vhd

```

-- Signals
signal a_bus : reg_data_type;
signal b_bus : reg_data_type;
signal c_bus : reg_data_type;

```

Punto critico nell'implementazione dei bus è però la disciplina di accesso utilizzata. Per quanto riguarda A, si tratta semplicemente dell'insieme di fili che collega il registro H al primo ingresso dell'ALU; il bus C invece è collegato all'uscita dello shifter della componente ALU:

datapath.vhd

```

-- ALU instantiation
alu : entity work.alu
port map (
  control      => alu_control,
  operand_a    => a_bus,
  operand_b    => b_bus,
  sh_result    => c_bus,
  negative_flag => alu_n_flag,
  zero_flag     => alu_z_flag);

```

Come abbiamo visto nella sezione relativa ai registri, il bus C è connesso a tutti i registri e un segnale di abilitazione per ciascuno di essi funge da abilitazione in scrittura. Infine, il bus B è implementato mediante un multiplexer, implementato usando il costrutto VHDL *with/select* (*selected assignment*):

datopath.vhd

```
with reg_to_b_control select b_bus <=
    mdr_reg      when b_ctrl_mdr,
    pc_reg       when b_ctrl_pc,
    mbr_s        when b_ctrl_mbr,
    mbr_u        when b_ctrl_mbru,
    sp_reg       when b_ctrl_sp,
    lv_reg       when b_ctrl_lv,
    cpp_reg      when b_ctrl_cpp,
    tos_reg      when b_ctrl_tos,
    opc_reg      when b_ctrl_opc,
    (others => '0') when others;
```

1.1.4 Implementazione del Protocollo con la Memoria

La lettura e la generazione del segnale di *write enable* sono implementate nello stesso process utilizzato per gli altri registri; il dato proveniente dalla memoria è considerato come un'ulteriore possibile sorgente per il dato, in aggiunta al bus C:

datopath.vhd

```
-- Processor registers
reg_proc : process(clk) is
begin

    [...]

    if c_to_reg_control(c_ctrl_mdr) = '1' then
        mdr_reg <= c_bus;
    elsif rd_ff = '1' then
        mdr_reg <= mem_data_in;
    end if;

    [...]
```

```

wr_ff <= mem_control(mem_ctrl_write);
end process;

```

I registri dell’interfaccia e il segnale di *write enable* sono dunque assegnati alle uscite dell’entity:

datapath.vhd

```

-- Output
mem_data_out  <= mdr_reg;
mem_data_addr <= mar_reg;
mem_data_we   <= wr_ff;

```

Nota 1.6

L’interfaccia istruzioni PC/MBR opera in modo simile in lettura, e non prevede operazioni di scrittura.

1.2 Le Microistruzioni

Per controllare l’unità operativa del processore, presentata in fig. 1.2, sono necessari 29 segnali:

- 9 segnali per controllare la scrittura dei dati dal bus C ai registri;
- 9 segnali per controllare quale registro è collegato al bus B e va in ingresso all’ALU;
- 8 segnali per controllare l’ALU;
- 2 segnali per abilitare lettura/scrittura sull’interfaccia MAR/MDR;
- 1 segnale per abilitare il fetch sull’interfaccia PC/MBR.

Poiché al più un registro alla volta può essere collegato al registro B, non c’è bisogno di utilizzare effettivamente 9 segnali: dato che al più uno di essi può essere asserito, le 10 configurazioni possibili possono essere codificate su $\lceil \log_2 10 \rceil = 4$ bit.

I 24 segnali individuati permettono di controllare l’unità operativa per un ciclo di clock; per completare una microistruzione, è necessario aggiungere due campi aggiuntivi, che chiamiamo Addr e JAM, che descriveremo approfonditamente quando parleremo dell’unità di controllo; per il momento, è sufficiente sapere che sono necessari a determinare la microistruzione da eseguire nel ciclo di clock successivo.

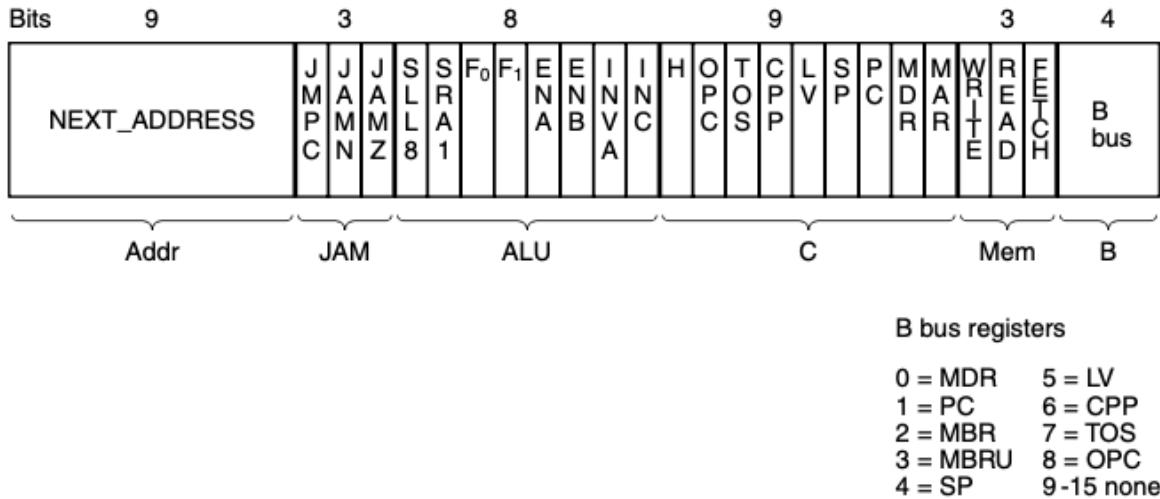


Figura 1.5: Il formato delle microistruzioni del Mic-1

In fig. 1.5 è presentato il formato che adotteremo per le microistruzioni, rappresentate su 36 bit; come anticipato, ogni microistruzione comprende i seguenti campi:

- Addr - Indirizzo di una potenziale prossima microistruzione;
- JAM - Determina come è selezionata la prossima microistruzione;
- ALU - Controllo dell'ALU e dello shifter;
- C - Controlla quali registri vengono scritti dal bus C;
- Mem - Controlla le operazioni di memoria;
- B - Seleziona il registro connesso al bus B (codifica in fig. 1.5).

1.2.1 Esempio pratico: l'istruzione IADD

Ora che abbiamo un'idea del funzionamento dell'unità operativa e della struttura di una microistruzione, possiamo analizzare più nel dettaglio il modo in cui viene eseguita un'istruzione IJVM. Consideriamo ad esempio l'istruzione IADD: quando l'abbiamo introdotta, abbiamo detto che “esegue il pop dei due elementi in cima allo stack e il push della loro somma”; questo significa che il processore, in prima analisi, dovrebbe eseguire due accessi in memoria in lettura, un'operazione di somma con l'ALU ed un accesso in memoria in scrittura. In pratica, però, l'implementazione del micropogramma Mic-1 mantiene due utili invarianti:

- il registro SP (*Stack Pointer*), al termine dell'esecuzione di un'istruzione IJVM, contiene sempre l'indirizzo dell'elemento in testa allo stack;

- il registro TOS (*Top of Stack*), al termine dell'esecuzione di un'istruzione IJVM, contiene sempre il valore dell'elemento in testa allo stack.

È dunque sufficiente un singolo accesso in memoria, e l'istruzione può essere eseguita in tre cicli di clock (i.e. tre microistruzioni); supponendo che lo stack si trovi nello stato in fig. 1.1 (b), per quanto detto in precedenza TOS contiene il valore a3; da questo stato:

1. il contenuto di SP viene decrementato di 1 e scritto sia in SP che in MAR, e si avvia l'operazione di lettura; si noti che SP punta ora alla posizione da cui deve essere letto il valore di a2 e in cui in seguito sarà scritta la somma a2 + a3;
2. il contenuto di TOS è posto nel registro H;
3. i dati contenuti in TOS e MDR (dato proveniente dalla memoria) sono sommati e posti sia in TOS che in MDR, si avvia una operazione di scrittura.

È possibile verificare manualmente che queste operazioni rispettano le invarianti enunciate in precedenza, e che il valore in cima allo stack è ora proprio la somma dei due operandi che vi erano stati posti in precedenza.

Trascurando per ora i campi Addr e JAM, siamo in grado di codificare gli altri campi delle microistruzioni:

1.
 - ALU: 110110 (decremento dell'operando B);
 - C: 000001001 (bus C scrive su SP e MAR);
 - Mem: 010 (lettura dati);
 - B: 0100 (SP controlla il bus B).
2.
 - ALU: 010100 (l'operando B passa invariato);
 - C: 100000000 (bus C scrive su H);
 - Mem: 000 (nessuna operazione di memoria);
 - B: 0111 (TOS controlla il bus B).
3.
 - ALU: 111100 (somma degli operandi A e B);
 - C: 001000010 (bus C scrive su MDR e TOS);
 - Mem: 100 (scrittura dati);
 - B: 0000 (MDR controlla il bus B).

1.2.2 Il linguaggio MAL: parte 1

Scrivere a mano le microistruzioni è perfettamente possibile, ma è molto semplice commettere errori; inoltre, il microprogramma così ottenuto sarebbe tedioso da comprendere e modificare.

DEST = H
DEST = SOURCE
DEST = \bar{H}
DEST = $\bar{\text{SOURCE}}$
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE - H
DEST = SOURCE - 1
DEST = -H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = -1

Figura 1.6: Le operazioni permesse nel linguaggio MAL

Il linguaggio usato in [1] per semplificare la scrittura del microprogramma è denominato MAL (*MicroAssembly Language*); un tool detto *microassemblatore* esegue dunque la traduzione nel formato di fig. 1.5, completamente equivalente.

Introduciamo anzitutto la notazione necessaria a specificare il controllo dei bus B e C e dell'ALU; le operazioni permesse sono riportate in fig. 1.6, dove SOURCE è uno dei registri connessi al bus B e DEST una combinazione di registri connessi al bus C, eventualmente separati dal simbolo =; ad esempio, per incrementare il contenuto del registro SP e scriverlo sia in SP che in MAR, si scrive:

```
SP = MAR = SP + 1
```

A ciascuna di queste operazioni può essere applicato uno shift a sinistra di 1 byte aggiungendo << 8, ad esempio:

```
H = MBR << 8
```

Una seconda parte della microistruzione, separata dalla precedente da un punto e virgola (;), riguarda le istruzioni di memoria; lettura e scrittura sull'interfaccia dati sono indicate rispettivamente da rd o wr, mentre la lettura sull'interfaccia istruzioni è indicata da fetch; in una singola microistruzione è possibile avere un'operazione per ciascuna delle due interfacce, eventualmente separate da ;.

A questo punto siamo in grado effettivamente di tradurre in linguaggio MAL le microistruzioni dell'istruzione IADD:

```
MAR = SP = SP - 1; rd
H = TOS
MDR = TOS = MDR + H; wr
```

Questo frammento di microprogramma è tuttavia incompleto; per capire la funzione dei campi Addr e JAM è necessario introdurre l'unità di controllo del Mic-1.

1.3 L'Unità di Controllo

In fig. 1.7 è riportato il diagramma completo del processore Mic-1, in cui oltre all'unità operativa sono riportate le componenti dell'unità di controllo.

L'unità di controllo del Mic-1 si comporta come un *sequencer*, producendo in ciascun ciclo:

1. lo stato dei segnali di controllo;
2. l'indirizzo della prossima microistruzione da eseguire.

Il microprogramma è effettivamente memorizzato in una memoria a sola lettura, interna al processore, detta *control store*; in pratica, il control store contiene le microistruzioni del microprogramma allo stesso modo in cui la memoria principale contiene le istruzioni (di livello ISA) del programma da eseguire.

Anche l'accesso al control store richiede un'interfaccia, controllata dai due registri MPC (*MicroProgram Counter*) e MIR (*MicroInstruction Register*); non è necessario un segnale di lettura, in quanto il control store è letto continuamente ad ogni ciclo.

Un'importante differenza è legata al fatto che, mentre le istruzioni del programma sono eseguite generalmente in ordine sequenziale (a meno ovviamente di istruzioni di salto), ciascuna microistruzione specifica esplicitamente l'indirizzo della successiva, riportandolo nel campo Addr.

Le microistruzioni forniscono tuttavia un supporto al controllo dei salti mediante il campo JAM; i due bit JAMN e JAMZ sono combinati con i valori dei flag N e Z dell'ALU, e permettono di modificare il bit più significativo dell'MPC:

$$\text{MPC}[8] = (\text{JAMZ AND Z}) \text{ OR } (\text{JAMN AND N}) \text{ OR } \text{Addr}[8]$$

Inoltre, se il bit JMPC è asserito, gli 8 bit meno significativi di MPC sono messi in OR bit-a-bit con gli 8 bit contenuti nel registro MBR.

1.3.1 Il linguaggio MAL: parte 2

Possiamo a questo punto introdurre le parti mancanti del linguaggio MAL; osserviamo il microcodice completo per l'istruzione IADD:

```
iadd = 0x65:  
    MAR = SP = SP - 1; rd  
    H = TOS  
    MDR = TOS = MDR + H; wr; goto main
```

Abbiamo aggiunto una label, iadd, che permette di identificare la locazione del control store, seguita dall'indirizzo nel control store a partire dal quale verrà posizionato

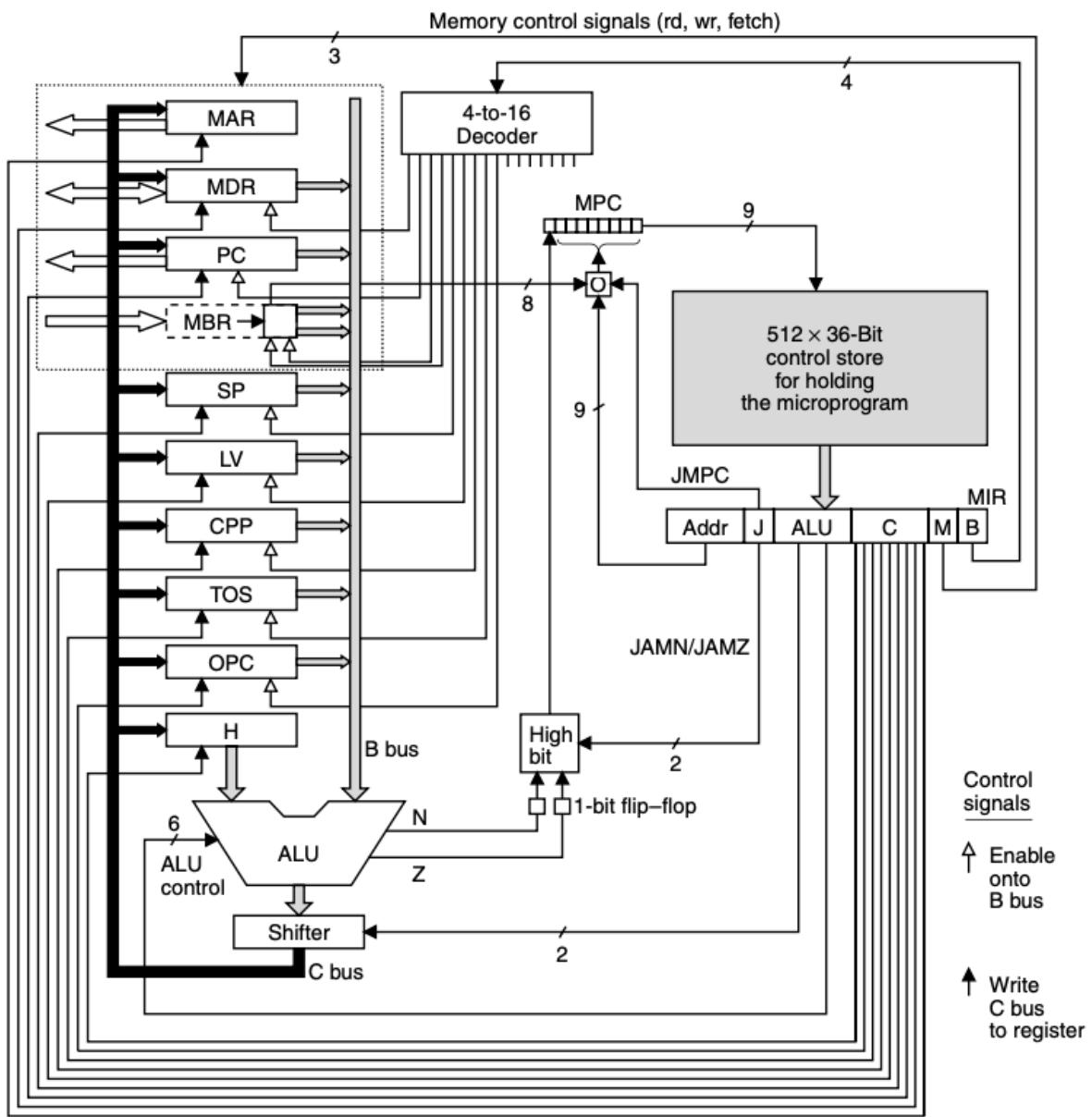


Figura 1.7: Diagramma completo del processore Mic-1

il microcodice seguente (dunque 0x65-0x67); in assenza di indicazioni esplicite, il campo Addr contiene semplicemente l'indirizzo della microistruzione successiva (dunque in questo caso, le prime due istruzioni, situate agli indirizzi 0x65 e 0x66 nel control store, avranno campo Addr posto rispettivamente a 0x66 e 0x67).

Consideriamo invece l'ultima microistruzione; abbiamo aggiunto un'ulteriore parte alla microistruzione, che influenza il suo campo Addr:

```
goto main
```

indica al programma microassemblatore, che traduce il linguaggio MAL nel formato delle microistruzioni, che il campo Addr per questa microistruzione deve essere posto uguale all'indirizzo della microistruzione che ha come label main.

Osserviamo anche il microcodice relativo alla label main:

```
main:
```

```
    PC = PC + 1; fetch; goto (MBR)
```

Questa microistruzione è particolarmente importante, in quanto ogni istruzione IJVM si conclude tornando ad essa; la sua funziona è incrementare il program counter, fare il fetch del successivo byte di istruzione e usare il contenuto del registro MBR come prossimo valore del microprogram counter; il campo

```
goto (MBR)
```

è codificato usando il bit JMPC nel campo JAM.

È chiaro da quanto detto che l'indirizzo che assegniamo ad una istruzione IJVM nel control store deve essere uguale alla sua codifica in linguaggio macchina prodotta dall'assemblatore IJVM: l'istruzione IADD è codificata come 0x65, e memorizzata proprio all'indirizzo 0x65 nel control store; quando la microistruzione main esegue il fetch del byte 0x65, l'MPC è settato a 0x65 e il microcodice relativo all'istruzione IADD viene eseguito.

Per concludere, il controllo di flusso basato sui flag dell'ALU è scritto in linguaggio MAL come:

```
if (flag) goto T; else goto F
```

dove flag può essere N o Z, mentre T e F sono due label.

1.3.2 Implementazione dell'Unità di Controllo

Il control store presenta un'interfaccia molto semplice:

```
control_store.vhd
```

```
entity control_store is
  port (
    --! Address of the desired word
    address : in ctrl_str_addr_type;
    --! Content of the addressed word
```

```
    word      : out ctrl_str_word_type
  );
end entity control_store;
```

Le microistruzioni sono inserite direttamente all'interno del codice:

control_store.vhd

Tutti i tipi sono definiti, come negli altri casi, nel file `common_defs.vhd`:

common_defs.vhd

```
--! Control store address
subtype ctrl_str_addr_type is
    std_logic_vector(ctrl_str_addr_width - 1 downto 0);
--! Control store word
subtype ctrl_str_word_type is
    std_logic_vector(ctrl_str_word_width - 1 downto 0);

[...]

--! Control store word
subtype ctrl_str_word_type is
    std_logic_vector(ctrl_str_word_width - 1 downto 0);

[...]

--! Control store content
type ctrl_str_type is array (ctrl_str_words - 1 downto 0)
    of ctrl_str_word_type;
```

L'implementazione della control unit è molto semplice: un process aggiorna il registro MIR con la microistruzione corrente, i cui bit del campo Addr, opportunamente mascherati, costituiscono il registro “virtuale” MPC:

control_unit.vhd

```
-- Registers
reg_proc : process(clk) is
begin
    if rising_edge(clk) then
        if reset = '1' then
            mir_reg <= "000000001000000000000000000000000001001";
            n_ff    <= '0';
            z_ff    <= '0';
        else
            mir_reg <= control_store_word;
            n_ff    <= alu_n_flag;
            z_ff    <= alu_z_flag;
        end if;
    end if;
end process reg_proc;

-- MPC virtual register
ctrl_nxt_addr_no_msb <=
    → mir_reg(ctrl_nxt_addr_no_msb_type'range);
jmpc_addr <= ctrl_nxt_addr_no_msb or mbr_reg_in when
    → mir_reg(ctrl_jmpc) = '1' else ctrl_nxt_addr_no_msb;
high_bit <= (alu_n_flag and mir_reg(ctrl_jamn)) or
    → (alu_z_flag and mir_reg(ctrl_jamz));
mpc_virtual_reg <= (mir_reg(ctrl_nxt_addr_msb) or
    → high_bit) & jmpc_addr;
```

Il decoder per la selezione del registro che controlla il bus B è implementato come un process puramente combinatorio:

control_unit.vhd

```
-- B_BUS control decoder
reg_to_b_decoder : process(mir_reg(ctrl_b'range)) is
begin
    reg_to_b_decoder_out <= (others => '0');
```

```

if unsigned(mir_reg(ctrl_b'range)) < b_ctrl_width then
    reg_to_b_decoder_out(to_integer(
        unsigned(mir_reg(ctrl_b'range)))) <= '1';
end if;
end process reg_to_b_decoder;

```

1.3.3 Collegamento tra UO e UC

Siamo finalmente in grado di definire le entity relative all’unità operativa e all’unità di controllo.

Entrambe le unità devono ricevere:

- un segnale di clock;
- un segnale di reset.

Questi segnali sono comuni.

L’unità di controllo fornisce all’unità operativa i segnali relativi:

- al controllo dell’ALU;
- al controllo del bus C;
- al controllo del bus B;
- al controllo della memoria.

Riceve invece da essa:

- i flag dell’ALU;
- il contenuto del registro MBR.

control_unit.vhd

```

entity control_unit is
  port (
    --! Clock
    clk : in std_logic;
    --! Synchronous active-high reset
    reset : in std_logic;
    --! Content of the MBR register
    mbr_reg_in : in mbr_data_type;
    --! ALU negative flag

```

```

    alu_n_flag      : in  std_logic;
    --! ALU zero flag
    alu_z_flag      : in  std_logic;
    --! Control signals for the ALU
    alu_control     : out alu_ctrl_type;
    --! Control signals for the C bus
    c_to_reg_control : out c_ctrl_type;
    --! Control signals for memory operations
    mem_control     : out mem_ctrl_type;
    --! Control signals for the B bus
    reg_to_b_control : out b_ctrl_type
  );
end entity control_unit;

```

In aggiunta a questi segnali, ci aspettiamo di trovare nell'entity dell'unità operativa anche i porti relativi all'interfaccia con la memoria:

- indirizzo dati;
- ingresso dati;
- uscita dati;
- write enable dati;
- indirizzo istruzioni;
- ingresso istruzioni.

datapath.vhd

```

entity datapath is
  port (
    --! Clock
    clk           : in  std_logic;
    --! Synchronous active-high reset
    reset         : in  std_logic;
    --! Control signals for the ALU
    alu_control   : in  alu_ctrl_type;
    --! Control signals for the C bus
    c_to_reg_control : in  c_ctrl_type;
    --! Control signals for memory operations
    mem_control   : in  mem_ctrl_type;

```

```

--! Control signals for the B bus
reg_to_b_control : in b_ctrl_type;
--! Content of the MBR register
mbr_reg_out      : out mbr_data_type;
--! ALU negative flag
alu_n_flag       : out std_logic;
--! ALU zero flag
alu_z_flag       : out std_logic;
--! Memory data write enable
mem_data_we     : out std_logic;
--! Port for memory data read
mem_data_in     : in reg_data_type;
--! Port for memory data write
mem_data_out    : out reg_data_type;
--! Memory address for memory data operations
mem_data_addr   : out reg_data_type;
--! Port for memory instruction read
mem_instr_in    : in mbr_data_type;
--! Memory address for memory instruction fetch
mem_instr_addr  : out reg_data_type
);
end entity datapath;

```

Infine, introduciamo il componente processor all'interno del quale sono istanziati sia il datapath che la control_unit. Tutte le coppie di porti corrispondenti sono connesse internamente, mentre sono esposti quelli relativi:

- al clock;
- al reset;
- all'interfaccia con la memoria.

processor.vhd

```

entity processor is
port (
  --! Clock
  clk           : in std_logic;
  --! Synchronous active-high reset
  reset         : in std_logic;
  --! Memory data write enable
  mem_data_we   : out std_logic;

```

```
--! Port for memory data read
mem_data_in      : in  reg_data_type;
--! Port for memory data write
mem_data_out     : out reg_data_type;
--! Memory address for memory data operations
mem_data_addr   : out reg_data_type;
--! Port for memory instruction read
mem_instr_in    : in  mbr_data_type;
--! Memory address for memory instruction fetch
mem_instr_addr  : out reg_data_type
);
end entity processor;
```

2 La Toolchain Mic-1

2.1 Installazione

Per installare il microassemblatore `mal` e l'assemblatore `ajvm` è anzitutto necessario installare il version control system Git ed il build system Gradle; inoltre, per alcuni task useremo il build system Cmake. La procedura può variare leggermente a seconda del sistema operativo usato, ma in una tipica distribuzione Linux Debian-based (come Ubuntu) è sufficiente eseguire:

```
$ sudo apt install git gradle cmake
```

Conviene dunque creare una nuova cartella per i repository da scaricare, e posizionarvi; al suo interno, creeremo anche una directory per i tool compilati:

```
$ mkdir esercitazione_mic
$ cd esercitazione_mic
$ mkdir tools
```

È possibile a questo punto compilare i tool (ignorando eventuali warning in fase di test):

```
$ git clone https://github.com/albmoriconi/mal.git
$ cd mal
$ ./gradlew build
$ cd ..
$ git clone https://github.com/albmoriconi/ajvm.git
$ cd ajvm
$ ./gradlew build
$ cd ..
$ tar -xvf mal/build/distributions/mal.tar -C tools/
$ tar -xvf ajvm/build/distributions/ajvm.tar -C tools/
```

Per poter usare i tool nella shell attiva, le loro directory vanno aggiunte al path di

ricerca dei binari; può essere utile a questo scopo creare un file `settings.sh` con il seguente contenuto:

```
export PATH="$HOME/esercitazione_mic/tools/ajvm/bin:$HOME/
esercitazione_mic/tools/mal/bin:$PATH"
```

Sarà dunque possibile configurare correttamente l'ambiente eseguendo semplicemente il comando:

```
$ source settings.sh
```

Infine, è possibile eseguire il clone del repository contenente il codice VHDL del processore, e predisporne la directory per la build:

```
$ git clone https://github.com/albmoriconi/amic-0.git
$ cd amic-0
$ mkdir build
$ cd build
$ cmake ..
```

2.2 Struttura del Progetto

Da qui in avanti, ogni volta che faremo riferimento a un percorso sarà rispetto alla directory in cui è stato scaricato il repository `amic-0`.

La directory principale del repository è organizzata come segue:

- `src` - codice sorgente
 - `main` - implementazione (subdirectory ordinate per linguaggio)
 - `test` - testbench (subdirectory ordinate per linguaggio)
- `util` - utility necessarie per il flow di sviluppo (subdirectory ordinate per linguaggio)
- `build` - contiene i file prodotti dal build system (questa directory deve essere creata manualmente)

Per il momento non ci interessano le altre directory e file presenti; è importante tuttavia notare che ogni directory che contiene codice (eventualmente nelle subdirectory) contiene anche un file `CMakeLists.txt` che serve al build system per tenere traccia dei sorgenti e aggiungerli ai target opportuni.

2.3 Flow di Sviluppo

Per utilizzare il flow di sviluppo da linea di comando è necessario installare due tool aggiuntivi: il simulatore ghdl e il visualizzatore di forme d'onda gtkwave:

```
$ sudo apt install ghdl gtkwave
```

Dalla directory build è ora possibile lanciare il tool make con una serie di target utili per lanciare microassemblatore, assemblatore e alcuni test.

In particolare, è possibile editare il microprogramma in linguaggio MAL e il programma in linguaggio IJVM situati rispettivamente nelle directory `src/main/mal` e `src/main/ajvm` e dunque rigenerare control store e RAM lanciando, dalla directory build, i target:

```
$ make create_control_store  
$ make create_ram
```

Il target `check` lancia invece i testbench contenuti nella directory `src/test/vhdl`:

```
$ make check
```

Attenzione 2.1

Modificando il programma contenuto nella RAM, il testbench `processor_tb.vhd` potrebbe fallire in quanto verifica il risultato prodotto dal programma precaricato; due possibili soluzioni sono:

- modificare le condizioni alle righe 97-100 in modo da verificare le nuove condizioni sul risultato; oppure
- più semplicemente, si possono rimuovere le stesse righe; in questo caso, se si usa il simulatore GHDL, sarà necessario terminare manualmente l'esecuzione del tasto (`Ctrl + C`) o modificare gli argomenti a linea di comando per specificare la durata del test.

Un esempio dei risultati della suite di test è riportato in fig. 2.1. L'esecuzione dei test produce anche le waveform relative alle esecuzioni dei testbench, come quella riportata in fig. 2.2, che possono essere trovate nella directory `build/src/test/vhdl` e visualizzate utilizzando il tool `gtkwave`.

```

make — tmux — 100x30
Built target src.test.vhdl.processor_tb
Scanning dependencies of target src.test.vhdl.alu_tb
analyze /Users/albmoriconi/Progetti/amic-0/src/test/vhdl/alu_tb.vhd
elaborate alu_tb
Built target src.test.vhdl.alu_tb
Scanning dependencies of target src.test.vhdl.control_unit_tb
analyze /Users/albmoriconi/Progetti/amic-0/src/test/vhdl/control_unit_tb.vhd
elaborate control_unit_tb
Built target src.test.vhdl.control_unit_tb
Scanning dependencies of target src.test.vhdl.datapath_tb
analyze /Users/albmoriconi/Progetti/amic-0/src/test/vhdl/datapath_tb.vhd
elaborate datapath_tb
Built target src.test.vhdl.datapath_tb
Scanning dependencies of target check
Test project /Users/albmoriconi/Progetti/amic-0/build
  Start 1: src.test.vhdl.alu_tb ..... Passed  0.37 sec
  Start 2: src.test.vhdl.control_unit_tb .... Passed  0.19 sec
  Start 3: src.test.vhdl.datapath_tb ..... Passed  0.18 sec
  Start 4: src.test.vhdl.processor_tb ..... Passed  0.21 sec

100% tests passed, 0 tests failed out of 4

Total Test time (real) =  0.95 sec
Built target check
[-P/a/build] [0] 1:make* 2:fish- (master|✓)
19:13 25-Nov-19

```

Figura 2.1: Risultati dei testbench

Attenzione 2.2

Il comando `make check` potrebbe non funzionare con tutte le versioni di GHDL e CMake. Qualora si riscontrassero errori nell'esecuzione dei test con questa modalità, è sufficiente importare manualmente i sorgenti .vhd contenuti nella directory `amic-0/src/main/vhdl` e il testbench `amic-0/src/test/vhdl/processor_tb.vhd` in un progetto ISE o Vivado, e utilizzare il simulatore Xilinx.

Per aggiungere altri testbench alla suite è sufficiente salvarli nella directory `test/vhdl` e modificare il file `test/vhdl/CmakeLists.txt` aggiungendo alla macro `add_vhdl_test_sources` un'entry relativa al nuovo testbench.

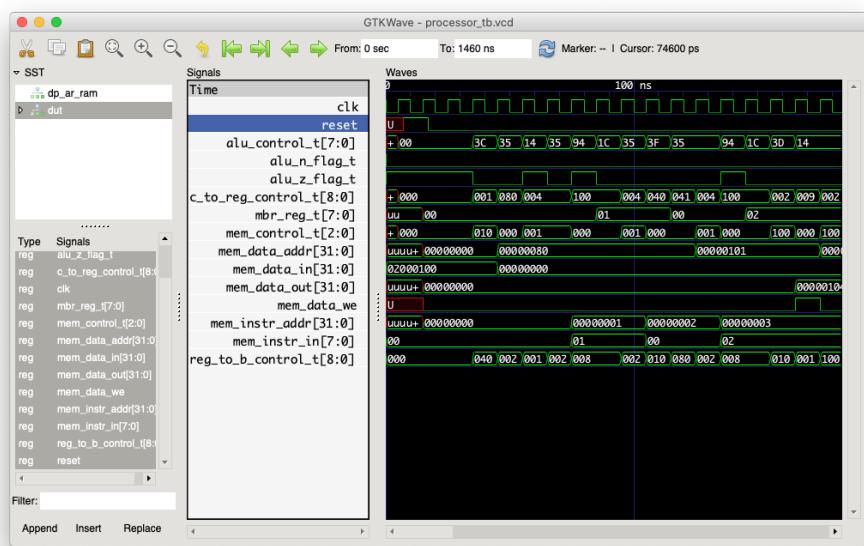


Figura 2.2: Waveform del testbench processor_tb.vhd

Bibliografia

- [1] G. Conte, A. Mazzeo, N. Mazzocca, P. Prinetto *Architettura dei calcolatori*, CittàStudi Edizioni, 2015.
- [2] A. S. Tanenbaum, T. Austin *Structured Computer Organization*. Pearson, 2013.
- [3] N. Mazzocca, A. Moriconi *Il processore MIC-1*.
- [4] Digilent Romania *RS232 Reference Component*.