POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Politecnico
di Torino

Tesi di Laurea Magistrale

Analisi di codici open-source per la sintesi di alberi di distribuzione del clock.

Relatori:

Ch.mo Prof. Mario Roberto CASU

Ch.mo Prof. Maurizio MARTINA

Ch.mo Ing. Alessandro DE LAURENZIS

Candidato:

Vito D'AVINO

Luglio 2022

Sommario

L'aumento della densità dei componenti e dell'area dei circuiti ha reso la sintesi del circuito di clock un'operazione critica nel flusso di implementazione fisica. Per distribuire opportunamente il segnale del clock ad un numero elevato di terminali è necessario effettuare un'ottimizzazione multi-obiettivo, che generi un circuito robusto e conforme ai vincoli fisici di progetto e dove possibile rispetti i limiti di temporizzazione del circuito e di potenza totale dissipata.

Questo lavoro di tesi è incentrato sullo studio degli algoritmi di sintesi di alberi di clock. Dopo aver formulato il problema e descritto le principali figure di merito che caratterizzano i risultati è stato analizzato lo stato dell'arte degli algoritmi], concentrando lo studio sulle tecniche volte a ottimizzare le risorse impiegate.

In particolare, sono state prese in esame le due versioni di TritonCTS, un programma open-source nato con l'obiettivo di individuare la soluzione ideale in termini di consumo di potenza dell'albero, dati dei valori limiti di skew e latenza.

La prima versione introduce il concetto di albero H generalizzato: una topologia ad albero formata da più livelli ortogonali fra loro con un fattore di diramazione per ogni livello arbitrario ed ottimizzato sulle caratteristiche del circuito. L'elevata complessità dell'algoritmo di costruzione topologica implementato, unita alle API di terze parti non aggiornate determina la saturazione della memoria RAM disponibile, determinando il fallimento della sintesi. Tale problema è stato analizzato e descritto nel capitolo 3.

TritonCTS 2.0 è la versione attualmente impiegata nel progetto OpenROAD per la creazione di un flusso di implementazione completamente automatizzato e open-source. A partire dai file sorgente è stato analizzato e descritto nel capitolo 3 il nuovo approccio di sintesi implementato che mira ad essere flessibile e poco oneroso a livello computazionale.

Sono state sintetizzate le strutture di distribuzione del clock per diversi circuiti con numero di componenti sequenziali e relativa distribuzione molto diversi tra loro.

A seguito dell'analisi delle strutture risultanti, i file sorgente sono stati parzialmente modificati, per aggiungere funzionalità volte a minimizzare il sovraccarico capacitivo dei ripetitori e delle interconnessioni dell'albero, riportando i miglioramenti ottenuti nel capitolo 4.

Infine, i risultati della versione modificata di TritonCTS 2.0 sono stati comparati con quelli di CCOpt di Innovus. È necessario sottolineare che è stata opportunamente disabilitata l'ottimizzazione concorrenziale del timing dei percorsi di dato svolta da CCOpt, funzionalità assente nei sintetizzatori di "prima generazione" come TritonCTS, al fine di effettuare un'equa comparazione.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

1.1 Scopo

La complessità progettuale dei circuiti integrati è aumentata esponenzialmente con lo scaling tecnologico, richiedendo una continua innovazione degli strumenti di sintesi. Ciò è particolarmente evidente se si osserva l'evoluzione della sintesi della rete di clock divenuta allo stato dell'arte una fase cruciale del design fisico da cui dipendono le performance del chip.

L'impiego di dispositivi di memoria nelle architetture VLSI è in costante aumento, soprattutto a causa della crescente complessità algoritmica delle funzioni svolte e dell'utilizzo di strutture con molti stadi di pipeline. L'andamento tecnologico evidenzia un incremento dell'area occupata dai circuiti con una conseguente crescita della lunghezza delle interconnessioni e quindi delle capacità ad esse associate. Pertanto il progetto della rete di distribuzione di una net a fan-out così elevato come quella di clock deve necessariamente rispettare i vincoli di temporizzazione richiesti dal sistema, ma non può prescindere dalla riduzione dei consumi di potenza.

In quest'ottica è stato analizzato lo stato dell'arte degli algoritmi per la sintesi di circuiti di distribuzione del segnale di clock a ridotto consumo di potenza. L'attenzione è stata poi concentrata sul progetto open-source TritonCTS, nato con l'obiettivo di individuare la soluzione ideale in termini di consumo di potenza dell'albero, dati dei valori limiti di skew e latenza.

1.2 Struttura

Il lavoro di tesi è stato strutturato come segue

- Nel Capitolo 2 sono stati coperti gli aspetti teorici della sintesi dell'albero di clock ed è stata effettuata un'analisi bibliografica delle tecniche note allo stato dell'arte.
- Nel Capitolo 3 è stato descritto il funzionamento di TritonCTS in entrambe le sue versioni.
- Nel Capitolo 4 sono descritte le modifiche apportate all'algoritmo di TritonCTS 2.0 e sono stati analizzati i risultati delle sintesi di diversi design.
- Nel Capitolo 5 è stato effettuato un confronto con i risultati di Innovus.

Capitolo 2

Sintesi dell'albero di Clock

2.1 Cenni sull'implementazione fisica

L'ultimo stadio del flusso di progetto VLSI [1] è l'implementazione fisica, in cui a partire dal circuito descritto al *Register Transfer Level* (RTL) si ricava una disposizione ottimale delle celle e delle interconnessioni.

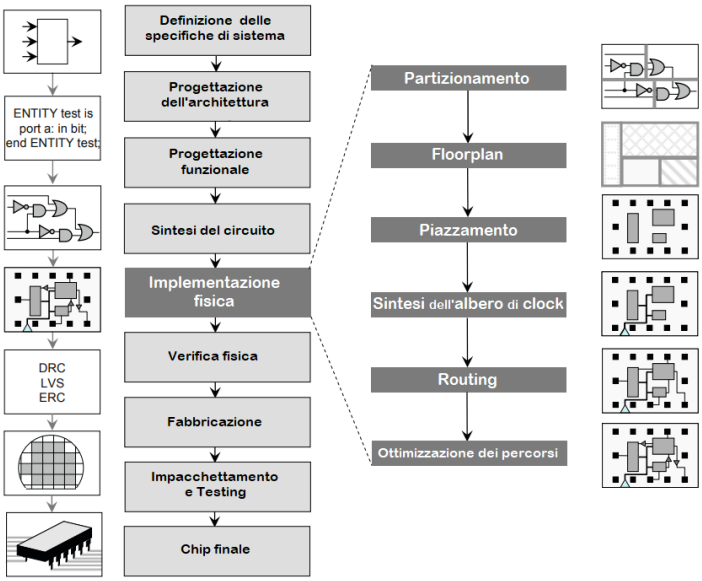

Figura 2.1: Flusso VLSI con dettaglio del Design Fisico.

Come evidenziato in figura 2.1, il design fisico inizia con la fase di *Partizionamento*, dove il sistema è suddiviso in macro-moduli isolando le regioni logiche di funzionamento entro spazi prefissati con l'obiettivo di minimizzare il numero totale

di interconnessioni tra macro diverse. Segue il *Floorplan*, in cui si individua la disposizione ottimale delle partizioni per minimizzare la lunghezza delle interconnessioni ed evitare problemi di congestione, per poi procedere all'effettiva disposizione ed orientamento di ogni cella standard effettuata nell'operazione di *Piazzamento*. La Sintesi dell'albero di Clock (CTS) ha come obiettivo il collegamento della sorgente del segnale di clock con tutti i componenti sequenziali, di gating o di test che lo richiedono, denominati *sink*, e opera sulle informazioni descritte dagli algoritmi delle fasi precedenti nei file Design Exchange Format (".def") e nei file HDL, con la possibilità di aggiungere elementi (quali i buffer di clock) o di ottimizzare la posizione dei componenti sequenziali.

2.2 Figure di Merito

2.2.1 Skew

Lo *skew* è la differenza del tempo di arrivo del segnale di clock su due diversi sink. Due registri si dicono correlati o adiacenti quando sono collegati lungo il loro percorso dati. La differenza nel tempo di arrivo del fronte dei clock su due sink correlati è definita *skew locale* ed influisce sulla durata del minimo e del massimo percorso combinatorio che è possibile inserire tra loro. Un sistema a skew locale minimizzato è il risultato ideale di una CTS, ma la risoluzione diretta di questo problema richiede una complessità algoritmica elevatissima e non ha ancora una soluzione. Infatti, oltre all'instradamento delle interconnessioni di un nodo ad alto fan-out, è necessario analizzare il percorso combinatorio che separa i componenti sequenziali per individuare ogni coppia di sink correlati, definendo al minimo un insieme di N relazioni per N sink e tenere conto di ognuna di queste coppie nel bilanciamento dei percorsi del segnale di clock. Per ovviare a questo problema durante la CTS si ottimizza lo *skew globale*, ovvero la differenza tra il massimo ed il minimo ritardo combinatorio del percorso di clock verso due elementi sequenziali qualsiasi, anche non adiacenti. Sebbene questo parametro sia più facile da misurare e tenere in conto, il suo annullamento comporta un alto costo in termini di interconnessioni e ripetitori impiegati. Consideriamo in prima approssimazione l'associazione del minimo e del massimo ritardo del segnale di clock al collegamento rispettivamente al sink più vicino e a quello più lontano dalla sorgente. Per bilanciare questi due percorsi, dovremo necessariamente inserire tratti di interconnessione e ripetitori ridondanti nel percorso più breve, definendo un percorso a lunghezza non minima con impiego di risorse che determina maggiore area occupata e potenza dissipata. Per questo motivo, al crescere del numero di sink e dell'area del *die*, si è passati dal definire strutture a "zero skew" verso soluzioni a skew limitato, accettando tempi di arrivo differenti tra i vari sink, soprattutto se

non correlati, beneficiando di una struttura meno complessa per la rete di clock e di un minor picco di potenza istantaneo grazie ai diversi istanti di commutazione.

2.2.2 Latenza

Con latenza del segnale di clock si intende il ritardo misurato a partire dalla sorgente fino ad un sink. Questa figura di merito, al pari del numero massimo di buffer presenti lungo un percorso, fornisce informazioni sulle dimensioni e sulla profondità della rete di distribuzione del segnale. La minimizzazione di questo valore determina una riduzione della sensibilità alle variazioni di processo e ambientali dei vincoli di temporizzazione, di conseguenza è importante che l'algoritmo di CTS misuri questo parametro e ne tenga conto nella scelta della soluzione topologica migliore. Di contro, è impossibile conoscere con esattezza il ritardo in questa fase, perché se i parametri resistivi dipendono esclusivamente dalla geometria dell'interconnessione stessa, il valore della capacità dipende anche dalle linee di metallo circostanti che ancora non sono state disposte. Di conseguenza per guidare l'algoritmo sono sufficienti dei modelli che offrono una risposta in tempi brevi anche se con un'accuratezza inferiore.

La rappresentazione di lunghi tratti di interconnessione è effettuata componendo tratti di linea di lunghezza unitaria ed utilizzando la relativa caratterizzazione presente nei file tecnologici. Trascurando in questa trattazione i componenti induttivi, i modelli a parametri concentrati utilizzati sono riportati in figura 2.2. Al modello a L è associata l'accuratezza peggiore, di conseguenza si utilizza una rappresentazione a T o ancora meglio a π (che definisce una quantità inferiore di nodi intermedi) utilizzando rispettivamente un valore di resistenza e di capacità pari ad $\frac{1}{2}$ di quello caratteristico del tratto di linea unitario.

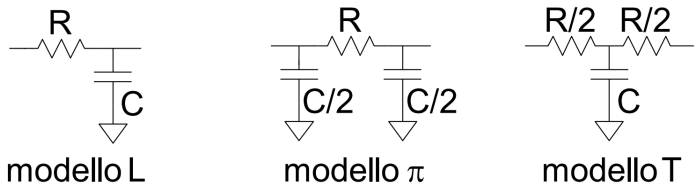

Figura 2.2: Modelli a parametri concentrati.

Definito il modello dell'interconnessione, è necessario stabilire un modello per il calcolo del ritardo complessivo di una linea. Quello a cui è associato il minor costo computazionale è quello lineare, dove il ritardo è direttamente proporzionale alla lunghezza dell'interconnessione ed è quindi pari alla semplice somma dei ritardi dei blocchi unitari. Per quanto semplice, questo modello è molto impreciso ed è valido soprattutto per tecnologie poco scalate in cui la resistenza delle interconnessioni

ha un valore trascurabile. Quando non è possibile trascurare il contributo resistivo, il ritardo dell'interconnessione varia con una relazione quadratica rispetto alla lunghezza.

Consideriamo il modello in figura 2.3 di un'interconnessione a scala RC, assumendo di applicare un segnale a gradino in ingresso e che ogni capacità sia inizialmente scarica. Ogni capacità C_i è riferita a massa ed è separata dal driver da una resistenza complessiva crescente con i, determinando N poli diversi. Sommando in T_P tutte le costanti di tempo del circuito come in (2.1).

$$T_P = R_1C_1 + (R_1 + R_2)C_2 + ... + (R_1 + R_2 + ... + R_N)C_N \tag{2.1}$$

La quantità T_P è la risposta di primo ordine all'impulso ed è denominata ritardo di Elmore. Questo valore tipicamente differisce dal ritardo effettivo misurato ai terminali di un'interconnessione, essendo un'approssimazione a singolo polo dominante, ma offre una valutazione pessimistica del suo valore e può essere usata per stimare il ritardo massimo delle interconnessioni in fase di CTS.

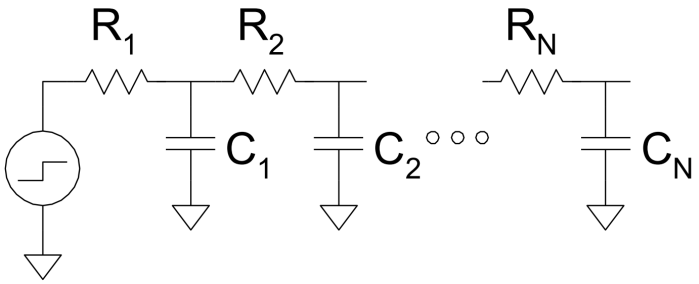

Figura 2.3: Modello circuitale di un'interconnessione a scala RC.

2.2.3 Tempo di Transizione

Considerando una transizione dal valore logico basso V_{OL} verso quello alto V_{OH}, definiamo il tempo di transizione di un nodo, detto anche slew, come il tempo richiesto per il passaggio del valore di tensione dal 10% (o 20%) al 90% (o 80%) dell'escursione complessiva ($\Delta V = V_{OH} - V_{OL}$). Nello specifico questo è anche conosciuto come "tempo di salita", mentre analogamente nel caso di un transizione da V_{OH} verso V_{OL} si considera il tempo di transizione o "di discesa" come l'intervallo di tempo impiegato per passare dal 10% (o 20%) al 90% (o 80%) di ΔV. Considerando il modello espresso in (2.1) una lunga interconnessione è composta da diversi nodi, ognuno con una costante di tempo τ crescente in funzione della distanza dal driver che determina una degradazione del segnale e un tempo di commutazione più elevato. Un elevato tempo di transizione sul nodo di ingresso di

un dispositivo in logica CMOS causa un largo intervallo temporale in cui entrambe le reti complementari sono accese, aumentando il contributo della potenza di corto-circuito. Di contro, un ridotto slew introduce un forte rumore di crosstalk per le interconnessioni adiacenti e richiede l'inserimento di diversi buffer, anche di grandi dimensioni, aumentando l'area occupata, il valore di capacità totale della linea ed il consumo di potenza. Valori di slew accettabili si aggirano tipicamente intorno al $10\% - 20\%$ del periodo di clock.

2.2.4 Design Rule Constraints

I file *liberty* nello Standard IEEE contengono informazioni tabellate delle caratteristiche funzionali, elettriche, di temporizzazione e potenza delle celle di una tecnologia. La caratterizzazione di celle in tecnologie non fortemente scalate può essere effettuata secondo il modello del ritardo non lineare (*NLDM*), misurando la risposta al variare del tempo di transizione del segnale di ingresso e del carico capacitivo applicato in uscita. Ogni file liberty contiene una caratterizzazione tecnologica effettuata in determinate condizioni operative riguardanti le variazioni di Processo, la tensione di alimentazione (V) e la Temperatura che formano il punto o *corner PVT*. Il risultato di queste caratterizzazioni è un insieme discreto di punti da cui è possibile ricavare il comportamento della cella in tutte le condizioni, con un grado di approssimazione contenuto e associato al passo dello *sweep* delle variabili indipendenti (tempo di transizione in ingresso e capacità in uscita). Di contro, per valori che eccedono i limiti superiori ed inferiori testati in fase di caratterizzazione non è possibile ottenere un risultato accurato tramite estrapolazione, di conseguenza in ogni fase di design è necessario rispettare questi vincoli, denominati *Design Rule Constraint* (DRC), per ottenere un comportamento predicibile. Esistono DRC legate al valore minimo applicabile, ma i vincoli più rilevanti da rispettare sono:

- *max_fanout*, ovvero il massimo numero di gate che è possibile collegare in uscita ad un componente;
- *max_capacitance*, ovvero la massima capacità di carico applicabile sull'uscita di una cella;
- *max_transition* descritto più nel dettaglio nel paragrafo precedente.

In fase di progetto è possibile modificare le DRC inserendo esclusivamente valori più stringenti rispetto a quelli fissati dalla tecnologia.

2.2.5 Tempo di Setup e di Hold

In riferimento ad un flip flop, definiamo tempo di setup t^S l'intervallo in cui il segnale di Dato deve essere stabile prima del fronte attivo di clock, mentre il tempo

di hold t^H è minimo intervallo di tempo successivo al fronte attivo di clock in cui il valore del Dato deve essere mantenuto costante. Entrambi sono parametri caratteristici delle celle dei FF di una data libreria e non possono essere modificati in fase di implementazione. La mancata stabilità del segnale di ingresso del FF durante questi due intervalli di tempo può determinare la cattura di un valore errato o una condizione di metastabilità del FF.

Consideriamo un circuito composto da due generici flip flop FF_i e FF_j adiacenti. Il vincolo dell'operazione di CTS è di soddisfare le equazioni (2.2) e (2.3) mantenendo un tempo di slack (in cui non è svolta alcuna attività) $T_{slack} \geq 0$.

Riguardo l'equazione (2.2), il periodo di clock T deve essere maggiore della differenza del tempo di arrivo del segnale di clock sui due FF $t_j - t_i$ (ovvero dello skew), del tempo di propagazione del primo FF t_{ij}^{CQ}, del massimo tempo di propagazione della logica combinatoria tra i due dispositivi t_{ij}^{max} e delle incertezze dovute alle variazioni di processo δ_i e δ_j. Far fronte a violazioni di setup in fase di CTS richiede o l'aumento del periodo di clock, causando un peggioramento delle prestazioni dell'intero integrato, o la minimizzazione dello skew tra FF_i e FF_j.

$$T_{slack}^{setup_{ij}} = (t_j - t_i) + T - t_{ij}^{CQ} - t_{ij}^{max} - \delta_i - \delta_j - t_j^{setup} \tag{2.2}$$

Per quanto riguarda il vincolo di hold è evidente dalla (2.3) che questi è indipendente dal periodo di clock T e per far fronte a questo tipo di violazioni è necessario modificare o l'albero di clock o il percorso del dato.

$$T_{slack}^{hold_{ij}} = (t_i - t_j) + t_{ij}^{CQ} + t_{ij}^{min} - \delta_i - \delta_j - t_j^{hold} \tag{2.3}$$

2.2.6 Potenza

La potenza totale dissipata in un circuito integrato si divide in due contributi: la potenza statica dissipata costantemente per alimentare ogni cella e la potenza dinamica dissipata durante le commutazioni. La prima è principalmente legata alla corrente di sotto-soglia che scorre nei dispositivi quando questi sono spenti, alle correnti di dispersione associate alle giunzioni p-n inversamente polarizzate e a quelle associate ad effetti di canale corto tipici di tecnologie fortemente scalate. Il consumo di potenza statico è direttamente proporzionale al numero e alle dimensioni delle celle implementate e al valore degli ingressi nel caso di porte con più transistori in serie.

La potenza dinamica è definita come la potenza necessaria per caricare e scaricare i nodi intermedi e di uscita della rete. Analiticamente il suo valore è pari a $P_d = \alpha f C_L V_{DD}^2$ dove α denominata *Switching Activity* è la probabilità che il nodo commuti in un periodo di clock, f è la frequenza operativa, C_L è la capacità totale di carico mentre V_{DD} è la tensione di alimentazione. Un ulteriore contributo di

potenza dissipata che rientra nella potenza dinamica è la potenza di cortocircuito, legata alla corrente istantanea che scorre dall'alimentazione verso massa ad ogni commutazione delle porte logiche, attraverso il canale formato dalla contemporanea apertura sia della rete di pull-up che di quella di pull-down. Il suo valore è direttamente proporzionale al tempo di transizione del segnale posto in ingresso alla porta logica, di conseguenza è inversamente proporzionale alla dimensione del *driver*, con una tendenza opposta al caso della potenza statica.

2.3 Formulazione del problema

L'algoritmo di CTS opera su di un set di terminali, in cui distinguiamo la sorgente s_0 e l'insieme dei sink $S = s_1, s_2, ..., s_n$ a cui sono associate le rispettive posizioni nel piano ed un insieme di buffer e livelli di metallizzazione utilizzabili, con le relative librerie di caratterizzazione. In alcuni algoritmi è possibile definire anche una serie di vincoli, sia fisici, come ad esempio DRC più stringenti di quelli espressi nelle librerie, che geometrici, fornendo un set di ostruzioni dove sono contenute informazioni su parti del circuito in cui non è possibile piazzare buffer o interconnessioni.

La soluzione G è l'insieme di nodi intermedi (intesi sia come buffer che come semplici diramazioni) e di linee che connette tutti gli end-point alla sorgente che consente l'opportuna distribuzione del segnale di temporizzazione. Ogni arco della soluzione e è caratterizzato da un costo in termini di lunghezza L_e e capacità, mentre ogni nodo intermedio u, per definizione, riceve un solo ingresso e genera più di un filo di uscita, fino al limite superiore del *max_fanout*. Le funzioni obiettivo più comunemente utilizzate mirano a minimizzare skew e latenza, mentre allo scopo di ridurre il consumo di potenza è opportuno affiancare alla minimizzazione della latenza massima la riduzione della somma totale dei costi di ogni linea e della capacità totale dell'albero.

Il corpo dell'algoritmo può essere diviso in due fasi: inizialmente si definisce la topologia della rete, specificando il numero di nodi intermedi e le relative ramificazioni, e successivamente si procede al posizionamento di ogni tratto di interconnessione e buffer.

2.4 Algoritmi di costruzione ad albero

La soluzione topologica più diffusa e da cui prende il nome questa fase di design, è quella ad albero. Dalla teoria dei grafi[1], si definisce albero una struttura in cui due vertici sono connessi da uno e un solo cammino. In questa struttura la sorgente prende il nome di radice (*root*) e, considerando il caso di un albero orientato, è possibile definire per ogni arco del grafo orientato un nodo padre dal quale nasce la congiungente ed un nodo figlio che la riceve. Ogni nodo intermedio è sia un nodo

padre che figlio, mentre tutti i terminali, da cui non parte nessun collegamento, sono denominati foglie (*leaves*). Le strutture ad albero possono essere classificate in base al numero di archi figli per ogni nodo e tra le diverse classi ricordiamo la topologia binaria, con la definizione di due figli per ogni nodo intermedio. È possibile definire il livello di ogni nodo in maniera gerarchica, come il livello del relativo nodo padre incrementato di 1, associando alla radice il livello nullo. Da qui possiamo definire anche la profondità dell'albero, pari al massimo livello e quindi pari al numero massimo di archi presenti lungo un percorso. Se tutte le foglie sono allo stesso livello di profondità e quindi il numero minimo di archi coincide con il numero massimo, l'albero si definisce bilanciato.

Si evidenzia che tutte le soluzioni prevedono l'inserzione di invertitori o buffer che rigenerino il segnale. Inoltre, dove possibile, è stata riportata la complessità computazionale degli algoritmi, utilizzando la notazione "$O()$" ("o" grande), che indica la proporzionalità del numero di operazioni da svolgere in funzione del numero di ingressi n.

2.4.1 Albero ad H

La creazione di un albero ad H [2] è l'approccio più semplice per generare una struttura che mira a minimizzare lo skew sfruttando la simmetria della topologia. In questo algoritmo si posiziona il buffer sorgente nel centro geometrico dall'area del circuito integrato, per poi dividere il piano in 4 settori identici di forma rettangolare. A partire dalla radice si tracciano due percorsi lungo uno dei due assi del piano, con direzione opposta tra loro e lunghezza pari ad $\frac{1}{4}$ della lunghezza dell'asse. Quindi, si definisce una nuova diramazione binaria si tracciano due interconnessioni, con direzione ortogonale a quella precedente, verso i centri dei settori, formando una struttura ad H. La procedura viene quindi iterata a partire da ogni centro fino al raggiungimento degli n sink, per un totale di $k = \frac{1}{2}\log_2(n)$ volte. Infine, introducendo i buffer in maniera regolare, si ottiene una struttura perfettamente bilanciata, con una lunghezza del percorso uguale per tutti i terminali e quindi skew totale nullo.

In sistemi ad alta frequenza, se in corrispondenza di una diramazione non è stato inserito un buffer, si crea una giunzione a T che genera riflessioni. Di conseguenza, si utilizza una struttura *Conica*[3] in cui a valle di ogni diramazione si dimezza l'ampiezza delle interconnessioni (W) così da ottenere impedenze adattate e cancellare le onde riflesse.

Il discorso teorico fatto finora, però, trascura alcune problematiche reali:

- le variazioni di processo determinano uno skew non nullo, con un'incertezza tra i sink proporzionale al numero di livelli che li separa dall'ultimo punto di diramazione in comune;

- la divisione del chip è effettuata in maniera geometrica e non tiene conto dell'effettiva distribuzione dei sink e di eventuali regioni ad alta concentrazione di elementi di memoria;

- eventuali aree in cui il piazzamento delle celle o delle interconnessioni è vietato.

2.4.2 Metodo delle Medie e Mediane

Il Metodo delle Medie e Mediane (MMM) si propone di risolvere il problema di una distribuzione disomogenea dei sink nella costruzione di un albero binario, evitando una divisione geometrica dell'area del circuito integrato. Nel primo step, si calcola il baricentro dell'insieme totale delle posizioni dei sink per posizionare il buffer sorgente. A questo punto, si calcola il valore della coordinata mediana della distribuzione lungo cui tagliare l'insieme dei sink ed effettuare una divisione in due sottogruppi uguali o al più differenti per un'unità. Per scegliere la direzione migliore di taglio, ovvero la coordinata lungo cui separare gli insiemi, si può utilizzare un approccio *look-ahead* di primo livello, effettuando entrambi i tagli e costruendo un ulteriore livello dell'albero per poi scartare l'opzione che presenta lo skew maggiore sul terminale individuato. Per entrambi i nuovi insiemi, viene quindi calcolata la posizione del baricentro, che sarà connesso al buffer sorgente, per poi iterare l'operazione fino a che ogni nuovo sottoinsieme creato contenga un solo elemento.

Questa soluzione, però, non annulla lo skew, perché non considera un bilanciamento globale della latenza. Lungo due percorsi diversi possono esserci forti disomogeneità nella distribuzione dei componenti, richiedendo lunghezze di interconnessioni diverse tra diversi sottogruppi lungo lo stesso livello. Ciò deriva anche dalla forte approssimazione legata alla direzione di taglio e all'impossibilità di raggruppare tra loro elementi sequenziali correlati. Infine, questo algoritmo non tiene conto di eventuali blocchi e non assicura di minimizzare i problemi di cross-talk tra interconnessioni.

2.4.3 Recursive Geometric Matching

L'approccio MMM può essere considerato *dall'alto verso il basso*, lavorando prima sul sistema generale e avvicinandosi man mano ai terminali della rete. Al contrario, l'algoritmo Recursive Geometric Matching (RGM) prevede l'approccio inverso per la costruzione di un albero binario, individuando ricorsivamente per n sink, $\frac{n}{2}$ segmenti di linea che connettano coppie di sink con l'obiettivo di minimizzare la lunghezza totale delle interconnessioni. Lungo la linea viene individuato il punto medio, denominato *tapping*, tale che lo skew del gruppo sia nullo. All'iterazione successiva si applica la stessa analisi, utilizzando i punti di *tapping* ed eventuali

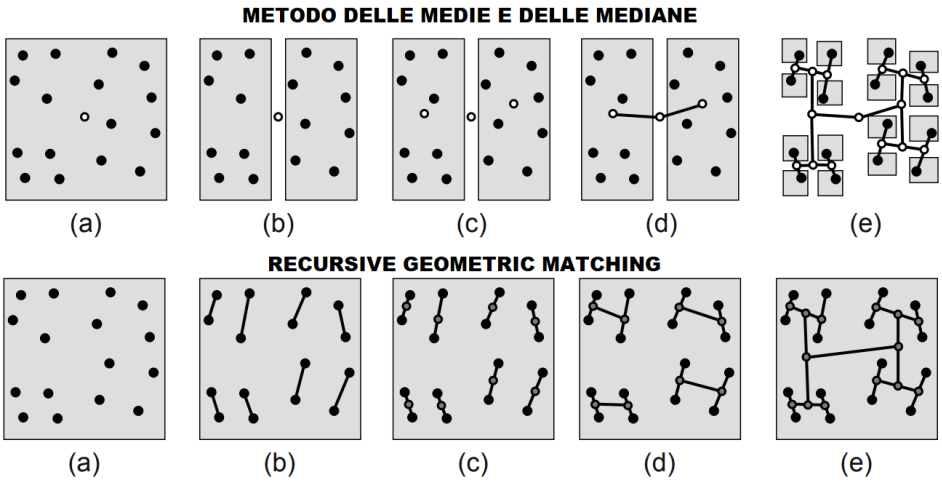

Figura 2.4: Illustrazioni degli algoritmi MMM (serie in alto) e RGM(in basso).

elementi rimasti esclusi dall'analisi precedente nel caso in cui il numero degli elementi sia dispari.

Se, per iterazioni successive alla prima, non è possibile individuare la migliore posizione del punto di equilibrio che minimizzi la differenza tra i percorsi, è possibile effettuare una trasformazione topologica, definita *H-flipping*, che effettua una redistribuzione dei gruppi che individui una soluzione a skew nullo, aumentando il costo di interconnessione totale, come mostrato in figura 2.5.

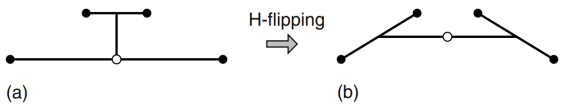

Figura 2.5: H-flipping.

L'approccio RGM presenta un costo computazione pari a $O(n^{2.5} \cdot \log(n))$, che se confrontato con il costo del metodo MMM, $O(n \cdot \log(n))$, evidenzia una complessità algoritmica molto superiore. Di contro i risultati ottenuti con il metodo RGM hanno uno skew e una lunghezza totale delle interconnessioni minori rispetto al metodo concorrente.

La pecca principale di questo metodo riguarda nuovamente il bilanciamento livello per livello, che trascura la latenza introdotta nei tratti di albero precedentemente individuati ed appartenenti a livelli inferiori. In questo modo per ogni livello si ottiene una minimizzazione dello skew, ma si rischia di incorrere in forti skew globali legando in egual modo sottogruppi a latenza potenzialmente diversa.

2.4.4 Exact Zero Skew (EZS)

L'algoritmo EZS [4] supera la limitazione del metodo RGM utilizzando un approccio dal basso verso l'alto e tenendo conto delle diverse latenze dei sotto-alberi individuati ai livelli precedenti. Inoltre, storicamente è uno dei primi ad utilizzare il modello di Elmore per la stima dei ritardi, ottenendo un'accuratezza maggiore.

Partendo dai sink viene effettuato un primo accoppiamento, definendo un insieme iniziale di sotto-alberi caratterizzati da una certa latenza t, dal valore della capacità di ingresso C_{sub} e dalla posizione del relativo punto di tapping. Una volta individuati i due insiemi da collegare, come mostrato in figura 2.6, è necessario individuare una coppia di archi con un estremo in comune tale per cui si eguaglino le latenze dei due insiemi totali, come espresso nell'equazione (2.4).

$$r_1\Big(\frac{c_1}{2} + C_{sub1}\Big) + t_1 = r_2\Big(\frac{c_2}{2} + C_{sub2}\Big) + t_2 \tag{2.4}$$

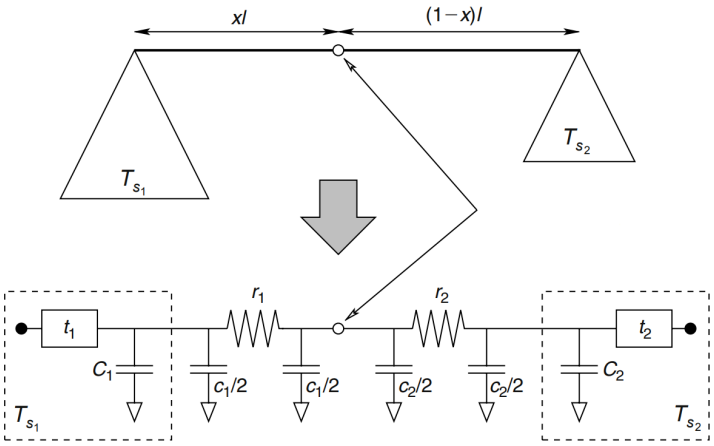

Figura 2.6: Circuito equivalente per il calcolo del punto di tapping nel metodo EZS.

Esprimendo la lunghezza di entrambi i segmenti in termini parametrici come xl e $(1-x)l$ dove l è il valore di lunghezza unitario a cui sono associate resistenza r_{unit} e capacità c_{unit} per unità di lunghezza, è possibile ricavare la lunghezza di Manhatthan dei due segmenti sostituendo nell'equazione (2.4) e ricavando il parametro x come

$$x = \frac{t_2 - t_1 + r_{unit}l\Big(\frac{c_{unit}l}{2} + C_{sub2}\Big)}{r_{unit}l(c_{unit}l + C_{sub1} + C_{sub2})} \tag{2.5}$$

Il risultato deve essere compreso tra $0 \leq x \leq 1$ per ottenere un risultato fisicamente sensato e definire i due tratti. Se il valore eccede dal range, vuol dire

che i due sotto-insiemi sono talmente sbilanciati da richiedere l'estensione di uno dei due tratti, tipicamente attraverso un'interconnessione a serpentina, bilanciando i ritardi a fronte di un incremento del costo e della capacità totale dell'albero.

2.4.5 Deferred Merge Embedding

Il flusso di definizione topologica degli algoritmi presentati è unidirezionale, ovvero non consente la modifica di soluzioni individuate nei livelli precedenti al fine di un'ottimizzazione globale dell'albero. Consideriamo i casi riportati in figura 2.7 dove 4 generici nodi intermedi sono stati connessi con una struttura binaria con l'algoritmo EZS. Una volta stabilita la distanza del punto di tapping E dai nodi A e B è possibile individuare diversi percorsi che soddisfino i requisiti dell'algoritmo. Salendo di livello, però, è evidente che la scelta di una posizione del nodo E più vicina a quello F consente un ridotto costo in termini di interconnessione e quindi di capacità e latenza totale.

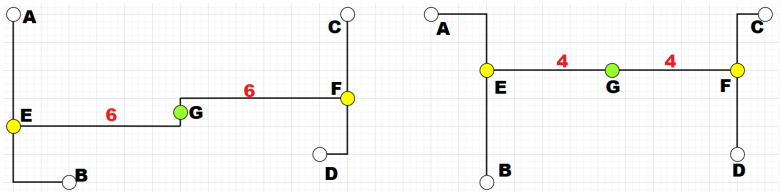

Figura 2.7: Soluzioni del metodo EZS con temporizzazione equivalente e costo differente.

Per risolvere queste ambiguità sono stati definiti alcuni algoritmi che operano su tutti gli input degli algoritmi presentati più una soluzione topologica precedentemente descritta su cui effettuare ottimizzazioni ricorsive in fase di integrazione fisica, tra cui annoveriamo l'algoritmo *Deferred Merge Embedding* (DME). Questo metodo può essere applicato ai risultati di tutti gli algoritmi descritti precedentemente e consta di due fasi: la prima in cui l'albero viene percorso dal basso verso l'alto per indicare gli insiemi dei punti ottimale per posizionare il relativo nodo padre, mentre la seconda individua l'ideale posizione della diramazione effettuando una nuova scansione dell'albero partendo dalla radice fino ad arrivare ai sink.

Consideriamo il flusso della fase "dal basso". Il punto di partenza nella prima iterazione è la posizione precisa e non modificabile dei sink s_i dalla quale è calcolata la distanza di Manhattan d_{ij} dal relativo nodo padre u_j determinato dalla topologia. L'insieme dei punti distanti d_{ij} dal sink viene denominato regione a rettangolo inclinato (TRR), per via della sua forma, la posizione esatta del sink prende il nome di "nucleo" della TRR mentre la distanza "raggio". Per ogni nodo padre u_j esisterà un insieme non nullo di punti, evidenziato in 2.8 per il nodo u_1, in cui le TRR dei

relativi nodi figli si sovrapporranno denominato "segmento di unione" $ms(u_1)$, che assicura uno skew nullo per costruzione con la stessa accuratezza dell'algoritmo che ha descritto la topologia. Nelle successive iterazioni, partendo dai segmenti di una coppia di nodi, si calcolano i raggi delle nuove regioni TRR tali per cui il costo di interconnessione sia minimo, definendo due effettivi rettangoli grazie al nucleo non più puntiforme e definendo un nuovo segmento di unione $ms(u_3)$ da utilizzare nelle successive iterazioni.

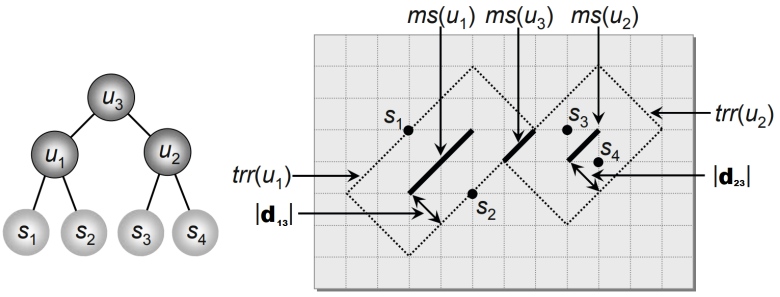

Figura 2.8: Rappresentazione grafica di un'iterazione DME applicata a nodi intermedi dell'albero.

Algorithm 1 DME prima fase

for ogni nodo v partendo dall'ultimo livello verso i superiori **do**
  if v è un nodo terminale **then**
    $ms(v) = (x_v, y_v)$;
  else
    Individua i nodi figli (a, b);
    Calcolo degli archi d_{av}, d_{bv};
    $TRR_a[nucleo, raggio] = [ms(a), d_{av}]$;
    $TRR_b[nucleo, raggio] = [ms(b), d_{bv}]$;
    $ms(v) = TRR_a \bigcap TRR_a$
  end if
end for

Una volta raggiunta la radice, sarà possibile partire con la seconda parte dell'algoritmo con l'approccio dall'alto verso il basso. Nella prima iterazione si analizza la posizione della radice, ma senza un livello superiore ogni punto del segmento di unione $ms(root)$ assicurerà il minimo costo e zero skew. Per tutte le iterazioni successive conoscendo la locazione del nodo padre e ricordando che ogni punto del segmento $ms(i)$ assicura uno skew nullo per costruzione, sarà sufficiente individuare il percorso a costo minore per ottimizzare il livello.

Algorithm 2 DME seconda fase

1: **for** ogni nodo intermedio v partendo dal livello superiore **do**
2: **if** v è la radice **then**
3: $loc(v) = qualunque(ms(v))$;
4: **else**
5: Individua il nodo padre *padre* di v;
6: $TRR_{padre}[nucleo, raggio] = [loc(padre), d_{padre\_v}]$;
7: $loc(v) = qualunque(ms(v) \bigcap TRR_{padre})$
8: **end if**
9: **end for**

La complessità computazionale di entrambe le fasi è lineare con il numero di nodi, determinando una complessità dell'algoritmo DME pari a $O(2u)$ che deve essere sommata al costo dell'algoritmo utilizzato per determinare la topologia. È dimostrato che l'albero risultate annulla lo skew usando un modello lineare per il calcolo del ritardo delle interconnessioni, mentre utilizzando il modello di Elmore ciò non avviene, vanificando le ottimizzazioni ottenute utilizzando un algoritmo EZS per la topologia.

2.5 Clustering

Finora le principali criticità riscontrate nella sintesi delle reti di clock riguardano la distribuzione disomogenea dei sink e il relativo numero, da cui ha una dipendenza più che lineare la complessità computazionale degli algoritmi. Il bilanciamento dei carichi in nodi ad alto fan-out è ottenuto attraverso l'operazione di raggruppamento o *clustering* dei componenti sequenziali, che unisce in un unico nodo padre diversi sink consentendo il riutilizzo di buffer e tratti di linea e riducendo cospicuamente la ridondanza dei nodi intermedi.

Esistono molti algoritmi per il raggruppamento dei sink, ognuno caratterizzato da peculiari funzioni costo con cui si stima la qualità dei sotto-gruppi creati e si assicura il rispetto sia dei vincoli temporali e fisici, sia dei parametri con cui l'utente personalizza l'operazione. La principale problematica di queste operazioni riguarda l'approssimazione con cui sono valutate le capacità delle interconnessioni che collegano i diversi elementi del sotto-insieme. Una stima precisa del costo e dei ritardi può essere fatta solo in seguito alla definizione delle interconnessioni stesse, ma al tempo stesso è impossibile a livello algoritmico effettuare il routing di tutte le possibili combinazioni. Di conseguenza è necessario individuare l'insieme di figure di merito ed approssimazioni che garantisca la bontà della soluzione.

La maggior parte di queste tecniche coincide con quelle impiegate in fase di piazzamento. Una è la stima del semi-perimetro ($HRPM$) che associa il costo dell'interconnessioni al semi-perimetro del rettangolo che racchiude tutti i pin, oppure il computo del minimo albero ricoprente MST, problema ampiamente affrontato nella teoria dei grafi. Un'ulteriore metrica è stata presentata dal professor Edahiro e prende il nome di $HRPM$ scalata, dove il costo delle interconnessioni è pari a $\sqrt{N} \cdot D$, dove N è il numero dei sink del gruppo mentre D è pari alla massima distanza tra una coppia di pin.

In letteratura è presente uno studio sulla qualità delle approssimazioni HRPM, MST ed HRPM scalata [5] effettuata misurando l'effettiva lunghezza delle interconnessioni di un albero di distribuzione del clock con $N = 1846$ e un massimo fan-out dei nodi variabile da 1 a 32. I risultati, riportati in figura 2.10, evidenziano come al crescere dell'effettiva lunghezza delle interconnessioni tutte le metriche si discostino

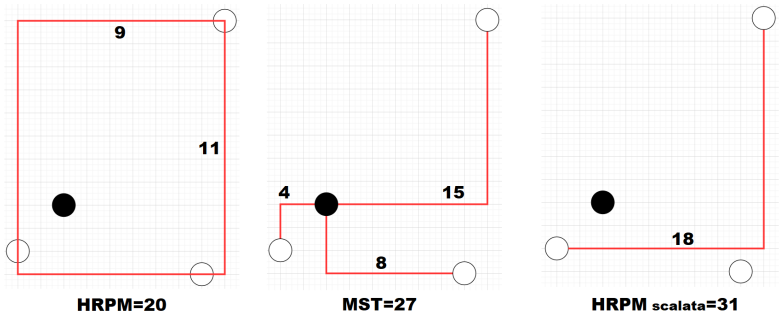

Figura 2.9: Rappresentazione grafica delle tecniche di stima HRPM, MST ed HRPM scalata.

fortemente dal valore atteso, con un errore massimo presente nella HRPM scalata pari a circa due volte la lunghezza effettiva. La correlazione statistica dei risultati per la MST e per la HRPM scalata è rispettivamente $\rho = 0.96$ e $\rho = 0.97$. Ciò consente di applicare liberamente queste approssimazioni, soprattutto per distanze contenute. È infine importante sottolineare come la HRPM scalata sia una tecnica pessimistica in quanto genera un risultato tipicamente maggiore di quello effettivo, a differenza delle altre metriche.

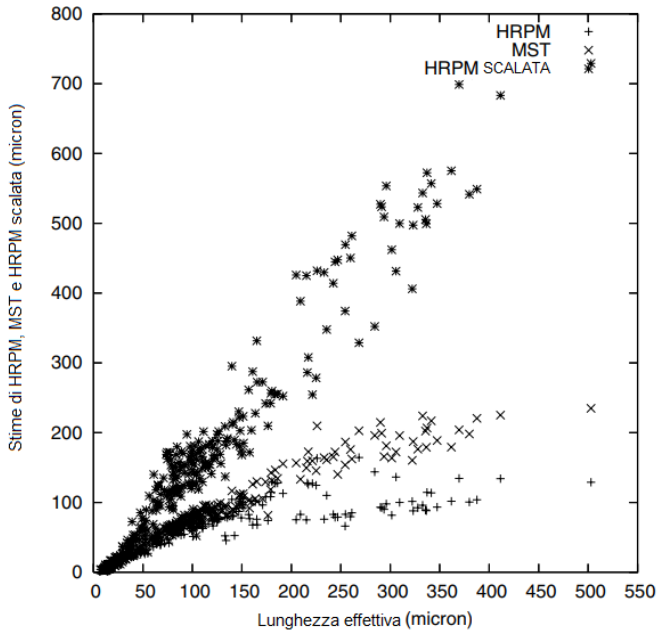

Figura 2.10: Accuratezza delle stime dei diversi metodi rispetto all'effettiva lunghezza delle interconnessioni.

Tipicamente gli algoritmi di raggruppamento non sono applicati esclusivamente ai componenti sequenziali, ma una volta definito un livello intermedio di buffer è possibile iterare nuovamente l'operazione di aggregazione, al fine di ridurre ulteriormente l'insieme dei sink in ingresso agli algoritmi topologici. In alcuni casi è possibile anche estremizzare l'operazione e costruire l'intero albero di distribuzione iterando l'operazione fino a raggiungere la radice.

2.5.1 Partizione

Tra i primi algoritmi presentati per la suddivisione in gruppi dei terminali di clock, ritroviamo una suddivisione del piano del circuito integrato tramite tagli longitudinali in macro-aree di dimensione decrescente [6].

L'algoritmo richiede in ingresso l'insieme S dei sink con posizione e capacità di ingresso c_i, il rettangolo R che contiene l'insieme da dividere (nella prima iterazione questa coincide con l'intero integrato), il numero di cluster richiesti K ed un parametro tipico dell'algoritmo denominato *vincolo d'equilibrio* B, pari al rapporto tra il massimo ed il minimo costo di un cluster ed usato per fornire all'utente un grado di libertà aggiuntiva sul bilanciamento della struttura. L'uscita è una lista L contenente tutti i possibili sotto-insiemi P, ognuno caratterizzato da tre parametri che ne indicano il carico medio avg, massimo max e minimo min. Questi valori sono ricavati attraverso l'equazione (2.6) sommando le singole capacità di ingresso c_i dei pin presenti nell'aggregato ed effettuando una stima del carico delle interconnessioni attraverso la metrica *HRPM* scalata.

$$C(P_j) = \alpha \sum_{i=1}^{N} c_i + \beta \sqrt{N} D \tag{2.6}$$

Lo pseudo-codice è riportato nell'algoritmo 3. La procedura è ricorsiva e, ricevuti gli ingressi, controlla se l'insieme è stato già analizzato o se è possibile effettuare ottimizzazioni (righe 2-6). In seguito effettua tutte le possibili divisioni della superficie in due parti non sovrapposte, attraverso tagli sia orizzontali sia verticali (riga 9). Per ognuna di queste individua l'insieme dei pin all'interno e la funzione si reitera due volte, una per ogni sotto-insieme creato, per definire i livelli sottostanti. Per ogni livello, a partire dal più basso, se una soluzione soddisfa il vincolo di bilanciamento B (riga 17) questa viene aggiunta alla lista d'uscita completa dei suoi parametri caratteristici (riga 18-19).

Questo algoritmo non è strettamente legato al raggruppamento di componenti sequenziali, ma può essere applicato anche a livelli superiori dell'albero per aggregare i buffer in gruppi bilanciati, fornendo le relative posizioni e la capacità d'ingresso dei buffer.

Algorithm 3 Raggruppamento tramite partizione

```
1: procedure Raggruppamento_partizione(S, R, K, B)
2:     if (R,K) è stato processato then
3:         return L di (R,K);
4:     end if
5:     if K=1 then                          ▷ Se non devo dividere in ulteriori cluster
6:         return L=costo(S),costo(S),costo(S);
7:     else
8:         for k=1 to K-1 do
9:             Effettua tutti i tagli P = ((R_1, k), (R_2, K − k)|(R_1 ∩ R_2 = ∅);
10:        end for
11:        for ogni coppia in P do
12:            Definisci gli insiemi di pin S_1, S_2 di R_1, R_2;
13:            L_1 = Raggruppamento_partizione(S_1, R_1, k, B);
14:            L_2 = Raggruppamento_partizione(S_2, R_2, K − k, B);
15:            for ogni soluzione (avg_1, min_1, max_1) in L_1 do
16:                for ogni soluzione (avg_2, min_2, max_2) in L_2 do
17:                    if max(max_1, max_2)/min(min_1, min_2) ≤ B then
18:                        Avg = (avg_1 · k + avg_2 · (K − k))/K;
19:                        Append(Avg,Max,Min) to L
20:                    end if
21:                end for
22:            end for
23:        end for
24:    end if
25:    return L;
26: end procedure
```

2.5.2 Raggruppamento di Shelar

Questo metodo di raggruppamento individuato da Shelar [7] divide le foglie in gruppi puntando a minimizzare il consumo di potenza dell'ultimo livello dell'albero. Ovviamente, per farlo è necessario stimare la potenza dissipata dagli insiemi prima che le interconnessioni siano state definite, di conseguenza è necessario effettuare una semplificazione. Consideriamo la potenza dissipata in un generico cluster di sink c pilotato da un buffer come nell'equazione (2.7)

$$P_c = \left(1 + \frac{1}{g'}\right)(C_{Seq} + C_{Inter})V^2 f + \frac{J_{Leakage}}{g'}(C_{Seq} + C_{Inter}) \tag{2.7}$$

dove V ed f sono rispettivamente la tensione di alimentazione e la frequenza

del sistema, g' è il guadagno del buffer utilizzato[1], $J_{Leakage}$ è un valore costante che tiene in conto delle correnti di perdita del buffer, C_{Seq} è la capacità di ingresso dei sink, mentre C_{Inter} è il carico capacitivo associato alle interconnessioni.

I parametri nell'equazione che possono essere modificati in fase di CTS riguardano il buffer utilizzato per pilotare il cluster e la capacità delle interconnessioni dell'insieme C_{Inter}. Di conseguenza, per individuare il raggruppamento con il minor consumo di potenza è sufficiente minimizzare il carico capacitivo aggiuntivo C_{clust_sol} legato alle interconnessioni dei gruppi e ai buffer che li pilotano, espresso in versione semplificata come nell'equazione (2.8).

$$C_{clust\_sol} = \sum_{i=1}^{n} \Big(\alpha + MST(c_i)\Big) \tag{2.8}$$

In cui viene effettuata la sommatoria per tutti gli insiemi $C = c_1...c_n$ della capacità delle interconnessioni, stimata attraverso il metodo dell'albero ricoprente minimo $MST(c_i)$, e del costo del buffer che pilota l'insieme, definito attraverso un parametro modificabile dall'utente α.

Lo pseudo-codice 4 descrive il funzionamento dell'algoritmo. Come primo passo, ricevuto l'insieme di componenti sequenziali S, viene creato un grafo G a cricca, che in teoria dei grafi corrisponde ad un insieme di nodi in cui tutti i vertici sono collegati a tutti gli altri $(G(S,E)\, con\, E = e(s_i, s_j)\,|\, i,j \in \{1,...,n\} \wedge i \neq j)$ come riportato nell'esempio di figura 2.11.

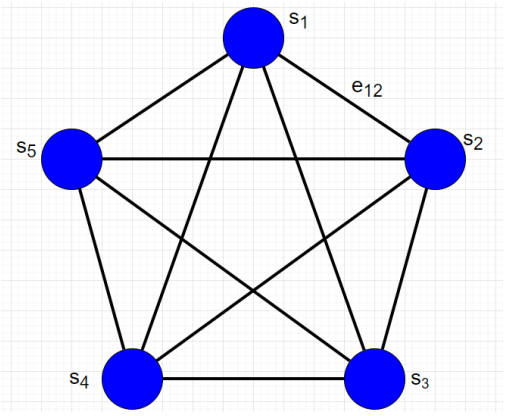

Figura 2.11: Esempio di un grafo a cricca con 5 elementi.

Ad ogni arco dell'insieme E viene assegnato un peso w legato alla capacità dell'interconnessione. La stima di questo valore è effettuata considerando la distanza di Manhattan tra i due estremi ed utilizzando la capacità per unità di lunghezza

[1]Inteso come il rapporto tra la capacità massima di carico e la capacità di ingresso della cella.

del livello orizzontale C_H e verticale C_V di metallizzazione (riga 2). L'insieme degli archi E viene quindi ordinato per valori di peso $w(e_{ij})$ crescenti ed è inizializzato un vettore di cluster di n elementi (pari al numero degi n sink), $C = \{c_1, ..., c_n\}$ (righe 3-4) e si inizializza il consumo di potenza della soluzione C_{clust_sol} considerando la presenza di un buffer dal costo α per ognuno degli n insiemi ed un costo di interconnessione nullo (riga 5).

Successivamente si analizza ogni arco della cricca che unisce due sink appartenenti a cluster diversi (righe 6-9). L'obiettivo della funzione è unire gli n cluster minimizzando C_{clust_sol}. Per farlo però è necessario rispettare il vincolo della massima capacità di carico C_{max} applicabile in uscita al buffer, di conseguenza per ogni cluster c_i deve essere verificata la condizione (2.9), dove C_{s_j} è la capacità di ingresso del sink s_j.

$$\sum_{s_j \in c_i} (C_{s_j} + MST(c_i)) \leq C_{max} \tag{2.9}$$

Se il carico capacitivo di entrambi i cluster può essere applicato in uscita al buffer e se il peso dell'arco da aggiungere non è superiore al peso di un buffer è conveniente unire i due gruppi (righe 10-12). Prima di procedere con l'analisi si aggiornano l'insieme dei cluster C e il consumo di potenza della soluzione considerando il peso del nuovo tratto di interconnessione $w(e_{ij})$ e rimuovendo il peso α del buffer del gruppo c_j (righe 13-15).

La complessità dell'algoritmo è generalmente proporzionale a $O(n^2 \cdot logn)$, soprattutto a causa dell'ordinamento degli archi in funzione del costo. Di contro, è possibile ottimizzare il tempo di esecuzione eliminando tutti gli archi dal costo troppo elevato e non effettivamente implementabile definendo una soglia limite.

2.5.3 Raggruppamento in K-gruppi

L'algoritmo K-gruppi, o *K-means*, si propone di dividere in K gruppi l'insieme di elementi ricevuto. Esistono diverse varianti di questo algoritmo, applicate anche al raggruppamento dei componenti sequenziali e per i nostri fini concentriamo l'attenzione sulle procedure che mirano a minimizzare la somma dei costi di interconnessione WL tra le foglie s ed il centro dell'agglomerato c, calcolato come nell'equazione (2.10).

$$WL = \sum_{i=1}^{m} \sum_{s_j \in c_i} (\left|x_{s_j} - x_{c_i}\right| + \left|y_{s_j} - y_{c_i}\right|) \tag{2.10}$$

La qualità del risultato dipende soprattutto dalla scelta del parametro K. Una regola empirica prevede un numero di gruppi pari a $K = \alpha \frac{N}{max\_fanout}$ con α

Algorithm 4 Raggruppamento di Shelar

crea un grafo a cricca $G(S, E)$;
assegna un peso ad ogni arco $W : E \leftarrow R \,|\, w(e(s_i, s_j)) = C_H \cdot |x_{s_i} + x_{s_j}| + C_V \cdot |y_{s_i} + y_{s_j}|$;
Ordinare ogni arco di E in ordine crescente di W;
Inizializzare il set di gruppi $C = c_1, ..., c_n \,|\, c_i = s_i$;
$C_{clust\_sol} = n \cdot \alpha$;
for ogni arco di E **do**
    $c_i \leftarrow Cluster(s_i)$;
    $c_j \leftarrow Cluster(s_j)$;
    if $c_i \neq c_j$ **then**
        if $c_i \bigcup c_j$ non viola (2.9) **then**
            if $w(e) \leq \alpha$ **then**
                $c_i \leftarrow c_i \bigcup c_j$
                $C \leftarrow C - c_j$
                $C_{clust\_sol} \leftarrow C_{clust\_sol} + w(e) - \alpha$
            end if
        end if
    end if
end for

corrispondente ad un generico numero maggiore di 1 per evitare violazioni delle DRC, ma non considera affatto la disposizione dei componenti. In figura 2.12(a) viene riportata una distribuzione disomogenea di sink ed in 2.12(b) e 2.12(c) viene applicato l'algoritmo di raggruppamento rispettivamente con $K = 3$ e $K = 5$, ottenendo due risultati sub-ottimali esclusivamente a causa del parametro K inserito.

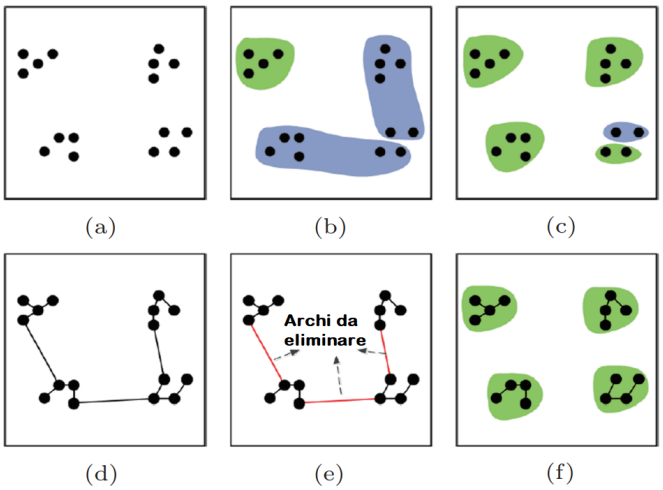

Figura 2.12: Applicazione del raggruppamento in K gruppi (a) topologia, (b) K=3, (c) K=5, (d) Albero ricoprente minimo, (e) dettaglo degli archi dal costo eccessivo, (f) raggruppamento con il valore ottimale K=4 scelto attraverso l'algoritmo 5.

L'approccio classico è quello di iterare la funzione per diversi valori di K ed individuare il risultato migliore. Considerato che l'algoritmo *K-means* richiede un numero di iterazioni minore del numero di punti n per la maggior parte delle iterazioni, è spesso praticabile l'approccio "prova e sbaglia", ma è giusto evidenziare che nel caso peggiore l'algoritmo può richiedere un numero di operazione con proporzionalità esponenziale rispetto al numero di elementi $O(2^n)$.

L'algoritmo 5 mira ad individuare automaticamente il valore di K ottimale per una data distribuzione. Dopo aver definito una cricca di archi pesati sull'insieme di componenti sequenziali S (righe 1-2), la procedura calcola l'albero ricoprente minimo applicando una funzione di terze parti (l'algoritmo di *Kruskal*) (riga 3), ottenendo il risultato riportato in figura 2.12(d). Successivamente viene calcolata la soglia limite per la lunghezza di un'interconnessione EL (riga 4), considerando l'area libera del die $W \cdot L$ al netto dell'eventuale presenza di ostruzioni $Size_{ob}$ attraverso la formula (2.11).

$$EL = \alpha\sqrt{\frac{W \cdot L - Size_{ob}}{N}} \tag{2.11}$$

Per ogni arco dal peso superiore alla soglia EL del circuito, l'algoritmo incrementerà il valore di K (riga 8) fino al raggiungimento del numero di cluster ideale.

Algorithm 5 Generatore_di_K

1: Crea un grafo a cricca $G(S, E)$;
2: Assegna ad ogni arco E il peso $w(e(s_i, s_j)) = \left|x_{s_i} - x_{s_j}\right| + \left|y_{s_i} - y_{s_j}\right|$;
3: Definisci il minimo albero ricoprente (MST) $T(S, E_T) = Kruskal(G)$
4: Applica la (2.11);
5: $K = 1$;
6: **while** $w(E_T) > EL$ **do**
7: Rimuovi E_T;
8: $K = K + 1$;
9: **end while**

Possiamo quindi applicare l'algoritmo K-gruppi che minimizzi il costo delle interconnessioni. Il principio di funzionamento di questa procedura è molto semplice ed è riportato in Alg. 6. Viene creato un insieme di K gruppi, i cui centri sono momentaneamente inizializzati in corrispondenza delle locazioni dei primi K registri dell'insieme S e si inizializzano le variabili $NewWL$ e $OldWL$ utilizzate per verificare la bontà della soluzione, in maniera tale da entrare nella procedura iterativa successiva (righe 1-3). In seguito inizia una procedura iterativa in cui ogni FF viene assegnato all'insieme più vicino, fintanto che siano rispettate le DRC sul carico dell'agglomerato (riga 7). Una volta definita la prima disposizione, il centro degli agglomerati c viene ricalcolato attraverso l'equazione (2.12) come il punto medio delle coordinate dei p registri s contenuti nel cluster (riga 10).

$$x_{c_i} = \frac{1}{p}\sum_{i=1}^{p} x_{s_i} \quad ; \quad y_{c_i} = \frac{1}{p}\sum_{i=1}^{p} y_{s_i} \tag{2.12}$$

Calcolati i nuovi centri, i valori di $NewWL$ e $OldWL$ sono aggiornati e l'operazione viene iterata fino a quando non è più possibile ottimizzare la lunghezza delle interconnessioni per questo ultimo livello.

2.5.4 Algoritmo di ricerca "avida" GSR

Con il termine "avido" si indica un algoritmo che sceglie l'opzione migliore al momento dell'analisi e non considera eventuali ottimizzazioni globali da svolgere

Algorithm 6 K-gruppi

1: Crea un insieme di K raggruppamenti centrati sui primi registri;
2: $OldWL = 0$;
3: $NewWL = 1$;
4: **while** $|NewWL - OldWL| \geq 1$ **do**
5: Rimuovi ogni sink dai gruppi;
6: **for** $i = 1$ to N **do**
7: Collega s_i al centro più vicino rispettando le DRC;
8: **end for**
9: **for** $i = 1$ to K **do**
10: Calcolo (x_{c_i} e x_{c_i});
11: **end for**
12: $OldWL = NewWL$;
13: $NewWL = WL$ calcolata con (2.10);
14: **end while**

in iterazioni successive, applicando una procedura sequenziale che non modifica nessuna decisione presa precedentemente. Di conseguenza è possibile ottenere un risultato ottimale solo se questo coincide con l'ottimo di ogni sotto-problema di cui è composto. Questi algoritmi sono vantaggiosi soprattutto grazie alla contenuta complessità computazionale, ottenuta grazie all'assenza di ricorsività delle decisioni.

Il funzionamento di questo programma per il raggruppamento dei registri è riportato nell'algoritmo 7 e si basa sul presupposto di fissare a priori una dimensione massima dei cluster calcolando la massima distanza del centro dell'agglomerato con l'equazione (2.13). Quest'ultima deriva sia dal limite empirico previsto per il numero dei cluster, sia dall'equazione (2.11) sul massimo costo accettabile per un arco.

$$Raggio_{max} = \alpha\sqrt{\frac{W \cdot L - Size_{ob}}{Nmax\_fanout}} \tag{2.13}$$

L'algoritmo riceve l'insieme dei registri S e associa il primo registro s_1 al primo cluster c_1. Quest'ultimo viene caratterizzato dal costo capacitivo del suo carico $Load_{c_1}$, dal fan-out totale $FanOut_{c_1}$ e dalla sua posizione centrale (x_{c_1}, y_{c_1}) calcolata attraverso l'equazione (2.12) che coincide con la posizione del registro s_1 in questa prima iterazione (righe 1-3).

Algorithm 7 GSR

1: crea il gruppo c_1 con dentro s_1;
2: $Load_{c_1} = Load_{r_1}$ e $FanOut_{c_1} = 1$;
3: Calcolo di x_{c_1} e y_{c_1};
4: Calcolo $Raggio_{max}$;
5: **for** ogni registro dell'insieme S **do**
6: Individua il cluster c_f più vicino a s_i
7: **if** $(Dist(c_f, r_i) < Raggio_{max})\&(Load_{c_f} + Load_{s_i} < Load_{max})\&(FanOut_{c_f} + 1 < FanOut_{max})$ **then**
8: $s_i \leftarrow c_i$
9: $Load_{c_f} = Load_{c_f} + Load_{s_i}$
10: $FanOut_{c_f} = FanOut_{c_f} + 1$
11: **else**
12: Nuovo gruppo c_j con s_i
13: $Load_{c_j} = Load_{r_j}$ e $FanOut_{c_j} = 1$;
14: Calcolo di x_{c_j} e y_{c_j};
15: **end if**
16: **end for**

2.5.5 Comparazione

Gli algoritmi di Shelar, KSR e GSR sono stati applicati sui circuiti di riferimento dell'*ISPD 2010*, fornendo la stessa libreria di buffer in ingresso e attraverso l'utilizzo di eseguibili scritti in C++ presenti sulla stessa workstation [8]. In figura 2.13 e 2.14 sono riportati i risultati della capacità totale dell'albero al termine dell'implementazione e il tempo di esecuzione. I diversi design sono ordinati per numero dei registri decrescente, a partire dal $10f02$ che conta $N_{02} = 2249$ registri e a seguire $N_{07} = 1915$, $N_{04} = 1845$ e $N_{03} = 1200$.

In termini di costo capacitivo i risultati degli algoritmi KSR e GSR risultano quasi equivalenti, con un valore mediamente inferiore per quest'ultimo ottenendo un risparmio medio del 3% sui 4 circuiti testati. Se confrontati all'algoritmo di Shelar, invece i risultati dimostrano un netto miglioramento della capacità totale con picchi nel circuito $10f04$ del 43,3% di risparmio rispetto al GSR e del 41,6% del KSR, mostrando come il raggruppamento tramite partizione non converga verso il risultato ottimale. Sono stati analizzati anche i valori di skew e latenza massima, mostrando un risultato bilanciato tra i diversi metodi per quanto riguarda il percorso critico dell'albero, mentre l'algoritmo di Shelar definisce soluzioni con uno skew massimo doppio rispetto ai concorrenti.

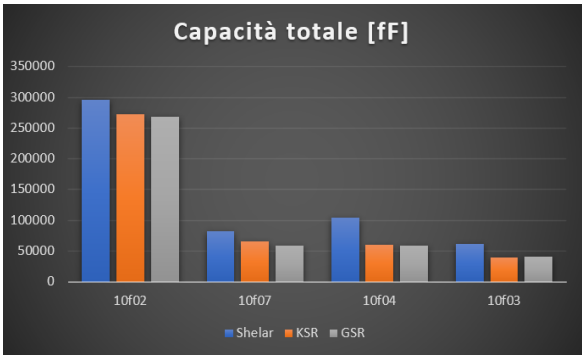

Figura 2.13: Capacità totale dell'albero a seguito delle tre diverse operazioni di raggruppamento.

Esaminando infine il tempo di esecuzione riportato in figura 2.14, appare evidente come l'algoritmo KSR sia quello associato alla maggiore complessità, con tempi di risposta per la sola operazione di raggruppamento che richiedono mediamente 6 volte il tempo richiesto dall'algoritmo GSR, che risulta il migliore di questa analisi.

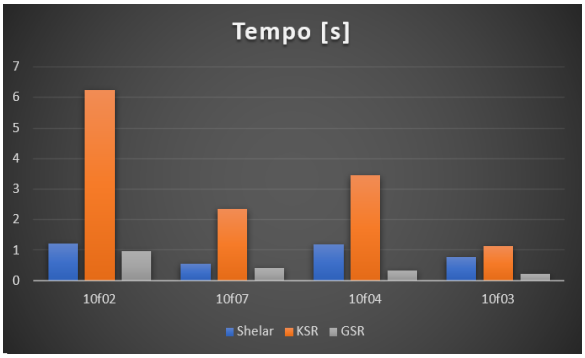

Figura 2.14: Tempo computazionale impiegato dagli algoritmi di raggruppamento su diversi circuiti di test.

2.5.6 Ottimizzazioni post-piazzamento e Multi-Bit Flip-Flop

La CTS opera sul risultato del piazzamento e le posizioni dei FF sono individuate per rispettare i vincoli temporali del percorso dati. A seguito o in concomitanza delle operazioni di raggruppamento è possibile applicare delle strategie di ottimizzazione del piazzamento dei componenti sequenziali per avvicinarli fra loro e ridurre la lunghezza delle interconnessioni di clock [9]. Ovviamente, questa ottimizzazione deve tenere conto della temporizzazione dei percorso di dato per non compromettere l'integrità logica delle operazioni.

Consideriamo il collegamento tra un generico FF ed il driver connesso al suo pin di dato (D). Effettuando la *Static Timing Analysis* (STA) sul percorso è possibile misurare il tempo di arrivo effettivo sul nodo D (AAT(D)), pari all'ultimo tempo di transizione sul nodo misurato a partire dall'inizio del ciclo di clock. Allo stesso modo definiamo il tempo di arrivo richiesto (RAT(D)) come l'intervallo di tempo tra l'inizio del ciclo di clcok e l'istante prima del quale deve essere completata l'ultima transizione su quel nodo per assicurare il corretto funzionamento del circuito all'interno del determinato periodo di clock.

La differenza tra questi due valori è denominata tempo di *slack* $t_{slack} = RAT(D) - AAT(D)$ e affinché siano rispettati i vincoli di setup del percorso questi deve essere non nullo $t_{slack} \geq 0$.

Partendo dal valore dello slack del percorso del dato è possibile ricavare la lunghezza del tratto di interconnessione equivalente. Questo valore corrisponde alla massima distanza rispetto all'uscita del driver in cui è possibile posizionare il pin D nel rispetto dei vincoli di temporizzazione del sistema e prende il nome di **TVFD** (*timing-violation-free distance*). A partire da questo risultato e come riportato in figura 2.15, delineiamo la regione racchiusa dall'insieme dei punti con una distanza di Manhattan pari a TVFD rispetto al pin di uscita del driver, entro la quale è

possibile posizionare il pin di dato del FF senza incorrere in una violazione. La stessa operazione è effettuata rispetto al percorso a latenza più stringente di quelli in uscita al FF.

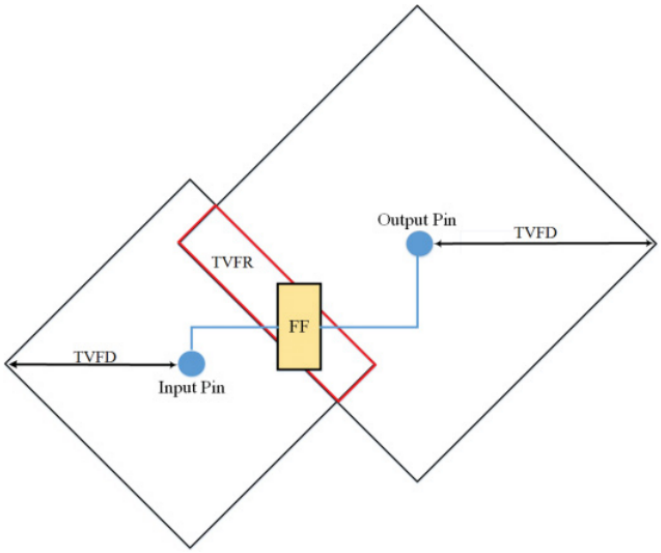

Figura 2.15: Rappresentazione grafica delle regioni che assicurano il rispetto dei vincoli di temporizzazione per un FF con TVFR evidenziata in rosso.

Definite queste due regioni, esisterà sempre un'area non nulla in cui le due regioni si sovrappongono, denominata **TVFR** (*timing-violation-free region*), colorata in rosso in figura, in cui sarà possibile posizionare il FF senza incorrere in violazioni di temporizzazione lungo il percorso dati.

Una volta analizzati i percorsi è possibile procedere all'ottimizzazione della posizione effettuando un concorrenziale raggruppamento. Innanzitutto, si effettua una ricerca delle coppie di FF tra loro più vicini dividendo il piano del circuito in settori. Categorizzate le distanze, le coppie sono ordinate per valori di distanza crescenti. Consideriamo nell'esempio riportato in figura 2.16(a) che la coppia più vicina non ancora processata sia (FF_A, FF_B, d_{ab}). Se sono rispettati i vincoli di massima capacità e tempo di transizione e se le TVFR dei due FF lo permettono, questi elementi sono raggruppati in un unico insieme e avvicinati il più possibile tra loro, come mostrato in 2.16(b).

Segue la coppia (FF_A, FF_C, d_{ac}). Entrambi i FF sono già stati processati ed uniti in un gruppo, di conseguenza l'algoritmo opera sulle due coppie AB e CD, controlla nuovamente che l'unione in un unico cluster non violi i vincoli e se possibile avvicina le celle fra loro, come riportato in figura 2.16(c).

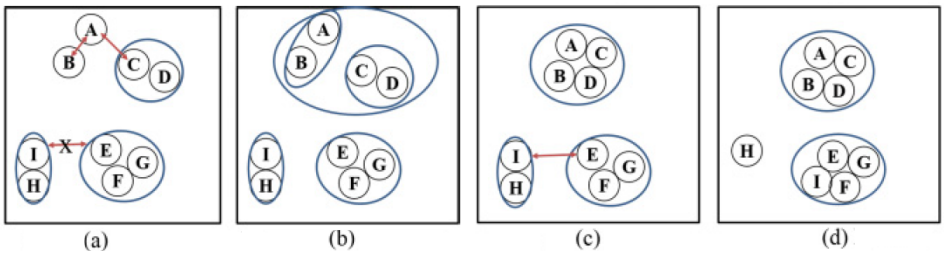

Figura 2.16: Applicazione degli algoritmi di post-piazzamento.

Successivamente viene effettuata un'operazione di rifinitura dei cluster costruiti in precedenza. Consideriamo i buffer E, F, G, H ed I ed ipotizziamo che non sia possibile unire direttamente i due insiemi a causa del vincolo sulla capacità massima. L'algoritmo effettua momentaneamente l'unione dei due gruppi e stima la lunghezza delle interconnessioni eliminando uno per volta i diversi sink che lo compongono. Se una delle combinazioni comporta una riduzione della lunghezza totale delle interconnessioni, l'algoritmo adotta questa nuova soluzione come nel caso di figura 2.16(d).

In termini di tempo di esecuzione dell'algoritmo, sia la parte di definizione dei gruppi sia quella di rifinitura degli insiemi hanno una complessità lineare con il numero di sink $O(N)$. Sulla complessità dell'intero algoritmo gli autori non sono riusciti a misurare precisamente il numero di operazioni effettuate, ma comparando i tempi di esecuzione con algoritmi di complessità nota, hanno stimato una complessità sub-quadratica.

Un problema di questa tecnica riguarda il possibile aumento della lunghezza delle interconnessioni di dato, che in alcuni casi può comportare un aumento della potenza complessiva e limitare il risparmio ottenuto sulla rete di clock. Inoltre, in alcuni agglomerati, è possibile orientare opportunamente le celle, come mostrato in figura 2.17, così da ridurre la lunghezza delle interconnessioni di clock, a spese di una maggiore congestione dei tratti di metallizzazione.

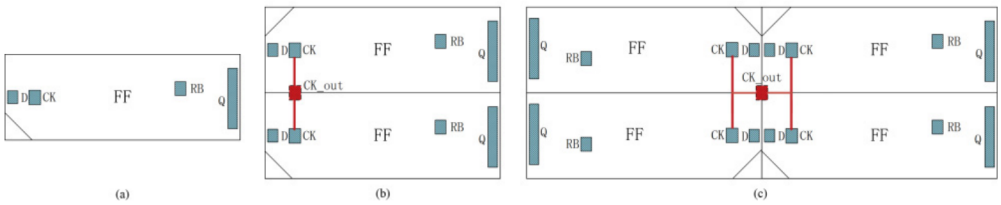

Figura 2.17: Disposizione ottimale di un gruppo di 2 e 4 FF per la minimizzazione delle interconnessioni di clock.

In figura 2.18 è rappresentata la struttura interna di un FF, costituito da due latch pilotati da un segnale complementare, ottenuto tramite l'inserimento di due

inverter. Questa circuiteria si ripropone in ogni FF del design, determinando un notevole consumo di potenza interna. Il principio dietro i Multi-Bit Flip Flop è quello di creare una cella sequenziale che condivida la circuiteria interna di divisione del clock, abbattendo il consumo di potenza interno, l'area occupata e il numero di pin visti dall'albero di clock.

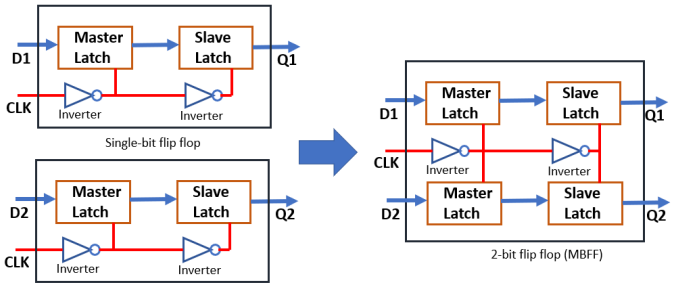

Figura 2.18: Struttura interna di FF e MBFF.

Le principali problematiche legate all'uso di queste celle riguardano la complessità interna di queste strutture, che necessita tipicamente di più di un livello di metallizzazione, definendo una struttura densa di interconnessioni e con una denstità di pin maggiore visto il ridotto spazio in cui si estende. Per questo motivo la dimensione massima di una cella MBFF è pari a 4 bit interni. Inoltre, la richiesta di corrente dei dispositivi contenuti nel MBFF ad ogni commutazione del clock è elevata e deve essere tenuta in conto nella gestione delle linee di alimentazione. Per evitare problemi di IR drop o di elettromigrazione sulle linee di alimentazione solitamente si dispongono delle capacità di disaccoppiamento utilizzate come serbatoio di carica per ridurre il carico sulle linee di alimentazione.

L'uso combinato dei MBFF (laddove la libreria tecnologica lo permetta) e delle procedure di ottimizzazione post-piazzamento consente di ridurre drasticamente la lunghezza delle interconnessioni dell'albero di clock, con un conseguente risparmio di potenza dissipata.

2.6 Dimensionamento di Buffer ed interconnessioni

L'introduzione di ripetitori lungo il percorso di distribuzione del clock è un'operazione necessaria e che offre diversi benefici. Il tempo di propagazione in interconnessioni lunghe è dominato dal ritardo associato alle resistenze R e capacità C parassite delle interconnessioni stesse, ed ha una dipendenza quadratica rispetto alla lunghezza della linea $t_{pd} \propto l^2$. Per ridurre la latenza complessiva viene spezzato il percorso

delle interconnessioni con i ripetitori, definendo tratti di linea a lunghezza inferiore e riducendo il contributo dei parametri parassiti nel tempo di propagazione.

La divisione in tratti comporta ulteriori vantaggi: il carico totale viene distribuito su più dispositivi, ognuno con un proprio ritardo intrinseco, separando temporalmente l'operazione di carica delle diverse capacità e richiedendo una corrente di picco minore. Inoltre, per ogni ripetitore si definiscono nuovi collegamenti alle linee di alimentazione e massa, distribuendo in più punti della superficie dell'integrato la richiesta di corrente. In questo modo riduciamo la massima potenza istantanea dissipata, rilassando le richieste alla rete di alimentazione e al contempo definendo un minore *IR drop*.

I lati negativi riguardano l'introduzione delle capacità di ingresso dei dispositivi, un aumento dell'area woccupata, della potenza statica dissipata e della sensibilità alle variazioni di processo. In letteratura è stato ampiamente affrontato il problema del numero ottimale di ripetitori lungo un percorso per ottimizzare il tempo di propagazione del segnale. L'ottimizzazione dei consumi di potenza è invece un problema più complesso, soprattutto a causa di tutte le variabili elencate.

2.6.1 Rilassamento dello slew

La priorità in questa fase di integrazione fisica dell'albero è il rispetto delle DRC, quindi il posizionamento dei buffer deve assicurare il rispetto dello slew massimo. Consideriamo come riferimento la topologia di figura 2.19. I buffer X ed Y sono direttamente collegati ai sink S. È stato già evidenziato come ogni FF debba ricevere un fronte di clock netto, per evitare violazioni di setup e di hold. Inoltre, a seguito dei raggruppamenti, il fan out dei buffer dell'ultimo livello è molto elevato ed un tempo di transizione rilassato determina un aumento della potenza di corto-circuito dei dispositivi collegati. Di conseguenza è evidente come i ripetitori collegati direttamente ai sink abbiano un grande impatto sul sistema. Al contrario, il buffer Z come ogni altro buffer lungo il percorso dell'albero, non inficia direttamente sulle prestazioni della logica sottostante ed ha un fanout tipicamente contenuto.

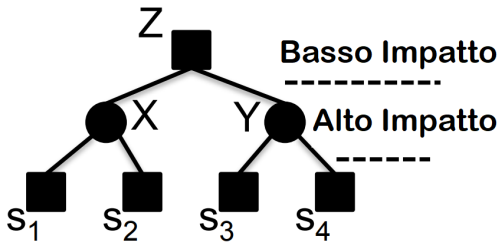

Figura 2.19: Topologia di un ramo di clock con buffer a basso e alto impatto.

È possibile definire due diversi vincoli di slew, uno globale S_{global} per i ripetitori "ad alto impatto" direttamente collegati ai sink ed un altro rilassato S_{relax} per altri buffer che permetta di introdurre un minor numero di dispositivi e risparmiare potenza [10]. Questo principio nell'inserzione dei buffer è stato provato sui circuiti dell'*ISPD2010*. I risultati mostrano che per circuiti congestionati e con un'elevata profondità dell'albero (intesa come numero massimo di buffer lungo il percorso), il risparmio di potenza raggiunge anche il 49.94%, con una riduzione anche della latenza complessiva, mentre per alberi a profondità limitata la riduzione di potenza è molto inferiore con una media circa del 2%.

Capitolo 3

TritonCTS

In questo capitolo viene descritto TritonCTS, un programma open-source che sintetizza la rete di distribuzione del clock, nato con l'obiettivo di individuare la soluzione ideale in termini di consumo di potenza, skew e latenza totale dell'albero.

Il software è attualmente disponibile in due versioni. La più recente, *TritonCTS 2.0*, è la soluzione attualmente impiegata nel più ampio pacchetto di programmi **OpenROAD**, descritto nel paragrafo 3.1. In questa tesi sono state esplorate le potenzialità di questo software, analizzando nel dettaglio gli algoritmi interni e studiando le differenze rispetto alla prima versione.

3.1 OpenROAD

OpenROAD è un sintetizzatore di circuiti open-source che effettua sia la sintesi logica, sia l'implementazione fisica di un design descritto a livello HDL. Lo scopo di questa suite di programmi è la creazione di un sistema autonomo che implementi circuiti in silicio. In figura 3.1 sono confrontati gli andamenti del numero di transistor per chip previsto dalla legge di Moore e del costo di progetto hardware per le fasi di design e verifica per alcune tecnologie fino al 2015.

Per ogni nuovo nodo tecnologico, il costo totale di progetto e verifica (rappresentato in rosso in figura) è aumentato in maniera esponenziale nel periodo considerato. La causa di questo trend è legata all'aumento della complessità di progetto e di realizzazione dei circuiti, cresciuta in funzione della riduzione delle dimensioni ottenibili tramite i processi fotolitografici. Per fronteggiare questo problema, le principali aziende produttrici di System-on-Chip (SoC) hanno diviso la gestione delle diverse fasi di progetto in più gruppi di lavoro, ma questa soluzione per realtà più modeste non è accessibile, determinando tempi di progettazione dai 12 ai 36 mesi. Inoltre, le licenze d'uso degli stumenti di automazione del design elettronico

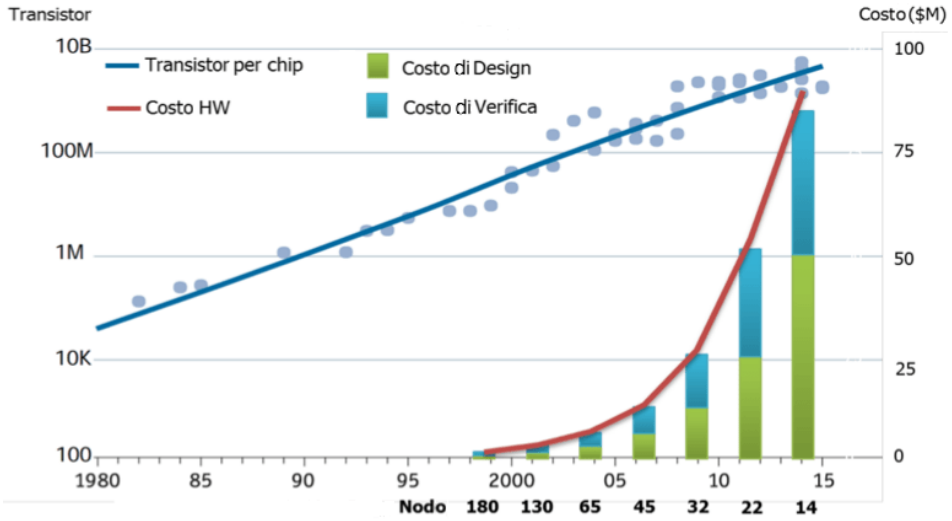

Figura 3.1: Andamento negli anni della legge di Moore e dei costi di progettazione hardware.

(EDA) hanno un costo notevole e gli algoritmi interni per lo svolgimento delle diverse fasi di layout sono vincolati da stretti accordi di non divulgazione.

Per queste ragioni, l'obiettivo della suite di programmi OpenROAD è la creazione di un generatore di layout ad accesso aperto e senza costi di licenza, completamente automatico e con un tempo di risposta massimo di 24 ore, che permetta a qualsiasi designer di ottenere un layout completo del circuito descritto.

In figura 3.2 viene riportato l'intero flusso di progetto ed in OpenROAD per ogni fase di progetto è stato individuato un programma dedicato. Una volta termina un'operazione, le sue uscite sono conformi agli ingressi dell'operatore successivo. L'uso di questa struttura modulare consente in qualunque momento di accedere ai risultati parziali dell'implementazione e allo stesso modo di poter fornire in ingresso ai software intermedi i risultati parziali di altri strumenti CAD.

Per quanto concerne la sintesi dell'albero di clock in figura sono specificati anche gli ingressi richiesti dal software TritonCTS ovvero:

- Il file .def contenente le informazioni sul posizionamento delle standard cell;
- Il file .lef tecnologico;
- La descrizione Verilog del circuito al livello delle porte logiche;
- I file tecnologici .lib contenenti la caratterizzazione delle standard cell.

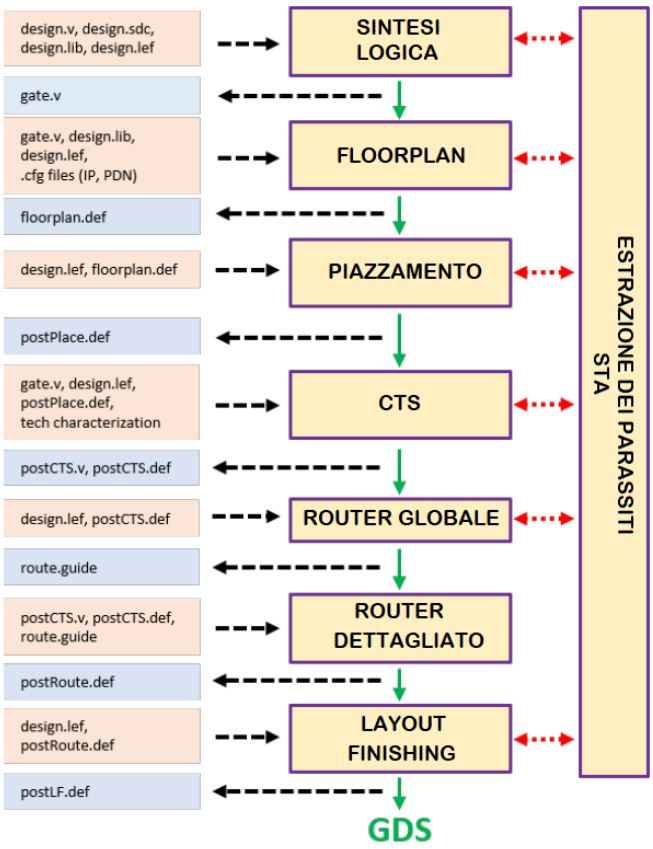

Figura 3.2: Il flusso di OpenROAD.

In uscita, il programma aggiorna i file .def e .v inserendo i buffer della struttura di clock sintetizzata tenendo conto delle ostruzioni per poi renderli disponibili per i successivi algoritmi di routing. Inoltre, il programma interagisce con un software esterno di *Static Timing Analysis* (STA) ovvero *OpenSTA* per l'analisi dei percorsi sintetizzati e degli elementi circuitali che lo compongono, ma è dimostrata la compatibilità anche con altri strumenti commerciali come *Synopsys PrimeTime*.

3.1.1 Modalità d'uso

Per utilizzare TritonCTS 2.0 è conveniente installare l'intero eseguibile di OpenROAD, così da assicurare il corretto funzionamento anche dei software correlati. Oltre alla già citata dipendenza con OpenSTA, infatti, è conveniente usufruire delle funzionalità di altri software presenti nella suite. Ad esempio, con il comando *repair_clock_inverters* si ricerca la presenza di inverter a monte dei componenti sequenziali per poi posizionarli in prossimità del sink e rinominarli in maniera

univoca così da assicurare che il software di CTS non consideri l'uscita dell'inverter come una nuova radice di clock da sintetizzare.

Attraverso il comando *clock_tree_synthesis* è possibile richiamare il programma e di seguito è riportata la sua descrizione.

Comando per richiamare le funzionalità di TritonCTS in OpenROAD

```
clock_tree_synthesis -buf_list <list_of_buffers> \
                     [-root_buf <root_buf>] \
                     [-wire_unit <wire_unit>] \
                     [-clk_nets <list_of_clk_nets>] \
                     [-out_path <lut_path>] \
                     [-post_cts_disable] \
                     [-distance_between_buffers] \
                     [-branching_point_buffers_distance] \
                     [-clustering_exponent] \
                     [-clustering_unbalance_ratio] \
                     [-sink_clustering_enable] \
                     [-sink_clustering_size <cluster_size>] \
                     [-sink_clustering_max_diameter <max_diameter>]
```

- $-buf\_list$ è l'insieme di buffer che l'utente specifica per la costruzione dell'albero ed è l'unico argomento che non può essere omesso. Il primo buffer specificato viene usato anche come riferimento per definire altri parametri;
- $-root\_buf$ è il primo buffer utilizzato come radice dell'albero;
- $-wire\_unit$ è un parametro che rappresenta la minima distanza tra buffer da utilizzare nella fase di caratterizzazione tecnologica iniziale;
- $-clk\_nets$ è la lista di nodi su cui verrà effettuata l'operazione;
- $-out\_path$ è il percorso in cui saranno salvate le LUT della tecnologia;
- $-post\_cts\_disable$ disabilita l'ottimizzazione dei sink sbilanciati;
- $-distance\_between\_buffers$; è il valore con cui l'utente può inserire una distanza minima (in μm) tra i buffer dell'albero;
- $-branching\_point\_buffers\_distance$ è la distanza (in μm) che deve intercorrere tra l'uscita di un buffer ed una diramazione per forzare l'inserimento di un nuovo ripetitore. Richiede che l'argomento $-distance\_between\_buffers$ sia stato specificato;
- $-clustering\_exponent$ è un parametro utilizzato in una formula dell'operazione di raggruppamento;

- $-clustering\_unbalance\_ratio$ è il rapporto tra il numero massimo di sink in un cluster ed il numero di sink di una regione;
- $-sink\_clustering\_enable$ abilita l'iniziale raggruppamento per processare i sink prima di iniziare la costruzione dell'albero;
- $-sink\_clustering\_size$ il numero massimo di sink per cluster nell'operazione di raggruppamento iniziale;
- $-sink\_clustering\_max\_diameter$ la dimensione geometrica (in μm) dei cluster nell'operazione di raggruppamento iniziale;

3.2 Descrizione tecnologica tramite LUT

Nella prima fase del flusso algoritmico di TritonCTS sono compilate delle *Look-Up Table* (LUT) contenenti le caratterizzazioni di vari tratti di interconnessione di lunghezze differenti, con e senza buffer, caratterizzati in latenza, tempo di transizione e carico capacitivo. Le informazioni su questi blocchi costitutivi sono poi utilizzate dagli algoritmi successivi per ottimizzare i punti delle diramazioni e posizionare al meglio i ripetitori in base alle specifiche assegnate.

Nel capitolo precedente è stata evidenziata l'importanza di una stima pronta e affidabile della propagazione del segnale nelle interconnessioni per verificare sia la realizzabilità sia la qualità di una soluzione, ma è stato anche sottolineato come l'accuratezza della stima sia inversamente proporzionale al costo computazionale per effettuarla.

TritonCTS utilizza un approccio diverso: anziché analizzare ad ogni passo la soluzione topologica individuata, l'algoritmo accede alla tabella in cui sono state memorizzate le informazioni su ogni percorso implementabile ed individua la soluzione ottimale in base ai vincoli di temporizzazione e potenza.

3.2.1 Caratterizzazione delle LUT

Le variabili che influenzano la caratterizzazione sono:

- La lunghezza dell'interconnessione L;
- La presenza di buffer in diversi punti dell'interconnessione e la minima distanza tra loro;
- La lista dei buffer e dei livelli di metal disponibili per la sintesi;
- Il tempo di transizione t_{tr} del segnale posto in ingresso;
- La capacità di carico C_{load} applicata all'altra estremità.

I primi tre parametri influenzano il numero di soluzioni possibili da testare, mentre gli ultimi due definiscono il numero di simulazioni che è necessario effettuare per ogni tratto di interconnessione considerato. La qualità del risultato della sintesi dipende molto dalla granularità con cui viene effettuata la caratterizzazione della tecnologia, ma un'ampia quantità di soluzioni determina anche un elevato tempo computazionale.

Consideriamo ad esempio un'interconnessione lunga $L = 60\mu m$, con una distanza minima tra due buffer pari a $30\mu m$ e una lista con 4 diverse dimensioni dei ripetitori. Le locazioni in cui è possibile disporre i buffer sono $n = 2$ (a metà e alla fine dell'interconnessione) e per ognuna dobbiamo considerare sia il caso in cui non sia posizionato alcuni buffer, sia l'uso delle 4 celle listate, definendo $m = 5$ diverse possibilità. Ne deriva che è necessario considerare un totale di $m^n = 5^2 = 25$ soluzioni possibili, determinando un numero di simulazioni da effettuare considerevole soprattutto per tratti molto lunghi.

All'utente viene concessa la possibilità di personalizzare la distanza minima tra due buffer e il numero di buffer da utilizzare modificando gli argomenti $-wire\_unit$ e $-buf\_list$ quando viene richiamata l'operazione di TritonCTS, mentre non è possibile variare la lunghezza minima e massima delle interconnessioni da testare. I valori di default testati sono $L_{min} = 20\mu m$ e $L_{max} = 80\mu m$, mentre nel caso in cui non sia specificata la distanza minima tra due buffer il programma utilizza automaticamente un valore pari a 10 volte l'altezza della cella del primo buffer specificato. Inoltre, l'utente riceve un aggiornamento in tempo reale sul terminale riguardo il numero di soluzioni testate, con un messaggio di avviso ogni 50000 percorsi creati e l'informazione sul numero finale, riportato anche nel logfile.

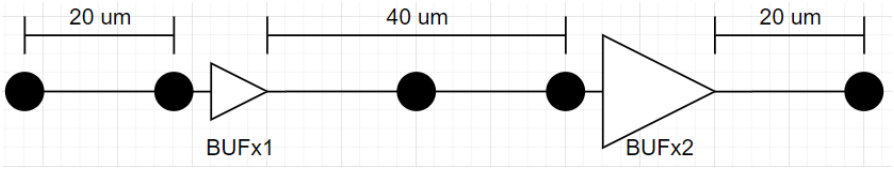

Figura 3.3: Circuito utilizzato per la caratterizzazione.

Una volta identificate tutte le possibili soluzioni topologiche da analizzare, il risultato viene inserito all'interno di una LUT denominata *"sol_list.txt"*. La dimensione di ogni riga varia in base alla topologia e contiene il nome dell'eventuale buffer e la lunghezza dei tratti che la compongono espressa in multipli interi della lunghezza minima tecnologica λ. Consideriamo come esempio il caso di un'interconnessione di lunghezza massima pari ad $L = 80\mu m$ con l'inserzione di due buffer *"BUFx1"* e *"BUFx2"* rispettivamente a $20\mu m$ e $60\mu m$ dall'origine del tratto, come rappresentato in figura 3.3. Ipotizzando che la soluzione sia la prima testata, la descrizione della soluzione è:

0	20	BUFx1	40	BUFx2	20

Definite le soluzioni, la caratterizzazione avviene simulando il circuito riportato in figura 3.4, effettuando uno sweep sulle due variabili rimanenti, ovvero tempo di transizione t_{tr} e della capacità di carico applicata C_{load} per misurare la risposta in tutte le condizioni possibili.

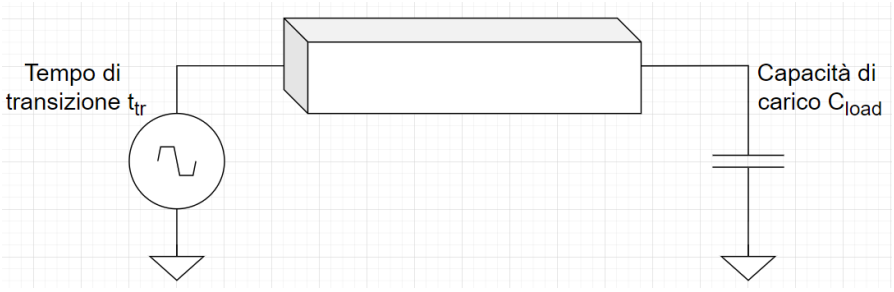

Figura 3.4: Circuito utilizzato per la caratterizzazione.

Il numero di simulazioni effettuate per tratto dipende dal valore massimo di entrambe le variabili e dei passi utilizzati per la caratterizzazione. Prima di iniziare l'operazione di CTS, l'utente ha la possibilità di personalizzare questi valori utilizzando il comando *configure_cts_characterization* di cui riportiamo la descrizione.

Comando per regolare la caratterizzazione delle LUT

```
configure_cts_characterization [-max_slew <max_slew>] \
                               [-max_cap <max_cap>] \
                               [-slew_inter <slew_inter>] \
                               [-cap_inter <cap_inter>]
```

Gli argomenti riguardanti il tempo di transizione devono essere espressi in secondi $[s]$, mentre quelli sulla massima capacità in farad $[F]$.

Caratterizzando i parametri $-slew\_inter$ e $-cap\_inter$ si definisce sia il passo della caratterizzazione (intervallo), sia il minimo valore di test con cui inizia la simulazione. In assenza di specifiche, i valori predefiniti adottati dal programma sono rispettivamente $5.0ps$ e $5.0fF$.

Per quanto riguarda gli argomenti di $-max\_slew$ e $-max\_cap$ il discorso è più complesso. Innanzitutto, per scelta di design, è necessario che l'utente specifichi sempre entrambi i limiti massimi. Nel caso in cui uno o entrambi i limiti non siano personalizzati, il programma procede ad analizzare i valori massimi utilizzati nei file .lib per caratterizzare il primo buffer indicato nella lista.

Terminata la personalizzazione il programma procede nella creazione di due vettori, contenti i valori di di capacità di carico e tempo di transizione massimo del segnale di ingresso. La dimensione di questi vettori è però fissata da due costanti, inserite all'interno del file "*CtsOptions.h*" denominate "*charLoadIterations_*" e "*charSlewIterations_*" che fissano le dimensioni massime dei vettori rispettivamente a 34 e 12 unità. Questa scelta è stata probabilmente adottata per limitare il tempo di esecuzione della caratterizzazione in linea con l'obiettivo del team di OpenROAD.

Per ogni simulazione vengono analizzati diversi parametri del circuito e ognuno di questi viene memorizzato in una linea della tabella seguendo la seguente divisione in colonne:

- Indice della simulazione;
- Lunghezza totale dell'interconnessione L;
- Carico applicato al nodo terminale C_{load};
- Il tempo di transizione misurato al nodo terminale;
- Potenza dissipata dalla soluzione (diversa da zero in presenza di buffer);
- Latenza introdotta;
- Capacità misurata all'ingresso;
- Tempo di transizione in ingresso t_{tr};
- Una variabile booleana che identifica la presenza di buffer nel tratto (pari a 1 nel caso di un'interconnessione pura);
- Ulteriori colonne nel caso di buffer ognuna contenente la distanza dal nodo precedente espressa in frazione della lunghezza totale[1].

Per evitare soluzioni ridondanti viene effettuata una scrematura dei risultati con buffer. Le soluzioni sono raggruppate in insiemi che condividono distanza, capacità in ingresso, slew sul nodo terminale e capacità di carico e per ogni insieme viene immagazzinata nel file di uscita (denominato "*lut.txt*") solo la soluzione a potenza minore. Questa scelta determina una riduzione della dimensione della LUT finale di

[1]Es. considerando la soluzione di figura 3.3 lunga $L = 80\mu m$, per indicare la posizione del primo buffer posto a $20\mu m$ dall'origine viene inserita una nuova colonna, la decima, contenente [0.25]; per descrivere il secondo ripetitore, posizionato a $60\mu m$ dall'origine, ma a $40\mu m$ dal nuovo buffer, nell'ulteriore undicesima colonna viene inserito [0.5].

circa il 94% ed hanno empiricamente stimato una perdita nella qualità del risultato finale dell'albero di clock, in termini di skew e latenza ottimale, intorno al 5%.

Terminata la caratterizzazione nella LUT, all'utente viene fornito uno schema riassuntivo all'interno del logfile, schematizzato come segue

Min. len	Max. len	Min. cap	Max. cap	Min. slew	Max. slew

È bene notare che i valori inseriti non sono riferiti ai limiti della caratterizzazione, bensì al massimo slew della tabella a seguito della degradazione lungo le interconnessioni e alla massima capacità misurata dal terminale di un buffer considerando anche i valori parassiti delle interconnessioni.

L'uso dei segmenti di dimensione ridotta caratterizzati nelle LUT consente di stimare la latenza totale delle interconnessioni più lunghe con un errore trascurabile. La principale causa di errore, inoltre, non è legata strettamente alla lunghezza minima considerata, ma al passo di caratterizzazione dello slew e della capacità di carico applicata.

3.3 Pre-raggruppamento

Terminata la caratterizzazione tecnologica, TritonCTS 2.0 analizza il file circuitali ed elabora le informazioni in un formato utile per l'algoritmo. Se non specificate nell'argomento $-clk\_nets$, l'algoritmo individua autonomamente ed in maniera affidabile i nodi che si comportano come segnali di clock nel file verilog a seguito del piazzamento.

Se abilitata l'opzione $-sink\_clustering\_enable$, la prima operazione di TritonCTS 2.0 è un pre-raggruppamento dei sink per generare un ultimo livello dell'albero ottimale e per ridurre il numero di ingressi all'algoritmo di costruzione dell'albero e quindi il tempo di operazione complessivo. È possibile effettuare il raggruppamento in due diverse tecniche: il primo è un metodo a K-medie capacitivo, mentre l'altro può essere definito geometrico.

Raggruppamento Geometrico

Se gli argomenti *sink_clustering_max_diameter* e *sink_clustering_size* sono stati specificati dall'utente, il raggruppamento iniziale viene effettuato secondo il metodo che denominiamo geometrico, in assenza di una specifica nomenclatura assegnata dagli autori.

Innanzitutto, il vettore contenente le coordinate dei sink S viene elaborato, calcolando per ogni elemento il valore dell'arcotangente θ rispetto ad un sistema di riferimento centrato nel centro geometrico del piano del circuito, per poi ordinare

il vettore per valori di θ crescenti. Per comprendere il metodo di raggruppamento, consideriamo il caso di figura 3.5, in cui sono riportati 3 sink ordinati in base ai rispettivi angoli come $\theta_1 < \theta_2 < \theta_3$.

Supponiamo di iniziare l'analisi dal sink s_1 e, sulla sua posizione, viene centrato il primo insieme. Scorrendo il vettore ordinato dei sink S, si passa ad analizzare la posizione di s_2 e si vaglia la possibilità di inserirlo in cluster che non hanno raggiunto ancora il numero massimo di elementi, misurando la distanza di Manhattan massima con gli elementi che lo compongono. In questo caso semplificato è sufficiente valutare la distanza con il sink s_1, ed ottenendo un valore superiore al diametro massimo di un cluster, si istanzia un nuovo insieme centrato nella posizione di s_2.

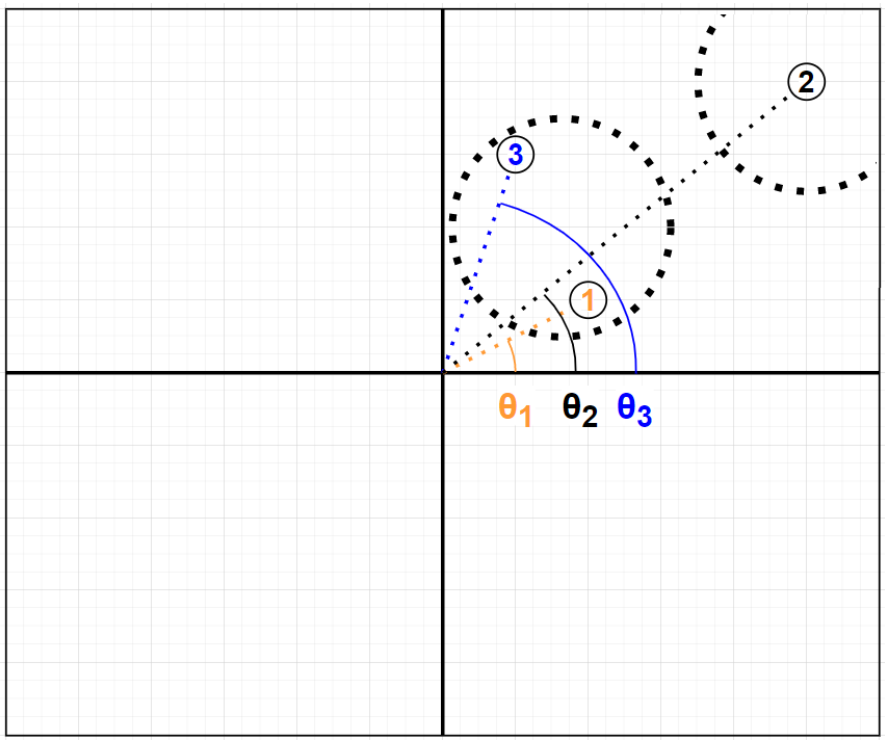

Figura 3.5: Flusso di assegnazione di un sink ad un cluster nel metodo CKMeans.

Si procede quindi a popolare i diversi insiemi, scorrendo il vettore dei sink. Passando al terminale s_3, si svolge un'analisi analoga a quella descritta precedentemente e, nel caso di idoneità con più di un cluster, si procede ad assegnare il sink all'insieme meno distante. Terminata una prima analisi, questo procedimento viene ripetuto più volte con un sink di partenza differenti, comparando le diverse soluzioni e memorizzando quella a costo minore.

Raggruppamento a K-medie capacitivo (*CKMeans*)

Il metodo CKMeans [11] è un algoritmo che, nonostante il nome, si propone di raggruppare gli N sink di un design in $M = N/K$ insiemi ognuno di dimensione massima K, ottimizzato per tecnologie fornite di Multi-Bit Flip-Flop.

La forma degli insiemi definiti, infatti, è rettangolare e con una dimensione ed un rapporto tra lunghezza e altezza proporzionali al numero di bit contenuti, come per le celle MBFF. Nonostante sia stata evidenziata la scarsa convenienza nell'adottare celle con più di 4 bit, a causa della congestione nel layout e dell'aumento del consumo di potenza nei percorsi di dato che ne deriva, l'algoritmo esplora la possibilità di creare cluster con un elevato numero di sink per analizzare le diverse possibilità di raggruppamento.

Un generico cluster t_m è caratterizzato in questo metodo dalla posizione del suo centro geometrico X_m, Y_m, che equivale al terminale di clock dell'ipotetico MBFF che rappresenta, e dal numero delle K locazioni interne in cui sono memorizzati i singoli bit $f_{m1}...f_{mK}$. Queste locazioni sono disposte in maniera regolare nel rettangolo del cluster e le rispettive coordinate sono espresse in funzione della distanza rispetto al centro dell'insieme $(x'_{m1}, y'_{m1})...(x'_{mK}, y'_{mK})$.

Di conseguenza, il primo passo dell'algoritmo riguarda l'inizializzazione delle coordinate dei centri del vettore dei cluster T. Si utilizzano come riferimento le posizioni di alcuni sink S, scelti con una tecnica quasi-randomica che assicura il rispetto di una distanza minima dei nuovi centri rispetto a quelli già selezionati. Il successivo flusso dell'algoritmo è rappresentato in figura 3.6. Le posizioni degli N sink sono messe in relazione con le $M \times K$ possibili locazioni dei diversi cluster a seguito dell'inizializzazione dei centri, definendo un totale di $N \times M \times K$ coppie e ad ognuna viene assegnato un peso, che tiene conto della capacità di carico (pari ad 1 in tutti i casi dell'esempio) e della distanza di Manhattan $d_{n,mk}$ tra la posizione x_n, y_n in cui il FF s_n è stato piazzato e la locazione di memoria all'interno del cluster $X_m + x'_{mk}, Y_m + y'_{mk}$ espressa in funzione del centro del gruppo, descritta nell'equazione (3.1).

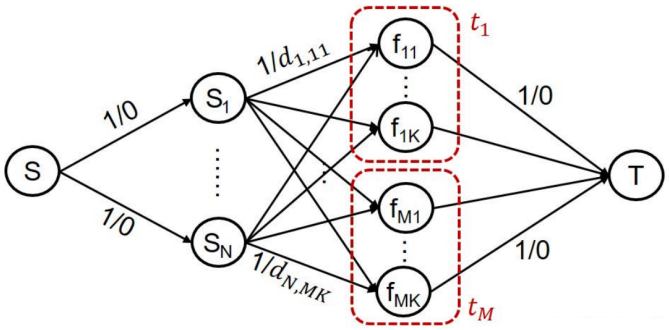

Figura 3.6: Flusso di assegnazione di un sink ad un cluster nel metodo CKMeans.

$$d_{n,mk} = |X_m + x'_{mk} - x_n| + |Y_m + y'_{mk} - y_n| \tag{3.1}$$

L'algoritmo procede quindi ad associare univocamente ognuno degli N sink ad una locazione, scegliendo le N coppie che minimizza la sommatoria delle distanze $D = \sum^{N} d_{n,mk}$. Terminata l'assegnazione dei sink, il centro degli insiemi X_m, Y_m viene ricalcolato, con l'obiettivo di minimizzare la somma delle distanze interne al cluster e si re-iterata la procedura fino a quando non è possibile trovare una soluzione che minimizzi ulteriormente D. Nella figura 3.7 è riportato il risultato finale di un raccoglimento CKMeans con $K = 64$.

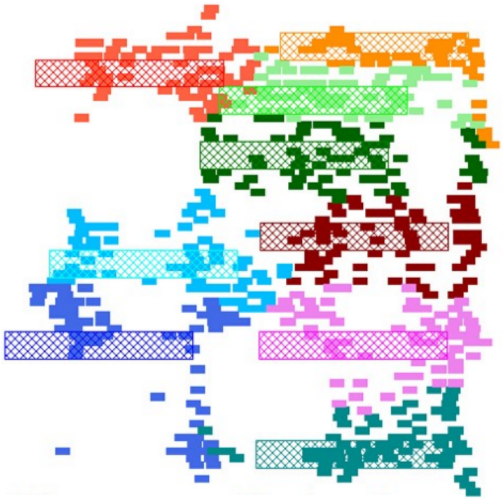

Figura 3.7: Risultato del raggruppamento con cluster di 64 bit.

Il problema principale della versione dell'algoritmo implementata in TritonCTS riguarda il limite della massima capacità applicabile ad ogni cluster, che è fissata a $(2 \times cap\_inter)$ e non può essere modificata dall'utente quando viene richiamata la CTS, rendendo impraticabile l'uso di questo metodo.

3.4 Costruzione dell'albero

Dopo l'eventuale raggruppamento iniziale, TritonCTS 2.0 procede alla costruzione dell'albero con un approccio dall'alto verso il basso molto simile al Metodo delle Medie e delle Mediane (MMM) descritto nel paragrafo 2.4.2. Nel primo passo si definisce la regione del piano che racchiude tutti i sink e la radice dell'albero è posizionata al centro geometrico di quest'area. Successivamente si procede a suddividere l'area operativa lungo la coordinata mediana della distribuzione dei sink e a definire il punto della diramazione come il valore medio di questa distribuzione.

A questo punto l'algoritmo si assicura che i sink siano stati suddivisi in maniera ottimale tra i due gruppi. Consideriamo il caso di figura 3.8 in cui è già stata effettuata la suddivisione e sono stati individuati i punti di diramazione u_1 ed u_2. L'algoritmo calcola la distanza tra ogni sink (in rosso) ed entrambi i centri per assicurarsi che la distribuzione sia ottimale. Nel caso del sink s_1 la distanza verso il suo punto di diramazione u_2 è maggiore rispetto a quella dell'altro insieme u_1.

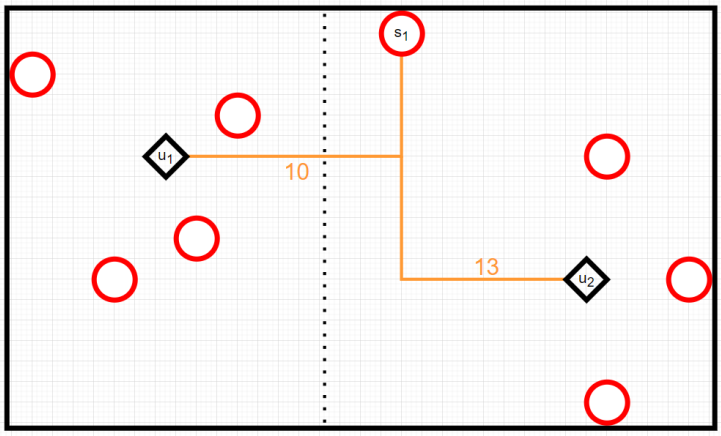

Figura 3.8: Rifinitura della suddivisione dei sink nella costruzione dell'albero.

In questo caso viene applicato l'algoritmo di raggruppamento CKMeans per equilibrare nuovamente i gruppi ed il centro delle diramazioni viene ricalcolato. Questa procedura viene ripetuta fino a che ogni sotto-insieme creato presenta al massimo 15 sink, valore soglia definito dai programmatori e non modificabile dall'utente.

Durante la costruzione dell'albero si considera anche la disposizione dei ripetitori. I vincoli sul massimo tempo di transizione e sulla massima capacità applicabile in uscita non possono essere modificati dall'utente e sono legati a multipli di *cap_inter* e *slew_inter*. All'utente viene fornita la possibilità di determinare la distanza minima tra due ripetitori consecutivi attraverso l'argomento $distance\_between\_buffers$, fornendo un grado di libertà aggiuntivo.

3.5 Differenze con la prima versione

Creazione delle LUT

La caratterizzazione tecnologica in TritonCTS 2.0 deve essere ripetuta ad ogni nuova sintesi e non è possibile riutilizzare LUT compilate precedentemente, nonostante l'operazione sia indipendente dal circuito sintetizzato. Il risultato, infatti, è legato esclusivamente al passo e agli estremi di caratterizzazione e alla lista di buffer

utilizzabili nell'albero e la bontà del risultato finale della sintesi è legato alla granularità delle tabelle. Di conseguenza, l'uso di più buffer e di intervalli ampi di caratterizzazione determina un notevole incremento del numero di operazione da svolgere ad ogni nuova iterazione dell'algoritmo.

Al contrario, nella prima versione di TritonCTS questa fase è scorporata dal flusso di sintesi dell'albero. Individuato il file liberty che associato alla tecnologia su cui sintetizzare il circuito, l'utente effettua la caratterizzazione dei blocchi fondamentali per la costruzione dell'albero e memorizza le LUT. Queste saranno poi fornite in ingresso a TritonCTS in sostituzione dei file .lib, riducendo il numero di operazioni da svolgere per ogni nuovo design.

Albero ad H generalizzato

La topologia definita è il tratto peculiare della prima versione di TritonCTS. Gli alberi ad H, grazie alla loro struttura simmetrica, minimizzano lo skew, ma hanno un costo delle interconnessioni elevato e a tratti ridondante a causa della natura esclusivamente binaria dell'albero.

Di conseguenza, il programma esplora le potenzialità di una struttura bilanciata a più livelli ortogonali tra loro, tipica degli alberi ad H, in cui il numero di diramazioni per livello non è limitato a 2 , bensì viene scelto sulla base delle caratteristiche geometriche del circuito. Un esempio di topologia ad Albero ad H generalizzato è riportato in figura 3.9, operata su un insieme di 1024 sink disposti uniformemente e con una radice posta al centro del design. Analizzata la forma del circuito, TritonCTS individua l'uso di un fattore di diramazione per il primo ed il quinto livello pari a 4, consentendo una riduzione del numero di livelli complessivo (8, rispetto ai $\log_2(1024) = 10$ previsti per una struttura binaria) ed anche del costo totale delle interconnessioni.

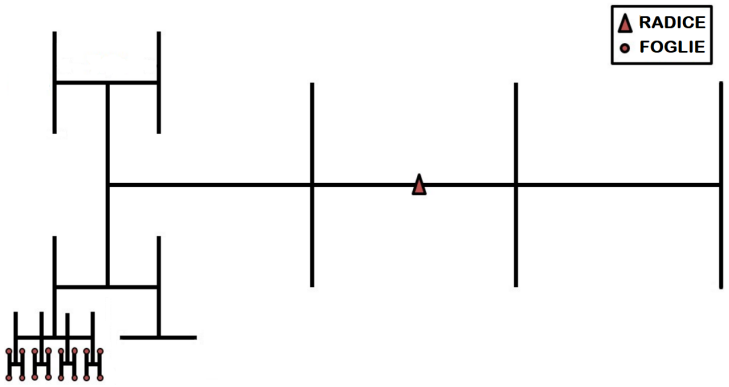

Figura 3.9: Albero ad H generalizzato con 8 livelli di profondità e fattore di diramazione rispettivamente pari a (4, 2, 2, 2, 4, 2, 2, 2).

Il problema della sintesi di una topologia ad albero ottimale per il circuito è affrontato con una tecnica di programmazione dinamica e scegliendo un approccio *bottom-up*, ovvero partendo dai terminali del nodo di clock. In prima approssimazione, si considera una distribuzione dei sink omogenea e regolare nelle aree considerate. L'obiettivo dell'algoritmo è compilare lo spazio delle soluzioni topologiche possibili di un design al variare:

- Della profondità dell'albero P (intesa come numero di livelli);
- Dell'altezza h e della larghezza w delle regioni;
- Del numero n di sink presenti nella regione;
- Della latenza massima e minima dell'albero;
- Della capacità totale vista dalla radice.

In figura 3.10 è rappresentato graficamente l'approccio utilizzato dall'algoritmo. Ipotizziamo che il primo livello da costruire sia orientato orizzontalmente rispetto al piano e che H e W siano rispettivamente l'altezza e la larghezza dell'intero design. Come punto di partenza, si descrive l'insieme di alberi a singolo livello di profondità al crescere dell'altezza delle regione considerata $h \in [h_{int}, H]$, della sua larghezza $w \in [w_{int}, W]$ e del numero di sink che contiene $n \in [2, (N \times w \times h)/(W \times H)]$, con passi di caratterizzazione rispettivamente pari a h_{int}, w_{int} e 2.

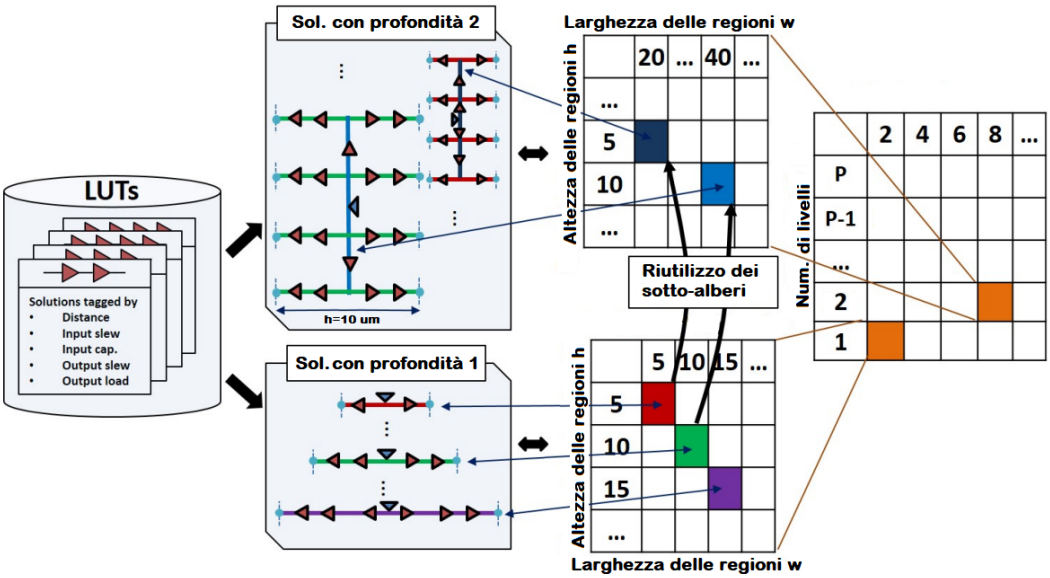

Figura 3.10: Ottimizzazione della topologia e del piazzamento.

Ad esempio, in figura è riportata la tabella delle soluzioni a singolo livello di profondità contenenti due sink $n = 2$, con dettaglio grafico sulle soluzioni con dimensioni della regione pari a $w = h = w_{int} = h_{int} = 5\mu m$ (in rosso), $w = h = 10\mu m$ (in verde) e $w = h = 15\mu m$ (in viola). Si evidenzia che nel caso di alberi a singolo livello di profondità, l'altezza delle regioni non influisce sulle diverse soluzioni, ma deve essere tenuta in conto per le successive fasi dell'algoritmo per mantenere la corretta indicizzazione.

Terminata la caratterizzazione di queste soluzioni, si procede a considerare gli alberi con un livello di profondità aggiuntivo $P = 2$. Sia l'orientamento, sia il riferimento di ogni nuovo livello è ortogonale al precedente ed il valore minimo del numero di regioni viene raddoppiato, per considerare la soluzione esclusivamente binaria, aggiornando anche il minimo valore di ampiezza delle regioni da considerare. Inoltre, per la costruzione dell'albero l'algoritmo accede nuovamente alle LUT tecnologiche per la costruzione del nuovo livello, ma riutilizza la soluzione più adatta individuata nella costruzione dei livelli inferiori così da ridurre la complessità computazionale.

Per esplicitare il concetto, sempre in figura 3.10 viene considerata la matrice delle soluzioni con alberi a 2 livelli di profondità, contenenti 8 sink. La larghezza minima della regione in cui è possibile disporre 4 sink tiene in conto delle dimensioni minime delle soluzioni precedenti, di conseguenza otteniamo $w \in [4 \times w_{int} = 20\mu m, W]$. Nel dettaglio, è rappresentata graficamente in blu scuro la soluzione con un $h = 5\mu m, w = 20\mu m$. Per connettere in due livelli 8 sink, è necessario che il livello superiore abbia un fattore di diramazione pari a 4. L'algoritmo quindi accede alle soluzioni indicizzate "$P = 1, n = 2, w = 5, h = 5$" e le utilizza come "sotto-alberi" per la costruzione dell'albero. Nel caso in cui sia possibile ottenere più combinazioni caratterizzate da eguali latenze minima e massima, l'algoritmo memorizza sempre la soluzione a costo minore.

La complessità dell'algoritmo è $O\left(N^2 \frac{H \cdot W}{h_{int} \cdot w_{int}} \frac{\gamma_{max}}{\gamma_{int}}\right)$ dove γ_{int} è l'intervallo temporale minimo, mentre γ_{max} è la latenza massima dell'albero, espressa dall'utente nella caratterizzazione dell'operazione di sintesi. Il numero di operazioni da svolgere, quindi, risulta molto elevato e per ridurre il tempo di esecuzione totale è previsto uno sfoltimento delle soluzioni, eliminando quelle che non rispettano i vincoli di latenza e skew totali dell'albero e quelle in cui si eccede il fanout massimo dei buffer. Nonostante lo sfoltimento, però, il peso nella memoria RAM per memorizzare lo spazio delle soluzioni indirizzato su 7 grandezza è notevole. Di conseguenza non è stato possibile testare questo algoritmo su un comune PC, ma è probabilmente richiesto l'utilizzo di una workstation.

Per completezza, successivamente l'algoritmo ottimizza la struttura dell'albero effettuando un raggruppamento livello per livello, così da bilanciare il carico capacitivo tra i diversi rami e ottimizzare le soluzioni in base all'effettiva distribuzione delle celle sequenziali. L'algoritmo di raggruppamento si basa su quello a K-medie capacitivo e sono applicati due diverse approssimazioni per i livelli superiori ed inferiori. Per i livelli superiori viene effettuato un raggruppamento "globale", trascurando la capacità delle interconnessioni per non incrementare eccessivamente la complessità computazionale. Quando il numero di sink per cluster individuato scende sotto una certa soglia definita è possibile considerare l'algoritmo di cluster "locale" in cui viene considerata anche la capacità delle interconnessioni stimata col metodo HRPM.

Capitolo 4

Sintesi e risultati

Per valutare l'operato di TritonCTS 2.0 è stata effettuata la sintesi dell'albero di clock di tre design con dimensioni e numero di sink diversi tra loro, le cui caratteristiche sono riassunte nella tabella 4.1.

		Sink	Dimensioni
(1)	Risc-V 32	1088	$454\,\mu m \times 454\,\mu m$
(2)	Encoder JPEG	4380	$1499\,\mu m \times 1499\,\mu m$
(3)	Comparatore RNS	25868	$2749\,\mu m \times 2749\,\mu m$

Tabella 4.1: Caratteristiche dei circuiti di test.

I circuiti sono stati implementati utilizzando il kit tecnologico open-source *SKY130HD*, progettato in collaborazione tra Google e la fabbrica di semiconduttori SkyWater Technology, mentre il flusso di sintesi, floorplan e piazzamento è stato effettuato tramite la suite di OpenROAD.

4.1 Risultati di default

Nell'appendice sono riportati gli script generalizzati di personalizzazione ed esecuzione della sintesi. Nel dettaglio, in figura 4.1 sono riportati i comandi caratteristici di TritonCTS, con gli argomenti più rilevanti ai fini delle analisi.

Dato l'elevato numero di sink, per ridurre il numero di livelli e in ottica di riduzione dei consumi è conveniente utilizzare sempre il comando "*sink_clustering _enable*", che abilita l'operazione di raggruppamento iniziale delle foglie dell'albero in agglomerati pilotati da buffer, che diventeranno a tutti gli effetti le nuove foglie della successiva operazione di sintesi.

L'algoritmo di clustering utilizzato è quello geometrico, descritto nel paragrafo 3.3. Oltre ai limiti superiori per il diametro ed il numero di sink per gruppo,

```
configure_cts_characterization -max_slew $max_slew \
                               -max_cap  $max_cap  \
                               -slew_inter $min_slew

clock_tree_synthesis -root_buf [list "BUF_1"] \
                     -buf_list [list "BUF_1" "BUF_2" "BUF_N" ] \
                     -post_cts_disable \
                     -sink_clustering_enable \
                     -sink_clustering_size $cluster_size \
                     -sink_clustering_max_diameter $cluster_diameter \
                     -distance_between_buffers $dist_min

report_cts [-out_file "CTS_report.txt"]
```

Figura 4.1: Script di caratterizzazione delle LUT e lancio della CTS.

un ulteriore argomento che influenza questa operazione è il tipo di cella indicato "$root\_buff$". Questo tipo di buffer, infatti, sarà usato sia come radice dell'intero albero, sia come radice dell'insieme dei sink. Non è attualmente possibile inserire una lista di diverse celle e delegare all'algoritmo il compito di individuare la dimensione più adatta.

Per quanto riguarda il valore del diametro massimo degli insiemi è stato scelto un valore sufficientemente elevato, tale da non essere il collo di bottiglia del raggruppamento. Utilizzare un valore non sufficientemente elevato, infatti, preclude la possibilità di riempire totalmente i cluster e richiede l'impiego di un numero maggiore di ripetitori con fan-out non ottimizzato. Ad esempio, considerando il caso del circuito (2) di tabella 4.1 è conveniente usare il valore di $450\,\mu m$, mentre per il circuito (3) di tabella 4.1 $600\,\mu m$.

Per quanto riguarda il ripetitore da utilizzare, consideriamo che nel PDK della tecnologia SkyWater è presente un set di buffer di clock con diverse *drive strength* nello specifico $x1, x2, x4, x8, x16$.

Siccome non è possibile inserire una lista di diverse celle e lasciar scegliere all'algoritmo la forza più adatta, in maniera empirica è stata scelta la cella "$sky130\_fd\_sc\_hd\_\_clkbuf\_8$", poiché assicura il rispetto dei vincoli sul massimo tempo di transizione e sulla massima capacità applicabile in uscita per le dimensioni massime dei cluster considerati.

Di seguito riportiamo l'analisi sulle sintesi effettuate al variare del numero di sink per gruppo. Alcuni dei seguenti parametri sono stati estrapolati dal logfile, come informazioni sulla profondità dell'albero intesa sia come numero di livelli, sia come numero di buffer costruiti lungo il percorso e la lunghezza media dei tratti di interconnessione per sink, calcolata come l'effettiva distanza di Manhattan dell'origine di ogni nodo figlio del percorso rispetto al relativo nodo padre fino al raggiungimento del pin di ingresso del segnale di clock, tenendo quindi conto anche del tratto di linea che pilota il buffer radice dell'albero. Per ampliare l'analisi,

sono stati considerati dei parametri aggiuntivi, calcolati elaborando i dati presenti nei file DEF, come la lunghezza massima di un percorso e le caratteristiche dei cluster, utilizzando un approccio identico a quello di TritonCTS per il calcolo della lunghezza media dei percorsi. Ogni misura riferita ad una lunghezza è da considerarsi in μm, mentre lo skew è in ps.

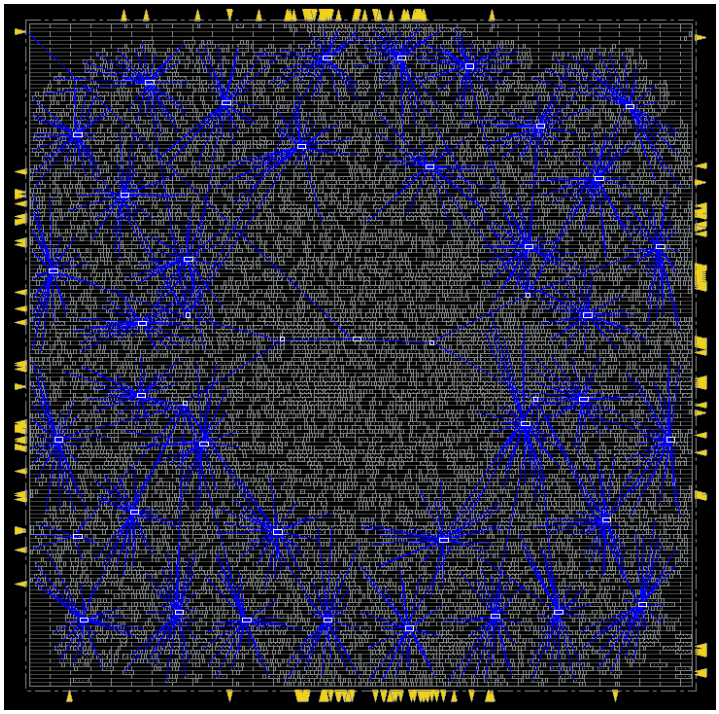

Figura 4.2: Albero completo del circuito Risc-V con numero massimo di sink per cluster pari a 30. Ogni elemento evidenziato in bianco rappresenta un buffer della struttura, mentre ogni linea blu è chiamata *trendline* e rappresenta l'insieme di celle collegate alle net del relativo componente.

	Numero massimo di sink per insieme				
(1)	20	25	30	35	40
Lunghezza massima	812	834	818	820	832
Lunghezza media dei percorsi	756	764	759	780	739
Skew	0,07	0,09	0,11	0,10	0,14
Buffer inseriti	62	51	44	39	31
Profondità dell'albero	4	4	4	4	3
Distanza sink-buffer massima	102,01	106,73	97,78	127,43	140,17
Distanza sink-buffer media	29,13	32,64	35,58	37,12	42,53
Cluster pieni	53	43	36	31	27
Cluster creati	55	44	37	32	28

Tabella 4.2: Analisi dell'albero di clock per il circuito Risc-V al variare del numero massimo di sink.

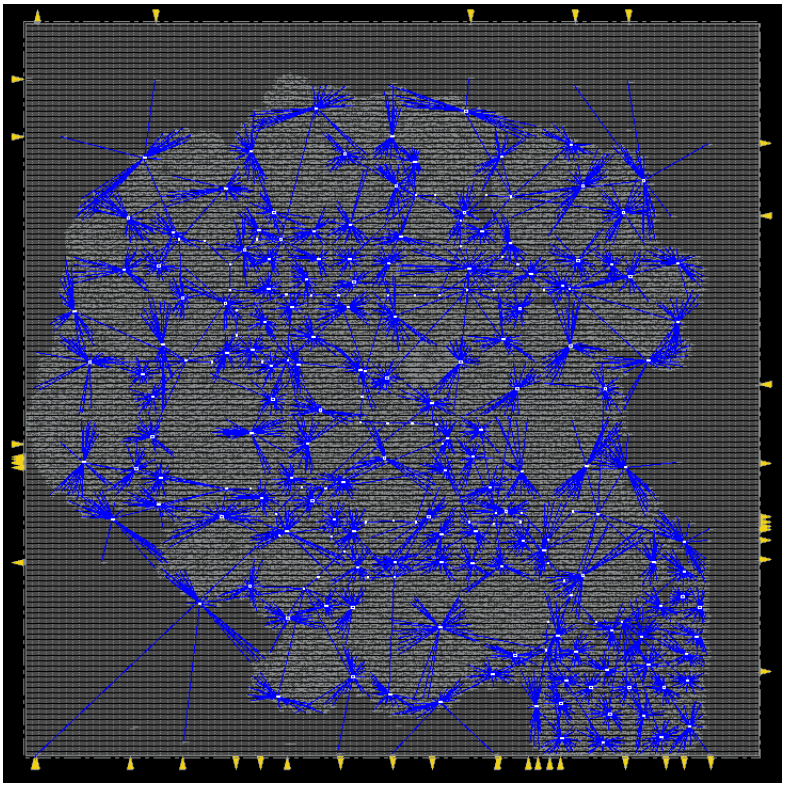

Figura 4.3: Albero completo dell'encoder JPEG con numero massimo di sink per insieme pari a 20.

	Numero massimo di sink per insieme				
(2)	20	25	30	35	40
Lunghezza massima	2930	2920	2930	2920	2850
Lunghezza media dei percorsi	2390	2390	2390	2380	2380
Skew	0,17	0,20	0,23	0,26	0,44
Buffer inseriti	288	245	213	193	145
Profondità dell'albero	13	13	12	12	12
Distanza sink-buffer massima	307,82	307,14	330,66	332,58	296,38
Distanza sink-buffer media	40,35	47,47	51,30	56,20	61,05
Cluster pieni	219	175	146	125	109
Cluster creati	219	176	146	125	109

Tabella 4.3: Analisi dell'albero di clock per l'encoder JPEG al variare del numero massimo di sink.

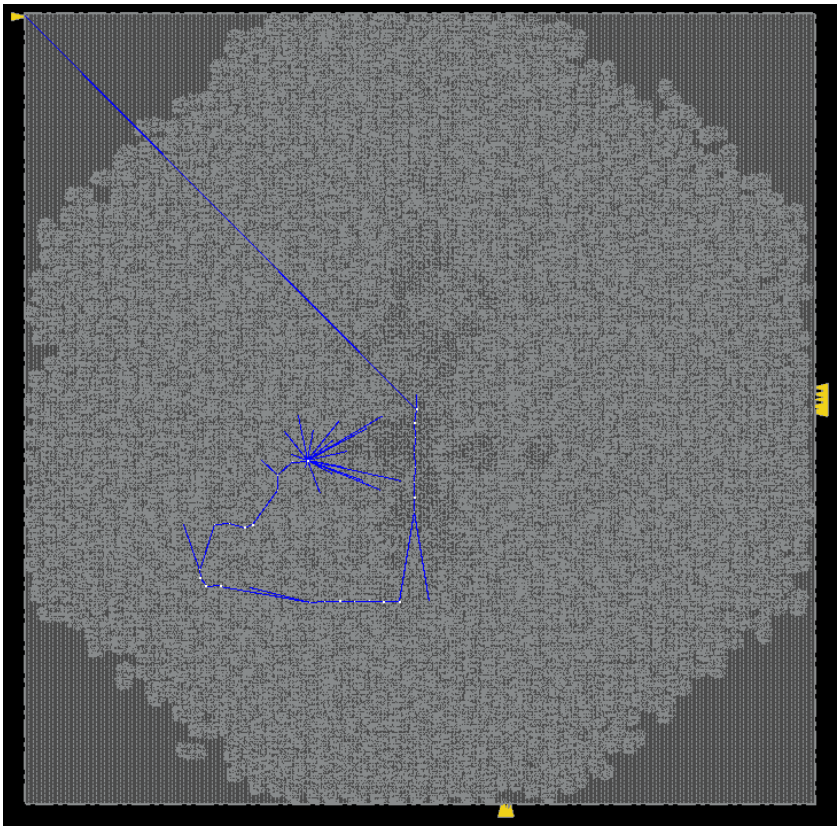

Figura 4.4: Dettaglio del percorso dalla lunghezza maggiore del circuito (3).

	Numero massimo di sink per insieme				
(3)	20	25	30	35	40
Lunghezza massima	5330	5330	5220	5220	5190
Lunghezza media dei percorsi	5190	5200	5160	5160	5170
Skew	0,28	0,34	0,33	0,36	0,44
Buffer inseriti	1786	1494	1194	1071	978
Profondità dell'albero	29	29	28	28	28
Distanza sink-buffer massima	467,04	478,18	476,73	495,35	502,07
Distanza sink-buffer media	75,33	75,62	75,81	74,98	75,12
Cluster pieni	1290	1031	861	737	644
Cluster creati	1296	1037	864	743	651

Tabella 4.4: Analisi dell'albero di clock per il circuito (3) al variare del numero massimo di sink.

È necessario sottolineare che l'algoritmo di sintesi dell'albero di TritonCTS 2.0, a differenza di quello della prima versione, non permette di personalizzare il valore massimo di skew accettabile, per la natura stessa del Metodo delle Medie e Mediane. L'assenza di questo grado di libertà limita la costruzione di alberi a basso consumo di potenza, poiché non è possibile rilassare la differenza totale tra il massimo ed il minimo percorso di clock, causando una ridondanza di elementi. Di conseguenza, in tutte le sintesi effettuate il numero di ripetitori minimo e massimo individuato lungo un percorso è uguale.

La costruzione di diramazione binaria dei primi livelli dell'albero è ideale soprattutto per il circuito (1), che, come evidenziato in figura 4.2, è caratterizzato da una distribuzione regolare dei sink. Per circuiti di dimensioni maggiori e con distribuzione dei sink irregolare non è possibile dire lo stesso, visto il corposo aumento del numero di buffer complessivo e lungo il singolo percorso. Questa ridondanza è evidenziata soprattutto in figura 4.4 dove è riportato il percorso a massima distanza dalla radice, in cui si riscontrano ripetitori ridondanti e posizionati in prossimità. Il tempo di transizione in questi nodi intermedio è infatti legato ad un valore molto contenuto espresso in multipli del passo di caratterizzazione e ben lontano dal tempo di transizione massimo accettabile in ingresso alla cella successiva. Un eventuale riduzione del numero di ripetitori in questi tratti non comporta, infatti, nessuna violazione dei vincoli fisici di design.

4.2 Il massimo tempo di transizione

Il numero di buffer utilizzati in questi circuiti di distribuzione del segnale di clock risulta piuttosto elevato considerando il pre-raggruppamento e soprattutto se confrontati con il numero totale di sink. Di conseguenza è stato analizzato nel dettaglio il funzionamento con l'obiettivo di identificare le operazioni che limitavano la generazione di un risultato a basso consumo di potenza.

Innanzitutto, nel logfile è contenuto un resoconto della caratterizzazione tecnologica effettuata, di cui riportiamo un esempio in figura 4.5. Ogni parametro è espresso come multiplo rispettivamente dei valori minimi di lunghezza ($20\mu m$), capacità ($5fF$) e tempo di transizione ($5ps$). Inoltre, i massimi di capacità e tempo di transizione rappresentano rispettivamente la massima capacità misurata sul nodo di ingresso e il massimo tempo di transizione misurato al nodo terminale dei blocchi caratterizzati, di conseguenza è corretto che non rappresentino gli argomenti del comando $configure\_cts\ \_characterization$.

Ciononostante, il loro valore risulta molto minore di quello desiderato ed appare totalmente indipendente da quelli indicati dall'utente. Di conseguenza sono stati osservati i codici sorgente ed in particolare il file *TechChar.cpp* nel quale è contenuta la funzione *initCharacterization* che svolge l'operazione di interesse.

```
[INFO CTS-0084] Compiling LUT.
Min. len    Max. len    Min. cap    Max. cap    Min. slew   Max. slew
2           8           1           36          1           30
```

Figura 4.5: Valori massimi e minimi delle LUT tecnologiche.

Osservando la struttura della funzione, nella definizione delle variabili dei massimi tempo di transizione e capacità di carico viene effettivamente data priorità ai valori personalizzati dall'utente rispetto a quelli estrapolati dai file .lib (nel caso in cui sia *max_slew* che *max_cap* siano stati indicati). Di contro, la condizione di fine ciclo dell'operazione *for* usata per creare i vettori di test è ottenuta tramite la *AND* tra le variabili massime ed un multiplo predefinito dei passi di caratterizzazione, come è possibile vedere nello stralcio di codice riportato di seguito.

Creazione del vettore con i tempi di transizione da caratterizzare.

```
for (float slewInter = _charSlewInter;
       (slewInter <= _charMaxSlew)
      && (slewInter <= slewIterations * _charSlewInter);
       slewInter += _charSlewInter) {
    _slewsToTest.push_back(slewInter);
    }
```

Il valore di *slewIterations* viene inizializzato nel file *CtsOptions.h* e posto pari al parametro *_charSlewIterazion* fissato a 12, mentre per il massimo valore di capacità di carico, inizializzato sempre nello stesso file, si fa riferimento al parametro $\_charLoadIterations = 34$. La soluzione più semplice e meno invasiva per personalizzare la condizione di fino ciclo è stata aumentare arbitrariamente i valori inizializzati, così da far prevalere sempre gli estremi specificati dall'utente.

Inoltre, il valore del massimo tempo di transizione utilizzato nella costruzione topologica dell'albero non è equivalente a quello dell'operazione di caratterizzazione. Infatti, analizzando tramite Static Timing Analysis il tempo di transizione dei tratti dell'albero di clock si evidenziano valori estremamente contenuti, associati nuovamente a multipli interi del passo di caratterizzazione utilizzato.

La funzionalità di personalizzazione del massimo tempo di transizione dell'albero è necessaria per la costruzione di sistema di distribuzione a basso consumo di potenza, di conseguenza è stato ritenuto opportuno modificare nuovamente i file sorgente. La costruzione dell'albero è effettuata soprattutto all'interno del file *HTreeBuilder.cpp*. Per non modificare i comandi di avvio dell'operazione di CTS è stato scelto di riutilizzare l'argomento *max_slew* definito nel comando $configure\_cts\ \_characterization$. Di conseguenza, dopo aver reso generali le variabili dell'operazione di caratterizzazione è stato posta la soglia massima del tempo di transizione nella topologia $SLEW\_THRESHOLD$ pari al metodo $\_techChar->getMaxSlew()$.

4.3 Integrazione dell'albero di Clock

Le operazioni di lettura e la scrittura dei file in formato DEF e LEF richiedono solitamente strumenti dedicati, esterni al flusso operativo dei software di CTS. Di conseguenza effettuata l'analisi del layout post-piazzamento, le informazioni sono elaborate in un formato fruibile dall'algoritmo così da evitare continue operazioni di lettura e scrittura dei file e velocizzare l'analisi topologica e la conseguente integrazione delle componenti aggiuntive.

Inoltre, individuata la struttura ottimale dell'albero di clock, è necessario assicurarsi che le successive fasi di routing e ottimizzazione dei percorsi di dato non modifichino le posizioni delle celle sequenziali, dei buffer di clock e delle interconnessioni individuate. Di seguito, è riportata la sintassi parziale della lista dei componenti piazzati, con alcuni degli argomenti di interesse pensati per fissare posizione degli elementi ed evitare successivi spostamenti.

Sintassi parziale dei componenti nel formato DEF

```
1 COMPONENTS numComps;
2         [- compName modelName
3           [+ {FIXED pt orient | COVER pt orient | PLACED pt orient
4                | UNPLACED} ]
5           [+ WEIGHT weight]
6           ... ; ]
7 END COMPONENTS
```

Ogni sezione nel formato DEF inizia con la parola chiave riguardante il tipo di elemento descritto, che nel caso delle celle è "*COMPONENTS*" e con il numero complessivo degli elementi che contiene (*numComps*).

L'istanza di una cella della lista comincia con il carattere −, seguita dal nome personalizzato del componente *compName* e dal nome della tipo di cella da piazzare *modelName* presente nel file della tecnologia utilizzata. Successivamente è possibile definire la locazione della cella, specificando le coordinate dell'origine nell'argomento *pt* (nel formato $(x\,y)$) e l'orientazione *orient* della cella rispetto all'origine, come descritto graficamente in figura 4.6.

A secondo dell'attributo si caratterizza la mobilità della cella per le operazioni successive. Tipicamente a seguito della fase di piazzamento ogni componente è descritto attraverso l'attributo "*PLACED*", che rappresenta una posizione del piano che può essere modificata dai successivi strumenti automatici di layout. Al contrario, una locazione contrassegnata dall'attributo "*FIXED*" non è modificabile da strumenti automatici, ma solo da comandi interattivi utilizzati dall'utente, mentre una cella con la specifica "*COVER*" non può essere spostate né da strumenti automatici, né da comandi interattivi. Nel caso di cella istanziata, ma senza specifica locazione è possibile utilizzare l'attributo "*UNPLACED*".

LEF/DEF	Definition	LEF/DEF	Definition
N		FN	
S		FS	
W		FW	
E		FE	

Figura 4.6: Orientamenti possibili delle celle con relativa posizione dell'origine *pt* marcata in nero.

Infine l'attributo "*WEIGHT*" assegna un peso al componente che determina se uno strumento di piazzamento automatico debba provare a rispettare la locazione specificata nell'attributo precedente. Il peso è espresso con un valore numerico, di default pari a 0. Inoltre, durante l'iniziale fase di piazzamento, ogni valore non nullo è trattato allo stesso modo degli altri, indipendentemente dal peso assegnato. 14 Come esempio, in figura 4.7 è riportato uno stralcio del file DEF risultato di TritonCTS.

```
- clkbuf_4_1__f_clk sky130_fd_sc_hd__buf_16 + PLACED ( 728180 402560 ) FS ;
- clkbuf_4_2__f_clk sky130_fd_sc_hd__buf_16 + PLACED ( 377200 546720 ) N ;
- clkbuf_4_3__f_clk sky130_fd_sc_hd__buf_16 + PLACED ( 637100 584800 ) N ;
- clkbuf_4_4__f_clk sky130_fd_sc_hd__buf_16 + PLACED ( 1053400 168640 ) FS ;
- clkbuf_4_5__f_clk sky130_fd_sc_hd__buf_16 + PLACED ( 1222220 206720 ) FS ;
- clkbuf_4_6__f_clk sky130_fd_sc_hd__buf_16 + PLACED ( 909880 503200 ) N ;
- clkbuf_4_7__f_clk sky130_fd_sc_hd__buf_16 + PLACED ( 1208880 492320 ) N ;
- clkbuf_4_8__f_clk sky130_fd_sc_hd__buf_16 + PLACED ( 272780 780640 ) N ;
- clkbuf_4_9__f_clk sky130_fd_sc_hd__buf_16 + PLACED ( 558900 794240 ) FS ;
- clkbuf_leaf_0_clk sky130_fd_sc_hd__buf_16 + PLACED ( 211600 448800 ) N ;
- clkbuf_leaf_100_clk sky130_fd_sc_hd__buf_16 + PLACED ( 915860 620160 ) FS ;
- clkbuf_leaf_101_clk sky130_fd_sc_hd__buf_16 + PLACED ( 921840 669120 ) FS ;
- clkbuf_leaf_102_clk sky130_fd_sc_hd__buf_16 + PLACED ( 999580 609280 ) FS ;
- clkbuf_leaf_103_clk sky130_fd_sc_hd__buf_16 + PLACED ( 990380 552160 ) N ;
- clkbuf_leaf_104_clk sky130_fd_sc_hd__buf_16 + PLACED ( 922760 522240 ) FS ;
```

Figura 4.7: Istanze dei buffer di clock definiti da TritonCTS contenuti nel file DEF di uscita.

È evidente che le celle descritte non siano minimamente vincolate nella locazione individuata, di conseguenza è stato ritenuto opportuno modificare il sistema di scrittura dei componenti.

Nella suite di OpenROAD si utilizza un software gestore di basi di dati, denominato *OpenDB*, che funge da tramite tra i software di lettura e scrittura dei file DEF e LEF e l'algoritmo di TritonCTS. Questo programma crea un file temporaneo in formato binario, rappresentando ogni elemento circuitale attraverso la programmazione ad oggetti, con classi e metodi facilmente fruibili dall'algoritmo. Operando sulle diverse proprietà degli oggetti si evitano le continue operazioni di lettura e scrittura dei file, velocizzando quindi sia l'analisi topologica sia integrazione.

È necessario sottolineare che la documentazione sull'interfaccia di programmazione del software (*API*) è assente, di conseguenza è possibile che alcune potenzialità dell'applicazione non siano state esplorate a fondo in questo lavoro, perché non ancora ultimate dal team di sviluppo.

Terminata l'analisi, TritonCTS scrive all'interno del database le modifiche da apportare al layout del circuito, istanziando nuovi oggetti associati ai buffer che desidera piazzare per poi caratterizzarne le diverse proprietà. Di conseguenza per modificare l'attributo del piazzamento è sufficiente modificare il relativo metodo, attraverso il comando *setPlacementStatus* ed utilizzando l'attributo *FIRM* che corrisponde allo stato "*FIXED*" nel formato DEF.

Effettuando questa modifica, è stato individuato un problema nella gestione delle ostruzioni di TritonCTS. Nella compilazione della base di dati, infatti, si tiene conto esclusivamente della posizione dell'origine del componente *pt* e non dell'area effettivamente occupata della cella. Di conseguenza, le posizioni definite in OpenDB non sono fisicamente implementabili e creano sovrapposizioni sia tra i buffer ed altri componenti del design, sia tra due diversi buffer dell'albero individuati in fase di CTS.

Per evidenziare questi errori è necessario effettuare l'operazione di piazzamento dettagliato, utilizzando un ulteriore software della suite denominato *OpenDP*. Questo programma riceve le informazioni contenute nel database ed individua le posizioni legali in cui disporre le celle attraverso il comando TCL *detailed_placement*. Di contro, essendo uno strumento automatico, OpenDP non può modificare la locazione delle celle con attributo "*FIXED*". Per questo motivo, effettuando la legalizzazione del layout con il comando *check_placement*, il programma aggiorna il logfile evidenziando le violazioni e arresta il flusso operativo, impedendo l'effettiva scrittura dei file di uscita.

In alternativa, è stato preferito assegnare un peso ai buffer, così da differenziarli dal resto delle celle durante le successive fasi di ottimizzazione, caratterizzando il metodo di ogni istanza con il comando *setWeight(2)* nella funzione adibita alla generazione delle celle *createClockBuffers* del file TritonCTS.cpp.

Per quanto riguarda le interconnessioni dell'albero di clock, la situazione è più complessa. Anche nella prima versione di TritonCTS, infatti il flusso prevede l'uso di uno strumento di routing esterno al programma, di conseguenza non è mai stato possibile specificare la locazione ottimale di punti di diramazione senza ripetitori. Le ragioni dietro l'assenza di questa funzionalità sono legate alle LUT tecnologiche utilizzate dall'algoritmo per comporre l'albero. All'interno della tabella "*sol_list*", infatti, sono specificati i nomi degli eventuali buffer utilizzati in quella specifica soluzione, ma non i livelli di metallizzazione utilizzati per la specifica soluzione. Infatti, durante l'analisi TritonCTS opera conoscendo la lunghezza, le caratteristiche temporali e la capacità dei tratti di linea, ma non può risalire al livello specifico utilizzato in quella caratterizzazione. Di conseguenza, nella

descrizione in OpenDB delle interconnessioni, l'unico metodo compilabile è quello dei terminali dell'interconnessione.

4.4 Comparazione dei risultati finali

A seguito delle modifiche algoritmiche, sono state ripetute le sintesi degli alberi di clock dei circuiti precedenti al variare del numero di sink per insieme nella fase di pre-raggruppamento. La lista dei buffer utilizzati e la dimensione massima degli insiemi non sono stati modificati, mentre è stato impostato il massimo tempo di transizione concesso di $1.4ns$ ed è stata effettuata correttamente una caratterizzazione tecnologica con massimo slew pari a $1.4ns$ e passo di caratterizzazione pari a $0.025ns$, così da aumentare l'accuratezza complessiva dei blocchi. Ovviamente, avendo eliminato la limitazione interna, il tempo di pre-caratterizzazione aumenta notevolmente. Inoltre, valori inferiori di passo di caratterizzazione determinano un raggiungimento della dimensione massima della RAM del dispositivo utilizzato, portando al fallimento dell'operazione.

	Numero massimo di sink per insieme					
(1)	20		30		40	
Lunghezza massima	803	-0,1%	807	-0,1%	860	+3.3%
Lunghezza media dei percorsi	781	+3,3%	781	+2,9%	769	+4,1%
Skew	0,07	+0,0%	0,10	-9,1%	0,18	+28,6%
Buffer inseriti	60	-3,2%	42	-4,5%	28	-9,7%
Profondità dell'albero	3	-1	3	-1	3	+0

Tabella 4.5: Analisi delle sintesi per il circuito Risc-V con confronto rispetto ai valori delle sintesi di default.

Per quanto riguarda il circuito Risc-V, con disposizione dei sink regolare e contenuta, sopratutto a seguito del pre-raggruppamento, si riscontra una riduzione di 2 unità del numero di buffer complessivi rimuovendo i buffer dal primo livello, come si evince dalla figura 4.8. È necessario sottolineare che nonostante sia stata ridotta la profondità dell'albero intesa come numero di ripetitori per percorso, il numero di livelli topologici è invariato, in quanto dipende esclusivamente dal numero di sink. Di conseguenza, nonostante la diramazione centrale mostri un fattore di diramazione pari a 4, a livello topologico durante l'algoritmo è stato definito un livello ulteriore costituito da pure interconnessioni non visualizzabile senza la creazione delle interconnessioni stesse. Le variazioni sulle lunghezze massime e medie dei percorsi e sullo skew sono molto contenute. Da segnalare il caso anomalo della sintesi a seguito del clustering con un numero massimo di 40 elementi per gruppo, in cui si ha un netto peggioramento percentuale dello skew complessivo

che passa dagli 0,14 *ps* ottenuti con l'algoritmo precedente agli attuali 0,28 *ps*, che restano nominalmente valori estremamente contenuti per la tecnologia di interesse.

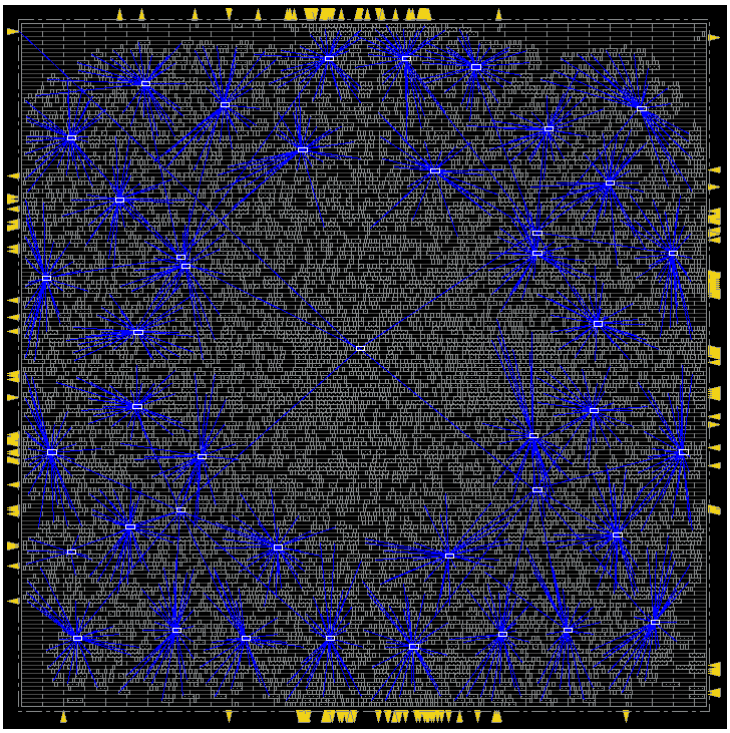

Figura 4.8: Albero completo dell'encoder JPEG con numero massimo di sink per insieme pari a 20.

	Numero massimo di sink per insieme					
(2)	20		30		40	
Lunghezza massima	2400	-18,1%	2610	-10,9%	2450	-14,0%
Lunghezza media dei percorsi	2340	-2,1%	2320	-2,9%	2330	-2,1%
Skew	0,11	-35,3%	0,13	-43,5%	0,27	-38,6%
Buffer inseriti	240	-16,7%	167	-21,6%	123	-15,2%
Profondità dell'albero	4	-9	4	-8	4	-8

Tabella 4.6: Analisi delle sintesi per l'encoder JPEG con confronto rispetto ai valori delle sintesi di default.

Per quanto riguarda i risultati dell'encoder JPEG è evidente un netto miglioramento di tutti i parametri presi in esame come evidenziato nella tabella 4.6. La profondità complessiva dell'albero è stata profondamente ridotta, eliminando diversi ripetitori ridondanti a cui non erano associati punti di diramazione. Grazie

alla riduzione del numero di livelli complessivo, inoltre, lo skew misurato al termine della sintesi presenta netti miglioramenti in tutti i casi considerati. In tal senso, la forte riduzione del percorso più lungo, pari al $-18{,}1\%$ nel caso di insiemi di 20 elementi associata ad un generale miglioramento della lunghezza media dei percorsi determina un risultato ottimale in termini di riduzione del costo capacitivo totale legato alle interconnessioni e quindi al consumo di potenza.

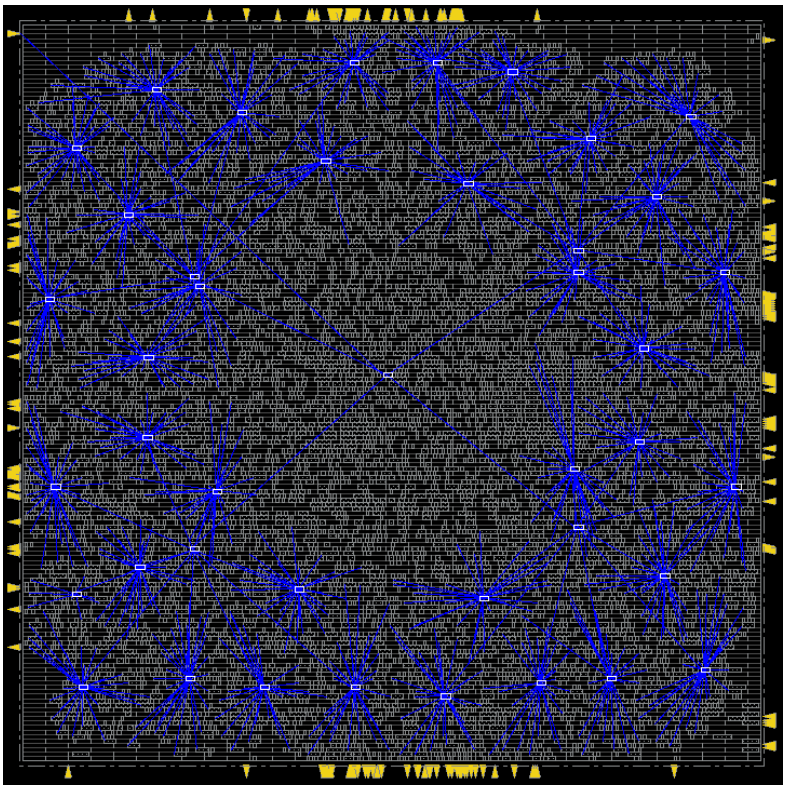

Figura 4.9: Dettaglio dell'albero dell'encoder JPEG con numero massimo di sink per insieme pari a 20. Per rendere fruibile l'immagine non sono stati evidenziati i sink effettivi, bensì in rosso sono stati evidenziati i buffer "foglia" ottenuti con l'operazione di pre-raggruppamento.

	Numero massimo di sink per insieme					
Comparatore RNS	20		30		40	
Lunghezza massima	5240	-1,7%	5090	-2,5%	5130	-1,1%
Lunghezza media dei percorsi	5120	-1,3%	5030	-2,5%	5040	-2,5%
Skew	0,19	-32,1%	0,16	-51,5%	0,15	-65,9%
Buffer inseriti	1519	-14,0%	959	-19,7%	743	-24,0%
Profondità dell'albero	8	-21	7	-21	7	-21

Tabella 4.7: Analisi delle sintesi per il comparatore RNS con confronto rispetto ai valori delle sintesi di default.

Infine, i risultati ottenuti per il comparatore RNS evidenziano il netto miglioramento nella rimozione dei vincoli sul tempo di transizione. La riduzione complessiva del numero di buffer di 267 nel caso di gruppi da 20 elementi e di 235 con il raggruppamento a 30 e 40, consente una grande riduzione della ridondanza complessiva, soprattutto se consideriamo la riduzione di ripetitori ridondanti sui livelli superiori che ha consentito la cospicua riduzione della profondità complessiva dell'albero di ben 21 buffer per percorso. Questa peculiarità ha portato nel caso migliore alla riduzione dello skew globale del $-65{,}9\%$, che è un risultato piacevole anche se non funzionale all'obiettivo globale di una riduzione dei consumi, considerando i valori nominalmente molto bassi visto il nodo tecnologico considerato.

Possiamo concludere che le modifiche apportate alla fase di caratterizzazione tecnologica e alla personalizzazione del massimo tempo di transizione sui nodi del tronco sono molto influenti soprattutto per i circuiti più complessi e con un elevato numero di sink.

Capitolo 5

Confronto con Innovus

In questo capitolo sono stati confrontati i risultati di TritonCTS con quelli di *Innovus*, lo strumento commerciale per l'implementazione fisica dei SoC sviluppato da *Cadence*.

È necessario evidenziare, però, che la versione attualmente disponibile del software effettua la sintesi dell'albero di clock con un approccio differente da quello del tipico flusso di implementazione mostrato in figura 2.1, unificando in una singola operazione, denominata *Clock Concurrent Optimization* (CCOpt), sia la definizione del circuito di distribuzione del segnale, sia l'ottimizzazione della temporizzazione dei percorsi di dato[12].

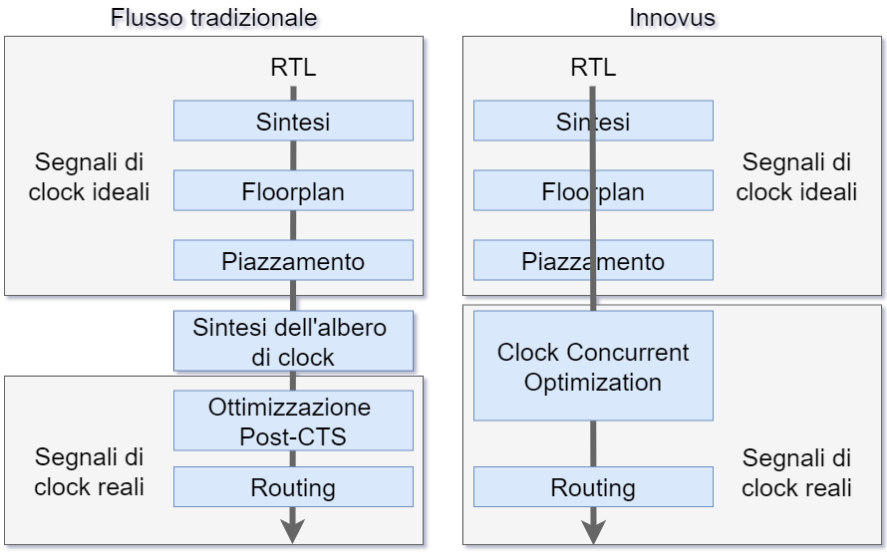

Figura 5.1: Confronto dei flussi di implementazione fisica tradizionale e di Innovus.

Le ragioni che hanno portato allo sviluppo di questa tecnica sono da ricondursi principalmente all'impatto delle variazioni on-chip (OCV) per le tecnologie più scalate ($65\,nm$ e $32\,nm$) e alla necessità di ridurre il consumo di potenza complessivo del circuito che ha portato all'impiego massiccio della circuiteria di clock gating, complicando ulteriormente la struttura di distribuzione del segnale di clock.

5.1 Sintesi con CCOpt

Il funzionamento interno del programma è coperto da stringenti accordi di non divulgazione, di conseguenza non è possibile conoscere nel dettaglio l'algoritmo utilizzato per la sintesi dell'albero in CCOpt. Da ciò che è noto, l'algoritmo definisce inizialmente un albero di distribuzione che rispetti i vincoli fisici di progetto, per poi apportare modifiche locali che correggano le violazioni dei tempi di setup e di hold. Di conseguenza, la minimizzazione dello skew nell'albero non rientra tra i principali obiettivi dell'algoritmo e se il bilanciamento dei percorsi richiede un impiego eccessivo di risorse, questo non viene semplicemente effettuato.

Questo tipo di algoritmi può essere definito di "seconda generazione" e le funzioni svolte, così come i suoi obiettivi differiscono da quelli di sintetizzatori più "classici" come TritonCTS. Confrontandoci con il supporto di Cadence e osservando il manuale d'uso fornito, è stata disabilitata l'ottimizzazione concorrenziale della temporizzazione dei percorsi di dato aggiungendo al comando "*ccopt_design*", che richiama la sintesi dell'albero di clock, l'opzione "*-cts*", rendendo l'operazione svolta confrontabile con quella di TritonCTS 2.0.

Inoltre, attraverso il comando "*set_ccopt_property*" sono stati imposti lo stesso tempo di transizione massimo ($1.5\,ns$) scelto in TritonCTS ed il numero massimo di elementi applicabili in uscita ad un buffer è stato modificato conformemente a quanto effettuato nei casi precedenti. Tra le altre proprietà è stato modificato anche il valore di skew obiettivo della sintesi, inserendo il valore risultante dalle sintesi di TritonCTS, ma per costruzione dell'algoritmo di CCOpt questo vincolo non è necessariamente rispettato. Nell'analisi dei parametri, inoltre, non è stata considerata la distanza effettiva dei componenti rispetto ai tratti di interconnessione utilizzati, bensì è stata considerata la distanza di Manhattan in conformità con l'analisi effettuata per TritonCTS.

I risultati del circuito RiscV, il numero (1) di tabella 4.1, sono riportati in tabella 5.1. Complessivamente le soluzioni dei due circuiti appaiono molto simili, con un lieve aumento della lunghezza dei percorsi massimi nella soluzione sintetizzata con CCOpt, ma una complessiva riduzione della lunghezza media dei percorsi. Considerando il numero di buffer che compongono l'albero, il numero complessivo non differisce di molto. Di contro è necessario evidenziare che la maggior parte dei buffer sono impiegati nell'ultimo livello dell'albero, quello definito attraverso

l'operazione di raggruppamento dei sink. Mentre per l'albero sintetizzato da TritonCTS sono stati utilizzati i buffer con driving strenght $x8$, nella soluzione con CCOpt sono stati impiegate soluzioni con dimensioni minori, utilizzando soprattutto buffer $x2$ e $x4$ determinando un netto risparmio delle risorse impiegate. Ciò evidenzia l'importanza di valutare buffer di diverse dimensioni nella fase di raggruppamento, caratteristica purtroppo assente in TritonCTS e che preclude l'ottimizzazione delle risorse.

	Numero massimo di sink per insieme					
Risc-V	20		30		40	
Lunghezza massima	850	+6%	869	+8%	870	+1%
Lunghezza media dei percorsi	650	-17%	735	-6%	725	-6%
Skew	0,11	+50%	0,05	-52%	0,18	+0%
Buffer inseriti	59	-2%	40	-5%	30	+7%
Profondità dell'albero	3	+0	3	+0	3	+0

Tabella 5.1: Confronto delle sintesi per il circuito Risc-V di tabella 4.1 effettuate con CCOpt rispetto ai valori delle sintesi di TritonCTS 2.0 modificato.

In figura 5.3 è evidenziata la struttura ad albero del circuito JPEG encoder di tabella 4.1 omettendo la visualizzazione delle interconnessioni per conformità con le immagini risultanti da TritonCTS.

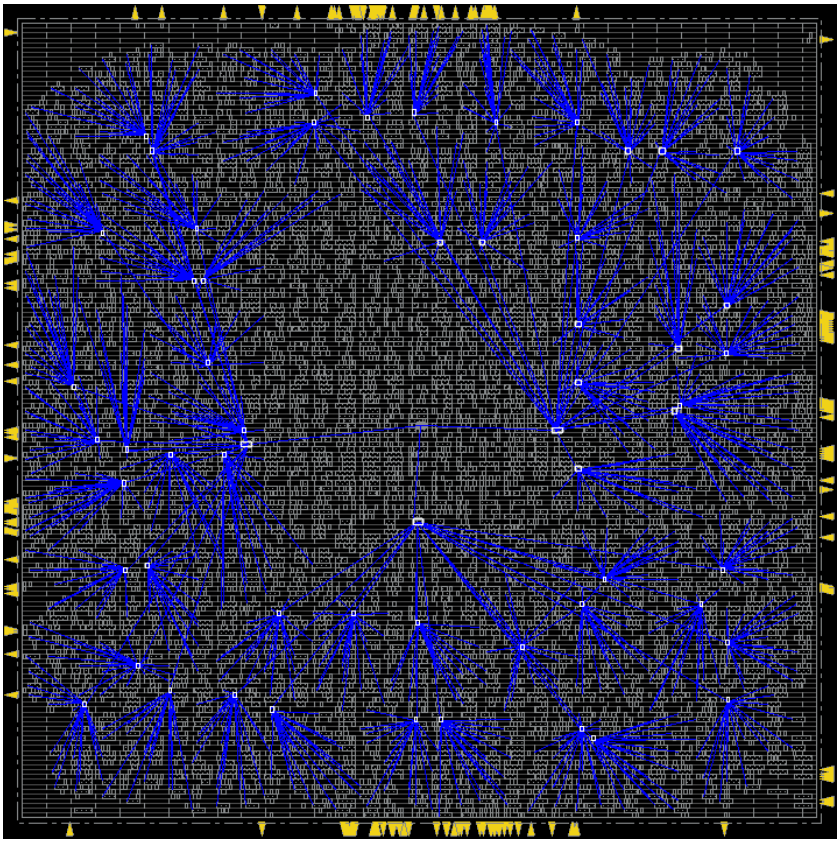

Figura 5.2: Struttura dell'albero di clock del circuito (1) di tabella 4.1 sintetizzato con Innovus con massimo fan-out pari a 20.

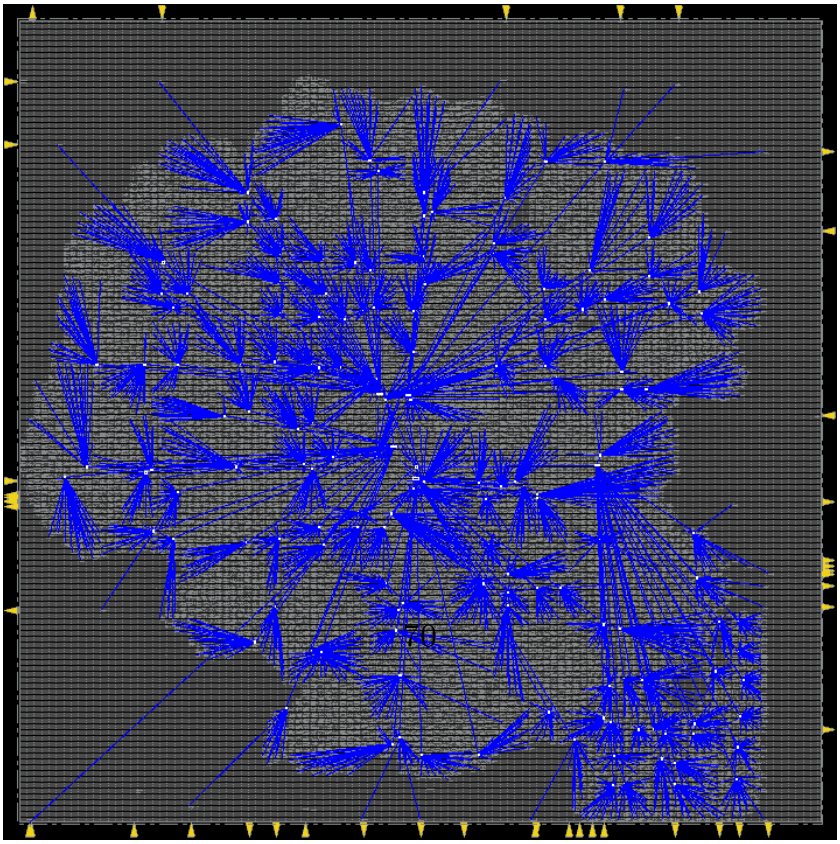

Figura 5.3: Struttura dell'albero di clock del circuito JPEG encoder di tabella 4.1 sintetizzato con Innovus con massimo fan-out pari a 30.

Come si evince dai parametri di 5.2, la soluzione di Innovus determina una riduzione delle risorse utilizzate a fronte di un cospicuo aumento proporzionale dello skew, che resta comunque entro valori molto contenuti per una tecnologia 130 nm. La lunghezza media dei percorsi riscontra una riduzione media del 32,3%, nonostante un generale aumento della lunghezza del percorso massimo. Ciò è possibile grazie alla diversa topologia utilizzata per il tronco, che assicura la creazione di soluzioni con un fattore di diramazione diverso da uno e generalmente associato al massimo fan-out specificato nella caratterizzazione dell'operazione. Inoltre, oltre alla diminuzione del numero totale di buffer utilizzati, si riscontra nuovamente un impiego di ripetitori a driving strength generalmente inferiore rispetto a quelli impiegati nella soluzione di TritonCTS.

	Numero massimo di sink per insieme					
(2)	20		30		40	
Lunghezza massima	2768	+15%	2774	+6%	2774	+13%
Lunghezza media dei percorsi	1534	-34%	1574	-32%	1614	-31%
Skew	0,194	+76%	0,22	+69%	0,27	+0%
Buffer inseriti	233	-3%	154	-8%	115	-7%
Profondità dell'albero	3-4	+0	3-4	+0	3-4	+0

Tabella 5.2: Confronto delle sintesi per il circuito (2) di tabella 4.1 effettuate con CCOpt rispetto ai valori delle sintesi di TritonCTS 2.0 modificato.

Una peculiarità dell'approccio di CCOpt, riguarda la creazione degli insiemi di sink. Come è possibile osservare nel dettaglio di figura 5.4, in molti insiemi il buffer che pilota l'insieme è posto a debita distanza dal centro dell'insieme. Questo si traduce in un generale aumento della distanza media tra sink e driver, che nel caso in esame con un fan-out massimo di 30 è pari a 76,1 μm, che rispetto ai 51,3 μm determina un aumento del 48%. Purtroppo, non conoscendo l'algoritmo di raggruppamento utilizzato non è possibile conoscere le ragioni dietro questa operazione.

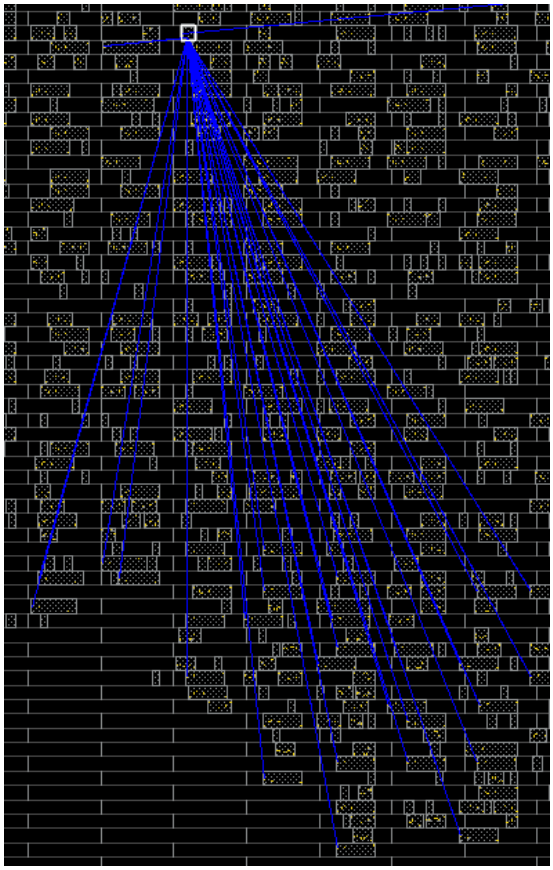

Figura 5.4: Esempio di cluster di 30 elementi sbilanciato definito nel circuito (2) di tabella 4.1 sintetizzato con Innovus.

Infine, nella tabella 5.3 sono riportati i parametri del circuito (3) di tabella 4.1. In particolare per questo design, l'uso di una topologia non binaria ha consentito di ridurre di molto sia la lunghezza media dei percorsi, come già evidenziato per i circuiti precedenti, sia la lunghezza del percorso massimo.

	Numero massimo di sink per insieme					
(3)	20		30		40	
Lunghezza massima	4259	-19%	4509	-11%	4339	-15%
Lunghezza media dei percorsi	2879	-44%	2914	-42%	2949	-41%
Skew	0,23	+21%	0,25	+56%	0,27	+80%
Buffer inseriti	1372	-10%	914	-5%	676	-9%
Profondità dell'albero	6	-2	6-8	+1	5-6	-1

Tabella 5.3: Confronto delle sintesi per il circuito (3) di tabella 4.1 effettuate con CCOpt rispetto ai valori delle sintesi di TritonCTS 2.0 modificato.

In particolare in figura 5.5 è riportato nel dettaglio il percorso a lunghezza maggiore. In questo caso è evidente come la struttura sintetizzata con CCOpt non preveda tratti di interconnessione ridondanti e in questo caso il percorso a distanza maggiore sia quello effettivamente più lontano dalla posizione della radice.

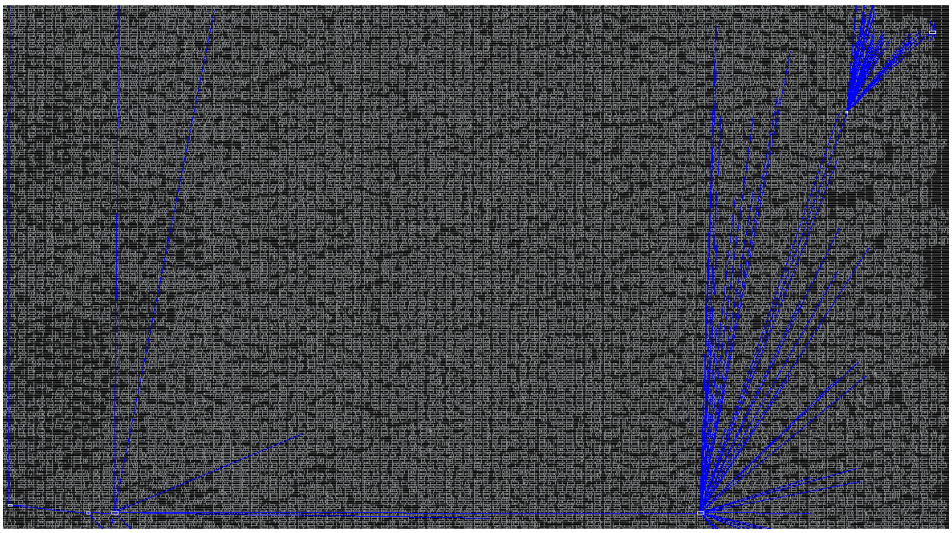

Figura 5.5: Dettaglio del percorso con lunghezza sorgente-sink maggiore dell'albero sintetizzato con Innovus con massimo fan-out pari a 40 del circuito (3) di tabella 4.1.

Capitolo 6

Conclusioni

TritonCTS 2.0 è risultato un valido strumento open-source che ha mostrato affidabilità nella sintesi dell'albero di distribuzione del clock per una varietà di design, indipendentemente dalle dimensioni del circuito e dalla numerosità ed irregolarità delle distribuzioni dei sink. I suoi punti di forza sono la facilità d'uso e l'automazione del flusso, che è totalmente compatibile sia in ingresso che in uscita con altri strumenti di implementazione fisica.

Anche grazie al confronto con gli sviluppatori, è emerso che l'utente medio del software è interessato alla sua semplicità. Di conseguenza, le sue operazioni mirano a creare un risultato rapidamente ed il suo flusso è ottimizzato per l'elaborazione di strutture semplici ed in presenza di un ridotto numero di variabili da considerare, limitando con vincoli interni i parametri di caratterizzazione della sintesi. Inoltre, l'uso di un'ampia selezione di buffer da considerare nella costruzione dell'albero risulta sconveniente, in quanto incrementa esponenzialmente i tempi di caratterizzazione tecnologica ed il loro impiego è considerato solo per i ripetitori del "tronco" dell'albero. Infatti, la scelta della cella utilizzata sia come radice, sia nella creazione dell'ultimo livello dell'albero, creato tramite raggruppamento dei sink, è delegata all'utente. Questo si traduce in un grado di libertà in meno nell'ottimizzazione automatica dell'albero di clock ed è una scelta algoritmica in disaccordo con il principio di totale automazione del progetto OpenROAD.

Grazie alla sua natura di software open-source è stato possibile aggiungere funzionalità e gradi di libertà nella personalizzazione della sintesi, che hanno permesso la creazione di alberi di clock meno ridondanti e quindi con una gestione delle risorse più ottimizzata. In particolare, è stato possibile ottimizzare la disposizione dei ripetitori lungo la disposizione dell'albero, riducendo mediamente la profondità per ramo dell'albero in circuiti di medie e grandi dimensioni del 70%. Una limitazione alla libertà di riprogrammazione dell'algoritmo è causata dall'assenza di

documentazione del database su cui si appoggia e su cui il team del progetto sta attualmente lavorando.

L'obiettivo della prima versione di TritonCTS di definire la topologia ottimale in termini di consumi di potenza dati un range di latenza e skew massimi, è stato completamente abbandonato in questa seconda versione. L'impiego dell'albero ad H generalizzato individuato in questa prima versione è la prospettiva più promettente per la sintesi di alberi a basso consumo di potenza, ma è necessario semplificare l'algoritmo di costruzione topologica, magari memorizzando in locale le soluzioni parziali analizzate così da evitare la saturazione della RAM dei terminali utilizzati. Questo può essere possibile una volta aggiornate le API di terze parti che impediscono sia la corretta lettura e scrittura dei file DEF e LEF, sia la gestione ottimale del database di soluzioni interno.

Dal confronto effettuato con CCOpt escludendo l'ottimizzazione della temporizzazione dei percorsi dei dati è emerso che per design di piccole dimensioni e con una distribuzione omogenea dei nodi terminali, gli alberi risultanti hanno caratteristiche molto simili. Per circuiti di medie e grandi dimensioni, invece, i risultati di CCOpt presentano una forte riduzione sia della lunghezza media dei percorsi, sia del numero e delle dimensioni dei buffer impiegati nella struttura, ottenendo un risparmio di risorse impiegate notevole a spese di un aumento dello skew globale nell'ordine dei decimali di ps, che per una tecnologia $130\,nm$ come quella considerata risulta ampiamente accettabile. Per compensare il divario il principale punto di forza del progetto TritonCTS è la collettività di sviluppatori che lavora al programma, sia interni al DARPA, l'agenzia governativa del Dipartimento della Difesa degli Stati Uniti che finanzia lo sviluppo del progetto OpenROAD, sia appassionati che tramite il forum *GitHub* collaudano il programma e forniscono spunti importanti per il miglioramento del software.

Appendice A

Makefile per richiamare TritonCTS in OpenROAD

```
DESIGN_CONFIG=./designs/sky130hd/riscv32/config.mk
ifndef DISPLAY
export QT_QPA_PLATFORM ?= offscreen
endif

.DEFAULT_GOAL := finish

SHELL           = /bin/bash
.SHELLFLAGS     = -o pipefail -c

export FLOW_HOME ?= .
export DESIGN_HOME    ?= $(FLOW_HOME)/designs
export PLATFORM_HOME ?= $(FLOW_HOME)/platforms
export WORK_HOME      ?= .
export UTILS_DIR      ?= $(FLOW_HOME)/util
export SCRIPTS_DIR    ?= $(FLOW_HOME)/scripts
export TEST_DIR       ?= $(FLOW_HOME)/test
# Include design and platform configuration
include $(DESIGN_CONFIG)
PUBLIC=nangate45 sky130hd sky130hs asap7
ifneq ($(wildcard $(PLATFORM_HOME)/$(PLATFORM)),)
  export PLATFORM_DIR = $(PLATFORM_HOME)/$(PLATFORM)
else ifneq ($(findstring $(PLATFORM),$(PUBLIC)),)
  export PLATFORM_DIR = ./platforms/$(PLATFORM)
else ifneq ($(wildcard ../../$(PLATFORM)),)
  export PLATFORM_DIR = ../../$(PLATFORM)
else
  $(error [ERROR][FLOW] Platform '$(PLATFORM)' not found.)
```

```
29 endif
30
31 $(info [INFO][FLOW] Using platform directory $(PLATFORM_DIR))
32 include $(PLATFORM_DIR)/config.mk
33
34 # Setup working directories
35 export DESIGN_NICKNAME ?= $(DESIGN_NAME)
36 export DESIGN_DIR  = $(dir $(DESIGN_CONFIG))
37 export LOG_DIR     = $(WORK_HOME)/logs/$(PLATFORM)/$(DESIGN_NICKNAME)
       /$(FLOW_VARIANT)
38 export OBJECTS_DIR = $(WORK_HOME)/objects/$(PLATFORM)/$(
       DESIGN_NICKNAME)/$(FLOW_VARIANT)
39 export REPORTS_DIR = $(WORK_HOME)/reports/$(PLATFORM)/$(
       DESIGN_NICKNAME)/$(FLOW_VARIANT)
40 export RESULTS_DIR = $(WORK_HOME)/results/$(PLATFORM)/$(
       DESIGN_NICKNAME)/$(FLOW_VARIANT)
41 export CITIESS_DIR = $(WORK_HOME)/citiess/$(PLATFORM)/$(
       DESIGN_NICKNAME)
42
43 export CTS_CLUSTER_SIZE     = 20
44 export CTS_CLUSTER_DIAMETER = 100
45
46 ifdef BLOCKS
47   $(info [INFO][FLOW] Invoked hierarchical flow.)
48   $(foreach block,$(BLOCKS),$(info Block ${block} needs to be
       hardened.))
49   $(foreach block,$(BLOCKS),$(eval BLOCK_CONFIGS += ./designs/$(
       PLATFORM)/$(DESIGN_NICKNAME)/$(block)/config.mk))
50   $(foreach block,$(BLOCKS),$(eval BLOCK_LEFS += ./results/$(PLATFORM
       )/$(DESIGN_NICKNAME)_$(block)/$(FLOW_VARIANT)/${block}.lef))
51   $(foreach block,$(BLOCKS),$(eval BLOCK_GDS += ./results/$(PLATFORM)
       /$(DESIGN_NICKNAME)_$(block)/$(FLOW_VARIANT)/6_final.gds))
52   export ADDITIONAL_LEFS += $(BLOCK_LEFS)
53   export ADDITIONAL_GDS += $(BLOCK_GDS)
54   export GDS_FILES += $(BLOCK_GDS)
55 endif
56
57 ifeq (, $(strip $(NPROC)))
58   # Linux (utility program)
59   NPROC := $(shell nproc 2>/dev/null)
60   ifeq (, $(strip $(NPROC)))
61     # Linux (generic)
62     NPROC := $(shell grep -c ^processor /proc/cpuinfo 2>/dev/null)
63   endif
64   ifeq (, $(strip $(NPROC)))
65     # BSD (at least FreeBSD and Mac OSX)
66     NPROC := $(shell sysctl -n hw.ncpu 2>/dev/null)
67   endif
68   ifeq (, $(strip $(NPROC)))
```

```
    # Fallback
    NPROC := 1
  endif
endif
export NUM_CORES := $(NPROC)
export LSORACLE_CMD ?= $(shell command -v lsoracle)
ifeq ($(LSORACLE_CMD),)
  LSORACLE_CMD = $(abspath $(FLOW_HOME)/../tools/install/LSOracle/bin/lsoracle)
endif

# setup all commands used within this flow

TIME_CMD = /usr/bin/time -f 'Elapsed time: %E[h:]min:sec. CPU time: user %U sys %S (%P). Peak memory: %MKB.'
TIME_TEST = $(shell $(TIME_CMD) echo foo 2>/dev/null)
ifeq (, $(strip $(TIME_TEST)))
  TIME_CMD = /usr/bin/time
endif

OPENROAD_EXE               ?= $(shell command -v openroad)
ifeq ($(OPENROAD_EXE),)
  OPENROAD_EXE              = $(abspath $(FLOW_HOME)/../tools/install/OpenROAD/bin/openroad)
endif
OPENROAD_ARGS              = -no_init $(OR_ARGS)
OPENROAD_CMD               = $(OPENROAD_EXE) -exit $(OPENROAD_ARGS)
OPENROAD_NO_EXIT_CMD       = $(OPENROAD_EXE) $(OPENROAD_ARGS)
OPENROAD_GUI_CMD           = $(OPENROAD_EXE) -gui $(OR_ARGS)

YOSYS_CMD                  ?= $(shell command -v yosys)
ifeq ($(YOSYS_CMD),)
  YOSYS_CMD                = $(abspath $(FLOW_HOME)/../tools/install/yosys/bin/yosys)
endif

KLAYOUT_CMD                ?= $(shell command -v klayout)
KLAYOUT_FOUND              = $(if $(KLAYOUT_CMD),,$(error KLayout not found in PATH))

WRAPPED_LEFS = $(foreach lef,$(notdir $(WRAP_LEFS)),$(OBJECTS_DIR)/lef/$(lef:.lef=_mod.lef))
WRAPPED_LIBS = $(foreach lib,$(notdir $(WRAP_LIBS)),$(OBJECTS_DIR)/$(lib:.lib=_mod.lib))
export ADDITIONAL_LEFS += $(WRAPPED_LEFS) $(WRAP_LEFS)
export LIB_FILES += $(WRAP_LIBS) $(WRAPPED_LIBS)
export DONT_USE_LIBS    = $(addprefix $(OBJECTS_DIR)/lib/, $(notdir $(LIB_FILES)))
```

```
109 export DONT_USE_SC_LIB ?= $(OBJECTS_DIR)/lib/$(notdir $(firstword $(
    LIB_FILES)))
110 # Stream system used for final result (GDS is default): GDS, GSDII,
    GDS2, OASIS, or OAS
111 STREAM_SYSTEM ?= GDS
112 ifneq ($(findstring GDS,$(shell echo $(STREAM_SYSTEM) | tr '[:lower:
    ]' '[:upper:]')),)
113     export STREAM_SYSTEM_EXT := gds
114     GDSOAS_FILES = $(GDS_FILES)
115     ADDITIONAL_GDSOAS = $(ADDITIONAL_GDS)
116     SEAL_GDSOAS = $(SEAL_GDS)
117 else
118     export STREAM_SYSTEM_EXT := oas
119     GDSOAS_FILES = $(OAS_FILES)
120     ADDITIONAL_GDSOAS = $(ADDITIONAL_OAS)
121     SEAL_GDSOAS = $(SEAL_OAS)
122 endif
123 export WRAPPED_GDSOAS = $(foreach lef,$(notdir $(WRAP_LEFS)),$(
    OBJECTS_DIR)/$(lef:.lef=_mod.$(STREAM_SYSTEM_EXT)))
124 # Targets to harden Blocks in case of hierarchical flow is triggered
125 build_macros: $(BLOCK_LEFS)
126 $(foreach block,$(BLOCKS),$(eval ./results/$(PLATFORM)/$(
    DESIGN_NICKNAME)_$(block)/$(FLOW_VARIANT)/${block}.lef: ./designs/
    $(PLATFORM)/$(DESIGN_NICKNAME)/${block}/config.mk))
127 $(foreach block,$(BLOCKS),$(eval ./results/$(PLATFORM)/$(
    DESIGN_NICKNAME)_$(block)/$(FLOW_VARIANT)/6_final.gds: ./results/$
    (PLATFORM)/$(DESIGN_NICKNAME)_$(block)/$(FLOW_VARIANT)/${block}
    .lef))
128 $(BLOCK_LEFS):
129     $(MAKE) DESIGN_CONFIG=$< generate_abstract
130 .SECONDEXPANSION:
131 $(DONT_USE_LIBS): $$(filter %$$(@F),$(LIB_FILES))
132     @mkdir -p $(OBJECTS_DIR)/lib
133     $(UTILS_DIR)/markDontUse.py -p "$(DONT_USE_CELLS)" -i $^ -o $@
134 $(OBJECTS_DIR)/lib/merged.lib:
135     $(UTILS_DIR)/mergeLib.pl $(PLATFORM)_merged $(DONT_USE_LIBS) > $@
136 $(OBJECTS_DIR)/klayout_tech.lef: $(TECH_LEF)
137     @mkdir -p $(OBJECTS_DIR)
138     sed '/OR_DEFAULT/d' $< > $@
139  $(OBJECTS_DIR)/klayout.lyt: $(KLAYOUT_TECH_FILE) $(OBJECTS_DIR)/
    klayout_tech.lef
140     sed 's,<lef-files>.*</lef-files>,$(foreach file, $(OBJECTS_DIR)/
    klayout_tech.lef $(SC_LEF) $(ADDITIONAL_LEFS),<lef-files>$(abspath
     $(file))</lef-files>),g' $< > $@
141 $(OBJECTS_DIR)/klayout_wrap.lyt: $(KLAYOUT_TECH_FILE) $(OBJECTS_DIR)/
    klayout_tech.lef
142     sed 's,<lef-files>.*</lef-files>,$(foreach file, $(OBJECTS_DIR)/
    klayout_tech.lef $(WRAP_LEFS),<lef-files>$(abspath $(file))</
    lef-files>),g' $< > $@
```

```
143 WRAP_CFG = $(PLATFORM_DIR)/wrapper.cfg
144
145 export TCLLIBPATH := util/cell-veneer $(TCLLIBPATH)
146 $(WRAPPED_LEFS):
147     mkdir -p $(OBJECTS_DIR)/lef $(OBJECTS_DIR)/def
148     util/cell-veneer/wrap.tcl -cfg $(WRAP_CFG) -macro $(filter %$(
    notdir $(@:_mod.lef=.lef)),$(WRAP_LEFS))
149     mv $(notdir $@) $@
150     mv $(notdir $(@:lef=def)) $(dir $@)../def/$(notdir $(@:lef=def))
151 $(WRAPPED_LIBS):
152     mkdir -p $(OBJECTS_DIR)/lib
153     sed 's/library(\(.*\))/library(\1_mod)/g' $(filter %$(notdir $(
    @:_mod.lib=.lib)),$(WRAPPED_LIBS)) | sed 's/cell(\(.*\))/cell(\1_mod)
    /g' > $@
154
155 finish: $(RESULTS_DIR)/4_cts.def \
156      $(RESULTS_DIR)/4_cts.sdc
157 # Run TritonCTS
158
159 $(RESULTS_DIR)/4_1_cts.def: $(CITIESS_DIR)/$(DESIGN_NICKNAME).def $(
    CITIESS_DIR)/$(DESIGN_NICKNAME).sdc
160     ($(TIME_CMD) $(OPENROAD_CMD) $(SCRIPTS_DIR)/cts.tcl -metrics $(
    LOG_DIR)/4_1_cts.json) 2>&1 | tee $(LOG_DIR)/4_1_cts.log
161
162 # Filler cell insertion
163
164 $(RESULTS_DIR)/4_2_cts_fillcell.def: $(RESULTS_DIR)/4_1_cts.def
165     ($(TIME_CMD) $(OPENROAD_CMD) $(SCRIPTS_DIR)/fillcell.tcl -metrics
     $(LOG_DIR)/4_2_cts_fillcell.json) 2>&1 | tee $(LOG_DIR)/4
    _2_cts_fillcell.log
166 $(RESULTS_DIR)/4_cts.sdc: $(RESULTS_DIR)/4_cts.def
167 $(RESULTS_DIR)/4_cts.def: $(RESULTS_DIR)/4_2_cts_fillcell.def
168     cp $< $@
169 clean_cts:
170     rm -rf $(RESULTS_DIR)/4_*cts*.def $(RESULTS_DIR)/4_cts.sdc $(
    RESULTS_DIR)/4_cts.v
171     rm -f  $(REPORTS_DIR)/4_*
172     rm -f  $(LOG_DIR)/4_*
```

Appendice B

Script di caratterizzazione della sintesi

```
if {![info exists standalone] || $standalone} {
  # Read lef
  read_lef $::env(TECH_LEF)
  read_lef $::env(SC_LEF)
  if {[info exist ::env(ADDITIONAL_LEFS)]} {
    foreach lef $::env(ADDITIONAL_LEFS) {
      read_lef $lef
    }
  }

  # Read liberty files
  source $::env(SCRIPTS_DIR)/read_liberty.tcl
  # Read design files
  read_def $::env(CITIESS_DIR)/$::env(DESIGN_NICKNAME).def
  # Read SDC file
  read_sdc $::env(CITIESS_DIR)/$::env(DESIGN_NICKNAME).sdc
  if [file exists $::env(PLATFORM_DIR)/derate.tcl] {
    source $::env(PLATFORM_DIR)/derate.tcl
  }
} else {
  puts "Starting CTS"
}

# Clone clock tree inverters next to register loads
# so cts does not try to buffer the inverted clocks.
repair_clock_inverters
source $::env(PLATFORM_DIR)/setRC.tcl

```

```
29 # Run CTS
30 if {[info exist ::env(CTS_CLUSTER_SIZE)]} {
31   set cluster_size "$::env(CTS_CLUSTER_SIZE)"
32 } else {
33   set cluster_size 40
34 }
35 if {[info exist ::env(CTS_CLUSTER_DIAMETER)]} {
36   set cluster_diameter "$::env(CTS_CLUSTER_DIAMETER)"
37 } else {
38   set cluster_diameter 10000
39 }
40
41 clock_tree_synthesis -root_buf [list "sky130_fd_sc_hd__buf_8"] \
42               -buf_list [list "sky130_fd_sc_hd__buf_16" "
    sky130_fd_sc_hd__buf_8" "sky130_fd_sc_hd__buf_4" "
    sky130_fd_sc_hd__buf_2" "sky130_fd_sc_hd__buf_1"] \
43               -post_cts_disable \
44                       -sink_clustering_enable \
45                       -sink_clustering_size $cluster_size \
46                       -sink_clustering_max_diameter $cluster_diameter
47
48 report_cts [-out_file "Out.txt"]
49
50 set_propagated_clock [all_clocks]
51
52 set_dont_use $::env(DONT_USE_CELLS)
53 source $::env(SCRIPTS_DIR)/report_metrics.tcl
54
55 estimate_parasitics -placement
56 report_metrics "cts pre-repair"
57 #repair_clock_nets
58 #estimate_parasitics -placement
59 #report_metrics "cts post-repair"
60
61 set_placement_padding -global \
62     -left $::env(CELL_PAD_IN_SITES_DETAIL_PLACEMENT) \
63     -right $::env(CELL_PAD_IN_SITES_DETAIL_PLACEMENT)
64 detailed_placement
65 estimate_parasitics -placement
66
67 puts "Repair setup violations..."
68 # process user settings
69 set additional_args ""
70 if { [info exists ::env(SETUP_SLACK_MARGIN)] && $::env(
    SETUP_SLACK_MARGIN) > 0.0} {
71   puts "Setup repair with slack margin $::env(SETUP_SLACK_MARGIN)"
72   append additional_args " -slack_margin $::env(SETUP_SLACK_MARGIN)"
73 }
74
```

```
75 #repair_timing -setup {*}$additional_args
76 puts "Repair hold violations..."
77 # process user settings
78 set additional_args ""
79 if { [info exists ::env(HOLD_SLACK_MARGIN)] && $::env(
       HOLD_SLACK_MARGIN) > 0.0} {
80   puts "Hold repair with slack margin $::env(HOLD_SLACK_MARGIN)"
81   append additional_args " -slack_margin $::env(HOLD_SLACK_MARGIN)"
82 }
83 #repair_timing -hold {*}$additional_args
84 detailed_placement
85 check_placement -verbose
86 report_metrics "cts final"
87 if {![info exists standalone] || $standalone} {
88   # write output
89   write_def $::env(RESULTS_DIR)/4_1_cts.def
90   write_verilog $::env(RESULTS_DIR)/4_cts.v
91   write_sdc $::env(RESULTS_DIR)/4_cts.sdc
92   # post CTS user TCL script hook
93   if { [info exists ::env(POST_CTS_TCL)] } {
94     source $::env(POST_CTS_TCL)
95   }
96   exit
97 }
```

Appendice C

Script di CCOpt

```
set_message -id IMPESI-3014 -severity info

setNanoRouteMode -reset -dbViaWeight

setNanoRouteMode -dbViaWeight {@1cut 1}
setNanoRouteMode -droutePostRouteSwapVia multiCut
setNanoRouteMode -drouteUseMultiCutViaEffort high

set_message -id IMPCCOPT-5067 -severity info

reset_ccopt_config
delete_ccopt_clock_tree_spec

set_ccopt_property update_io_latency false

setAnalysisMode -analysisType onChipVariation -cppr both

unset -nocomplain sparkinn_route_type_clk sparkinn_route_type_leaf
set sparkinn_route_type_clk  "create_route_type -name clk"
set sparkinn_route_type_leaf "create_route_type -name leaf"
append sparkinn_route_type_leaf " -top_preferred_layer met5
    -bottom_preferred_layer met1"
append sparkinn_route_type_clk " -top_preferred_layer met5
    -bottom_preferred_layer met1"

eval $sparkinn_route_type_clk
eval $sparkinn_route_type_leaf
unset -nocomplain sparkinn_route_type_clk sparkinn_route_type_leaf

set_ccopt_property -net_type top  -net clknet_0_clk route_type clk
set_ccopt_property -net_type trunk route_type clk
```

```
30 set_ccopt_property -net_type leaf  route_type leaf
31
32 set_ccopt_property buffer_cells [list "sky130_fd_sc_hd__buf_16" "
       sky130_fd_sc_hd__buf_8" "sky130_fd_sc_hd__buf_4" "
       sky130_fd_sc_hd__buf_2" "sky130_fd_sc_hd__buf_1"]
33
34 set_ccopt_property primary_delay_corner my_rc
35 set_ccopt_property target_max_trans 1.5
36 set_ccopt_property target_skew 0.19
37 set_ccopt_property max_fanout 20
38
39 create_ccopt_clock_tree_spec -file ./CTS/ccopt.spec
40
41 set_ccopt_property use_inverters true/false
42
43 ccopt_design -check_prerequisites
44
45 ccopt_design -cts -expandedViews -outDir ./CTS_Reports -prefix
       CTS_Setup
46
47 set dbgDefOutLefVias 1
48 set_message -id IMPDF-1012 -suppress
49 defOut -floorplan -netlist -noTracks -routing -placement ./CTS.def
50 set_message -id IMPDF-1012 -unsuppress
51
52 report_ccopt_clock_trees
53 report_ccopt_clock_tree_convergence
54 report_ccopt_clock_tree_structure
55 report_ccopt_skew_groups
56 report_ccopt_worst_chain
```

Bibliografia

[1] A. B. Kahng, J. Lienig, I. L. Markov e J. Hu. *VLSI Physical Design: From Graph Partitioning to Timing Closure*. 1st. Springer Publishing Company, Incorporated, 2011. ISBN: 9789048195909 (cit. alle pp. 3, 9).

[2] C. J. Alpert, D. P. Mehta e S. S. Sapatnekar. *Handbook of Algorithms for Physical design Automation*. CRC Press, 2009 (cit. a p. 10).

[3] Thucydides X. *Clocking in Modern VLSI Systems*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 1441902600 (cit. a p. 10).

[4] R.S. Tsay. «Exact Zero Skew». In: *IEEE Trans. Comput.-Aided Des.* 21 (dic. 1991), pp. 336–339. DOI: 10.1109/ICCAD.1991.185269 (cit. a p. 13).

[5] R. S. Shelar. «An Efficent Clustering Algorithm for Low Power Clock Tree Synthesis». In: ISPD '07 (2007), pp. 181–188. DOI: 10.1145/1231996.1232037. URL: https://doi.org/10.1145/1231996.1232037 (cit. a p. 17).

[6] A.D. Mehta, Yao-Ping Chen, N. Menezes, D.F. Wong e L.T. Pilegg. «Clustering and load balancing for buffered clock tree synthesis». In: *Proceedings International Conference on Computer Design VLSI in Computers and Processors*. 1997, pp. 217–223. DOI: 10.1109/ICCD.1997.628871 (cit. a p. 19).

[7] R. S. Shelar. «A Fast and Near-Optimal Clustering Algorithm for Low-Power Clock Tree Synthesis». In: 31.11 (nov. 2012), pp. 1781–1786. ISSN: 0278-0070. DOI: 10.1109/TCAD.2012.2206592. URL: https://doi.org/10.1109/TCAD.2012.2206592 (cit. a p. 20).

[8] Qiang Zhou Chao Deng Yi-Ci Cai. «Register Clustering Methodology for Low Power Clock Tree Synthesis». In: *Journal of Computer Science and Technology* 30.2, 391 (2015), p. 391. DOI: 10.1007/s11390-015-1531-4. URL: https://jcst.ict.ac.cn/EN/abstract/article_2136.shtml (cit. a p. 28).

[9] Chang Xu, Guojie Luo, Peixin Li, Yiyu Shi e Iris Hui-Ru Jiang. «Analytical Clustering Score with Application to Postplacement Register Clustering». In: *ACM Trans. Des. Autom. Electron. Syst.* 21.3 (mag. 2016). ISSN: 1084-4309. DOI: 10.1145/2894753. URL: https://doi.org/10.1145/2894753 (cit. a p. 29).

[10] S. Lerner, E. Leggett e B. Taskin. «Slew-down: analysis of slew relaxation for low-impact clock buffers». In: (2017), pp. 1–4. DOI: 10.1109/SLIP.2017.7974910 (cit. a p. 34).

[11] A. B. Kahng, J. Li e L. Wang. «Improved flop tray-based design implementation for power reduction». In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2016, pp. 1–8. DOI: 10.1145/2966986.2967047 (cit. a p. 45).

[12] P. Cunningham e S. Wilcox. «Clock Concurrent Optimization, Rethinking Timing Optimization to Target Clocks and Logic at the Same Time». In: *Cadence Design Systems, Inc. All rights reserved. Cadence and the Cadence logo are registered trademarks of Cadence Design Systems, Inc. All rights reserved*. 2011. URL: http://www10.edacafe.com/link/Clock-Concurrent-Optimization-Timing-Clocks-Logic-Same-Time/34504/link_download/No/Clock_Concurrent_Opt_WP_V2.pdf (cit. a p. 67).