POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Elettronica

Sviluppo di una Metodologia per la Verifica Accurata di Path Critici di System-on-Chip Nanometrici

Relatore: Prof. Giancarlo RIPAMONTI
Correlatore: Salvatore SANTAPÀ

Tesi di Laurea Magistrale di:
Andrea BARLETTA
Matricola 766073

Anno Accademico 2012/2013

Indice

Indice 2

Lista delle Figure 4

Introduction 6

Introduzione 9

1 Progettazione VLSI 12

2 Flussi di Verifica e Sistemi CAD 22

2.1 TIPI DI SIMULAZIONE 22
2.2 PRINCIPALI PRODOTTI CAD UTILIZZATI 24
2.2.1 Estrazione dei parassiti: Star-RCXT™ 24
2.2.2 Simulazione statica: PrimeTime 26
2.2.3 Simulazione dinamica: Eldo 28
2.2.4 Simulazione dinamica: CustomSim-XA 28
2.2.5 Analisi dei dati: Custom WaveView 30
2.2.6 Analisi dei dati: gnuplot 30

3 Static Timing Analysis 31

3.1 TIMING PATHS 32
3.2 STA VS TRANSISTOR-LEVEL TIMING VERIFICATION 40
3.2.1 Corners tecnologici 40
3.2.2 Necessità di ridurre il pessimismo 41
3.3 SIGNAL INTEGRITY 43

4 Il Nuovo Flusso: Accurate Path Verification 46

4.1 DESCRIZIONE DELL'INTERFACCIA GRAFICA 49
4.1.1 Ruolo di tasti e tabelle 49
4.1.2 Server fra APV e PrimeTime 50
4.1.3 Tipi di analisi disponibili 52
4.1.4 Variabili d'ambiente 55
4.2 DEFINIZIONE DEI PATH CRITICI 58
4.2.1 Premesse sul Signoff 58

4.2.2 Flessibilità nella scelta dei path critici ... *59*
4.3 COLLEZIONE DELLE INFORMAZIONI ... 60
4.3.1 Risoluzione dei conflitti ... *61*
4.3.2 Attributi generici del design ... *65*
4.3.3 Attributi dei clock ... *65*
4.3.4 Attributi dei path ... *66*
4.4 GENERAZIONE DELLA NETLIST SPICE ... 71
4.4.1 Top.cir ... *73*
4.4.2 spiceFinal#.spi ... *74*
4.4.3 opcond#.spi ... *76*
4.4.4 probe#.spi ... *78*
4.4.5 DSPF#.dspf e ptReducedSPEF#.spef ... *81*
4.4.6 dumpLibrary.spi ... *81*
4.4.7 corners.spi ... *81*
4.4.8 Altri files ... *82*
4.5 ANALISI E BACK-ANNOTAZIONE DEI RISULTATI ... 83
4.5.1 Simulazione dinamica iterativa ... *83*
4.5.2 Parsing dei .measure ... *84*
4.6 MODELLIZZAZIONE DEL CROSSTALK ... 87
4.6.1 Collezione delle informazioni sugli aggressori ... *87*
4.6.2 Necessità di introdurre pessimismo ... *90*
4.6.3 Aggressori in caso di clock-tree analysis ... *93*
4.7 DEVIAZIONI DAL FLUSSO APV ... 94
4.7.1 Clock-tree analysis ... *94*
4.7.2 Simulazioni manuali: NetFlow ... *94*
4.7.3 Simulazioni AMS ... *95*

5 Case studies ... 97

5.1 VERIFICA DI “PICCOLE” VIOLAZIONI DI SETUP ... 98
5.2 ANALISI DELLA CLOCK-TREE: POWER/EMI ... 106

Conclusioni e Sviluppi Futuri ... 110

Appendice A: SPICE ... 113

Appendice B: SPEF/DSPF ... 119

Appendice C: Verilog-A ... 123

Bibliografia ... 128

Ringraziamenti ... 131

Lista delle Figure

Figura 1. Legge di Moore: andamento del numero di transistori per singolo microprocessore....... 13

Figura 2. Evoluzione dei System on Chip... 14

Figura 3. VLSI verification challenges... 15

Figura 4. IPR (Intellectual Properties Reuse) 16

Figura 5. Passi del Flusso di Progettazione per Circuiti Integrati 19

Figura 6. Modello RC distribuito ricavato da layout per tre net... 26

Figura 7. STA nel flusso di progettazione 31

Figura 8. Esempi di data-path.. 32

Figura 9. Alcuni path particolari.. 33

Figura 10. Reg-to-reg data-path con clock attivi sul fronte di salita 34

Figura 11. Scelta dei fronti di clock da verificare.. 35

Figura 12. Andamento del delay delle celle al variare delle condizioni operative 41

Figura 13. Esempio di propagazione dei fronti con analisi OCV (in rosso) e PBA (in verde)........ 42

Figura 14. Aumento delle capacità di coupling con lo scaling tecnologico 43

Figura 15. Esempio di bump di tensione indotti in grado di anticipare/ritardare il fronte vittima... 44

Figura 16. Trade-off tra accuratezza e velocità di vari tipi di simulazione 46

Figura 17. APV Flow 47

Figura 18. Interfaccia grafica all'avvio.. 49

Figura 19. Console agganciata a PrimeTime 51

Figura 20. Menu delle preferenze 56

Figura 21. Environment variables 57

Figura 22. Prima fase del Flusso APV 60

Figura 23. Esempio di risoluzione di un conflitto fra load e path cells 62

Figura 24. Esempio di attributi di un oggetto path di PrimeTime 67

Figura 25. Tabella della GUI riempita con le informazioni prese da PrimeTime 70

Figura 26. Seconda parte del Flusso APV 71

Figura 27. Struttura dei file nell'ambiente UNIX 72

Figura 28. Esempio di connessione di due moduli Verilog-A per modellizzare il crosstalk 90

Figura 29. Clock reconvergence pessimism removal (CRPR) 92

Figura 30. Schema di funzionamento del NetFlow 95

Figura 31. Schema a blocchi di un complesso Phase Locked Loop 99

Figura 32. Interfaccia grafica con i dati della Static Timing Analysis 100

Figura 33. Tabella dei risultati al termine della simulazione 101

Figura 34. Schematico di due path registro-registro senza clock-tree 102

Figura 35. Forme d'onda simulate di un aggressore (verde) che induce un bump negativo sulla transizione rise di una net vittima (gialla) 104

Figura 36. Principali forme d'onda per un path (con e senza crosstalk) 105

Figura 37. Corrente istantanea fornita dall'alimentazione (in rosso), rispettiva FFT (in giallo, scala lineare, in verde, scala in dB_A) e clock principale (in arancione) 106

Figura 38. Snapshot durante un'analisi di clock-tree 107

Figura 39. Risultati dell'analisi di clock 108

Figura 40. Menu tasto destro 109

Figura 41. Distribuzione degli slew della clock-tree 109

Figura 42. Esempio di parassiti per due net 121

An Accurate Path Verification Flow to Secure and Speed Up Nanometer Design Closure

Introduction

Given the continuing demand for electronic devices with ever-greater performance, modern-day integrated circuits contain hundreds of millions of transistors. More and more, they are *System-on-Chip* (SoC), having analog, digital and RF blocks within a single chip, with significant complexity increase of the design and test steps.

Time-to-market and costs reduction make all more challenging. Could be successful, therefore, making the verification flow safer and faster, in order to reduce the number of design iterations (called ECO from Engineering Change Order) and avoid silicon failures.

The main problem is that it is impossible to simulate in the analog world circuits so big and complex. It is mandatory to use static analyses that are indeed very fast, but intrinsically pessimistic. The risk is being too much conservative, limiting the actual achievable performance or, worse, having unexpected failures.

The object of this thesis is to provide a new verification flow that supports, instead of replacing, the conventional Static Timing Analysis, making it more safe and accurate. The new flow is mainly meant for digital designers, it is their job to decide if the design can be

taped-out on silicon or it requires some changes on its critical paths first. A simple GUI (Graphical User Interface) has been specifically created to help the designer through the entire verification flow.
The choice of making this flow an integrative verification method, rather than an alternative one, has been taken because its main purpose it is to break the speed/accuracy trade-off that exists between static and dynamic analyses. In order to achieve this it is necessary to range over both simulation environments, static and dynamic ones. The flow takes full advantage of the **speed** of the *Static Timing Analysis* to build a small spice netlist on which to make accurate timing measurements (and willingly current/voltage/power measurements) with **accurate** analog simulations. The accuracy lies in the back-annotation of the netlist with post-layout parasitic that let you simulate better line delays and most of all crosstalk effects between adjacent lines.

Such a timing analysis adds robustness to the design flow, thus allowing to explore design implementation and verification tasks not so easy to achieve in the frame of a pure digital flow: accurate skew measurements on wide busses, clock tree implementations driven by power and EMI criteria, design quotation in advanced and/or preliminary technologies, measurements outside the standard libraries domain characterization.

This thesis has been the center of my one-year internship with the STMicroelectronics of Agrate Brianza (Milan), in the *Smart Power - AMS Design & Verification* group.
At the beginning, I had to learn some circuit description languages like SPICE, VHDL and Verilog-A and the programming language Tcl/Tk in the UNIX environment in order to build the computational engine and the graphical interface. It was also necessary to gain knowledge of the way of working of some simulation tools for being able to exploit them to the fullest and to interact with them in order to capture the info needed for building the netlist to be analyzed. During the flow development it has been needed being in touch with several STMicroelectronics design groups for the definition of specifications and the improvement of some sections of the software.

During the one-year internship program, I was able to play a number of important and challenging work experiences. The most satisfying among all of them was the presentation of an article at *SNUG France 2013* held in Grenoble [1]. The intent of that article was precisely the flow/tool developed in this Thesis.

After positioning the scope of the work within the **VLSI design**, will be described the various kinds of simulation.

Particular attention will be paid, in the third chapter, to the **Static Timing Analysis** with the Synopsys's tool named PrimeTime.

Follows a detailed description of the designed flow, called APV (**Accurate Path Verification**), that will try to highlight the problems faced and the choices made to solve them.

It will be presented the graphical interface written in Tk, the possible analyses, the UNIX simulation environment created, the modeling of crosstalk effect in the main verification flow. Also all possible secondary flows will be detailed. Everything will be provided with the most significant portions of the code created.

The appendices provide a quick reference to allow you to understand the various types of code shown throughout the text.

Introduzione

Data la continua richiesta di dispositivi elettronici con prestazioni sempre maggiori, si è arrivati oggi alla necessità di produrre circuiti integrati con centinaia di milioni di transistor. Sempre più spesso, sono circuiti *System-on-Chip* (SoC) ovvero racchiudono sia parti analogiche che digitali e anche a radiofrequenza. Ciò porta con sé un notevole incremento della complessità sia nella fase di progetto che in quella successiva di verifica. La riduzione del *time-to-market* e dei costi non fanno altro che aumentare le difficoltà. Può essere vincente, dunque, rendere più robusto e veloce il flusso di verifica in modo da ridurre il numero delle iterazioni di progetto (la cosiddetta fase di ECO) ed evitare fallimenti sul silicio.

Il problema principale risiede nel fatto che è impossibile simulare dinamicamente circuiti così complessi per cui si ricorre ad analisi di tipo statico che, seppur molto più veloci, sono intrinsecamente pessimistiche. Si rischia dunque di essere troppo conservativi, limitando le prestazioni ottenibili, o peggio, di avere dei fallimenti inaspettati.

Lo scopo della tesi è stato quello di sviluppare un nuovo flusso che non sostituisce la normale *Static Timing Analysis,* ma la affianca rendendola più sicura. Esso si prefigge di fornire in modo semplice i dati raccolti nel mondo analogico ai designer digitali che necessitano di prendere decisioni su se e come modificare i path critici del loro circuito prima di mandarlo in Produzione/Testing. Un'interfaccia grafica "digital-friendly" fa da guida all'interno dell'ambiente di simulazione appositamente creato per lo scopo.

La scelta di fare di questo nuovo flusso un metodo di verifica integrativo, e non alternativo, è dettata dal fatto che esso si prefigge di rompere il trade-off velocità/accuratezza che

esiste fra le analisi statiche e quelle dinamiche. Per ottenere ciò è necessario spaziare in entrambi gli ambienti di simulazione, quello statico e quello dinamico. Si sfrutta al massimo la **velocità** della *Static Timing Analysis* per costruire una netlist SPICE ridotta su cui fare accurate misure temporali (e volendo di corrente/tensione/power) con simulazioni analogiche **accurate**. L'accuratezza sta anche nel fatto che la netlist viene completamente corredata con i parassiti generati da un software di estrazione apposito per tenere conto nel modo più realistico possibile di ritardi dovuti alle linee e soprattutto a effetti di crosstalk fra linee vicine.

Un'analisi di questo tipo permette anche di esplorare situazioni difficilmente analizzabili in un flusso puramente digitale: misure accurate di *skew* su bus larghi, implementazione di una clock-tree guidata da criteri di dissipazione e/o EMI, valutazione della migrazione verso una tecnologia più scalata, simulazioni al di fuori del dominio di caratterizzazione delle librerie standard.

Il presente lavoro è stato il centro dell'esperienza di stage svolta presso STMicroelectronics di Agrate Brianza (MI) nel gruppo *Smart Power - AMS Design & Verification*.

Inizialmente è stato necessario acquisire le conoscenze relative ai linguaggi di descrizione hardware utilizzati, SPICE, VHDL e Verilog-A; al linguaggio Tcl/Tk in ambiente UNIX per costruire la parte di elaborazione e l'interfaccia grafica; quindi comprendere il funzionamento dei vari tool di simulazione per essere così in grado di sfruttarli al meglio e catturare le informazioni necessarie alla creazione della netlist SPICE da simulare. Durante lo sviluppo del flusso è stato necessario interagire con gruppi di design interni alla STMicroelectronics per la definizione delle specifiche e il miglioramento di alcune sezioni del software.

Durante il periodo di stage, durato un anno, mi è stato possibile svolgere una serie di importanti e stimolanti esperienze lavorative, tra cui spicca la presentazione di un articolo allo *SNUG France 2013* a Grenoble [1] avente come oggetto il flusso/tool sviluppato per questa Tesi.

Dopo aver inquadrato l'ambito di lavoro all'interno della **progettazione VLSI**, verranno descritti i vari tipi di simulazione.

Particolare attenzione sarà dedicata, nel terzo capitolo, alla ***Static Timing Analysis*** con il software PrimeTime.

Seguirà una descrizione dettagliata del flusso progettato, chiamato APV (***Accurate Path Verification***), che cercherà di mettere in evidenza le problematiche affrontate e le scelte fatte per risolverle. Verrà presentata l'interfaccia grafica, le analisi possibili, l'ambiente di simulazione UNIX creato, la modellizzazione del crosstalk nel flusso APV principale e i possibili flussi secondari. Il tutto sarà corredato dalle più significative porzioni di codice creato.

Infine due ***Case Study*** permetteranno di spiegare il funzionamento pratico con l'ottica di un progettista digitale. Il primo è il caso molto frequente di analisi di path con violazioni di setup, mentre il secondo descriverà l'analisi di una clock-tree.

Le appendici forniscono un veloce riferimento per permettere di capire i vari tipi di codice riportati nel corso della trattazione.

1 Progettazione VLSI

Una descrizione della progettazione VLSI è opportuna per poter comprendere il ruolo dei sistemi CAD (*Computer Aided Design*), la necessità di un loro continuo perfezionamento e le difficoltà riscontrate nell'ottenerlo. Il capitolo parte con l'evoluzione dei sistemi elettronici e le conseguenze che questa comporta sulle scelte di progettazione e realizzazione di un circuito; prosegue poi con una descrizione più tecnica del flusso di progettazione.

Con le nuove sfide imposte dal mercato, si è giunti alla progettazione e realizzazione di tutte le principali funzioni di un sistema completo su un unico circuito integrato (IC) realizzato in tecnologia sub-micrometrica (*Deep Sub-Micron* o DSM), definito appunto *System-on-Chip* (SoC).

Il continuo sviluppo dei sistemi microelettronici a semiconduttore, che segue la ben nota legge di Moore, con un aumento esponenziale del livello di integrazione pari ad un raddoppio ogni 18/24 mesi (Figura 1), è dovuto essenzialmente a due fattori distinti. Il primo è, indubbiamente, l'enorme e costante progresso della tecnologia di fabbricazione di questi dispositivi. Il secondo fattore, non meno importante, è rappresentato dall'evoluzione degli strumenti di ausilio alla progettazione.

Figura 1. Legge di Moore: andamento del numero di transistori per singolo microprocessore

In particolare gli strumenti CAD (*Computer Aided Design*) per circuiti elettronici digitali hanno avuto negli ultimi anni un ruolo cruciale nella progettazione e nello sviluppo della tecnologia VLSI (*Very Large Scale Integration*) nei più svariati campi di applicazione. Uno dei settori più interessanti è la sintesi e l'ottimizzazione automatica, ovvero la generazione di specifiche dettagliate a partire da modelli architetturali o logici e l'ottimizzazione di figure di merito quali le prestazioni e l'area. Ciò permette al progettista di concentrarsi sugli aspetti legati alla definizione delle specifiche lasciando ai programmi di CAD il compito di elaborare ed ottimizzare la relativa implementazione tecnologica. Questo è, con il passare del tempo, un fattore sempre più importante con il crescere del numero dei componenti da gestire. Se un tempo era semplice poter progettare a basso livello un sistema, ormai ciò è impossibile in quanto porterebbe via tanto di quel tempo che, una volta pronto, il sistema sarebbe ormai obsoleto. Il discorso della complessità inoltre non può prescindere dalla possibilità di commettere errori. La maggior parte della ricerca sulla sintesi e l'ottimizzazione è stata compiuta nel campo dei circuiti digitali

sincroni piuttosto che su quelli digitali asincroni ed analogici. La formulazione delle problematiche di sintesi ed ottimizzazione può essere convenientemente basata sulla teoria dei grafi e sull'algebra booleana e molti problemi possono essere ricondotti ad alcuni problemi fondamentali. Molti di questi sono computazionalmente intrattabili e devono essere risolti attraverso procedure euristiche. Una loro formulazione esatta aiuta tuttavia nella comprensione delle relazioni tra i problemi che possono incontrarsi nella sintesi e nell'ottimizzazione. In ogni caso, i recenti sviluppi sia degli algoritmi, sia della potenza di calcolo dei computer, hanno permesso di poter trattare un sempre maggior numero di problemi sia con soluzioni approssimate che esatte.

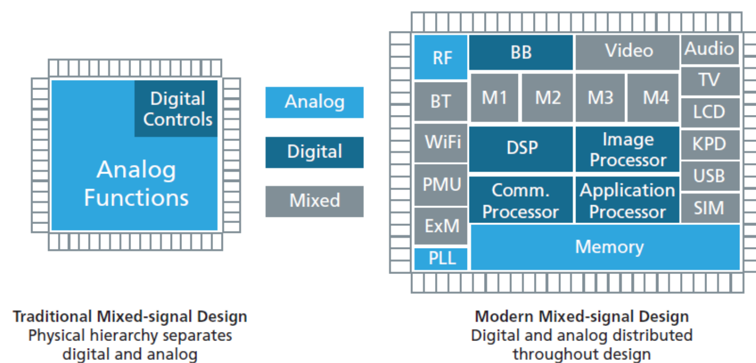


Figura 2. Evoluzione dei System on Chip

In una tecnologia di tipo DSM, è necessario tenere in considerazione gli effetti dovuti alle proprietà fisiche dei materiali all'interno del flusso di progettazione. Il DSM ha portato ad una progettazione di dimensione più contenuta nella geometria dei dispositivi, un aumento dei livelli di metal per le interconnessioni, una diminuzione delle tensioni di alimentazione, decine di milioni di dispositivi su di un singolo IC, tensioni di soglia più basse e maggiori frequenze di clock.

Questi fattori portano alla nascita di due problemi chiamati ***Signal Integrity (SI)*** e ***Design Integrity (DI)***. I problemi di SI includono crosstalk, IR drop, voltage e ground bounce. Al contrario DI include problemi di portatori caldi, surriscaldamento dei collegamenti e problemi legati alla migrazione delle cariche. A dimensioni maggiori problemi di SI e DI diventano di secondo ordine così da potere essere trascurati. Quando le dimensioni dei circuiti diminuiscono, invece, i problemi sopra elencati influiscono fino al punto da dover

procedere ad un processo di verifica dedicato. Nel caso vengano riscontrati errori dovuti a queste cause, è necessario operare un flusso di correzione dell'errore. Nelle tecnologie DSM, il numero di errori dovuti a problemi fisici, è talmente alto che diviene necessario che lo stesso processo di verifica debba sia trovare che fissare i possibili errori. Senza questa capacità "trova e risolvi" all'interno del processo di progettazione, di fronte a decine di migliaia di errori riscontrati, diviene impossibile risolvere questi errori attraverso un solo processo di post-elaborazione.

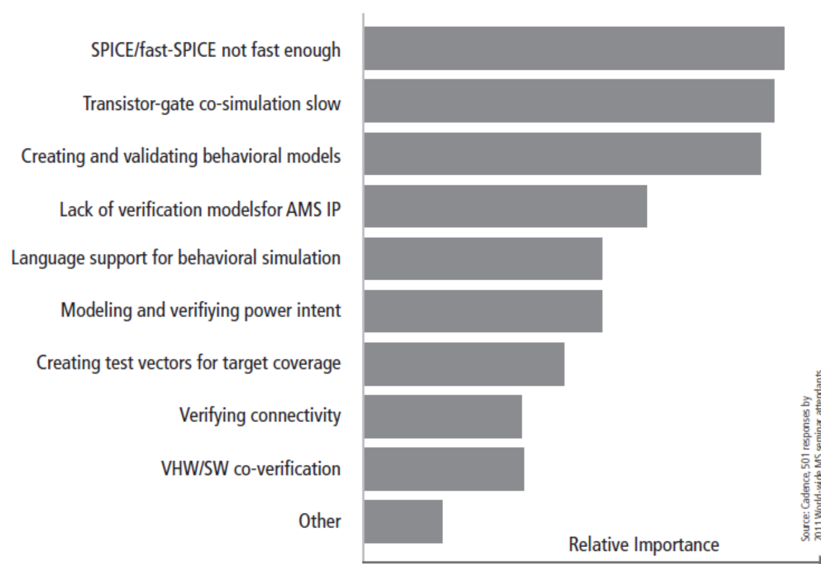


Figura 3. VLSI verification challenges

E' opportuno precisare, dunque, che il fattore limitante nella progettazione VLSI non è il numero di componenti massimo integrabile nel singolo chip, bensì il numero di componenti che gli ingegneri riescono a progettare nell'unità di tempo. Lo sviluppo di applicativi CAD che aiutino il progettista risulta essere in ritardo rispetto alla crescita della complessità dei circuiti. La Figura 3 mostra una lista di problemi che limitano questo sviluppo. La crescente differenza tra il miglioramento della produttività richiesto per soddisfare le esigenze del mercato e la produttività disponibile attraverso gli ultimi tool CAD viene denominato ***design productivity gap***. Il problema non può essere affrontato incrementando solamente il numero di persone al lavoro sul progetto. Al crescere del numero di componenti di un team di sviluppo, infatti, cresce anche la complessità nella

coordinazione per mantenere tutti sincronizzati. E’ stato necessario quindi utilizzare nuove soluzioni per rendere il flusso di progettazione più produttivo fra cui la ***progettazione gerarchica***, il ***riutilizzo di IP*** (Intellectual Properties) e un ***approccio di tipo Top-Down*** rispetto al più tradizionale ***stile Bottom-Up***.

In un flusso di ***progettazione gerarchica*** il progetto viene scomposto in diversi livelli. Il livello principale (*top level*) è composto dalla connessione di vari blocchi, mentre i sottostanti prevedono una descrizione più dettagliata di queste parti, sia nella definizione di sotto-blocchi che nell’introduzione di elementi presenti in librerie predefinite. Per migliorare questo concetto, deve essere possibile generare modelli astratti dei blocchi di ogni livello, in modo da poter essere utilizzati dai livelli superiori.

Il ***riutilizzo di IP*** nel design aiuta lo sviluppo di progetti DSM in due direzioni. Primo, dato che uno o più blocchi all’interno del progetto sono pre-progettati, la quantità di parti da progettare da zero viene notevolmente ridotta. Secondo, dato che le parti pre-progettate hanno quasi sicuramente subito un processo di verifica e validazione, possono essere istanziate all'interno del progetto come “scatole chiuse” (*black boxes*) che non hanno bisogno di un'ulteriore verifica.

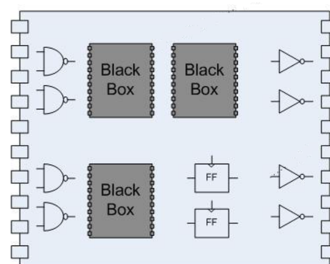


Figura 4. IPR (Intellectual Properties Reuse)

E’ possibile sfruttare questo ultimo punto grazie alla simulazione AMS: i *black boxes* vengono definiti a livello behavioral con linguaggi astratti tipo VHDL, Verilog e Verilog-AMS e pesano poco in termini di capacità computazionale dei tool di CAD.

Nello ***stile di progettazione tradizionale Bottom-Up*** si inizia dal progetto dei singoli blocchi a partire da un set di specifiche fino all'implementazione a livello di transistor. Ogni blocco viene sottoposto quindi a verifica come un'entità a sé stante e non come parte di un sistema globale. Infine i blocchi sono combinati insieme e l'intero sistema così ottenuto viene a sua volta verificato. Si noti che l'intero sistema è rappresentato a questo punto a livello di transistor. Questo stile di progetto può essere ancora di qualche utilità pratica solo per circuiti di piccole dimensioni mentre per circuiti di grandi dimensioni sarebbe fonte di numerose e importanti problematiche. Una volta che i blocchi sono combinati, la simulazione e la verifica dell'intero sistema diventa difficoltosa e costosa in termini di tempo. Una verifica inadeguata può determinare un notevole ritardo dovuto alla necessità di creare un numero superiore di prototipi in silicio. Per i progetti più complessi, un'analisi delle performance, dei costi e delle funzionalità può essere eseguita solo ad un livello architetturale e questo stile di progetto non prevede una visione globale di sistema. Qualsiasi errore o problema trovato nella fase di assemblaggio dei vari blocchi è difficile e temporalmente costoso da risolvere, in quanto implica la riprogettazione dei blocchi. Per assicurarsi poi che i blocchi funzionino insieme in modo corretto è necessaria una intensa comunicazione tra i progettisti di tali blocchi. Data, infatti, la limitata possibilità di verifiche a livello di sistema, qualsiasi errore di comunicazione si riflette nella necessità di realizzare più prototipi e quindi in ritardi.

Per le problematiche fino ad ora descritte risulta più adatto un ***progetto in stile Top-Down***. Quest'ultimo prevede che l'architettura complessiva del chip venga definita attraverso diagrammi a blocchi. Segue poi una fase di simulazione e ottimizzazione usando simulatori di sistema. Dai risultati della simulazione ad alto livello di astrazione vengono ricavati i requisiti che ciascun blocco deve soddisfare. Alla fine, viene realizzato il layout dell'intero chip e verificato in base ai requisiti originari. Più precisamente occorre realizzare uno schematico top level, ossia la definizione della partizione del progetto e dell'interfaccia per ogni blocco, ancor prima che sia stato progettato alcunché. A questo livello ogni blocco e ogni pin su ciascun blocco devono essere attentamente definiti e documentati, (ad esempio una linea di ingresso deve avere associato il nome e il tipo di ingresso) cosicché lo schematico di top level dia chiare informazioni sulle varie funzionalità coinvolte. Una

volta definito lo schematico di *top level*, devono essere scritti i modelli relativi al *top level* e il sistema viene verificato nella sua globalità secondo le specifiche desiderate. Modelli e schematico sono la base di partenza per il progetto vero e proprio dei singoli blocchi. I passi precedenti consentono di avere più progettisti che lavorino contemporaneamente e permettono una maggiore formalizzazione della loro comunicazione, che, di conseguenza, risulta più facilmente immune da errori. Ogni modifica sullo schematico di *top level* e il conseguente aggiornamento dei modelli può essere eseguita in ogni fase del progetto in base alle esigenze che possono sorgere dalla realizzazione circuitale. È necessaria in questo tipo di approccio una procedura di pianificazione della verifica formale dei risultati ottenuti: questo permette di trovare eventuali errori in fasi non ancora avanzate del progetto, quando cioè è più semplice e meno costoso identificare la causa dell'errore e trovarne una soluzione.

La Figura 5 mostra tutti i passi del Flusso di Progettazione Top-Down per Circuiti Integrati. Fino al ***Circuit Design***, momento in cui il progetto è stato sintetizzato fino al gate level, si parla di fase di ***Front-End***. Il ***Physical Design*** è un sinonimo della cosiddetta fase di ***Back-End***. In questa parte i device e le interconnessioni vengono tradotti in rappresentazioni geometriche a livello di silicio e di layer di metallizzazioni, si arriva cioè alla rappresentazione di layout. Il processo non è immediato, necessita anzi di numerosi passi di progettazione e validazione. Segue, infine, la fase di ***Sign-Off*** che prevede una serie di verifiche conclusive alla fine delle quali si iterano alcuni passi precedenti per risolvere eventuali problemi oppure si passa al ***Tape-Out*** del design per i test su silicio.

E' utile approfondire il passo di ***Physical Design*** in modo da comprendere l'uso della *Static Timing Analysis* (descritta nel capitolo seguente) nei vari punti in cui essa interviene. Il punto critico per cui il flusso APV è stato appositamente implementato è quello detto di ***Timing Closure*** ossia il passo in cui si verifica il rispetto dei vincoli temporali di tutti gli elementi digitali del design. In questa fase si dispone già dei parassiti estratti da layout.

Le varie fasi del ***Physical Design*** possono seguire flussi interni dedicati anche molto diversi al variare dei tool usati per la fase di ***Front-End*** e del processo produttivo usato.

Che sia un processo litografico a 65nm o a 28nm, con approccio n-Well o SOI si distinguono comunque alcune fasi principali: *Floorplanning, Placement, Clock-tree Synthesis (CTS), Routing*, *Design Closure* (detta anche *Signoff*, include la *Timing Closure*).

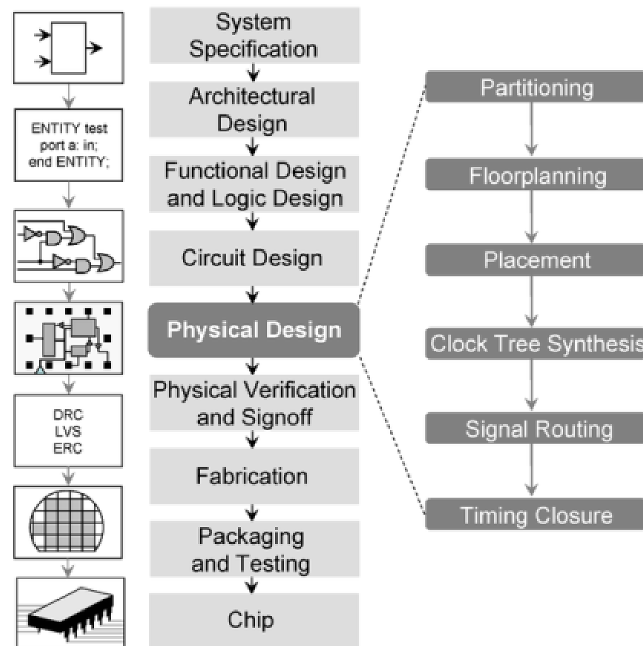


Figura 5. Passi del Flusso di Progettazione per Circuiti Integrati

Floorplanning

La fase di ***Floorplanning*** è quella in cui attraverso il supporto di specifici programmi CAD di piazzamento e routing si raccolgono le informazioni fondamentali del chip al fine di identificare le strutture da piazzare una vicina all'altra. La più importante tra le risorse caricate e visualizzate è proprio la netlist, generata nella fase precedente, che rappresenta una descrizione puntuale dell'intero dispositivo: in essa viene descritta (usando la sintassi del linguaggio Verilog o VHDL) ogni macro da utilizzare, la definizione di tutti i pin presenti (sia dei blocchi che delle celle standard usate) gli assegnamenti tra gli ingressi e le uscite di ogni elemento stanziato e altro. Insieme alla netlist sono caricate anche tutte le viste degli elementi da inserire (sia standard cell che eventuali blocchi) e i file di ***timing constraints***, che descrivono cioè tutte le limitazioni temporali che ogni componente presente deve rispettare. All'occorrenza, vengono caricati altri file utili come, ad esempio, i

LEF (Library Exchange Format) e i DEF (Design Exchange Format). Il primo contiene alcune informazioni di libreria come il numero e la dichiarazione degli strati di metallizzazione da usare per il routing, il numero e il tipo di via. Il file DEF fornisce, invece, le informazioni relative alle connessioni interne tra ogni elemento, i vincoli di piazzamento dei blocchi e i loro orientamenti.

Nel ***Floorplanning*** il progettista può avere una chiara visione di tutti gli elementi che saranno successivamente implementati nel design. In questa fase si configurano alcuni parametri fondamentali del design: viene definita, ad esempio, la forma del core e la sua dimensione (che dipende non solo dalle specifiche di progettazione ma anche dalla complessità del design). In questa fase vengono anche disegnate le net di alimentazione.

Placement

Ogni singola cella viene inserita nel design in una posizione precisa e definita, occupando tutta l'area del core a disposizione e distribuendosi intorno ai macro blocchi stanziati precedentemente. Per un piazzamento corretto ed ottimizzato ogni componente sarà posizionato in modo tale che soddisfi i requisiti di ritardo temporale caricati nel passo precedente.

Il ***Placement*** è una fase spesso lunga ed articolata del flusso poiché composta da operazioni di elaborazione che si succedono più volte in maniera iterativa. E' quindi completamente supportata da programmi specifici. Ad ogni passo di iterazione, la posizione delle celle viene valutata ed ottimizzata per soddisfare diversi requisiti, come ad esempio quello di ottenere delle connessioni con un carico capacitivo e resistivo il più piccolo possibile (il che significa avere un ritardo temporale ridotto) o rispettare criteri di minima congestione. Sono diversi, infatti, gli algoritmi utilizzati per ottenere un piazzamento valido e consistente.

Clock-tree synthesis

L'obiettivo della **sintesi dell'albero di clock** è quello di minimizzare lo skew fra i vari *endpoint* e sottostare ai vincoli di timing precedentemente verificati supponendo un clock ideale. Generalmente entrano in gioco anche restrizioni sulla potenza dissipata per cui risulta necessario ad esempio rilassare gli *slew* e quindi perdere la minimizzazione dello

skew oppure sulle emissioni elettromagnetiche per cui si possono ancora rilassare gli *slew* o schermare alcune net di clock con aumento dell'area dedicata alle connessioni. Anche l'inserimento di buffer e il riposizionamento di blocchi è possibile ai fini di bilanciare l'albero e rispettare i *constraints*. Ne consegue che è sempre necessario ripetere la fase di ***Placement*** al fine di riottimizzare il piazzamento.

Routing

Nella fase di ***Routing*** vengono utilizzate le risorse di routing, ossia vengono istanziate le connessioni nei vari layer di metallizzazione messi a disposizione dalla tecnologia utilizzata.

Physical Verification e Sign-Off

Tocca alla fase di ***Physical Verification*** verificare che il design a livello di layout generato fino ad ora sia corretto. Le verifiche sono di diversi tipi, ad esempio, il layout deve sottostare ai vincoli tecnologici (***Design Rule Checking*** o DRC), deve essere consistente con la netlist originale (***Layout vs Schematic*** o LVS) e non deve avere problemi elettrici (***Electrical Rule Checking*** o ERC).

Al fine di chiudere il design, manca un'importante verifica, quella dei *timing*. In realtà è stata compiuta in tutte le fasi precedenti, ma ora è possibile estrarre i parassiti da layout per un'analisi più precisa.

Come più volte evidenziato, un'analisi dinamica dei *timing* sull'intero design è al giorno d'oggi impossibile vista la complessità dei circuiti. Si esegue perciò un'analisi statica, più veloce ma pessimistica e meno precisa. Il flusso sviluppato in questa Tesi ha come principale scopo quello di rompere questo *trade-off*. Prima però verrà descritta la *Static Timing Analysis* di PrimeTime (Capitolo 3), tool di Synopsys maggiormente utilizzato per lo scopo. Nel Capitolo 2 verranno descritte le caratteristiche dei tipi di simulazione possibili e i tool utilizzati per lo sviluppo del flusso di verifica oggetto di questa Tesi.

2 Flussi di Verifica e Sistemi CAD

Nella prima parte di questo capitolo varrà descritto il panorama generale dei flussi di verifica in modo da poter approfondire, nel capitolo successivo, la *Static Timing Analysis*.
La seconda parte, invece, presenterà i vari tool utilizzati per lo sviluppo e l'esecuzione del flusso di verifica sviluppato in questa Tesi.

2.1 Tipi di simulazione

La simulazione per i circuiti integrati IC (*Integrated Circuits*) può essere classificata in **simulazione circuitale** (**analogica**) **simulazione logica** (**digitale**) e **di tipo misto** (**AMS**).
La **simulazione circuitale**, che impiega equazioni differenziali o equazioni circuitali, porta il circuito ad essere analizzato come un circuito analogico e lo analizza per cercare e valutare nel dettaglio tutti i parametri ed i comportamenti di un circuito elettronico, quali tensione tra i nodi, correnti nei rami ecc… Il linguaggio utilizzato per la descrizione dei circuiti a transistor level è lo SPICE (si veda l'Appendice A: SPICE). Le analisi che possono essere effettuate sul circuito in esame sono di vario tipo. E' possibile valutare il comportamento di una netlist nel dominio del tempo, cioè analizzare come variano tensioni e correnti durante un determinato intervallo di tempo, oppure nel dominio delle frequenze o ancora determinare il punto di lavoro del dispositivo.
La **simulazione logica**, nota anche come simulazione ad eventi, porta ad analizzare il circuito come se fosse di tipo digitale e ne valuta il comportamento e le operazioni logiche.
La simulazione digitale impiega una metodologia ad eventi (*event driven*), cioè analizza il

circuito ad intervalli discreti, regolari o irregolari, ma comunque legati a transizioni di stato del sistema. Per quanto riguarda i linguaggi utilizzati per la descrizione di un circuito a livello logico si hanno a disposizione il Verilog e il VHDL, scelte entrambe valide e che offrono un'ampia gamma di funzionalità. A prescindere dalla scelta fatta sul tipo di linguaggio, è possibile implementare diversi livelli di astrazione per un design a seconda di quello che si vuole verificare durante la fase di simulazione. I livelli d'astrazione per un design digitale sono:

- **Behavioral**: descrive il comportamento del progetto con qualche o addirittura nessun dettaglio sulla sua implementazione strutturale. Questo livello è utilizzato per simulare e testare i concetti di base del sistema e creare le specifiche per la sua implementazione strutturale.
- **Register Transfer Level (RTL)**: descrive le funzioni del design in termini di registri, circuiti combinatori, bus e circuiti di controllo senza alcun dettaglio sull'implementazione a livello di transistor. La simulazione a questo livello viene fatta per verificare la logica e il timing del design.
- **Gate level**: descrive le funzionalità, la tempistica e la struttura del design in termini di interconnessioni tra porte logiche. I blocchi logici comportamentali implementano funzioni booleane come NAND, NOR, NOT, AND, OR and XOR. La gate level viene utilizzata per verificare singolarmente la tempistica di ogni segnale.

La **simulazione mista** (mixed), che è una combinazione tra quella circuitale e quella logica, è stata sviluppata per la simulazione di circuiti LSI (Large Scale Integrated Circuits). Ne esistono due tipi a seconda dei linguaggi utilizzati nella descrizione del circuito e di conseguenza dei simulatori richiesti per poterli simulare. Nella prima, detta ***mixed-level***, la parte analogica viene astratta a livello comportamentale (non vi è traccia dell'implementazione circuitale utilizzata) utilizzando i linguaggi VHDL-AMS e Verilog-A/AMS. Nella seconda, detta ***mixed-signal***, vengono fatti coesistere nello stesso ambiente di simulazione due mondi completamente differenti, cioè quello analogico descritto in SPICE e quello digitale Verilog e/o VHDL. A causa della diversa natura dei linguaggi, le due parti devono essere simulate da due differenti simulatori; una porzione del circuito è soggetta alla simulazione di tipo circuitale, cioè viene trattata come parte analogica mentre

la rimanente parte è soggetta alla simulazione di tipo logico, è perciò valutata come circuito puramente digitale e quindi descritta a livello comportamentale. La sincronizzazione tra le differenti aree, avviene attraverso entrambi i simulatori in modo da poter trasferire i segnali dalla parte analogica a quella digitale e viceversa. Anche i grandi sistemi digitali includono un sempre maggiore numero di circuiti analogici come interfacce, temporizzatori, convertitori e altre funzioni. Sistemi completi di verifica SoC spesso richiedono per la verifica stessa che entrambe le parti, Digitale ed Analogica, lavorino insieme con un simulatore per la simulazione mista analogica/digitale.

Durante la trattazione si parlerà di simulazione statica in ambiente PrimeTime e di simulazione analogica nell'ambiente di lavoro creato.
E' opportuno, però, fare alcune precisazioni.
La simulazione statica a cui si fa riferimento è un particolare tipo di analisi statica che si occupa solo della verifica dei vincoli temporali del circuito, da cui il nome di ***Static Timing Analysis***. Quando i circuiti non erano complessi e grandi come quelli odierni era possibile un approccio dinamico (*Dynamic Timing Analysis*). Quest'ultimo approccio sarebbe più preciso ma indubbiamente molto più lento. Il flusso APV sviluppato si prefigge di fruttare la velocità e l'accuratezza dei due metodi contemporaneamente.
Le simulazioni analogiche eseguite all'interno del flusso APV, in realtà, non sono analogiche pure ma *mixed-level*, visto che si usano moduli in Verilog-A creati appositamente per sensitizzare la netlist SPICE (cioè fornire la tensione sui pin di dato dei Flip-Flop) e per catturare le misure, ma soprattutto per il fatto che APV è stato reso compatibile con la più generale situazione di circuito AMS in modo da essere versatile (si veda la sezione 4.7.3 - Simulazioni AMS).

2.2 Principali prodotti CAD utilizzati

2.2.1 Estrazione dei parassiti: Star-RCXT™

L'estrattore di parassiti di riferimento per l'intera industria della *Electronic Design Automation* (EDA) è Star-RCXT™ di Synopsys. Il singolo tool è valido per tutti i tipi di

flussi di design: *ASIC, System-on-Chip, Full-Custom e Semi-Custom Digital, memorie e circuiti analogici*. Oltre a una veloce estrazione accurata al sub-femtofarad, fornisce capacità avanzate per tecnologie scalate (da 65nm in poi) come l'*estrazione variation-aware* o *litho-aware*. E' in grado anche di estrarre induttanze e può essere usato per risolvere i problemi di modellizzazione dei processi in tecnologie fino alla 45nm.

Questo è un passo molto importante, soprattutto nella fase di *Sign-Off* del *Flusso di design* ***Physical Synthesis*** preso come riferimento per il flusso di verifica sviluppato in questa Tesi.

In ingresso, si richiedono un file di tipo DEF (*Data Exchange Format*) e uno di tipo LEF (*Layout Exchange Format*) che sono stati ottenuti dai passi precedenti di Back End. Il primo contiene alcune informazioni di libreria come il numero e la dichiarazione degli strati di metallizzazione da usare per il *routing* e il numero e il tipo di via. Il file DEF fornisce, invece, le informazioni relative al design specifico del circuito da realizzare che verranno poi utilizzate nelle fasi successive. In esso vengono descritte le connessioni interne tra ogni elemento, i vincoli di piazzamento dei blocchi e i loro orientamenti.
Una volta caricate queste informazioni il programma d'estrazione crea un proprio database, nel quale sono inseriti tutti i valori delle capacità e resistenze parassite estratte da ogni connessione del circuito fino ai singoli transistori, tenendo anche conto del tipo di tecnologia scelta per la realizzazione del dispositivo.
Tali dati sono poi resi disponibili attraverso la creazione di un file specifico e di struttura compatibile con gli altri programmi dell'ambiente: si identifica, infatti, con un formato conosciuto come **DSPF** (*Detailed Standard Parasitic Format*). Questo formato è simile ad una netlist SPICE, si veda l'Appendice B: SPEF/DSPF.
L'estrazione fatta risolvendo direttamente le equazioni di Maxwell può essere eseguita solo su design molto piccoli o porzioni di design. Con i moderni circuiti integrati è possibile solo un approccio approssimato che usa il *Pattern Matching*. Questa tecnica prevede la caratterizzazione di strutture standard tramite la risoluzione delle equazioni dei campi elettromagnetici. Le informazioni vengono poi ad essere memorizzate in delle librerie tecnologiche e usate per riconoscere nel circuito da analizzare delle strutture topologiche

note. Opportuni parametri correttivi vengono utilizzati caso per caso a seconda delle dimensioni fisiche del circuito attuale rispetto a quello di caratterizzazione.
La Figura 6 mostra un semplice esempio di traduzione da layout a rete di parassiti.

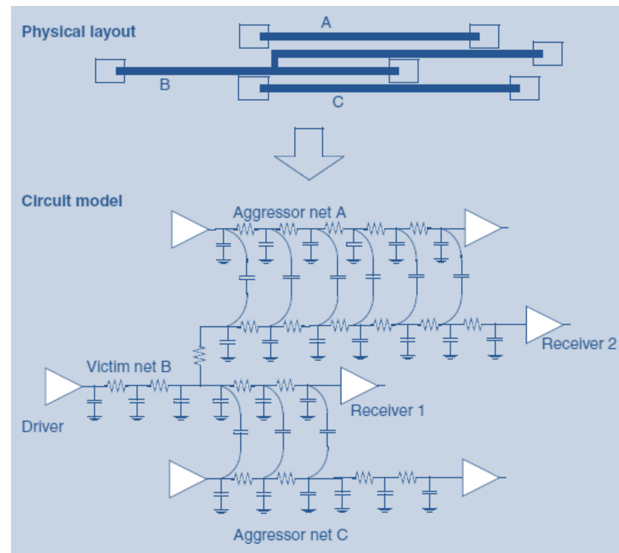


Figura 6. Modello RC distribuito ricavato da layout per tre net

2.2.2 Simulazione statica: PrimeTime

PrimeTime è un tool di Synopsys per l'analisi statica al gate-level capace di verificare un intero chip [2]. La *Static Timing Analysis* è una parte essenziale della progettazione dei moderni e complessi design composti da milioni di porte logiche. Questa verrà descritta in dettaglio nel capitolo successivo.
PrimeTime valuta in modo esaustivo le prestazioni del circuito in termini di tempistiche dei dati che vi viaggiano all'interno controllando tutti i possibili percorsi, chiamati path, che possono attraversare. I dati per essere catturati correttamente dai registri devono essere stabili un certo tempo prima del clock di cattura e anche un certo tempo dopo. L'esaustività è ottenuta senza usare vettori di test.
Il tool di Synopsys è cucito su misura sul flusso di progettazione fisico (Physical Synthesis Flow) ma può essere usato anche come analizzatore statico per altri flussi di design. Accetta ingressi in vari standard: netlist al gate-level in .db, in Verilog e in VHDL;

informazioni di delay in *Standard Delay Format* (SDF); parassiti estratti dai tool di layout nei formati *Standard Parasitic Exchange Format* (SPEF), *Detailed Standard Parasitic Format* (DSPF), *Synopsys Binary Parasitic Format* (SBPF) e *Reduced Standard Parasitic Format* (RSPF); infine vincoli temporali nel formato *Synopsys Design Constraints* (SDC).
Dopo aver letto il circuito descritto al gate-level, PrimeTime lo simula usando le informazioni sulle gate contenute nelle librerie della tecnologia utilizzata.
Se sono presenti delle violazioni temporali sarà necessario provvedere a una o più iterazioni di *Engineering Change Orders* (ECO) ossia sistemarle puntualmente a mano o con tool semi-automatici e rifare la sintesi del design, altrimenti si può passare ala fase di *placement and routing*.
A questo punto le informazioni sul layout, quindi i parassiti, vengono aggiornati e si ripete una o più *Timing Analysis* fino al *sign-off* e al *tape-out* del design.

PrimeTime esegue i seguenti tipi di controllo:

- Vincoli di setup, hold, recovery e removal
- Vincoli definiti dall'utente
- Periodo minimo e larghezza minima degli impulsi di clock
- Design rules (tempi di transizione, capacità e fanout massimi/minimi)

Il primo tipo di controllo è il più importante e riguarda la stabilità dei dati prima e dopo il fronte di cattura del clock in caso di flip-flop e latch.

Fra tutte le *add-on* di PrimeTime, quella riguardante la ***Signal Integrity*** è degna di nota in quanto viene usata dal nuovo flusso APV quando si vuole tenere conto anche del crosstalk.
PrimeTime SI (*Signal Integrity*) calcola i ritardi dovuti all'iniezione di carica da parte delle linee vicine alla linea vittima. Questa iniezione viene modellizzata con delle capacità fra aggressore e vittima e dipende sia dai materiali del processo tecnologico sia dalle caratteristiche geometriche delle linee.

2.2.3 Simulazione dinamica: Eldo

Il simulatore Eldo, così come tutti i simulatori analogici derivati da SPICE, basa il suo funzionamento sulla risoluzione di equazioni matematiche ricavate dal bilancio delle correnti in ogni nodo della rete. Tuttavia Eldo, a differenza della quasi totalità dei simulatori analogici, adatti soprattutto per circuiti discreti, è un simulatore nato appositamente per la simulazione di circuiti integrati. Essendo inoltre un simulatore recente, esso permette di ottenere prestazioni in termini di velocità di simulazioni un po' superiori rispetto ai tradizionali simulatori SPICE [19].

La principale caratteristica di Eldo è l'ampia gamma di modelli di dispositivi che offre, in particolare transistor CMOS di ultima generazione (BSIM3v3, BSIM4, Philips MOS MM9, EKV, HVMOS), transistor bipolari e componenti proprietari, e rende disponibili questi modelli per i processi di diffusione DSM (TSMC, UMC, Chartered, ST, AMS, ecc…). Il simulatore Eldo utilizza una combinazione di due differenti algoritmi di simulazione: l'algoritmo di Newton-Raphson (NR) e la tecnica OSR (One-Step Relaxation). Il primo è un algoritmo per il calcolo degli zeri di una funzione di cui si conosca la derivata e viene utilizzato da quasi tutti i simulatori per la linearizzazione dei componenti non lineari del circuito. La seconda è una tecnica specifica per l'analisi dei nodi con accoppiamento debole (cioè nodi poco influenzati tra loro nei problemi di calcolo dei tempi di ritardo) molto utile nei circuiti con un gran numero di transistor. Combinando i due algoritmi, il NR per i nodi con accoppiamento forte e l'OSR per quelli con accoppiamento debole, Eldo è in grado di migliorare la sua efficienza e la velocità media di simulazione al crescere delle dimensioni del circuito, consentendo l'utilizzo di diversi livelli di accuratezza per differenti parti del circuito.

2.2.4 Simulazione dinamica: CustomSim-XA

CustomSim è un simulatore FastSPICE al transistor-level. Supporta netlist nei formati HSPICE®, Eldo™ e Spectre®. E' in grado di simulare grossi System-on-Chip con SRAM e DRAM integrate, parti mixed-signal come OPAMP, pompe di carica, ADC/DAC, PLL e filtri, sia in fase di pre-layout che in fase di post-layout [20].

Il termine *FastSPICE simulator* indica una serie di algoritmi utilizzati per migliorare le prestazioni della simulazione in termini di tempo di esecuzione. Il tool fa un partizionamento gerarchico del circuito per risolvere più matrici piccole invece di una unica grande matrice che descrive tutti i nodi del circuito, fa un pre-processing dei sistemi lineari e poi risolve tutto usando algoritmi ottimizzati per la risoluzione di matrici sparse. Questi algoritmi rendono inoltre possibile sfruttare al meglio il multi-threading dei processori. L'effetto di questi (e altri) miglioramenti cresce, in termini di tempo di esecuzione medio richiesto per ogni elemento circuitale, al crescere della dimensione della netlist. CustomSim è un ordine di grandezza più veloce di un normale simulatore SPICE per netlist fino a 1'000 elementi e più ordini di grandezza per netlist con più di 1'000'000, ma ovviamente dipende molto anche dalla complessità del circuito.

Altre tecniche permettono di mantenere quasi tutta l'accuratezza di una simulazione normale, ad esempio il pieno supporto al metodo di risoluzione iterativo Newton-Raphson o, lasciando la notazione anglosassone, la *second-order integration*, il l*ocal truncation error control*, la *charge conservation*, così come il *multi-rate simulation* e *l'event-driven timestep control*. In pratica si riesce ad ottenere simultaneamente piena accuratezza sui componenti analogici e un'accuratezza lievemente minore su grosse porzioni digitali simulate in modalità *event-driven*.

L'utilizzo della gerarchia del design permette la creazione di strutture dati intelligenti, capaci di raccogliere le informazioni di istanze diverse della stessa cella, consentendo una notevole riduzione della capacità di memoria richiesta.

Esistono sette livelli di accuratezza, l'utente può sceglierli per scambiare velocità e accuratezza a seconda dei propri bisogni, sarà poi il tool a decidere come trattare le memorie, i componenti analogici, i rail delle alimentazioni e le soglie di tensione.
La compatibilità con i formati HSPICE, Eldo e Spectre permette di utilizzare i comandi più comuni, ad esempio i *probes*, i *measures*, gli *alter*, le espressioni condizionali e i *data sweeps*.

2.2.5 Analisi dei dati: Custom WaveView

Custom WaveView™ è un visualizzatore grafico di forme d'onda e anche uno strumento per il post-processing dei segnali in ambito analogico e mixed-signal [18]. E' in grado di caricare e elaborare la grossa mole di dati proveniente dai moderni circuiti integrati (IC).
La compatibilità con entrambi i mondi, digitale e analogico, consente di passare facilmente da uno all'altro. Ad esempio, caricando i dati di una simulazione analogica questi possono essere convertiti in digitale e salvati per essere dati in pasto ad un blocco digitale da simulare con un simulatore digitale.
Oltre a elaborazioni nel dominio del tempo e della frequenza si ha a disposizione un folto set di misure sia su grafico che sui dati binari. Alcune figure saranno riportate nei capitoli seguenti, come ad esempio la Figura 36. Principali forme d'onda per un path (con e senza crosstalk).
Custom WaveView è un prodotto Synopsys, ovviamente esistono competitors simili, come EZwave di Mentor Graphics e Simvision di Cadence Design Systems.

2.2.6 Analisi dei dati: gnuplot

Gnuplot è un programma per la realizzazione di grafici di funzioni matematiche in due o tre dimensioni e la rappresentazione grafica di dati grezzi. È un programma open source, tuttavia è possibile distribuire modifiche del codice sorgente solamente in forma di patch.
È disponibile per diversi sistemi operativi ed è in grado di esportare grafici nei più comuni formati grafici (tra cui PNG, EPS, SVG e JPEG).
A dispetto del nome, il programma non è correlato al progetto GNU e non utilizza licenze della Free Software Foundation [21].
Gnuplot possiede un'interfaccia a riga di comando. È interessante anche la modalità da shell, che consente di salvare le istruzioni per creare le immagini in semplici script. **Gnuplot** è inoltre implementato per il rendering da vari programmi, come GNU Octave, Emacs e altri.
Per il lavoro di questa Tesi non è fondamentale, serve solo a produrre delle statistiche sui dati alla fine delle simulazioni.

3 Static Timing Analysis

Come anticipato nel primo capitolo, la STA interviene in più punti del flusso di progettazione. La Figura 7 evidenzia come essa si trovi a trattare descrizioni anche molto diverse del circuito, dal *gate-level* con clock e interconnessioni ideali al *layout-level* con informazioni dettagliate sui parassiti. Quest'ultima condizione è la più importante da analizzare in quanto il passo successivo è il *tape-out* su silicio e si vorrebbe un circuito privo di violazioni. Violazioni inaspettate costano in termini di tempo e denaro, è, infatti, necessario sistemarle quindi rifare il passo di *Physical Design* e le verifiche di *Sign-Off*, nonché rifare, almeno parzialmente, le maschere litografiche. Il mancato rispetto del *time-to-market*, poi, potrebbe portare a perdite di settore.

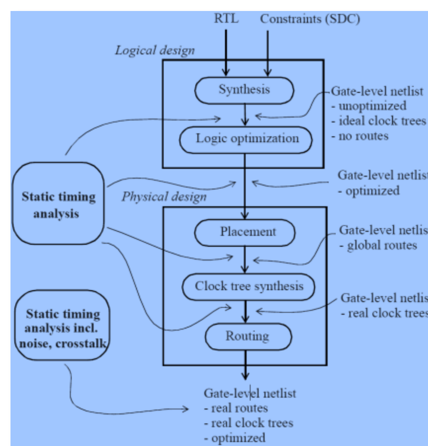


Figura 7. STA nel flusso di progettazione

Un altro modo di fare un'analisi di timing è quello dinamico, in cui si simula il comportamento completo del circuito con un certo numero di set diversi di stimoli in ingresso. Rispetto alla ***Dynamic Timing Analysis*** (**DTA**) la **STA** è molto più veloce perché non è necessario simulare il comportamento dei componenti ma solo il ritardo di propagazione dei dati. E' anche molto più completa in quanto controlla tutte le combinazioni possibili degli ingressi e non solo alcuni casi particolari come la DTA. Si parla dunque di 100% di ***coverage***. Ovviamente la STA controlla solo che i tempi siano rispettati e non la funzionalità del circuito, ossia che i dati in propagazione siano giusti.

3.1 Timing Paths

Fra i tool di STA quello più utilizzato a livello mondiale è PrimeTime di Synopsys per cui, d'ora in poi, con STA si farà implicito riferimento a quest'ultimo [2].
Al fine di eseguire la verifica alla ricerca di violazioni, PrimeTime suddivide il design in ***timing path***, calcola il ritardo di propagazione del dato attraverso ogni path e cerca le violazioni temporali.

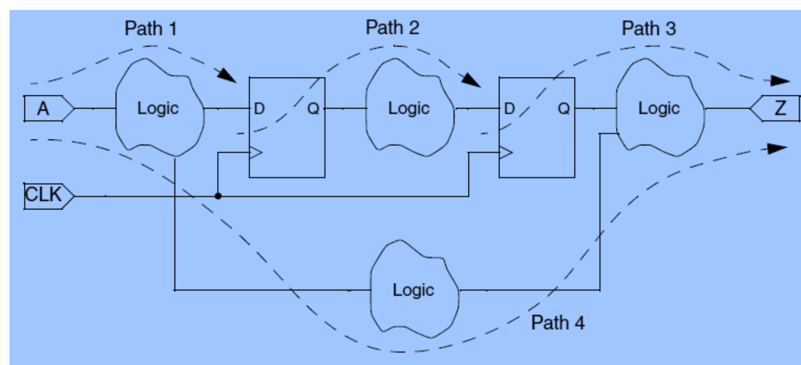


Figura 8. Esempi di data-path

Ciascun ***timing path*** ha uno ***startpoint*** e un ***endpoint***. La Figura 8 mostra una piccola porzione di design in cui sono evidenziati alcuni path. Il punto di partenza o *startpoint* è il punto del design da cui il dato viene lanciato da un fronte attivo di clock. Il dato viene poi fatto propagare attraverso della logica combinatoria fino al punto di arrivo o *endpoint*,

dove verrà catturato da un altro fronte attivo del clock. Possono essere *startpoint* i pin di clock degli elementi sequenziali o i pin d'ingresso del design. In quest'ultimo caso il dato sarà lanciato da una sorgente esterna. Fanno invece da *endpoint* gli ingressi di dato degli elementi sequenziali o i pin d'uscita del design. Per questi ultimi pin sarà della circuiteria esterna a dover catturare il dato.

Con la definizione appena data di *startpoint* e *endpoint* è chiaro come possano esistere quattro tipi di timing path: IN-to-reg, reg-to-reg, reg-to-OUT e IN-to-OUT. In realtà la situazione è un po' più complessa. Questi tipi di path vengono chiamati ***data-path sincroni***, e possono essere *single-cycle* o *multi-cycle*. Come è possibile vedere in Figura 9 esistono anche ***data-path asincroni***, ***path di clock*** e ***di clock-gating***.

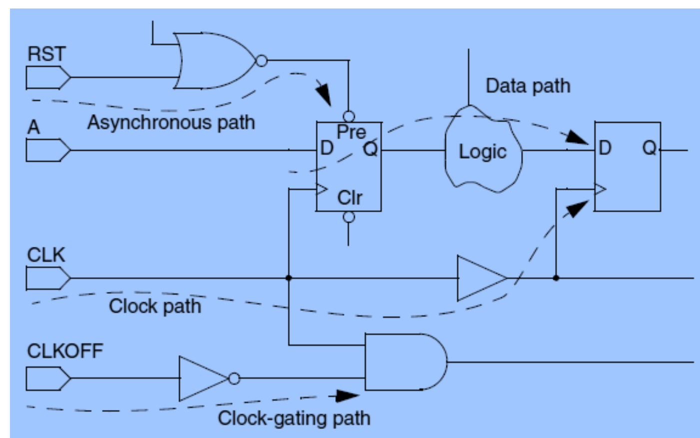


Figura 9. Alcuni path particolari

Al di là delle differenze tra i vari tipi di path, lo scopo dell'analisi di timing si riduce alla verifica che i *timing constraints* del progettista siano rispettati. In particolare un ***clock-gating path*** deve rispettare i timing per poter attivare/disattivare un dominio di clock correttamente, un ***data-path asincrono*** deve poter settare/resettare il registro nel momento opportuno, un ***clock-path*** deve garantire un ritardo al fronte che si propaga compreso fra un valore minimo e uno massimo. Ovviamente il caso di maggiore interesse risulta essere quello per cui il dato che si propaga nel ***data-path*** deve essere stabile per un certo tempo prima e dopo il fronte di cattura. I *timing constraints* in questo caso sono i ben noti tempi

di ***setup*** e di ***hold***. A seguire, il termine generico di *path* indicherà un *data-path*, a meno che non venga specificato diversamente.

Il caso più semplice con cui conviene illustrare le casistiche possibili per la verifica dei *timing constraints* di path critici è quello di data-path single-cycle da Flip-Flop a Flip-Flop. La Figura 10 ne mostra uno schema. In questo caso entrambi gli elementi sequenziali sono *positive-edge triggered*, ossia sono attivi sul fronte di salita del clock.

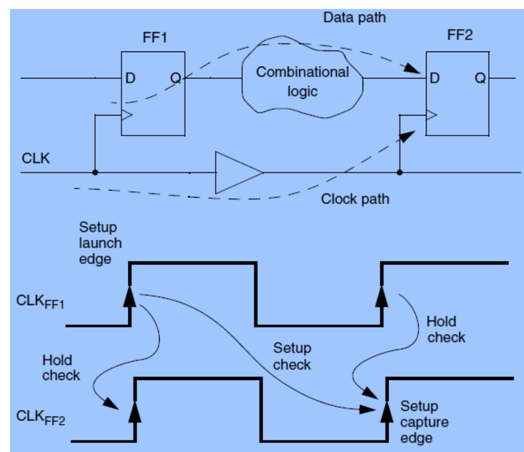


Figura 10. Reg-to-reg data-path con clock attivi sul fronte di salita

Ovviamente è possibile che uno o entrambi i Flip-Flop siano *negative-edge triggered* o siano dei Latch (attivi a livello alto o basso). Un altro fattore contribuisce ad aumentare le combinazioni possibili: clock di lancio e di cattura potrebbero essere diversi, sia in termini di periodo che di *duty-cycle*.

Due definizioni che raccolgono tutte le casistiche sono le seguenti:

- il **check di setup** va fatto fra il clock di lancio e il **successivo fronte di cattura** che si trova alla minore distanza fra tutti i casi possibili.
- il **check di hold** va fatto fra il clock di lancio e il **precedente** (o coincidente) **fronte di cattura** che si trova alla minore distanza fra tutti i casi possibili.

Un esempio è visibile in Figura 11. Il clock di lancio del primo ha periodo T1=6ns, mentre quello di cattura T2=8ns. Per avere tutte le combinazioni possibili dei fronti attivi di lancio e cattura fra cui cercare il caso pessimo da analizzare, è necessario considerare almeno un

tempo pari al minimo comune multiplo fra i due periodi T1 e T2, (24ns in questo caso). Si notino i fronti scelti (quelli cerchiati).

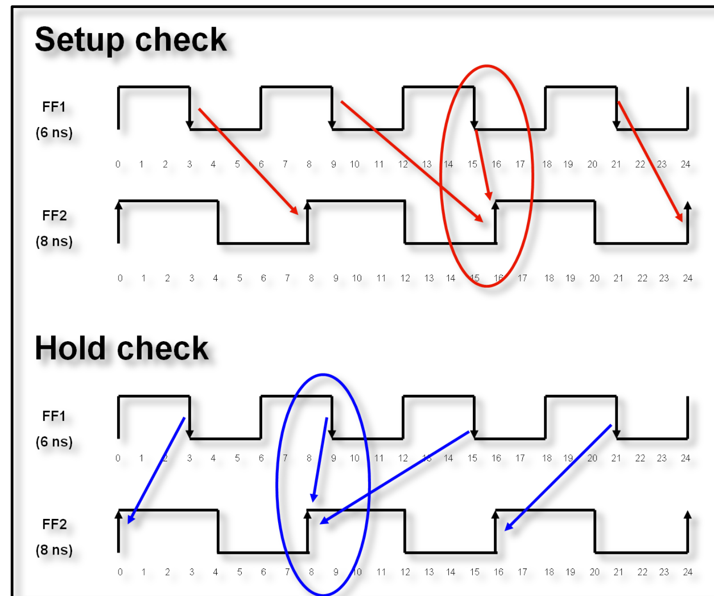


Figura 11. Scelta dei fronti di clock da verificare

Noti i fronti di riferimento la STA prosegue con il calcolo del ***required time*** e del ***data arrival time***.

Facendo riferimento ad un'analisi di setup con soli Flip-Flop, il primo è il tempo in cui arriva il fronte di cattura del clock (comprensivo di ritardi sulla clock-tree e di termini conservativi come ad esempio lo *skew*, detto anche *clock-uncertainty*) epurato del tempo di setup. Il *data arrival time* è semplicemente il tempo in cui il dato arriva all'ingresso del secondo Flip-Flop, ottenuto sommando ritardi e termini conservativi sulla clock-tree e i ritardi lungo il path.

E' immediato definire ora lo ***Slack*** come differenza fra *required time* e *arrival time* e notare come un valore negativo indichi la violazione dei vincoli temporali.

In caso di hold, invece, il dato dovrebbe arrivare almeno un tempo di hold dopo il fronte di cattura per cui, al fine di avere un valore negativo in caso di violazione, si definisce lo slack come differenza fra *arrival time* e *required time*.

Con i Latch al posto dei FF entrano in gioco anche i tempi “dati in prestito” da un elemento sequenziale al successivo. Si sfrutta spesso questa opportunità quando si vogliono rilassare i vincoli temporali di un certo path. La sostanza dello slack non cambia.

Lo strumento più utilizzato dai designer digitali, messo a disposizione da PrimeTime, è il cosiddetto ***Timing Report***. Con questo *report* è possibile analizzare in dettaglio i componenti costanti e quelli di propagazione del clock e del dato che vanno a formare lo slack del path sotto analisi. Le sezioni di codice seguenti mostrano un esempio.
La prima parte riporta le opzioni utilizzate per produrre il report. Segue l’indicazione di startpoint e endpoint; in questo caso, il path inizia con un Flip-Flop il cui clock si chiama INTDIV_OUT e termina su un altro FF comandato dallo stesso clock sfasato di 180°. Entrambi i FF sono attivi sul fronte di salita del clock.

```
****************************************
Report : timing
   -path_type full_clock_expanded
   -delay_type max
   -input_pins
   -nets
   -slack_lesser_than 0.00000
   -capacitance
Design : pll_dig_top
Version: G-2012.06-SP2
Date   : Thu Jun 20 11:15:06 2013
****************************************
  Startpoint: FREQ_CORR_D_regx0x
              (rising edge-triggered flip-flop clocked by INTDIV_OUT)
  Endpoint: DELTA_CEIL_SD_OUT_N_regx1x
              (rising edge-triggered flip-flop clocked by INTDIV_OUT')
  Path Group: INTDIV_OUT
  Path Type: max
  Max Data Paths Derating Factor  : 1.00000
  Min Clock Paths Derating Factor : 0.95000
  Max Clock Paths Derating Factor : 1.05000
```

Il report si riferisce ad un’**analisi di setup**, come indicato dal ***path type = max***. I vari fattori di *derating* servono per avere dati conservativi.

La seconda porzione mostra come il dato viene lanciato e come si propaga lungo il path fino all'endpoint. La prima colonna riporta i pin e le net, le altre riportano il fanout e la capacità dei pin, il ritardo di propagazione sulla singola net o attraverso la singola porta logica, l'indicazione che la net sia o meno corredata di parassiti RC (con l'&), il ritardo accumulato dalla *clock-root* fino al punto corrente e l'indicazione della transizione rise/fall.

```
Point                                                Fanout    Cap        Incr       Path
--------------------------------------------------------------------------------------
clock INTDIV_OUT (rise edge)                                               0.00000    0.00000
clock source latency                                                       0.00000    0.00000
INTDIV_OUT (in)                                                            0.01150 &  0.01150 r
INTDIV_OUT (net)                                          2  0.02008
ICC_CTS_0_HS65_GS_CNIVX62_BC/A (HS65_GS_CNIVX62)                           0.00018 &  0.01168 r
ICC_CTS_0_HS65_GS_CNIVX62_BC/Z (HS65_GS_CNIVX62)                           0.02026 &  0.03194 f
INTDIV_OUT_BC (net)                                       1  0.02761
ICC_CTS_0_HS65_GS_CNIVX62_BC_1/A (HS65_GS_CNIVX62)                         0.00297 &  0.03491 f
ICC_CTS_0_HS65_GS_CNIVX62_BC_1/Z (HS65_GS_CNIVX62)                         0.02257 &  0.05747 r
INTDIV_OUT_BC_1 (net)                                     3  0.03596
ICC_CTS_0_HS65_GS_CNIVX62_G3B1I1_1/A (HS65_GS_CNIVX62)                     0.00058 &  0.05805 r
ICC_CTS_0_HS65_GS_CNIVX62_G3B1I1_1/Z (HS65_GS_CNIVX62)                     0.02917 &  0.08722 f
INTDIV_OUT_BC_1_G3B1I1_1 (net)                            9  0.08407
ICC_CTS_0_HS65_GS_CNIVX24_G3B2I4/A (HS65_GS_CNIVX27)                       0.00136 &  0.08859 f
ICC_CTS_0_HS65_GS_CNIVX24_G3B2I4/Z (HS65_GS_CNIVX27)                       0.05177 &  0.14036 r
INTDIV_OUT_BC_1_G3B2I4_1 (net)                           35  0.06255
FREQ_CORR_D_regx0x/CP (HS65_GSS_DFPRQNX18)                                 0.00050 &  0.14085 r
FREQ_CORR_D_regx0x/QN (HS65_GSS_DFPRQNX18)                                 0.16094 &  0.30179 r
n178 (net)                                                2  0.01560
U2748/A (HS65_GS_NAND2X21)                                                 0.00009 &  0.30188 r
U2748/Z (HS65_GS_NAND2X21)                                                 0.02843 &  0.33030 f
xcellx118976xnet77011 (net)                               1  0.01215
U2693/B (HS65_GS_CNNOR2X24)                                                0.00012 &  0.33042 f
U2693/Z (HS65_GS_CNNOR2X24)                                                0.03441 &  0.36482 r
xcellx118976xnet77012 (net)                               4  0.02632
U2752/B (HS65_GS_NOR2AX13)                                                 0.00015 &  0.36497 r
U2752/Z (HS65_GS_NOR2AX13)                                                 0.02509 &  0.39006 f
n2091 (net)                                               1  0.00683
ICC_POSTCTS_INCR1_31/A (HS65_GS_NAND2X14)                                  0.00009 &  0.39015 f
ICC_POSTCTS_INCR1_31/Z (HS65_GS_NAND2X14)                                  0.02698 &  0.41713 r
net72529 (net)                                            2  0.00967
U2890/B (HS65_GS_XNOR2X18)                                                 0.00006 &  0.41719 r
U2890/Z (HS65_GS_XNOR2X18)                                                 0.05601 &  0.47320 f
n2250 (net)                                               1  0.00576
ICC_POSTCTS_INCR1_26/B (HS65_GS_NOR2AX13)                                  0.00008 &  0.47328 f
ICC_POSTCTS_INCR1_26/Z (HS65_GS_NOR2AX13)                                  0.02157 &  0.49485 r
```

```
N274 (net)                                        1  0.00180
DELTA_CEIL_SD_OUT_N_regx1x/D (HS65_GSS_DFPRQX35)             0.00001 &  0.49486 r
data arrival time                                                       0.49486
```

L’ultima riga riporta dunque il ***data arrival time***.

Stesse colonne per la clock-tree di cattura. L’ultima riga di questo blocco, però, riporta il ***data required time***.
Notare anche i vari componenti conservativi, *clock reconvergence pessimism* e *clock uncertainty*, e il vincolo temporale di setup.

```
clock INTDIV_OUT' (rise edge)                                0.60000    0.60000
clock source latency                                         0.00000    0.60000
INTDIV_OUT (in)                                              0.00530 &  0.60530 f
INTDIV_OUT (net)                                  2  0.01182
ICC_CTS_0_HS65_GS_CNIVX62_BC/A (HS65_GS_CNIVX62)             0.00009 &  0.60539 f
ICC_CTS_0_HS65_GS_CNIVX62_BC/Z (HS65_GS_CNIVX62)             0.01533 &  0.62072 r
INTDIV_OUT_BC (net)                               1  0.02039
ICC_CTS_0_HS65_GS_CNIVX62_BC_1/A (HS65_GS_CNIVX62)           0.00093 &  0.62166 r
ICC_CTS_0_HS65_GS_CNIVX62_BC_1/Z (HS65_GS_CNIVX62)           0.01437 &  0.63603 f
INTDIV_OUT_BC_1 (net)                             3  0.02393
U595/A (HS65_GS_CNIVX21)                                     0.00061 &  0.63664 f
U595/Z (HS65_GS_CNIVX21)                                     0.04388 &  0.68052 r
N429 (net)                                       32  0.06289
DELTA_CEIL_SD_OUT_N_regx1x/CP (HS65_GSS_DFPRQX35)            0.00140 &  0.68192 r
clock reconvergence pessimism                                0.00000    0.68192
clock uncertainty                                           -0.15000    0.53192
library setup time                                          -0.04996    0.48196
data required time                                                      0.48196
```

Infine, come da definizione, viene calcolato lo **slack**. In questo caso è negativo quindi, come indicato, il vincolo temporale di setup per questo path è violato.

```
data required time                                                      0.48196
data arrival time                                                      -0.49486
-------------------------------------------------------------------------------------
slack (VIOLATED)                                                       -0.01290
```

Durante l'analisi in PrimeTime, l'istruzione chiave che permette di investigare a fondo il design è ***get_timing_path*** che consente di creare una collezione di path completa di: informazioni topologiche, quali lo startpoint, l'endpoint, le intere clock-tree di lancio e di cattura; informazioni sulla STA, ad esempio lo slack, il tempo di arrivo del dato sull'endpoint e i vincoli di setup/hold. Il timing report precedentemente descritto è derivato proprio da oggetti di tipo path come quelli ottenuti grazie a *get_timing_path*.
La sintassi del comando è molto semplice e sostanzialmente implementa un filtro. Tutti i parametri sono facoltativi in quanto sono settati a dei valori di default. L'istruzione senza parametri restituisce al massimo un path, il peggiore per quanto riguarda l'analisi di setup, senza informazioni dettagliate sulla clock-tree e solo se il suo slack è negativo.
Conoscendo lo startpoint e/o l'endpoint e/o una cella in mezzo al path è possibile selezionare uno o più path specifici.
Altrimenti è possibile collezionare un generico numero di path (*-max_paths 50*) in un certo range di slack (usando le opzioni *-slack_lesser_than* e *-slack_greater_than*) appartenenti a uno o più gruppi (*-group "my_group"*) i cui endpoint sono fra quelli della lista indicata al parametro *-to*. Il codice sottostante mostra questo generico esempio in caso di analisi di hold (*-delay_type min*).

```
get_timing_paths -path_type full_clock_expanded -delay_type min \
                 -slack_lesser_than 0.1 -max_paths 50\
                 -to "list of custom endpoint pins" -group "my_group"
```

Un'altra istruzione molto utile in PrimeTime è ***get_attribute*** che permette di estrarre le varie informazioni dall'oggetto path, dai suoi oggetti interni, o dagli altri elementi del design, ad esempio pin, net, celle e librerie. Il codice mostra come estrarre lo slack peggiore per un'analisi di setup.

```
set WORST_SETUP_SLACK [get_attribute [get_timing_paths] slack]
```

3.2 STA vs Transistor-level Timing Verification

Al fine di considerare tutti i casi possibili e di non perdere nessuna violazione del circuito, la Static Timing Analysis deve necessariamente essere pessimistica.

Dopo uno sguardo alle variazioni del processo tecnologico, saranno descritte le modalità operative della STA. Il confronto con la DTA evidenzierà la necessità di ridurre il pessimismo, che è il principale motivo per cui si è ingegnerizzato il flusso oggetto di questa Tesi.

3.2.1 Corners tecnologici

Il processo tecnologico di circuiti integrati in VLSI è purtroppo affetto da una certa variabilità nei suoi parametri. Le cause sono principalmente le variazioni di umidità e temperatura durante il processo, e la posizione relativa del *die* rispetto al centro del *wafer* di silicio. La deviazione rispetto al valore nominale del drogaggio, dello spessore dell'ossido e degli altri parametri di processo può provocare significative differenze nei segnali che si propagano nel circuito e a volte questo è causa di fallimenti di interi sistemi.

I corners tecnologici rappresentano i casi estremi di queste variazioni in cui il circuito dovrà essere in grado di funzionare. Un circuito che lavora in un corner diverso da quello tipico sarà più o meno veloce rispetto alle specifiche. Ad esempio in caso di temperatura e/o tensione maggiore il circuito funziona solo rallentando il clock, mentre il caso opposto potrebbe avere delle violazioni di hold che renderebbero il chip inutilizzabile.

I corners sono molteplici e riguardano sia la mobilità dei dispositivi che le variazioni delle interconnessioni. Nel processo CMOS, ad esempio, esistono i corners "pari" FF TT e SS e quelli "dispari" FS, SF ecc… La prima lettera si riferisce agli NMOS e la seconda ai PMOS, mentre F, T e S indicano processi di tipo ***fast***, ***typical*** e ***slow***.

Generalmente si prendono in considerazione le cosiddette variazioni PVT (*Process*, *Voltage* and *Temperature*) in soli tre casi fra quelli possibili: il ***best***, il ***worst*** e il ***typical***. Questo vale sia per le interconnessioni che per i dispositivi in modo indipendente. La Figura 12 riassume l'andamento dei ritardi delle celle nelle diverse condizioni operative e lascia intuire le 27 combinazioni possibili (utilizzando solo le variazioni del processo pari).

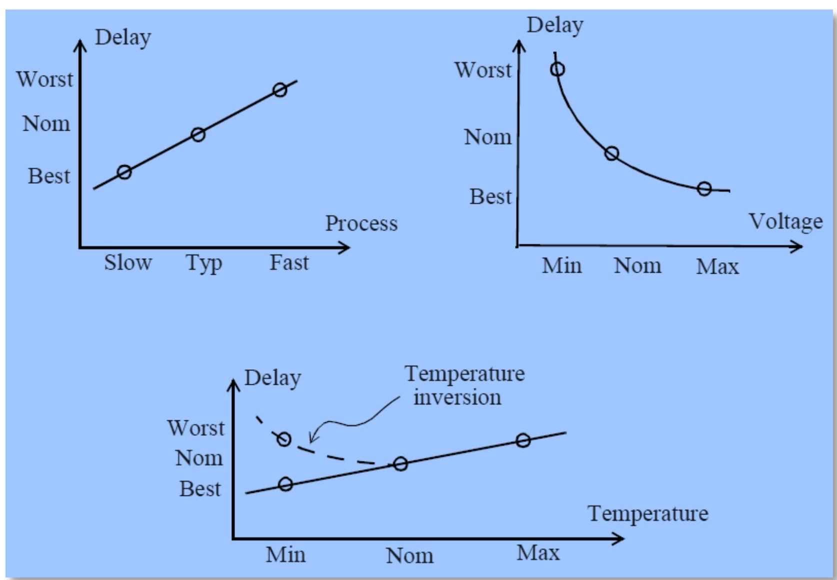


**Figura 12. Andamento del delay delle celle
al variare delle condizioni operative**

Per tenere conto di queste variazioni prima del *tape-out* su silicio, i simulatori dispongono delle cosiddette models, ossia di caratterizzazioni su silicio delle singole celle nei vari corners tecnologici.

3.2.2 Necessità di ridurre il pessimismo

La STA di PrimeTime utilizza la modalità di analisi chiamata OCV (*On-Chip Variation*) che consiste nell'usare fronti e ritardi differenziati nelle diverse sezioni del path, tutti selezionati all'interno di uno stesso corner tecnologico. La seguente tabella riassume tutte le casistiche [4].

	Clock-tree di lancio e path	**Clock-tree di cattura**
SETUP	Fronti lenti e ritardi maggiori	Fronti veloci e ritardi minori
HOLD	Fronti veloci e ritardi minori	Fronti lenti e ritardi maggiori

Un'analisi di tipo solo worst o solo best, ossia che usa lo stesso tipo di fronti e ritardi all'interno dello stesso path, presi all'interno dello stesso corner, rischia di mascherare alcune violazioni. Da qui la necessità di un'analisi OCV, che purtroppo introduce un consistente ammontare di pessimismo. A questo proposito, si osservi la Figura 13 che raffigura un path dal FF2 al FF3 e si supponga di voler verificare il vincolo di setup. La cella U1 ha due fronti di salita diversi, in base al pin d'ingresso che innesca la transizione. L'analisi di tipo OCV seleziona il caso peggiore, quando sarebbe sufficiente selezionare l'altro fronte.

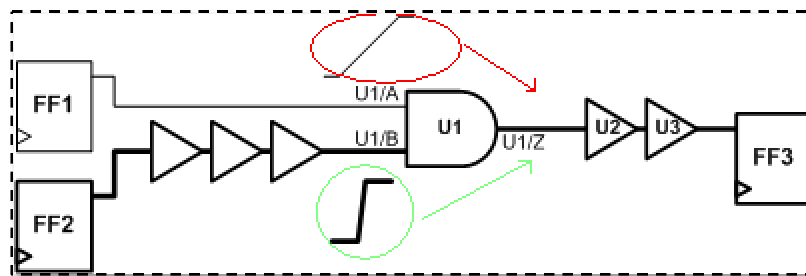


Figura 13. Esempio di propagazione dei fronti con analisi OCV (in rosso) e PBA (in verde)

Un'analisi che riduce questa componente di pessimismo viene detta **PBA** (***Path Based Analysis***) ed è disponibile anche in PrimeTime. Il problema della PBA è che richiede la valutazione dell'effettivo fronte che si propaga per ciascuna cella dell'intero design per cui il tempo richiesto alla *Static Timing Analysis* cresce considerevolmente. Per questo motivo la PBA viene usata solo sui path critici risultanti da una prima iterazione di STA in modalità OCV [2][4][6][9][10][11][12].

Il Flusso APV oggetto di questa Tesi si prefigge di implementare una verifica simile alla PBA in modo da eliminare più pessimismo possibile, fornendo al designer dati accurati e in poco tempo. L'analisi con APV sarebbe dinamica e non statica e, al prezzo di un tempo di simulazione leggermente maggiore rispetto ad una PBA, fornirebbe enormi vantaggi in termini di accuratezza, nonché una netlist completa su cui poter fare anche delle ulteriori analisi.

3.3 Signal Integrity

Il crosstalk dovuto all'accoppiamento capacitivo fra linee adiacenti influenza pesantemente i ritardi di propagazione dei segnali del circuito [7][8], soprattutto con l'utilizzo di tecnologie sempre più scalate (si veda la Figura 14).

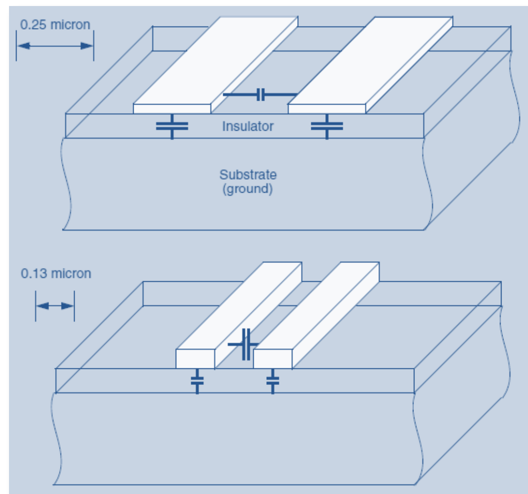


Figura 14. Aumento delle capacità di coupling con lo scaling tecnologico

La Figura 15 mostra un esempio in cui due transizioni (A e C), allineate temporalmente con una transizione vittima (B), sono in grado di anticipare o ritardare quest'ultima. Il fronte di aggressione inietta una corrente nella linea vittima proporzionale alla sua ripidezza e con segno positivo o negativo a seconda che sia in salita o in discesa ($i=C_{coupling}*dV/dt$). Questa corrente impulsiva si traduce in un *bump* di tensione sulla linea vittima, che in proporzione all'allineamento con il fronte vittima riesce a traslarlo temporalmente [14][15]. In realtà ciò avviene a livello infinitesimale lungo tutta la lunghezza di affacciamento delle due linee, ma la rappresentazione nei sistemi CAD non può non essere discretizzata. Il modello utilizzato per le linee è quasi sempre un modello a celle RC in cascata come quello mostrato in Figura 6.

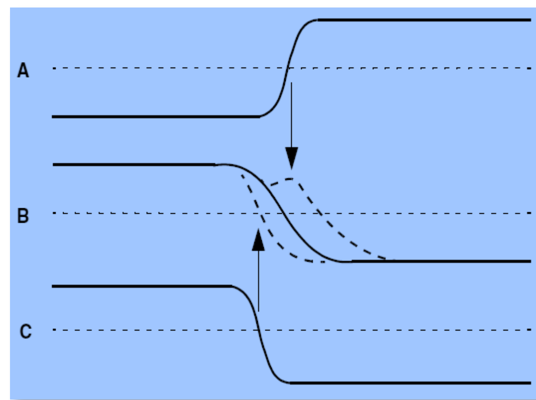


Figura 15. Esempio di bump di tensione indotti in grado di anticipare/ritardare il fronte vittima

Per consentire a PrimeTime SI (e quindi al nuovo flusso di verifica APV) di valutare i fenomeni di crosstalk sono necessarie alcune accortezze [2]:

- fare in modo che il file contenente i parassiti estratti da layout contenga le **capacità di accoppiamento** fra una linea e l'altra e non solo capacità verso massa;
- **caricare questi parassiti nella sessione** di PrimeTime usando l'istruzione *read_parasitics* con l'opzione *-keep_capacitive_coupling*;
- **abilitare l'analisi** di *signal integrity* settando al valore "*true*" la variabile d'ambiente di PrimeTime SI di nome *si_enable_analysis*.

A questo punto PrimeTime SI marchia tutte le net che hanno almeno una capacità di coupling come aggressori e le filtra prendendo solo quelle che sono in grado di indurre sostanziali ritardi.

Il filtraggio è effettuato in modo iterativo. Si esegue una prima STA durante la quale sono valutati tutti i ritardi indotti da tutti gli aggressori trascurando le finestre di allineamento temporale. Dalla seconda iterazione si tiene conto delle finestre temporali e si calcolano i ritardi aggiunti dagli aggressori, eliminando così gli aggressori troppo piccoli, disallineati temporalmente o non correlati logicamente.

I criteri di filtraggio possono essere controllati dall'utente modificando i valori di default di apposite variabili globali di PrimeTime SI, come ad esempio:

- *si_xtalk_exit_on_max_iteration_count* per fare più di due iterazioni di STA;
- *si_filter_per_aggr_xcap* per settare il valore assoluto di capacità sotto cui un aggressore non viene considerato;
- *si_filter_per_aggr_xcap_to_gcap_ratio* è come il precedente, ma il valore è normalizzato alla capacità verso massa della linea vittima;
- *si_filter_per_aggr_noise_peak_ratio* per settare una soglia all'ampiezza dell'impulso di tensione indotto (normalizzato al valore dell'alimentazione).

Se esistono più aggressori piccoli, correlati, vengono uniti in un unico aggressore fittizio per semplicità. Anche i criteri di raggruppamento sono personalizzabili dall'utente.

A questo punto PrimeTime SI ha creato un sottoinsieme di tutti gli *aggressors* marchiandoli come ***effective_aggressors*** ed ha aggiornato gli oggetti del design con le nuove informazioni, ad esempio negli oggetti path compaiono i singoli contributi di ritardo dovuti al crosstalk.

4 Il Nuovo Flusso: Accurate Path Verification

Il flusso è stato ingegnerizzato con il principale scopo di ridurre al massimo il pessimismo introdotto dalla simulazione statica. Il grafico in Figura 16 mostra il trade-off accuratezza/velocità che si vuole rompere sfruttando al meglio la combinazione fra analisi statiche e dinamiche.

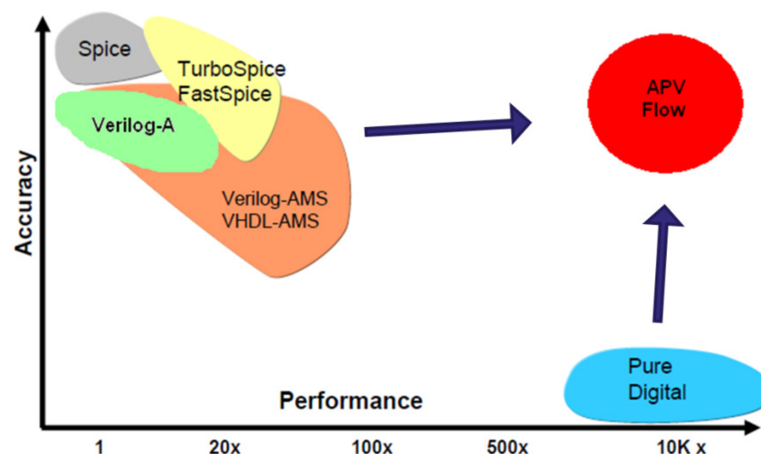


Figura 16. Trade-off tra accuratezza e velocità di vari tipi di simulazione

Il flusso APV è composto da più fasi e lavora in due ambienti diversi, all'interno del framework PrimeTime dove si raccolgono quasi tutte le informazioni sui path critici, e in un ambiente UNIX appositamente creato per l'elaborazione delle informazioni al fine di

generare la netlist SPICE da simulare, per lanciare una o più simulazioni e analizzarne i risultati. Nel seguito della trattazione l'ambiente UNIX verrà chiamato indifferentemente **advUtils** o **apvEasy**. In realtà il primo è un framework che racchiude vari tool per *l'Accurate Design/Digital Verification* tra cui l'*Accurate Path Verification Flow/Tool* oggetto di questa tesi. Le principali fasi del flusso sono illustrate in Figura 17.

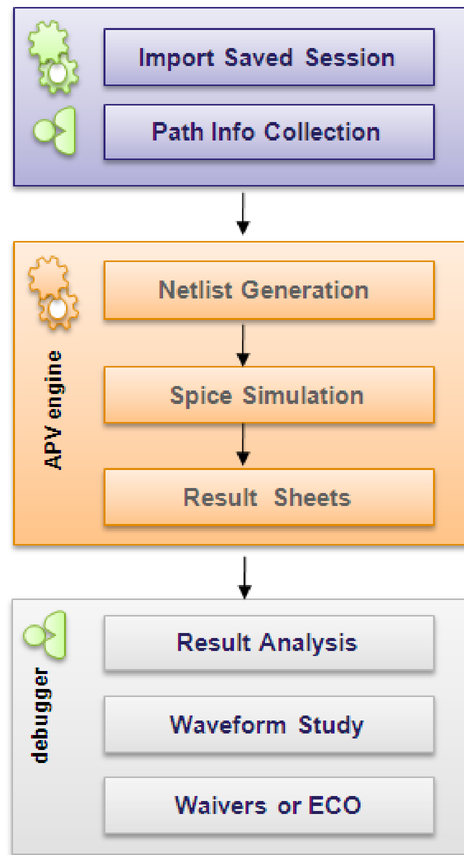


Figura 17. APV Flow

Il tutto inizia con una sessione di PrimeTime, ossia con un design definito al *gate-level* con linguaggio HDL e corredato di parassiti estratti da layout, librerie, *corners* e *timing constraints* da verificare. L'ambiente così composto è pronto per la *Static Timing Analysis*. In genere si eseguono alcune iterazioni di STA sistemando i problemi più evidenti (ad esempio inserendo buffer nella clock-tree per problemi di fan-out) e, a questo punto, il progettista si ritrova con una grossa parte (si spera) del circuito che funziona bene e lo farà

anche su silicio in quanto i dati della STA forniscono un abbondante margine di sicurezza, e un'altra parte su cui sono necessarie simulazioni più approfondite. Generalmente a questa seconda parte del design ci si riferisce col nome di ***Critical Paths***, seguendo quella naturale scomposizione in percorsi delimitati da due registri propria dell'analisi di timing su circuiti digitali. L'indecisione sui *Path Critici* è causa di due problematiche: il pessimismo introdotto durante la simulazione statica impedisce di ottenere il massimo delle prestazioni dal circuito in quanto, per evitare costosi fallimenti su silicio, si preferisce mantenere un certo margine di sicurezza sui dati della simulazione; un design aggressivo, cioè con piccoli margini, e fenomeni dinamici, in primis il crosstalk, possono portare al fallimento di path che secondo la STA dovrebbero funzionare.
Da quanto sopra si nota come i **Path Critici** non siano solo quelli con slack negativo ma tutti quei path per cui si vuole un livello di accuratezza maggiore rispetto ad un'analisi statica o presentano caratteristiche particolari, come ad esempio path al contorno di memorie, di pad di I/O o di blocchi analogici.

Definita la collezione di path critici da analizzare, il flusso prosegue con la cattura delle informazioni da PrimeTime, la loro elaborazione quindi la scrittura di tutti i file che compongono la netlist e infine il lancio delle simulazioni analogiche e la cattura dei risultati pronti per essere analizzati.
Tutti i passi verranno descritti dettagliatamente nei paragrafi successivi.

4.1 Descrizione dell’interfaccia grafica

Il tool **apvEasy** può essere utilizzato in modalità *batch*, ossia come se fosse un comando eseguibile da una *shell*. In questo caso prende il nome di apvBatch.

Indubbiamente, però, le caratteristiche e i vantaggi del flusso possono essere meglio compresi con l’ausilio della grafica. Verrà quindi presentata in questo paragrafo la GUI (*Graphical User Interface*), realizzata in linguaggio Tk, che accompagna la parte computazionale del tool, realizzata in linguaggio TCL.

Lo sviluppo di una interfaccia grafica è resa indispensabile per promuovere la suddetta metodologia: l’adozione di nuovi strumenti di design verte infatti, non solo sull’oggettiva utilità ma anche sulla facilità di impiego nel processo di verifica.

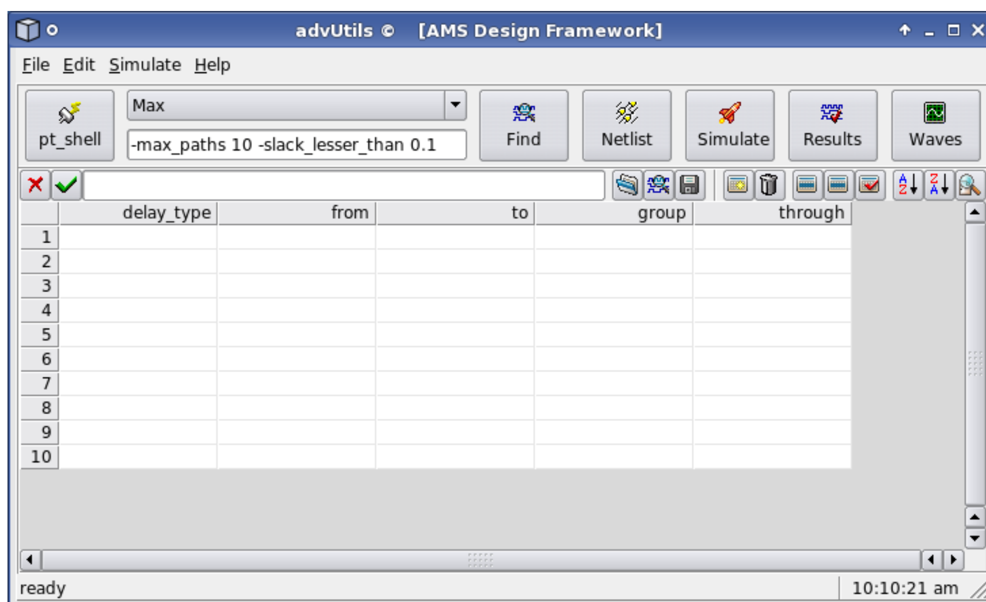


Figura 18. Interfaccia grafica all’avvio

4.1.1 Ruolo di tasti e tabelle

La Figura 18 mostra l’interfaccia grafica di apvEasy al suo avvio. Le due parti principali sono la fascia di bottoni e la tabella. La prima viene chiamata comunemente *ribbon* in ambito informatico e riassume in modo intuitivo, da sinistra verso destra, la successione dei passi necessari all’implementazione del flusso di verifica. La tabella può essere usata

come input per la definizione dei path (si veda la sezione 4.2 - Definizione dei Path Critici), ma è soprattutto la zona in cui visualizzare i risultati delle simulazioni statiche e dinamiche durante il flusso.

La barra dei menu, la barra di stato e la barra dei pulsanti posta al di sotto del *ribbon* sono stati aggiunti per ultimi e consentono rispettivamente di settare le variabili d'ambiente, avere informazioni durante l'elaborazione dei dati e interagire con la tabella, fornendo al designer utili strumenti per velocizzare le sue operazioni.

4.1.2 Server fra APV e PrimeTime

Il primo tasto del *ribbon* ha la funzione di connettere il tool al framework PrimeTime. Ciò serve a permettere lo scambio, bidirezionale, di tutte le informazioni necessarie al funzionamento del flusso. Dapprima il tool APV deve dire a PrimeTime secondo quali criteri deve selezionare i path critici all'interno del design. In seguito sarà PrimeTime a comunicare alcuni dei risultati dell'analisi statica al tool. Infine quest'ultimo interrogherà il database della STA per catturare le informazioni mancanti. E' inoltre possibile interagire manualmente con PrimeTime durante tutta la prima fase del flusso APV (ossia quella prima della simulazione dinamica), per modificare il design e i parametri della STA senza dover chiudere il tool e riavviarlo alla fine delle modifiche. L'interazione avviene per mezzo di una console, chiamata ***TclShell***, interamente realizzata in linguaggio TCL e capace di inviare comandi a PrimeTime e catturarne l'output per mostrarlo all'utente.

La Figura 19 mostra un esempio di utilizzo della Console. In giallo sono evidenziati alcuni comandi di PrimeTime: abilitazione e settaggi dell'analisi di *signal integrity*, lettura di un file di parassiti, settaggio del tipo di analisi (OCV), esecuzione dell'analisi e salvataggio della sessione.

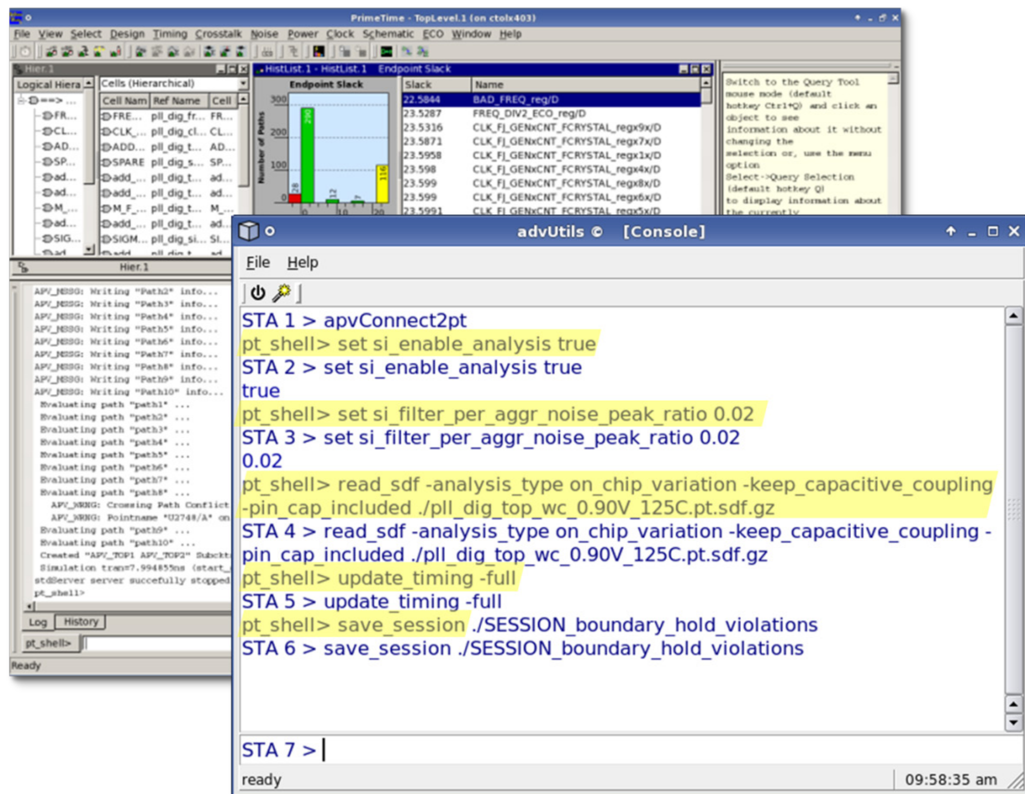


Figura 19. Console agganciata a PrimeTime

Il codice creato appositamente per far comunicare i tool di advUtils, fra cui apvEasy, e PrimeTime si basa sulla gestione di socket [16]. In pratica, l'ambiente è stato costruito in modo che lanciando apvEasy dall'interno di PrimeTime, si crea automaticamente un server in quest'ultimo e poi la connessione del client apvEasy. Molte procedure, poi, gestiscono la comunicazione smistando input output e codici di errore fra la shell di PrimeTime e la console.

Mostrare il codice sarebbe troppo lungo e farebbe distrarre il lettore dall'obiettivo principale della Tesi, per questo si preferisce ometterlo.

4.1.3 Tipi di analisi disponibili

Fino ad ora si è parlato in generale di analisi di path critici, mentre in realtà è possibile effettuare diversi tipi di analisi su di essi e su altri aspetti del design. E’ possibile selezionare il tipo di analisi, attraverso l’apposito menu a tendina, selezionando fra:

- Max
- MaxRise
- MaxFall
- Min
- MinRise
- MinFall
- Table
- ScanChain
- Clock
- Netlist

I primi otto punti riguardano essenzialmente l’analisi di path critici.

L’analisi di ***Max***, più comunemente nota come analisi di ***setup***, si occupa di verificare che i dati arrivino al secondo Flip-Flop almeno un tempo di setup prima del fronte di clock che li dovrà catturare.

Le analisi ***MaxRise*** e ***MaxFall*** sono analisi di setup che fissano il tipo di transizione (rise o fall) che il dato ha all’uscita del primo Flip-Flop.
E’ opportuno far notare che, cambiando il tipo di analisi da Rise a Fall o viceversa, cambiano i path critici in esame o almeno cambiano le sensitizzazioni dei suoi side-pins (cioè i livelli di tensione dei pin flottanti). Il motivo è principalmente dovuto a PrimeTime. Dato che le transizioni di rise/fall di ogni cella possono essere anche molto diverse fra loro, un path violato per un tipo di transizione può essere non violato per la transizione opposta, o magari non rientra più fra il numero massimo di path peggiori selezionati.

Con l’analisi di ***Min***, o di ***hold***, si verifica che il dato rimanga stabile per un certo tempo dopo il fronte di cattura. Per le analisi specifiche di Rise/Fall valgono le medesime considerazioni fatte per l’analisi di setup.

Il campo sotto il tipo di analisi è riempito con impostazioni di default e consente all’utente di personalizzare l’analisi. Nel caso di verifica di *setup* o di *hold* esso segue la sintassi del comando PrimeTime *get_timing_path* per facilitare il lavoro al progettista digitale. Per dettagli si veda la sezione 4.2 - Definizione dei Path Critici.

Selezionando ***Table*** la tabella dei risultati cambia momentaneamente funzione consentendo all’utente di immettere i dati necessari alla selezione dei path critici. Il formato è flessibile e permette di definire path disomogenei fra loro, ad esempio alcuni con transizione *rise* altri *fall*. Può essere utile anche quando si dispone già di un file in formato CSV (*Comma-Separated Values*) che descrive i path critici.
Per tutto il resto del flusso ci si riconduce sostanzialmente alle analisi di *setup* e *hold* descritte precedentemente.

Un’analisi di ***ScanChain*** consente di verificare tutto il design digitale (o una porzione) scompattandolo in tante catene costituite da più path accodati [27][28]. Il vantaggio consiste nel simulare tutte o solo alcune fra tutte le possibili configurazioni utilizzando un numero di colpi di clock inferiore al numero di scan della catena più lunga. Ciò è possibile sfruttando appositi pin di testing dei Flip-Flop pilotati in modo da verificare tutte le combinazioni di interesse e misurando più di una configurazione per ogni colpo di clock. In pratica si sfrutta il parallelismo nel settaggio dei dati da trasferire all’interno dei path che formano la catena.
Per creare le *SchanChain* l’utente deve listare gli *endpoint*, ossia i pin finali, e deve fornire alcuni parametri, come ad esempio i nomi dei pin di TestInput (TI) e TestEnable (TE).
Per il designer il resto del flusso è identico al flusso principale (setup&hold), anche se in realtà la procedura di creazione della netlist SPICE è diversa in quanto le informazioni di partenza sono diverse e la netlist che si vuole creare ha caratteristiche particolari.

Un'analisi diversa rispetto alle *Max* e alle *Min*, ma altrettanto importante, è l'analisi della ***clock-tree***. Tramite il campo sottostante è possibile dire a *PrimeTime* di restituire le informazioni relative a tutto l'albero di clock o solo ad alcune porzioni. Grazie a questo tipo di analisi è possibile valutare vari aspetti della clock-tree come il suo consumo e le sue emissioni elettromagnetiche (EMI). Si possono valutare i ritardi, e più importante, la distribuzione degli slew su tutti gli endpoint. Quest'ultimo aspetto, ossia la durata temporale delle transizioni sui pin di clock dei Flip-Flop, è molto importante per capire i trend del consumo di potenza e dell'EMI.
Essendo cambiati gli scopi della simulazione, il resto del flusso è ovviamente mutato rispetto al classico APV. La netlist viene infatti ad essere collezionata, sensitizzata e scritta in modo differente. Tuttavia l'impatto sulle operazioni che l'utente deve compiere è minimo e si concentra sulla fase di analisi dei dati. Si veda la sezione 5.2, che descrive l'analisi della clock-tree tramite un *Case Study*.

Ultima possibilità per l'utente è il ***NetFlow***, così chiamato perché parte appunto da una collezione di net. Questo flusso può anche essere eseguito al di fuori di una sessione di Static Timing Analysis aggiungendo così un livello ulteriore di flessibilità al tool APV. Ci si riferirà ora a questo caso, escludendo per un momento PrimeTime.
L'utente dispone già oppure crea sul momento un file con la **lista delle net** che si vogliono analizzare. Il secondo input dell'utente è un **file di parassiti**, in formato SPEF o DSPF, che altro non è che una sorta di netlist contenente tutti i collegamenti fra celle e net oltre, ovviamente, ai parassiti delle net. Visto che non si lavora direttamente con i path critici (anche se è possibile ricondursi a quella situazione), non si hanno a disposizione le informazioni riguardanti le sensitizzazioni di clock, *side-pins* e ingressi di dato. Per facilitare il compito del designer è stato creato l'apposito tab *Design* di impostazioni personalizzabili nel menu *Preferences* che verrà descritto nella sezione 4.1.4 che segue.
Questo flusso è molto utile nel caso in cui si voglia costruire un *testbench* su una porzione del design non facilmente selezionabile tramite la definizione di path critici (magari per la presenza di grossi blocchi analogici).
Verranno forniti ulteriori dettagli nella sezione 4.7.2 - Simulazioni manuali: NetFlow.

4.1.4 Variabili d'ambiente

Tramite il menu *Preferences*, l'utente ha la possibilità di effettuare delle scelte relative ad alcune variabili presenti nel tool. Queste sono state raggruppate in sezioni, in base alla loro funzione. Di seguito si trovano le descrizioni delle sezioni, mostrate poi in Figura 20.

- **Directories & Files**

  Permette di specificare il percorso delle directory utilizzate da apvEasy per i file di input e di output, nonché i nomi dei file contenenti le librerie, i parassiti, e infine la lista di net da analizzare in caso si utilizzi il NetFlow.

- **PrimeTime Timing Options**

  Consente di sovrascrivere le variabili catturate in PrimeTime nel caso di un normale flusso APV, o semplicemente di settarle in caso si utilizzi un flusso custom come il *NetFlow* o il *Clock-Tree Flow*.

  Le variabili che meritano di essere citate sono: le *slope*, ossia il tempo necessario ai dati per passare da V_{min} (solitamente 0V) a V_{supply}, in picosecondi; le *slope* degli aggressori (in ps); le soglie di scatto del valore logico (di default al 40% e 60% di V_{supply}) e infine le soglie usate per misurare le *slope* dei segnali nel mondo analogico (di solito 10% e 90% oppure 20% e 80%).

- **Design Options**

  Questa sezione permette principalmente di personalizzare quattro aspetti della netlist SPICE che verrà creata.

  La radice del nome da dare al o ai sottocircuiti generati e i nomi dei pin di alimentazione (per design *multi-voltage* e/o *multi-power*) non richiedono spiegazioni ulteriori.

  Il comportamento che il motore di generazione della netlist deve tenere nei confronti dei pin delle celle istanziate può essere personalizzato. Nello specifico è possibile marcare alcuni pin come ***common pins*** per poterli agevolmente pilotare. Può essere molto comodo nel caso dei pin di *Reset* o di *Test Input* e *Test Enable* in una analisi di *Scan Chains*. E' possibile anche indicare i pin di uscita da lasciare flottanti permettendo di caricarli manualmente con dei carichi noti e magari parametrici. I pin indicati come input vengono presi in considerazione solo in caso di analisi di *Scan Chains*. In seguito verranno introdotti nuovi campi per facilitare l'integrazione di

blocchi analogici. Infine il campo "*Floating pins to*" permette di assegnare un valore di tensione a tutti i pin rimasti flottanti, o ancora meglio di assegnare loro un'etichetta per poter poi "debuggare" la netlist in cerca di errori.

Il terzo aspetto riguarda questioni formali della netlist SPICE/verilog come ad esempio i caratteri usati come indicatori di bus e per la separazione gerarchica dei componenti o il *timescale*.

Infine, sia per fini statistici che di controllo, è possibile indicare i valori minimi di resistenze e capacità parassite da considerare.

- **Simulation Options**

  Tramite questo ultimo tab l'utente può settare il simulatore preferito, alcune sue opzioni, i file per le opzioni di simulazione avanzate e anche sovrascrivere alcuni valori calcolati da apvEasy come ad esempio la durata della simulazione di tipo *transient* e la temperatura.

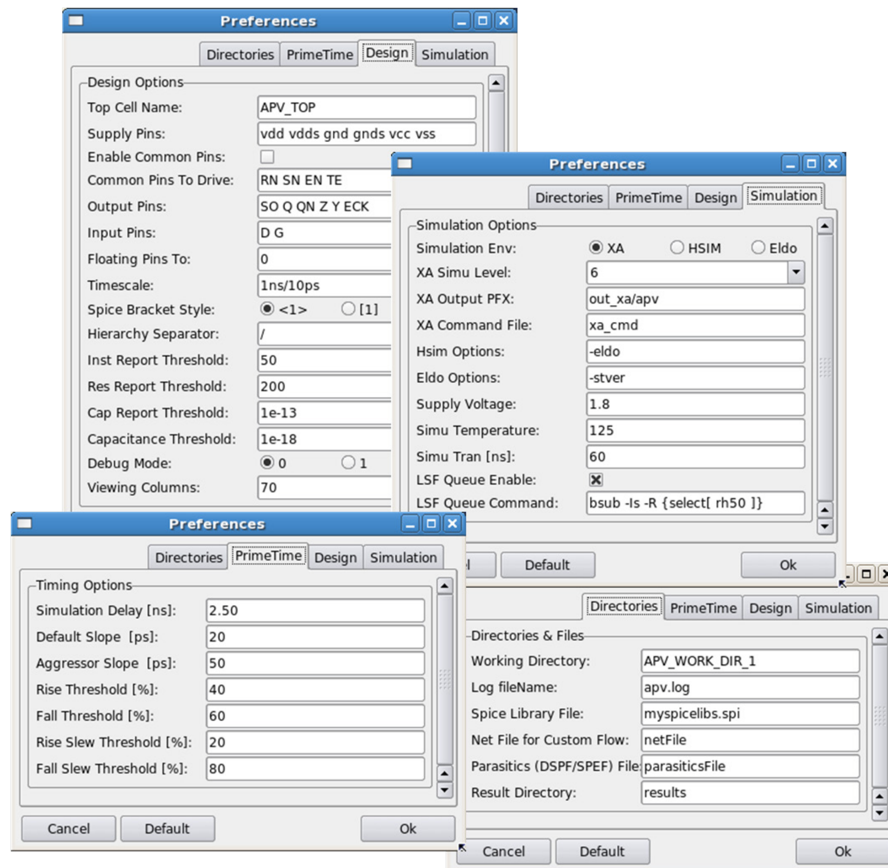


Figura 20. Menu delle preferenze

E' importante notare che la sezione *Environment Variables*, oltre a permettere all'utente una rapida e semplice configurazione delle principali opzioni, permette di selezionare i valori di default delle variabili al momento del caricamento di APV. Nella fase iniziale, quando si lancia il programma, vengono caricati i valori di default per le variabili presenti attraverso una procedura che ha il compito di leggere il contenuto di uno specifico file di inizializzazione; l'utente ha la possibilità di crearne uno proprio con i valori che ritiene opportuni che verrà salvato nella cartella di lavoro locale. Come si può vedere nella Figura 21, il file è composto da una lettera che identifica il tipo di elemento da inserire (per esempio r indica radio button), seguita da nome e chiave dell'array associato (tranne per le label e i tab) ed il testo da mostrare nella GUI.

```
t "Directories"
l "Directories & Files"
e icStd workDir   APV_HOLD "Working Directory"
e icStd logName   apv.log  "Log fileName"
e icStd spiFile   spiFile  "Spice Library File"
e icStd netFile   netFile  "Net File for Custom Flow"
e icStd parsFile  parsFile "Parasitics (DSPF/SPEF) File"
e icStd waveDir   out/apv  "Result Directory"

t PrimeTime
l "Timing Options"
e apvGlobals simu_delay             1  "Simulation Delay \[ns]"
e apvGlobals simu_slope             25 "Default Slope  \[ps]"
e apvGlobals aggressor_slope        40 "Aggressor Slope  \[ps]"
e apvGlobals delay_threshold_rise   40 "Rise Threshold \[%]"
e apvGlobals delay_threshold_fall   60 "Fall Threshold \[%]"
e apvGlobals slew_threshold_rise    20 "Rise Slew Threshold \[%]"
e apvGlobals slew_threshold_fall    80 "Fall Slew Threshold \[%]"

t Design
l "Design Options"
e icStd spiTop       DUT_SLACK                     "Top Cell Name"
e icStd pwrPins      "vdd vdds gnd gnds vcc vss"   "Supply Pins"
c icStd presvEn      0                             "Enable Common Pins"
e icStd presvPins    "RN SN EN TE"                 "Common Pins To Drive"
e icStd outPins      "SO Q QN Z Y ECK"             "Output Pins"
e icStd inPins       "D G"                         "Input Pins"
e icStd floatingPin  "Why_am_I_floating"           "Floating Pins To"
e icStd tScale       "1ns/10ps"                    "Timescale"
r icStd sBkt         "\<1>" "\[1]"                 "Spice Bracket Style"
e icStd hierStr      "/"                           "Hierarchy Separator"
e icStd iTh          50                            "Inst Report Threshold"
e icStd rTh          200                           "Res Report Threshold"
e icStd cTh          1e-13                         "Cap Report Threshold"
e icStd cSkip        1e-18                         "Capacitance Threshold"
r icStd debug        0 1                           "Debug Mode"
e icStd scktCols     70                            "Viewing Columns"

t "Simulation"
l "Simulation Options"
r icStd simEnv         "XA" "HSIM" "Eldo"  "Simulation Env"
r icStd xaLevel        6 3 4 5 7           "XA Simu Level"
e icStd xaOut          "out_xa/apv"        "XA Output PFX"
e icStd xaCmd          "xa_cmd"            "XA Command File"
e icStd hsimOpt        "-eldo"             "Hsim Options"
e icStd eldoOpt        "-stver"            "Eldo Options"
e icStd voltage        0.9                 "Supply Voltage"
e icStd temperature    125                 "Simu Temperature"
e icStd tran           50                  "Simu Tran \[ns]"
c icStd lsfen          1                   "LSF Queue Enable"
e icStd lsfcmd         "bsub -Is -R"       "LSF Queue Command"
```

Figura 21. Environment variables

In questo modo è possibile mantenere memorizzate diverse configurazioni per diverse analisi semplicemente utilizzando diverse directory di lavoro.

4.2 Definizione dei Path Critici

4.2.1 Premesse sul *Signoff*

Il primo passo del flusso APV, quello in cui si definisce la collezione di path da analizzare, è fondamentale per la buona riuscita dell'analisi approfondita che ci si appresta a intraprendere. La scelta è cruciale per vari aspetti.

Innanzitutto i path peggiori del design possono variare a seconda che la STA sia eseguita con o senza l'abilitazione del crosstalk. Di solito nel secondo caso il file di parassiti viene elaborato per aggiungere del pessimismo. Questa variabilità deve essere necessariamente tenuta sotto controllo con l'introduzione di pessimismo nella STA per evitare fallimenti su silicio, essendo questa uno strumento del *Signoff*, quindi una certificazione.

Senza tirare in ballo il crosstalk, il problema rimane se si pensa che tutto il chip viene analizzato in modalità OCV e un'analisi PBA sui path peggiori è facoltativa. La PBA elimina del pessimismo ma è pur sempre un'analisi statica (si riveda la sezione 3.2 - STA vs Transistor-level Timing Verification).

In caso di modellizzazione accurata del crosstalk (limitata solo dal tool di estrazione post-layout dei parassiti), i due aspetti precedenti costituiscono solo un limite al pieno utilizzo delle prestazioni ottenibili dal circuito, ma non portano a fallimenti inaspettati su silicio.

Le verifiche fatte in fase di *Signoff* devono essere eseguite seguendo alcuni passi obbligatori e necessitano di appositi ingressi per ottenere poi la certificazione per il *tape-out* su silicio. Al fine di non avere risultati aleatori, o comunque falsati, è necessario che i vari tool e i file da essi generati siano allineati durante tutto il flusso di verifica. In questa coerenza dei modelli rientra, ad esempio, la corretta gestione dei *corners* tecnologici. Se il corner di analisi corrente è il *worst-worst* sia i parassiti che la caratterizzazione delle celle dell'intero design dovranno essere di questo tipo.

Detto ciò si intuisce che la massima accuratezza ottenibile dal flusso APV non è semplicemente quella del simulatore analogico; per ottenerla è necessario utilizzare modelli adeguati e coerenti durante tutto il flusso di verifica.

In genere il designer digitale è incline a questo tipo di formalità, mentre quello analogico è più interessato alla malleabilità della netlist SPICE. Il flusso APV è in grado di

accontentarli entrambi, e a questo punto è necessario precisare la sua destinazione d'uso. Il flusso APV può essere usato:

- all'interno della fase di ***timing verification*** del *Signoff* per aumentare la precisione della STA e/o fare il debugging di path con violazioni.
- in qualsiasi fase del design flow come strumento avanzato
    - di **debugging**, per comprendere meglio il funzionamento di una determinata parte del circuito e risolvere eventuali problemi;
    - di **discovery**, per analizzare situazioni particolari come il cambio di tecnologia o la simulazione al di fuori dei domini di caratterizzazione disponibili.

Il primo caso è quello più importante. Ad oggi il flusso APV non è integrato nel *Signoff*, ma le premesse sulla rigida coerenza dei modelli utilizzati e la collaborazione intrapresa con Synopsys [22] fanno ben sperare. Una integrazione del flusso APV nel *Signoff*, o anche dell'intero tool direttamente in PrimeTime, potrà fornire un valido strumento integrativo veloce accurato e affidabile.

4.2.2 Flessibilità nella scelta dei path critici

Sono possibili vari metodi per la creazione della collezione di path da analizzare.

Il metodo principale consiste nel selezionare l'analisi (max o min) e fornire i parametri desiderati nella casella apposita della GUI. La sintassi segue le regole del comando PrimeTime *get_timing_path*. In questo modo è immediato ottenere la lista di tutti e soli i path violati, o solo i primi 10, o i 100 path peggiori che siano o no violati, o ancora i path con slack in un certo range, o magari appartenenti solo a una data lista di *endpoint*.

Una volta ottenuti i path in questo modo è possibile lanciare subito l'analisi o scartare/aggiungere alcuni path direttamente dalla tabella della GUI selezionando la modalità *Table*. La modalità *Table* consente di partire da una lista già nota di path definiti con *startpoint* e/o *endpoint* e/o *through* in formato *csv* o scritta direttamente a mano.

Un altro modo è quello di importare, tramite un menu apposito, un file contenente la lista dei *report timing file* dei path da analizzare.

In caso di analisi di *ScanChain* gli unici ingressi necessari sono gli *endpoint* e i pin attraverso cui espandere le *scan* all'indietro. Gli *endpoint* sono inseriti da tabella o da file, mentre i pin direttamente dall'interfaccia delle preferenze.

4.3 Collezione delle informazioni

Il primo passo del flusso APV è chiamato ***Find*** e può essere attivato attraverso l'apposito tasto della GUI. In questa fase il tool istruisce PrimeTime su quali path critici analizzare e si fa restituire le informazioni utili su di essi, insieme ad alcune informazioni sull'intero design. Nel seguito del paragrafo si descriveranno in dettaglio le informazioni catturate e come queste vengono elaborate al fine di poter creare la netlist completa nella successiva fase dedicata allo scopo. Dopo una premessa sulla risoluzione degli eventuali conflitti fra i vari path della collezione si descriveranno, nell'ordine, gli attributi dell'intero design, dei clock interessati e di tutti i path critici.

La Figura 22 riassume in modo grafico *inputs* e *outputs* di questa prima fase.

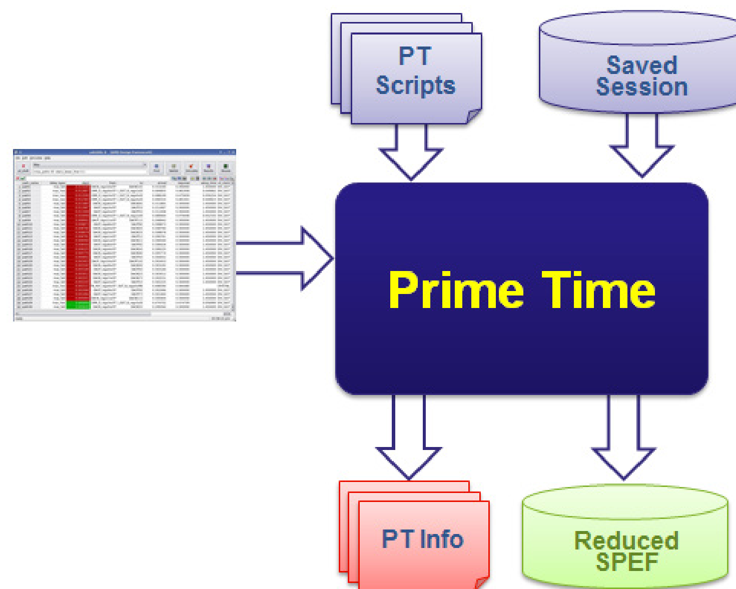


Figura 22. Prima fase del Flusso APV

4.3.1 Risoluzione dei conflitti

E' stato necessario organizzare la netlist SPICE in modo da accomodare più esigenze in contemporanea: la velocità della simulazione; l'accuratezza assoluta della porzione di circuito da simulare; l'accuratezza nella modellizzazione dell'influenza della restante parte; la corretta sensitizzazione di tutti i pin delle celle; infine, non meno importante, la facilità di lettura/modifica della netlist da parte del progettista. Per ottenere tutto ciò, durante lo sviluppo del flusso sono state prese delle decisioni e verificate una ad una con dei *test case*. Riporto qui di seguito, sotto forma di elenco, le decisioni finali che sono state prese.

1. Da tutta la **clock-tree** sono prese in considerazione solo le porzioni che dalla clock root arrivano agli *startpoint*/*endpoint* dei path critici.

2. La **clock-tree** ridotta del punto precedente viene corredata di tutti gli **aggressori** in grado di indurre apprezzabili ritardi in base a soglie standard o user-defined. Si veda l'apposita sezione 4.6.1.

3. Se due path critici condividono un **percorso comune**, dallo startpoint a un punto in mezzo al path, e i fronti del dato che lo attraversa sono concordi, vengono fusi in un solo path evitando la duplicazione inutile di celle del clock, del path e di carico, nonché di parassiti e aggressori.

4. Se una **cella di carico** di un path è anche una cella (non di carico) di un secondo path, aiutandoci con la Figura 23 possiamo vedere come essa venga duplicata e rinominata. Il numero di celle della netlist aumenta di una unità, ma bisogna notare che la soluzione alternativa sarebbe stata quella di porre i due path in due sottocircuiti separati con conseguente duplicazione di celle di clock e aggressori.

5. E' possibile che lo **startpoint** del path che si sta analizzando (path corrente) è un endpoint di un path già piazzato. Per start e endpoint si considerano in questo caso solo Flip-Flop e Latch e non ingressi/uscite primarie. In questo caso il path corrente viene

istanziato in un sottocircuito diverso per poter essere correttamente sensitizzato e per non intaccare le misure sul path con cui entrava in conflitto.

6. Caso del tutto analogo al precedente è quello per cui il path corrente ha un **endpoint** che coincide con uno startpoint già istanziato.

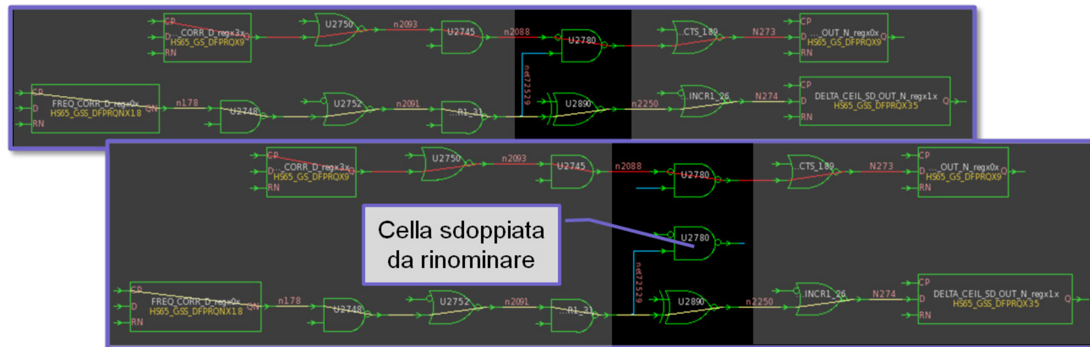


Figura 23. Esempio di risoluzione di un conflitto fra load e path cells

7. Se **due path si incrociano** a metà del path, cioè hanno almeno una net in comune e non hanno un percorso comune (per come è stato definito nel punto 3), è necessario istanziarli in due sottocircuiti differenti per evitare che il dato del path più veloce mascheri quello più lento.

8. Un esempio tipico di conflitto di ***clock management*** può verificarsi quando nella clock-tree è presente un multiplexer che permette la selezione di uno solo fra i diversi clock che ha ai suoi ingressi. Due path potrebbero in questo caso utilizzare due clock diversi.

   Le soluzioni sono due, o si separano i due path in due sottocircuiti diversi duplicando parte della clock-tree, o si allunga il tempo di simulazione cambiando a runtime la selezione del multiplexer.

9. Un **aggressore** del path che è anche una net di un *side-pin*, ossia di un pin di una cella del path che è necessario sensitizzare ad una tensione costante per poter permettere al dato di attraversarla, richiede una gestione specifica della netlist. E' necessario separare

la net aggressore dal pin caricandola con la capacità vista all'ingresso di quest'ultimo, che verrà poi pilotato in maniera identica agli altri *side-pins*.

10. Un caso particolare è quello in cui un **aggressore** aggredisce più di un path. Per evitare un allineamento troppo ottimistico fra aggressore e vittime si è preferito separare i path duplicando così gli aggressori. Questo punto risulterà più chiaro dopo la spiegazione di come vengono modellizzati gli aggressori in linguaggio Verolog-A nella sezione 4.6 - Modellizzazione del Crosstalk.

La risoluzione dei conflitti ha richiesto più tempo di quello preventivato in fase di definizione delle specifiche in quanto sono state necessarie delle modifiche, anche radicali, alla struttura del database contenente tutte le informazioni sui path e in generale sul design. La complessità di questo aspetto del flusso ha richiesto una minuziosa gestione delle precedenze fra le varie decisioni prese sui conflitti e una risoluzione dei conflitti sia a livello logico (algoritmo di ricerca dei conflitti per piazzamento dei path in sottocircuiti separati) che a livello strutturale lungo tutto il codice TCL (ridenominazione di celle, net, pin e gestione di moduli Verilog-A, generatori di tensione…).

Nella pagina seguente è riportata una porzione di codice che si occupa di risolvere alcuni dei suddetti conflitti.

```
########################################################
### Algorithm for solving conflicts between paths
########################################################
foreach_in_collection pp $path_points {
  set pointname [get_object_name [get_attribute $pp object]]
  set EDGES($pointname) [get_attribute $pp rise_fall]
  lappend POINTLIST $pointname
}
set size [llength $POINTLIST]; # PATH POINTS of current path
set sckt 0; while {1} {
  incr sckt; set INCR 0; set idx 1; set POINTS ""; set SUBCKTS($apvGlobals(spiTop)$sckt) 1
  foreach pointname $POINTLIST {
    set inst [ptpPoint2Inst $pointname]
    if {$idx < 2} {
    } elseif {$idx < 3} {
      if {[info exists endcells($sckt:$inst)]} {
        puts "APV_WRNG: EndPoint Conflict in Subckt$sckt"
        puts "          with previous \"$endcells($sckt:$inst)\" EndPoint."
        set INCR 1; break}
      set startcells($sckt:$inst) $path_name
    } elseif {$idx == $size} {
      if {[info exists startcells($sckt:$inst)]} {
        puts "APV_WRNG: StartPoint Conflict in Subckt$sckt"
        puts "          with previous \"$startcells($sckt:$inst)\" StartPoint."
        set INCR 1; break}
      set endcells($sckt:$inst) $path_name
    } elseif {[info exists apvPaths($sckt:$pointname)]} {
      if {![info exists COMMON($sckt:$POINTS)] ||
            $EDGES($pointname) != $ALLEDGES($sckt:$pointname)}{
        puts "APV_WRNG: Crossing Path Conflict in Subckt$sckt."
        puts "          between point \"$pointname\" and \"$apvPaths($sckt:$pointname)\""
        set INCR 1; break}
    } elseif [info exists TMPAGG($pointname)] {
      if [info exists apvScktInfo(paths:$sckt)] {
        foreach PATH $apvScktInfo(paths:$sckt) {
          if {[regexp \\\m$PATH\\\M $TMPAGG($pointname)]} {
            puts "APV_WRNG: Aggressor Conflict in Subckt$sckt"
            puts "         between point\"$pointname\"and\"$TMPAGG($pointname)\"in\"$PATH\""
            set INCR 1; break}
          }
        }
    }; incr idx
    set POINTS $POINTS/$pointname
    set COMMON($sckt:$POINTS) 1
    set POINTPATHS($sckt:$pointname) $path_name
    set ALLEDGES($sckt:$pointname) $EDGES($pointname)
  }
  if !$INCR break
} # sckt number is ready now!
```

4.3.2 Attributi generici del design

Fra gli svariati oggetti del database di PrimeTime si catturano all'inizio gli attributi del design. L'istruzione chiave che permette l'estrazione degli attributi dai vari tipi di oggetto è *get_attribute*.

Le tre istruzioni riportate sotto sono solo un esempio, tuttavia sono state scelte perché rappresentano i tre tipi di informazioni raccolte: a scopo **informativo** e di ***"logging"***, come il nome del design; a scopo di **controllo**, come la temperatura; a scopo **operativo**, come la/le tensione/i di alimentazione.

```
set curr_design [get_designs -filter "is_current==true"]
set temperature [get_attribute $curr_design temperature_$path_type]
set voltages    [get_attribute $curr_design voltage_$path_type]
```

Le informazioni catturate a scopo di controllo servono per mostrare dei messaggi di allerta all'utente nel caso in cui si rilevino delle discrepanze come ad esempio più di una temperatura operativa, soglie diverse o comunque "sospette" oppure tipi di analisi incompatibili.

Notare che il **path_type** che compare nel codice è legato all'analisi che si sta eseguendo, quindi può assumere solo il valore "max" o "min" a seconda che l'analisi sia di setup o di hold rispettivamente.

Un'informazione utile da catturare è la lista delle unità di misura usate. Nel caso il designer non le abbia modificate, PrimeTime utilizza di default il ns, il pF e l'Ω.

A questo punto si inizia a catturare tutti i vari tipi di informazioni legati ai path.

4.3.3 Attributi dei clock

All'inizio vengono raccolte tutte le informazioni sui clock. Si è scelto di lavorare sui due attributi "*launch_clock_paths*" e "*capture_clock_paths*", definiti per ciascun oggetto di tipo path, Come lascia intendere il nome, sono a loro volta degli oggetti, quindi contengono varie informazioni.

Il codice seguente mostra come vengono estratti i percorsi di lancio e cattura del clock per ogni path della collezione: ancora una volta si usa la funzione *get_attribute*.

```
set sp_clock_points [get_attribute -quiet [get_attribute -quiet \
                        $current_path launch_clock_paths] points]
set ep_clock_points [get_attribute -quiet [get_attribute -quiet \
                       $current_path capture_clock_paths] points]

### ENDPOINT CLOCK
if {[sizeof_collection $ep_clock_points] > 0} {
   set clock      [get_attribute -quiet $current_path endpoint_clock]
   set ep_clock   [get_attribute -quiet $clock full_name]

   set clkroot    [get_attribute [index_collection $ep_clock_points 0] object]
   set clknet     [get_attribute $clkroot full_name]
   set mclock     [get_clock -quiet $clknet]
   if {$mclock != ""} {set clock $mclock}
   set period     [get_attribute $clock period]
   set waveform   [get_attribute $clock waveform]
   set driver     [get_attribute -quiet $clkroot driving_cell_rise_$path_type]
   set driverpin  [get_attribute -quiet $clkroot driving_cell_pin_rise_$path_type]
   ...
}
### STARTPOINT CLOCK
...
```

Questa porzione di codice si occupa solo dell’estrazione delle seguenti informazioni dei clock: il nome; il suo punto di inizio, chiamato *clock-root*, che serve per poter poi inserire un generatore a onda quadra; il periodo, la fase, e il duty-cycle; infine la cella pre-driver che serve a smussare opportunamente i fronti.

E’ necessario fare ciò separatamente per la clock-tree di lancio e di cattura perché possono avere clock diversi.

Le informazioni sulle net e le celle saranno prese in una fase successiva.

4.3.4 Attributi dei path

In merito ai path si memorizzano tanti tipi di informazioni. Una lista di tutti gli attributi per un oggetto di questo tipo è visibile in Figura 24.

```
Design          Object              Type      Attribute Name          Value
--------------------------------------------------------------------------------
pll_dig_top     path                float     arrival                 0.506225
pll_dig_top     path                string    blocks                  _Top_
pll_dig_top     path                collection capture_clock_paths    path
pll_dig_top     path                float     clock_uncertainty       -0.150000
pll_dig_top     path                float     common_path_pessimism   0.000000
pll_dig_top     path                collection crpr_common_point      INTDIV_OUT
pll_dig_top     path                string    end_block               _Top_
pll_dig_top     path                collection endpoint               DELTA_CEIL_SD_OUT_N_regx1x/D
pll_dig_top     path                collection endpoint_clock         INTDIV_OUT
pll_dig_top     path                string    endpoint_clock_close_edge_type rise
pll_dig_top     path                float     endpoint_clock_close_edge_value 0.600000
pll_dig_top     path                boolean   endpoint_clock_is_inverted true
pll_dig_top     path                boolean   endpoint_clock_is_propagated true
pll_dig_top     path                float     endpoint_clock_latency  0.084405
pll_dig_top     path                string    endpoint_clock_open_edge_type rise
pll_dig_top     path                float     endpoint_clock_open_edge_value 0.600000
pll_dig_top     path                collection endpoint_clock_pin     DELTA_CEIL_SD_OUT_N_regx1x/CP
pll_dig_top     path                boolean   endpoint_is_level_sensitive false
pll_dig_top     path                float     endpoint_setup_time_value 0.048953
pll_dig_top     path                boolean   is_recalculated         false
pll_dig_top     path                boolean   is_recovered            false
pll_dig_top     path                collection launch_clock_paths     path
pll_dig_top     path                string    object_class            timing_path
pll_dig_top     path                collection path_group             INTDIV_OUT
pll_dig_top     path                string    path_type               max
pll_dig_top     path                collection points                 ...
pll_dig_top     path                float     required                0.485452
pll_dig_top     path                float     slack                   -0.020773
pll_dig_top     path                string    start_block             _Top_
pll_dig_top     path                string    start_end_blocks        _Top_  _Top_
pll_dig_top     path                string    start_end_blocks_sorted _Top_  _Top_
pll_dig_top     path                collection startpoint             FREQ_CORR_D_regx2x/CP
pll_dig_top     path                collection startpoint_clock       INTDIV_OUT
pll_dig_top     path                boolean   startpoint_clock_is_inverted false
pll_dig_top     path                boolean   startpoint_clock_is_propagated true
pll_dig_top     path                float     startpoint_clock_latency 0.146259
pll_dig_top     path                string    startpoint_clock_open_edge_type rise
pll_dig_top     path                float     startpoint_clock_open_edge_value 0.000000
pll_dig_top     path                boolean   startpoint_is_level_sensitive false
pll_dig_top     path                string    through_blocks          _Empty_
pll_dig_top     path                float     time_borrowed_from_endpoint 0.000000
pll_dig_top     path                float     time_lent_to_startpoint 0.000000
```

Figura 24. Esempio di attributi di un oggetto path di PrimeTime

Fra gli attributi, dopo i nomi di *startpoint* e *endpoint* e i nomi dei clock di lancio e di cattura, spiccano tre oggetti: *“**points**”*, *“**launch_clock_paths**”* e *“**capture_clock_paths**”*. Questi sono una collezione ordinata di “punti” e contengono tutte le informazioni sulla netlist riguardante il path sotto esame. Grazie a loro è possibile estrarre tutte le net e le celle connesse, sia del path che dei due rami di clock-tree. Gli oggetti “punto” contengono molte altre informazioni, ad esempio, i valori di tensione possibili e le capacità dei nodi. Infine, fra gli attributi di ciascun path, compaiono tutte le informazioni utilizzate per la *Static Timing Analysis* e i risultati della stessa. Per nominarne alcune abbiamo a disposizione il tempo di arrivo del dato (*arrival*), il tempo richiesto (*required*), naturalmente lo *slack* che è la loro differenza, il *setup time* (o l’*hold time*), vari termini di incertezza sul clock, l’indicazione della eventuale presenza di latch al posto di FF e i rispettivi *constraint* di prestito di tempo. Queste informazioni vengono catturate sia per essere mostrate al designer nell’interfaccia grafica, sia per la successiva fase di simulazione analogica. Il codice seguente mostra come vengono catturate le informazioni per ciascun path della collezione.

```
foreach_in_collection current_path $timing_paths {
    set path_points     [get_attribute $current_path points]
    set sp_clock_points [get_attribute -quiet [get_attribute -quiet \
                           $current_path launch_clock_paths] points]
    set ep_clock_points [get_attribute -quiet [get_attribute -quiet \
                           $current_path capture_clock_paths] points]

    set start_point  [get_attribute [get_attribute $current_path startpoint] full_name]
    set end_point    [get_attribute [get_attribute $current_path endpoint] full_name]
    set sp_clock     [get_attribute -quiet [get_attribute -quiet $current_path \
                        startpoint_clock] full_name]
    set ep_clock     [get_attribute -quiet [get_attribute -quiet $current_path \
                        endpoint_clock] full_name]
    set ep_clock_pin [get_object_name [get_attribute $current_path endpoint_clock_pin]]
    set ep_cedge     [get_attribute -quiet $current_path endpoint_clock_close_edge_type]
    set ep_cedgev    [get_attribute  -quiet $current_path endpoint_clock_close_edge_value]
    set required     [get_attribute -quiet $current_path required]
    set arrival      [get_attribute -quiet $current_path arrival]
    set slack        [get_attribute -quiet $current_path slack]
    set clk_uncert   [get_attribute -quiet $current_path clk_uncert]
    set ep_setup     [get_attribute -quiet $current_path endpoint_setup_time_value]
    set ep_hold      [get_attribute -quiet $current_path endpoint_hold_time_value]
    set ext_delay    [get_attribute -quiet $current_path endpoint_output_delay_value]
    ...
}
```

Il database è strutturato in modo da etichettare le informazioni con un identificativo di ciascun path. Questo è necessario per poter risolvere tutti i conflitti fra path. Ricordiamo che è possibile avere due path che si incrociano, oppure path che vorrebbero polarizzazioni diverse della stessa cella oppure, ancora, path che richiedono condizioni della clock-tree contrastanti.

A proposito di clock-tree, va sottolineato come essa stia diventando sempre più complessa nei moderni circuiti. Sono presenti al suo interno clock generati a partire da altri, clock smistati tramite multiplexer, divisioni in aree disattivabili in modo separato per ridurre il consumo di potenza (tecnica del *clock-gating*), così come logica di controllo che gestisce la frequenza di clock a run-time.
E' facile capire come due path possano entrare in conflitto in merito alle loro porzioni di clock-tree.

Le informazioni sulle tensioni dei **pin flottanti**, che siano *side-pins* (cioè pin del path) o pin di *clock-gating*, vengono catturati in questa fase sotto forma di *deck files*. Questi file vengono fatti scrivere a PrimeTime e saranno letti e interpretati nella fase successiva di creazione della netlist. L'Appendice A: SPICE mostra un esempio commentato di *deck file*.

Va precisato che, per motivi di accuratezza, durante l'esplorazione di ciascun punto del path e delle clock-tree di lancio e cattura, si catturano anche le celle carico che non sarebbero necessarie al funzionamento della netlist. Il prezzo è l'aumentata complessità computazionale dell'algoritmo di generazione della netlist, ma soprattutto l'accresciuta dimensione di quest'ultima con conseguente degradamento delle prestazioni in termini di velocità della simulazione. Tuttavia è stato verificato tramite varie simulazioni comparative che una capacità fissa al posto di ogni cella carico introduce incertezze inaccettabili nei ritardi di propagazione dei dati e dei clock. Ci si riferisce a questa modellizzazione della netlist con il nome di ***level 1 load modeling*** ad indicare che ci si espande di un livello.
Durante l'esplorazione di ciascun punto si collezionano anche tutti gli aggressori in modo da tenere in conto anche degli effetti del **crosstalk**. Anche la cattura di queste informazioni prevede l'etichettatura con l'identificativo di ciascun path. La decisione di quanti e quali aggressori debbano essere catturati è presa da PrimeTime durante la *Static Timing Analysis*, ma l'utente può personalizzare le soglie di default a proprio piacimento. Per i dettagli si riveda la sezione 3.3 - Signal Integrity. Per quanto riguarda, invece, la sensitizzazione di queste net si rimanda alla sezione 4.6 - Modellizzazione del Crosstalk.

Avendo ora a disposizione, in diversi array, tutte le net di tutti i path e delle rispettive porzioni di clock-tree, corredate degli aggressori, si procede alla creazione di uno *SPEF file*. Un solo comando di PrimeTime, riportato di seguito, è necessario per scrivere questo file che sarà cruciale per la creazione della netlist SPICE.

```
write_parasitics -format SPEF -nets $LIST_OF NETS reducedSPEF.spef
```

Consiste in un file dei parassiti di tutte e sole le net della porzione di design da analizzare, organizzate in una sintassi *SPICE-like* (si veda l'Appendice B: SPEF/DSPF) e quindi contenente tutte le informazioni sulla connettività delle celle. Il *parsing* di questo file è il centro dell'algoritmo di creazione della netlist.
Sarà spiegato che può esistere più di uno SPEF, ognuno legato ad un sottocircuito della netlist che si andrà a creare nella fase successiva.

La Figura 25 mostra cosa vede il progettista alla fine della fase "Find" di raccolta delle informazioni. La tabella dei risultati è riempita con tutti i dati della STA riguardanti i path oggetto di verifica. Spicca la colonna degli slack, rossa per tutti i path che violano il vincolo di setup.

advUtils © [AMS Design Framework]
File Edit Simulate Help
pt_shell | Max | -max_paths 30 -slack_lesser_than 0.1 | Find | Netlist | Simulate | Results | Waves

	path_name	delay_type	slack	from	to	arrival	required	setup_time	nt_clock
1	path1	max_fall	-0.013240	DACN_regx10x/CP	DACN[10]	0.313240	0.300000	1.450000	DIV_OUT
2	path2	max_rise	-0.012897	:ORR_D_regx0x/CP	)_OUT_N_regx1x/D	0.494855	0.481958	0.049962	DIV_OUT
3	path3	max_rise	-0.012590	:ORR_D_regx3x/CP	)_OUT_N_regx0x/D	0.488249	0.475659	0.056254	DIV_OUT
4	path4	max_rise	-0.012383	:ORR_D_regx0x/CP	)_OUT_N_regx4x/D	0.494324	0.481941	0.049923	DIV_OUT
5	path5	max_fall	-0.011865	DACN_regx6x/CP	DACN[6]	0.311865	0.300000	1.450000	DIV_OUT
6	path6	max_fall	-0.011687	DACP_regx3x/CP	DACP[3]	0.311687	0.300000	1.450000	DIV_OUT
7	path7	max_fall	-0.011008	DACP_regx5x/CP	DACP[5]	0.311008	0.300000	1.450000	DIV_OUT
8	path8	max_fall	-0.010056	:ORR_D_regx0x/CP	)_OUT_N_regx2x/D	0.489464	0.479408	0.052335	DIV_OUT
9	path9	max_fall	-0.008942	DACP_regx11x/CP	DACP[11]	0.308942	0.300000	1.450000	DIV_OUT
10	path10	max_fall	-0.008873	DACP_regx8x/CP	DACP[8]	0.308873	0.300000	1.450000	DIV_OUT
11	path11	max_fall	-0.008760	DACN_regx0x/CP	DACN[0]	0.308760	0.300000	1.450000	DIV_OUT
12	path12	max_fall	-0.008678	DACN_regx3x/CP	DACN[3]	0.308678	0.300000	1.450000	DIV_OUT
13	path13	max_fall	-0.006761	DACP_regx1x/CP	DACP[1]	0.306761	0.300000	1.450000	DIV_OUT
14	path14	max_fall	-0.006594	DACN_regx1x/CP	DACN[1]	0.306594	0.300000	1.450000	DIV_OUT
15	path15	max_fall	-0.006426	DACP_regx6x/CP	DACP[6]	0.306426	0.300000	1.450000	DIV_OUT
16	path16	max_fall	-0.006225	DACN_regx4x/CP	DACN[4]	0.306225	0.300000	1.450000	DIV_OUT
17	path17	max_fall	-0.005774	DACN_regx8x/CP	DACN[8]	0.305774	0.300000	1.450000	DIV_OUT
18	path18	max_fall	-0.004052	DACP_regx4x/CP	DACP[4]	0.304052	0.300000	1.450000	DIV_OUT
19	path19	max_fall	-0.003453	DACP_regx10x/CP	DACP[10]	0.303453	0.300000	1.450000	DIV_OUT
20	path20	max_fall	-0.003192	DACN_regx9x/CP	DACN[9]	0.303192	0.300000	1.450000	DIV_OUT
21	path21	max_fall	-0.003184	DACP_regx0x/CP	DACP[0]	0.303184	0.300000	1.450000	DIV_OUT
22	path22	max_fall	-0.003012	DACN_regx2x/CP	DACN[2]	0.303012	0.300000	1.450000	DIV_OUT
23	path23	max_fall	-0.002531	DACN_regx7x/CP	DACN[7]	0.302531	0.300000	1.450000	DIV_OUT
24	path24	max_fall	-0.002225	DACP_regx2x/CP	DACP[2]	0.302225	0.300000	1.450000	DIV_OUT
25	path25	max_rise	-0.002111	PN_RSY_regx4x/CP	OUT_N_regx0x/RN	0.496590	0.494480		CRYSTAL
26	path26	max_fall	-0.002066	DACP_regx9x/CP	DACP[9]	0.302066	0.300000	1.450000	DIV_OUT
27	path27	max_fall	-0.001064	DACP_regx7x/CP	DACP[7]	0.301064	0.300000	1.450000	DIV_OUT
28	path28	max_fall	-0.000400	DACN_regx11x/CP	DACN[11]	0.300400	0.300000	1.450000	DIV_OUT
29	path29	max_rise	0.000006	:ORR_D_regx3x/CP	)_OUT_N_regx3x/D	0.474702	0.474709	0.056990	DIV_OUT
30	path30	max_fall	0.000636	DACN_regx5x/CP	DACN[5]	0.299364	0.300000	1.450000	DIV_OUT

ready | 03:38:15 pm

Figura 25. Tabella della GUI riempita con le informazioni prese da PrimeTime

4.4 Generazione della Netlist SPICE

A questo punto si hanno tutte le informazioni, memorizzate in diversi formati, per generare la netlist SPICE. La Figura 26 mostra lo schema per questa seconda fase e le due successive in cui si lancia la simulazione e si analizzano i risultati.

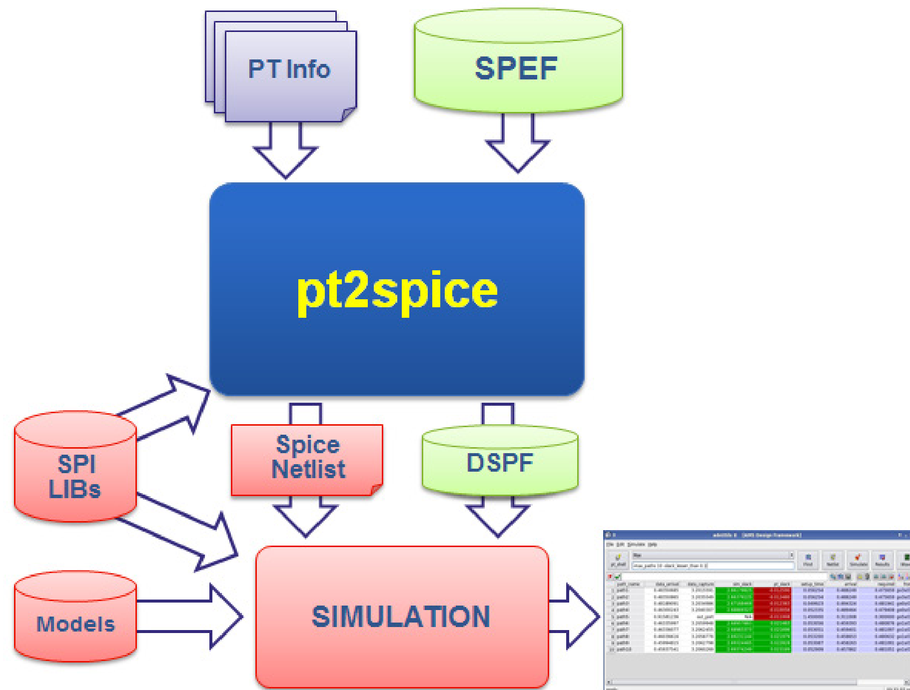


Figura 26. Seconda parte del Flusso APV

Memorizzate in formato database, si hanno istanze, pin e net delle celle dei path e della clock-tree, le informazioni sui generatori dei clock e degli aggressori, le net degli aggressori estranee ai path. Le informazioni sulle tensioni costanti di sensitizzazione dei pin flottanti saranno in parte catturate dai *deck files* e in parte generate dal tool. La denominazione e l'ordine delle porte di ciascuna cella di libreria usata sono ottenute facendo il parsing dei file di libreria. Infine i parassiti RC sono contenuti nello SPEF ridotto fatto scrivere a PrimeTime.

In realtà il file dei parassiti contiene anche tutte le informazioni sulla connettività del *testcase* sotto analisi. Per questo motivo, la creazione della netlist SPICE parte proprio con l'analisi di questo file. Durante la lettura viene generato un file di parassiti in formato

DSPF e contemporaneamente si crea un database della connettività. Il file DSPF sarà usato per la simulazione (si veda la sezione 4.4.5 - DSPF#.dspf e ptReducedSPEF#.spef), mentre il database delle connettività, assieme alle informazioni di PrimeTime, serve ad una complessa procedura (***pt2spice***) che ha il compito di scrivere i vari file costituenti la netlist. Per motivi di leggibilità, ma soprattutto per i diritti d'autore, nel resto del paragrafo 4.4 verranno mostrate solo alcune sezioni di codice TCL sviluppato.

Con il nome ***pt2spice*** viene in realtà indicato tutto l'insieme di procedure necessarie alla scrittura della netlist SPICE e dell'ambiente di simulazione. Assume questo significato anche in Figura 26. Nella medesima figura sono mostrati genericamente la netlist e le librerie in rosa e il DSPF in verde. Il dettaglio della struttura dell'ambiente può essere apprezzato con l'ausilio della Figura 27 e con le descrizioni dei singoli file che saranno qui di seguito riportate.

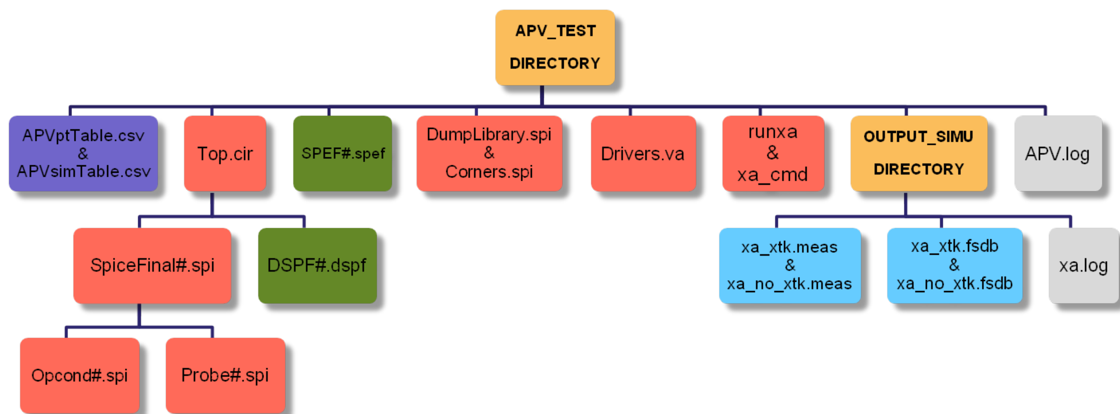


Figura 27. Struttura dei file nell'ambiente UNIX

E' doveroso precisare che la struttura interna dei singoli files, così come la struttura di tutto l'ambiente creato, hanno come principale obiettivo quello di **fornire al progettista una netlist facile da leggere e modificare** oltre che pronta per essere simulata.

4.4.1 Top.cir

Il ***Top.cir*** è il file principale della struttura della netlist. Esso contiene tutti i riferimenti agli altri file per cui è l'unico ad essere dato in pasto al simulatore. Ha una struttura fissa per facilitarne la lettura e la modifica, come è possibile vedere dall'esempio seguente.

```
************************************************
*** Created by apvEasy - v.2013.03.dev:1001
*** on Tue Apr 23 17:05:10 CEST 201304/18/2013
*** Design:    PLL_TOP
*** Subckt #:  2
*** Crosstalk: YES
************************************************
*** DUMP LYBRARY and CORNERS
.include dumpLibrary.spi
.include corners.spi

*** SUBCIRCUITS INSERTIONS
.include spiceFinal1.spi
.dspf_include ./DSPF1.dspf
.include spiceFinal2.spi
.dspf_include ./DSPF2.dspf
*** TOPLEVEL INSTANTIATIONS
XIDUT_SLACK_1 gnd gnds vdd vdds DUT_SLACK_1 supply=supply
XIDUT_SLACK_2 gnd gnds vdd vdds DUT_SLACK_2 supply=supply

*** SUPPLIES DEFINITIONS
.param supply = 0.9
Vgnd  gnd  0 0
Vgnds gnds 0 0
Vvdd  vdd  0 supply
Vvdds vdds 0 supply

*** SIMULATION PROPERTIES
.tran 0.001ns 10.5ns
.temp 125.0

*** CROSSTALK
.verilog drivers.va
.param aggr_enable = 0
.subckt apvaggressor vref in out inv=1 slope=20ps enable=aggr_enable
xagg vref in out aggressor inv=inv slope=slope enable=enable
.ends
.alter
.param aggr_enable = 1

.end
```

4.4.2 spiceFinal#.spi

Lo *spiceFinal* contiene la vera e propria netlist, ossia tutte le istanziazioni delle celle dei path, quelle della porzione di clock-tree usata e quelle delle celle di carico. Le *load cells* sono sia le celle carico della clock-tree che dei path. Il file *spiceFinal* viene suddiviso in tre aree, nell'ordine *path cells*, *clock cells* e *load cells*, per permettere una facile individuazione delle celle in caso il designer voglia modificare la netlist prima di simularla. Bisogna ricordare che in caso ci siano dei conflitti fra i critical path, questi vengono suddivisi in due o (raramente) più sottocircuiti per cui esisterà uno o più *spiceFinal*.

Sempre con l'obiettivo di rendere facile la navigazione all'interno della netlist, si è preferito inserire tutte le tensioni di sensitizzazione dei pin flottanti e tutti i moduli per la modellizzazione del crosstalk, così come tutti i diversi tipi di statement per catturare le informazioni della simulazione, tramite due soli *.include*. Lo statement *.include* consente di caricare esplicitamente il contenuto di un file in un altro file subito prima di fare la simulazione. I due file in questione (in realtà due per ogni *spiceFinal* esistente) sono gli *opcond* e i *probe* e verranno di seguito descritti.

Segue un esempio di *spiceFinal* in cui è possibile distinguere la definizione del sottocircuito (fra le parole chiave ".SUBCKT" e ".ENDS"), i *.include* dell'opcond e del probe file, e infine le tre zone in cui sono istanziate le celle dei path, della clock-tree e quelle di carico.

```
.SUBCKT APV_TOP1 vdd vdds gnd gnds vcc vss PARAM: supply=supply

.include opcond1.spi
.include probe1.spi

********************************************
*** Critical Path Instances
********************************************
XDACP_regx5x N429 DACP_regx5x/D
+STM_BOUNDARY_ISOLATION_NET_31570_DACPx5x APV_PWR_supply gnd gnds vdd
+vdds HS65_GS_DFPRQX9
XDELTA_CEIL_SD_OUT_N_regx1x N429 N274 DELTA_CEIL_SD_OUT_N_regx1x/Q
+APV_PWR_supply gnd gnds vdd vdds HS65_GSS_DFPRQX35
XFREQ_CORR_D_regx0x INTDIV_OUT_BC_1_G3B2I4_1 FREQ_CORR_D_regx0x/D n178
+APV_PWR_supply gnd gnds vdd vdds HS65_GSS_DFPRQNX18
XICC_POSTCTS_INCR1_26 APV_PWR_supply n2250 N274 gnd gnds vdd vdds HS65_GS_NOR2AX13
XICC_POSTCTS_INCR1_31 n2091 APV_PWR_supply net72529 gnd gnds vdd vdds HS65_GS_NAND2X14
```

```
XU2693 APV_PWR_0 xcellx118976xnet77011 xcellx118976xnet77012 gnd gnds
+vdd vdds HS65_GS_CNNOR2X24
XU2748 n178 APV_PWR_supply xcellx118976xnet77011 gnd gnds vdd vdds HS65_GS_NAND2X21
XU2752 APV_PWR_supply xcellx118976xnet77012 n2091 gnd gnds vdd vdds
+HS65_GS_NOR2AX13
XU2890 APV_PWR_0 net72529 n2250 gnd gnds vdd vdds HS65_GS_XNOR2X18

********************************************
*** CLOCK Instances
********************************************
XCLK_FJ_GENxN_M_UPDATE_RSY_regx1x INTDIV_OUT_BC_1_G3B2I4_1
+APV_GROUNDED:D CLK_FJ_GENxN_M_UPDATE_RSY_regx1x/Q APV_GROUNDED:RN gnd
+gnds vdd vdds HS65_GH_DFPRQX9
XDACN_regx0x N429 APV_GROUNDED:D DACN_regx0x/Q APV_GROUNDED:RN gnd
+gnds vdd vdds HS65_GS_DFPRQX9
XDACN_regx10x N429 APV_GROUNDED:D DACN_regx10x/Q APV_GROUNDED:RN gnd
+gnds vdd vdds HS65_GS_DFPRQX9
XDACN_regx2x N429 APV_GROUNDED:D DACN_regx2x/Q APV_GROUNDED:RN gnd
+gnds vdd vdds HS65_GS_DFPRQX9
XDELTA_CEIL_SD_OUT_N_regx7x N429 APV_GROUNDED:D
+DELTA_CEIL_SD_OUT_N_regx7x/Q APV_GROUNDED:RN gnd gnds vdd vdds HS65_GSS_DFPRQX18
XFREQ_ACC_regx4x INTDIV_OUT_BC_1_G3B2I4_1 APV_GROUNDED:D
+FREQ_ACC_regx4x/Q APV_GROUNDED:RN gnd gnds vdd vdds HS65_GS_DFPRQX9
XFREQ_CORR_D_regx2x INTDIV_OUT_BC_1_G3B2I4_1 APV_GROUNDED:D
+FREQ_CORR_D_regx2x/QN APV_GROUNDED:RN gnd gnds vdd vdds HS65_GSS_DFPRQNX18
XFREQ_SPAN_regx19x INTDIV_OUT_BC_1_G3B2I4_1 APV_GROUNDED:D
+FREQ_SPAN_regx19x/Q APV_GROUNDED:RN gnd gnds vdd vdds HS65_GS_DFPRQX9
XICC_CTS_0_HS65_GS_CNIVX10_G3IP_1 INTDIV_OUT_BC_1
+ICC_CTS_0_HS65_GS_CNIVX10_G3IP_1/Z gnd gnds vdd vdds HS65_GS_CNIVX10
XSIGMA_DACxSIGMA_RX_RET_DD_regx14x INTDIV_OUT_BC_1_G3B2I4_1
+APV_GROUNDED:D SIGMA_DACxSIGMA_RX_RET_DD_regx14x/QN APV_GROUNDED:RN
+gnd gnds vdd vdds HS65_GH_DFPRQNX9
XSTM_DIODE_PROTECTION_INTDIV_OUT INTDIV_OUT gnd gnds vdd vdds HS65_GS_ANTPROT3
XU595 INTDIV_OUT_BC_1 N429 gnd gnds vdd vdds HS65_GS_CNIVX21
...

********************************************
*** LOAD Instances
********************************************
XICC_POSTCTS_INCR1_34_LOAD xcellx118976xnet77012
+ICC_POSTCTS_INCR1_34_LOAD/Z gnd gnds vdd vdds HS65_GS_IVX27
XICC_PRECTS_140_LOAD n178 ICC_PRECTS_140_LOAD/Z gnd gnds vdd vdds HS65_GS_CNIVX21
XU2724_LOAD APV_GROUNDED:A xcellx118976xnet77012 U2724_LOAD/Z gnd gnds
+vdd vdds HS65_GS_NOR2X19
XU2780_LOAD APV_GROUNDED:A net72529 U2780_LOAD/Z gnd gnds vdd vdds HS65_GS_NAND2AX7

.ENDS
```

4.4.3 opcond#.spi

Come già accennato, ogni file di questo tipo sarà incluso nel suo rispettivo *spiceFinal*. Esso è diviso in quattro sezioni, di seguito descritte e accompagnate dalla rispettiva porzione di un file di esempio.

La **prima parte** è dedicata all'istanziazione dei generatori di clock. Possono essere fino al doppio del numero di path, ma molto spesso ciascun path è mono-clock e più path condividono la stessa radice di clock (*clock-root*). E' stata scelta la definizione compatta di ***pulse generator*** che si presta ad una rapida modifica da parte del designer. L'istanziazione della eventuale cella pre-driver non viene fatta in questo file ma nello spiceFinal.

```
*** Parameters
.param tdelay_ck=2.500000ns
.param slope_ck=10ps

*** Clock(s): pre-driver (if needed) and pulse generator
* CLOCK INTDIV_OUT 400 MHz
XPRE_INTDIV_OUT PRE_INTDIV_OUT INTDIV_OUT gnd gnds vdd vdds HS65_GH_BFX44
VINTDIV_OUT PRE_INTDIV_OUT 0 PULSE(0 supply
+ tdelay_ck slope_ck slope_ck {0.6ns-slope_ck} 2.500000ns)
```

La **seconda parte** ha l'importante compito di ospitare le istanziazioni dei moduli Verilog-A dedicati alla generazione del dato allo startpoint (FF, latch o porta) del path. Ancora una volta è stata curata la facilità di individuazione del componente all'interno della netlist e la sua modifica. Il primo compito è agevolato, come al solito, dall'inserimento di commenti nel file che riportano il nome del path e altre informazioni utili, mentre la modifica è resa intuitiva dalla struttura dei moduli Verilog-A che permette la modifica di uno o più parametri sia puntualmente che a livello globale. Un esempio potrebbe essere quello di modificare il tempo in cui viene fornito il dato ad un latch per valutare in diverse situazioni se il timing del path è violato o meno.

```
*** Data stimuli for path "path5"
XDATA_path5 XFREQ_CORR_D_regx0x.D XFREQ_CORR_D_regx0x.CP
+ apvdata vref=supply vout=supply rise=1 delay=1.55e-09
*VFREQ_CORR_D_regx0x/D FREQ_CORR_D_regx0x/D 0 PWL(0n 0
*+ {tdelay_ck+1.55ns} 0 {tdelay_ck+1.55ns+slope} supply 40.514107ns supply)

*** Data stimuli for path "path9"
XDATA_path9 XFREQ_CORR_D_regx3x.D XFREQ_CORR_D_regx3x.CP
+ apvdata vref=supply vout=supply rise=1 delay=1.55e-09
*VFREQ_CORR_D_regx3x/D FREQ_CORR_D_regx3x/D 0 PWL(0n 0
*+ {tdelay_ck+1.55ns} 0 {tdelay_ck+1.55ns+slope} supply 40.514107ns supply)
*** Data stimuli for path "path10"
*** APV_WRNG: Skip duplicated stimuli for "FREQ_CORR_D_regx3x/D".
```

La **terza parte** contiene tutti i generatori di tensione in DC necessari a sensitizzare i cosiddetti *side-pins*, ossia i pin flottanti delle celle che devono lasciar passare un dato o un clock. Questo tipo di informazione viene catturata tramite il parsing dei ***deck files*** fatti scrivere a PrimeTime nella precedente fase del flusso. Questa scelta segue l'impostazione generale data al flusso, ossia l'ottimizzazione delle risorse a disposizione. Inizialmente si procedeva al parsing dei *liberty files* (si veda l'Appendice A: SPICE) per catturare le funzioni logiche di ciascuna cella, una per ogni sua uscita. Bisognava poi trovare le condizioni degli n-1 ingressi flottanti in grado di far passare il dato dall'ingresso all'uscita d'interesse. Infine era necessario trovare la condizione pessima fra tutte quelle possibili. Soprattutto per via di quest'ultimo compito, l'algoritmo era diventato molto complesso e richiedeva molto consumo di memoria oltre a un discreto tempo di esecuzione. La soluzione ad oggi utilizzata sfrutta, come accennato, i *deck files*, fatti generare a comando a PrimeTime. Si veda l'Appendice A: SPICE per un esempio di uno di questi file, che non sono altro che delle piccole netlist SPICE riguardanti un singolo path. La simulazione dei *deck* richiederebbe delle modifiche da parte di un designer esperto in simulazioni analogiche, senza contare il fatto che sono file molto ridondanti per costruzione quindi richiederebbero lunghi tempi di simulazione. Il loro punto di forza è che contengono le informazioni complete sulla sensitizzazione dei *side-pins* in quanto sfruttano il database di PrimeTime a valle dell'obbligatoria acquisizione dei *liberty files*. A questo punto il parsing di tanti piccoli file quanti sono i path per ricavare solo il valore di tensione di ciascun pin è gratuito e immediato. La vecchia struttura complessa è stata comunque mantenuta per

l'esigenza di rendere il tool nel prossimo futuro il più possibile svincolato da un solo *vendor* di STA (nella fattispecie PrimeTime).

```
*** Pin Sensitization Voltages
VAPV_PWR_supply APV_PWR_supply 0 supply
VAPV_PWR_0 APV_PWR_0 0 0
VAPV_GROUNDED:RN APV_GROUNDED:RN 0 0
...
```

La **quarta ed ultima parte** è esclusivamente usata per istanziare i vari moduli Verilog-A necessari alla modellizzazione del crosstalk, che saranno meglio descritti nella sezione 4.6 - Modellizzazione del Crosstalk.

```
*** Aggressors (inst_name supply victim_net aggressor_net aggressor_cell [params ...])
XAGG1 vdd n180 FREQ_SU apvaggressor
XAGG2 vdd FREQ_CORR[3] n566 apvaggressor
XAGG3 vdd n9198 n47 apvaggressor slope=15e-12
XAGG4 vdd n9128 n38 apvaggressor slope=27e-12 inv=0
```

4.4.4 probe#.spi

Ogni probe file, relativo al proprio sottocircuito contiene tre tipi di *statements*: i **.measure**, i **.defwave** e i **.probe**.

I **.measure** servono a catturare tutte le informazioni temporali utili al calcolo degli slack. Sono state implementate anche misure di slew e delay per individuare possibili errori da indicare al designer, come ad esempio rise/fall time troppo lunghi che indicano celle sottodimensionate.

Più in dettaglio:

- ***data_arrival*** – E' la differenza fra il tempo di arrivo del dato all'*endpoint* e il tempo del fronte di riferimento del clock sulla clock-root. Concettualmente è identico alla definizione usata da PrimeTime.

- ***data_capture*** – E' l'istante temporale in cui arriva il fronte di cattura del dato. E' come il *required time* di PrimeTime tranne per il fatto che non contiene ancora il setup/hold time.(e nemmeno gli altri termini conservativi della STA).
- ***sp_clk_delay*** – E' il ritardo fra la clock-root e il pin di clock del primo Flip-Flop (lo *startpoint*) e coincide al *clock network delay* di PrimeTime. E' parte del *data_arrival*.
- ***ep_clk_delay*** – E' il ritardo sul ramo di cattura della clock-tree, quindi il ritardo accumulato dalla clock-root fino al pin del secondo FF.
- ***slew#*** – E' il rise/fall time relativo a un punto in mezzo al path o alla clock-tree.
- ***delay#*** – E' il ritardo di propagazione attraverso le celle del path (e della sua clock-tree) oppure attraverso le net che collegano le celle.

Il codice seguente mostra le quattro principali misure per un path.

```
*** Measures on path "path5"

.measure path5_data_arrival
+trig v(INTDIV_OUT) val = 0.36 td = 4.05ns rise = 1
+targ v(XDELTA_CEIL_SD_OUT_N_regx2x.D) val = 0.54 td = 0ns fall = 1

.measure path5_data_capture
+trig v(INTDIV_OUT) val = 0.36 td = 4.05ns rise = 1
+targ v(XDELTA_CEIL_SD_OUT_N_regx2x.CP) val = 0.36 td = 0ns rise = 3

.measure path5_sp_clk_delay
+trig v(INTDIV_OUT) val = 0.36 td = 0ns rise = 1
+targ v(XFREQ_CORR_D_regx2x.CP) val = 0.36 td = 0ns rise = 1

.measure path5_ep_clk__slew
+trig v(XFREQ_CORR_D_regx2x.CP) val = 0.18 td = 0ns rise = 3
+targ v(XFREQ_CORR_D_regx2x.CP) val = 0.72 td = 0ns rise = 3
...
```

Il costrutto **.measure** richiede un trigger e un target oltre ad altri parametri necessari e opzionali. Nella prima misura, ad esempio, è stata usata come trigger la tensione sulla net INTDIV_OUT (la clock-root). Si inizia a contare quando questa tensione supera il valore (*val*) 0.36 nel primo fronte di salita (*rise=1*) che occorre non prima di un certo ritardo td pari a 4.05ns.

Il target è il primo fronte di discesa della tensione sul pin CP dell'istanza XDELTA_CEIL_SD_OUT_N_regx2x.

I **.defwave** servono per semplificare molto il lavoro del progettista che vuole analizzare i path nel visualizzatore di forme d'onda. Come mostra il codice seguente si creano delle etichette per le forme d'onda dei punti caratteristici del path, ossia *data_pin*, *start_clock_pin*, *data_arrival_pin*, *end_clock_pin* e *clock_root*. Non è necessario ricordare nessuna delle net del path, infatti, ciascuna etichetta contiene anche il nome del path, permettendo di "plottare" tutte le forme d'onda di un path in maniera estremamente semplice.
Gli annessi **.probe** servono per esplicitare al simulatore di includere questi segnali all'interno del database delle forme d'onda.

```
*** Waveforms for path "path5"

.defwave path5_data_pin_1 = v(FREQ_CORR_D_regx2x/D)
.probe tran w(path5_data_pin_1)

.defwave path5_start_clock_pin = v(XFREQ_CORR_D_regx2x.CP)
.probe tran w(path5_start_clock_pin)

.defwave path5_data_arrival_pin = v(XDELTA_CEIL_SD_OUT_N_regx2x.D)
.probe tran w(path5_data_arrival_pin)
...
```

Ad integrazione delle misure e dei defwave automatici descritti fino ad ora, il framework *advUtils* che contiene il tool *apvEasy* permetterà al designer di introdurre misure e monitor personalizzabili. Questa *feature* è per ora in fase di sviluppo, ma è possibile vedere un esempio di monitor Verilog-A nell'Appendice C: Verilog-A.

4.4.5 DSPF#.dspf e ptReducedSPEF#.spef

Lo SPEF è il file di parassiti fatto generare a PrimeTime (uno per ogni sottocircuito della netlist SPICE) contenente le informazioni necessarie alle sole net estratte per l'analisi.
Ciascun DSPF viene generato a partire dal rispettivo SPEF. Il cambio di formato è quasi gratuito nel senso che viene fatto durante l'obbligatorio parsing dello SPEF, necessario al fine di memorizzare le informazioni sulla connettività delle celle da istanziare. La scelta del formato DSPF è dettata dal fatto che è lo standard per i simulatori analogici. Il formato preferito dai simulatori statici è invece lo SPEF, in quanto è ottimizzato per grossi design dal punto di vista dell'occupazione di memoria. Per i dettagli si veda l'Appendice B: SPEF/DSPF.
Bisogna specificare che i parassiti caricati nella sessione di PrimeTime sono quelli relativi al corner tecnologico desiderato per cui lo SPEF e il DSPF derivati si riferiscono al medesimo corner.

4.4.6 dumpLibrary.spi

Questo file raccoglie semplicemente tutte e sole le celle prese dalle varie librerie della tecnologia utilizzata. I file di libreria contengono le definizioni dei sottocircuiti, ossia la descrizione di ogni singola cella a livello transistor, usate per la simulazione analogica.
E' comodo avere un file unico, e con la dimensione strettamente necessaria, da portare assieme alla netlist senza doversi più preoccupare di link alle librerie.

4.4.7 corners.spi

Questo file contiene semplicemente una lista dei file delle models delle celle utilizzate nel design, ossia i corners tecnologici per tenere in conto delle condizioni operative del circuito (si riveda la sezione 3.2.1 - Corners tecnologici).
Visto che il file è necessario anche per la STA il designer deve solo copiarlo nella directory in cui è presente il file Top.cir della netlist da analizzare.

4.4.8 Altri files

Riguardando la Figura 27 ci si accorge che rimangono da descrivere i seguenti file:

- **Drivers.va** – contiene il codice Verilog-A che descrive i moduli per la sensitizzazione dei path e la modellizzazione del crosstalk.
- **runxa & xa_cmd** – contengono i comandi necessari per lanciare il simulatore CustomSim-XA, come ad esempio il livello di accuratezza e la directory di uscita impostati dall'utente. Due file del tutto simili saranno presenti per gli altri tipi di simulatori analogici.
- **APVptTable.csv & APVsimTable.csv** – sono i due file in formato CSV (*Comma-Separated Values*) che riempiono la tabella dei risultati dell'interfaccia grafica subito dopo la *Static Timing Analysis* e subito dopo l'importazione dei risultati della simulazione analogica, rispettivamente.
- **APV.log** – contiene le informazioni su tutte le operazioni compiute, sulle caratteristiche della netlist generata e sugli eventuali errori ottenuti.

La sottocartella dedicata ai risultati della simulazione analogica, ovviamente, si riempirà solo nella rispettiva fase del flusso, tuttavia si riportano qui di seguito le descrizioni per i file principali:

- **APV.fsdb** – è il file binario in formato fsdb, in pratica un database compresso, contenente tutte le informazioni sulle forme d'onda che è possibile visualizzare tramite un tool apposito. In realtà, nel caso sia presente anche un'analisi di crosstalk, i file di questo tipo sono due, con e senza l'abilitazione degli aggressori. Questo permette una più facile comparazione fra le forme d'onda.
- **xa.meas** – contiene i risultati dei *.measure* e ne possono esistere due, derivanti dalle simulazioni con e senza crosstalk. Si veda la sezione 4.5.2 - Parsing dei .measure.
- **xa.log** – è il file di log principale del simulatore analogico (in questo caso CustomSim-XA). Contiene informazioni dettagliate, *warnings* e errori riguardanti tutta la simulazione.
- **APV.valog** – è il file di log del simulatore analogico in merito alla sola parte di codice Verilog-A.

4.5 Analisi e Back-Annotazione dei Risultati

La netlist SPICE generata fino ad ora è pronta per essere simulata. Il designer ora ha tre possibili scenari davanti a se:

- lanciare la simulazione e proseguire con il flusso APV;
- modificare la netlist prima di continuare (ad esempio, si veda la sezione 4.7.3 - Simulazioni AMS);
- usare la netlist come testbench a proprio piacimento, ad esempio per fare analisi Monte Carlo, interrompendo il flusso APV.

Se lo scopo è quello di eseguire fino in fondo il flusso APV, il designer deve solo settare il simulatore analogico preferito e lanciare la simulazione tramite l'apposito tasto dell'interfaccia grafica.

4.5.1 Simulazione dinamica iterativa

La simulazione analogica è regolata dagli *statement* contenuti nel file Top.cir, riportati qui di seguito.

```
*** SIMULATION PROPERTIES
.tran 0.001ns 10.5ns
.temp 125.0

.verilog drivers.va
*** CROSSTALK
.param aggr_enable = 0
.subckt apvaggressor vref in out inv=1 slope=20ps enable=aggr_enable
xagg vref in out aggressor inv=inv slope=slope enable=enable
.ends
.alter
.param aggr_enable = 1
```

L'analisi è di tipo transitorio e dura circa 10.5ns nell'esempio riportato. La temperatura operativa è 125°C. L'istruzione *.verilog* serve a includere il codice dei moduli descritti in linguaggio Verilog-A: per il settaggio del dato all'ingresso dei path, per la modellizzazione del crosstalk e per i monitor personalizzati.

La seconda porzione si ha in caso si debba valutare anche il crosstalk. Si nota un primo settaggio del parametro che abilita i moduli per il crosstalk, la definizione esplicita del sottocircuito che li descrive, un'istruzione *.alter* e un secondo settaggio di *aggr_enable*. Questa struttura è stata appositamente costruita per fare in modo di lanciare automaticamente e, una dopo l'altra, due simulazioni distinte. La seconda simulazione è uguale alla prima se non per il fatto che ha i moduli che modellizzano il crosstalk abilitati.

Fino a questo punto la simulazione descritta è abbastanza standard, il *.alter* fa solo in modo di eseguire due simulazioni indipendenti in cascata. In realtà le simulazioni non sono del tutto indipendenti, i parametri della netlist SPICE, così come quelli definiti nei moduli Verilog-A, sono ereditati dalla simulazione relativa ad un *.alter* successivo. Questa possibilità ha permesso lo sviluppo di una nuova idea che ha le potenzialità di soddisfare alcune esigenze nate in fase di sviluppo del flusso APV. L'idea consiste nell'implementare una sorta di **simulazione iterativa**, controllata da alcuni parametri, attraverso la quale è possibile ottenere tre differenti livelli di vantaggi (con complessità crescente):

- semplice scambio di informazioni temporali fra una simulazione e la successiva (ad esempio per allungare il tempo di simulazione qualora non fosse sufficiente);
- cambio delle condizioni del path per fare più di una analisi contemporaneamente o risolvere conflitti fra path che richiedono sensitizzazioni differenti;
- complessa gestione di un *testbench* per l'analisi di path o *scan-chain*, con monitor Verilog-A che controllano l'andamento della simulazione e il passaggio a iterazioni successive sulla base del risultato dell'attività di monitoraggio che svolgono.

Il secondo livello è ancora in fase di test, mentre l'ultimo sarà ingegnerizzato in futuro.

4.5.2 Parsing dei .measure

Finita la simulazione, il tasto "*Results*" della GUI permette di catturare i risultati delle misure temporali e caricarle direttamente nella tabella dei risultati, integrando i dati della *Static Timing Analysis*.

Di seguito è riportato un esempio di file con estensione *.meas* generato dalla simulazione con CustomSim-XA e contenente i risultati dei *.measure*.

```
* CustomSim RHEL64 G-2012.06  (built 01:39:43 May 27 2012)
* Build id: 2346010
* Copyright (C) 2012 Synopsys Inc. All rights reserved.

xidut_slack_1.path1_data_arrival = 4.3406333e-10  targ = 5.4782458e-09  trig = 5.0441824e-09
xidut_slack_1.path1_data_capture = 6.7002266e-10  targ = 5.7142051e-09  trig = 5.0441824e-09
xidut_slack_1.path1_sp_clk_delay = 8.8629821e-11  targ = 5.1328122e-09  trig = 5.0441824e-09
...
xidut_slack_1.path8_data_arrival = 1.8653522e-09  targ = 6.9095346e-09  trig = 5.0441824e-09
xidut_slack_1.path8_data_capture = 2.5862560e-09  targ = 7.6304385e-09  trig = 5.0441824e-09
...
temper           =  1.2500000e+02
alter#           =  1.0000000e+00
```

La procedura che legge questi risultati è molto semplice: cattura il nome del path e il tipo di misura associandoli al primo valore della tripletta, che è la differenza fra gli altri due valori.

Successivamente le misure vengono usate per calcolare gli slack, includendo il vincolo di setup/hold e la *clock uncertainty*, in maniera simile ai *timing report* di PrimeTime mostrati nel Capitolo 3.

Va precisato che la definizione di slack della STA non è sempre direttamente applicabile nel caso di simulazioni analogiche quindi di DTA.

Ad esempio, i termini di prestito, in caso siano presenti dei Latch, devono essere considerati? Se si decide di non considerarli, supponendo che il dato in ingresso arrivi all'inizio della fase trasparente si è troppo ottimistici; viceversa, se il dato arriva poco prima che il Latch diventi opaco, si è troppo pessimistici. D'altra parte, se si decide di considerarli, si sta usando un dato statico con delle misure dinamiche, perdendo in accuratezza. In questo caso si permette all'utente la facoltà di includere o meno i *latch borrowed times*.

Un altro caso è quello dei *constraint* sulle porte ingresso/uscita del design [2]. La porzione esterna al design digitale che si occupa di lanciare o catturare il dato contiene

sostanzialmente un registro. Sarebbe consigliabile, nell'ambito di una simulazione analogica, includere anche queste porzioni per dati coerenti e più precisi.

Queste sottigliezze dipendono dal punto di vista del designer e dal modo in cui ha definito inizialmente i *constraints*. La cosa importante è sapere come sono calcolati gli slack per capire cosa rappresenta il suo valore numerico.

4.6 Modellizzazione del Crosstalk

Nel precedente paragrafo la modellizzazione del crosstalk è stata definita pessimistica. E' necessario però entrare più nel dettaglio per permettere di capire i vantaggi e i limiti del Flusso APV in merito ai dati delle simulazioni analogiche.

L'approccio scelto cerca, ancora una volta, di sfruttare i vantaggi di entrambi i tipi di simulazioni. Date le profonde differenze concettuali fra analisi statiche e dinamiche non è sempre possibile trovare un metodo ottimale, sono state fatte perciò alcune scelte di compromesso. Il caso ideale sarebbe quello di fornire un risultato della simulazione analogica coincidente con le misure su silicio (al variare dei corners tecnologici). Questo è possibile, ma vale solo per una specifica configurazione del circuito. Le variabili in gioco sono molteplici per cui le configurazioni sono in numero spropositato, basta pensare che il carico capacitivo visto all'ingresso di una porta logica dipende, oltre che dal valore corrente di tensione sul pin stesso, anche dalle tensioni sugli altri ingressi e dal carico che le sue uscite pilotano. Il problema delle configurazioni del circuito evidenziato è detto ***coverage***. Come già evidenziato, l'analisi dinamica è intrinsecamente latente in termini di coverage, mentre una STA riesce a raggiungere anche il 100%.

Si è scelto allora di impostare la modellizzazione e la simulazione del crosstalk in modo realistico quando possibile ed essere pessimistici al massimo in caso contrario [14][15].

Nella sezione 4.6.2 sono descritti i vari casi, ma prima verrà mostrato come vengono catturate e trattate le informazioni sugli aggressori. L'ultima parte, invece, evidenzierà brevemente le differenze in caso di analisi di clock-tree.

4.6.1 Collezione delle informazioni sugli aggressori

Per catturare tutte le info si parte dai punti del path, e in maniera del tutto analoga dai punti delle clock-tree di lancio e di cattura. Ciascun punto è un oggetto nel database di PrimeTime e contiene vari attributi, tra cui ***aggressors*** e ***effective_aggressors***. La differenza sostanziale fra i due è che gli aggressori effettivi sono un sottoinsieme di tutti i potenziali aggressori filtrati seguendo i parametri definiti dall'utente. Si riveda la sezione 3.3 - Signal Integrity per i dettagli. Ciascun aggressore effettivo catturato viene poi ad essere interrogato da apposite procedure mirate a catturare le slope e il valore

dell'alimentazione del rispettivo driver, e i carichi capacitivi connessi lungo la net. Come è possibile vedere dal codice riportato in seguito gli oggetti di PrimeTime vengono interrogati spesso con l'istruzione *get_attribute* come fatto per le informazioni sui path.

```
foreach_in_collection pp $path_points {
  set pointobj [get_attribute $pp object]
  set pointname [get_object_name $pointobj]
  set pinobj [get_pins -quiet $pointname]
  if {$pinobj == {}} continue
  set net [get_object_name [get_nets -quiet -segments -top_net_of_hierarchical_group \
                                     -of_object $pinobj]]
  set aggs [get_attribute [get_nets "$net"] effective_aggressors]
  foreach AGG $aggs {
    lappend AGGRESSORS($path_name:$AGG) $pointname
    set AGG_INFO($AGG) [apvGetAggInfo $AGG]
  }
}

# slopes, supply and loads
proc apvGetAggInfo {net} {
   set INFO {}
   set drivers [ptpPin2inst [ptpNet2pins $net out]]
   foreach d $drivers {
      set lib_cell [get_lib_cells -of_objects [get_cells $d]
      ...
   }
   return $INFO
}
```

La seconda porzione di codice invece riporta la descrizione in linguaggio Verilog-A del modulo ***aggressor***, le cui istanziazioni sono usate per modellizzare, quando è necessario, le transizioni di aggressione. Si veda l'Appendice C: Verilog-A.

Il modulo ha tre porte elettriche: la *vmax* per l'alimentazione (in modo da seguire le variazioni dei corners tecnologici), *in* per monitorare la net vittima e *out* per pilotare l'aggressore. Tre parametri consentono di modificare il suo comportamento: l'*enable* consente di disabilitare il modulo fornendo una bassa impedenza d'uscita, l'*inv* permette di gestire il segno della transizione rispetto a quella sulla net vittima e, infine, la *slope*, che può essere settata dal tool o dal designer.

```
module aggressor(vmax, in, out);
  input vmax, in;
  output out;
  electrical vmax, in, out;

  parameter slope=20e-12;
  parameter inv=1, enable=0;
  integer mode;
  real ivin, ivout, slpk, vth1, vth2, tr, tf;

  analog begin
      if (enable != 0) begin
          if (inv != 0) ivin = V(vmax, in); else ivin = V(in);
      end
      @(initial_step) begin
          slpk = V(vmax)/slope;
          vth1 = V(vmax)*0.1;
          vth2 = V(vmax)*0.9;
          if (enable != 0) mode = 0; else mode = 3;
      end
      case (mode)
          0: begin
              ivout = ivin;
          end
          1: begin
              ivout = max(vth2 - slpk*($abstime -tf), 0.001);
              if (ivin < 0.001) mode = 0;
          end
          2: begin
              ivout = min(vth1 + slpk*($abstime -tr), V(vmax) -0.001);
              if (V(vmax) - ivin < 0.001) mode = 0;
          end
          3: begin
              ivout = 0;
      endcase
      @(cross(ivin-vth2,-1)) begin
          mode = 1;
          tf = $abstime;
      end
      @(cross(ivin-vth1,+1)) begin
          mode = 2;
          tr = $abstime;
      end
      V(out) <+ ivout;
  end
endmodule
```

Sostanzialmente è implementata una piccola macchina a stati che monitora la net vittima e produce una transizione sull'aggressore allineata temporalmente. Il segno della transizione generata sarà opposto o uguale a quella della vittima a seconda che la si voglia rallentare o accelerare, rispettivamente [15].

4.6.2 Necessità di introdurre pessimismo

Aggressori esterni di net del path

Siamo nel caso in cui una net che non fa parte dell'insieme sotto verifica è un aggressore di una net di un path. In questa condizione è necessario l'uso di un modulo Verilog-A per ogni aggressore della net del path in questione, come è possibile vedere nell'esempio in Figura 28 in cui c'è accoppiamento capacitivo con due net.

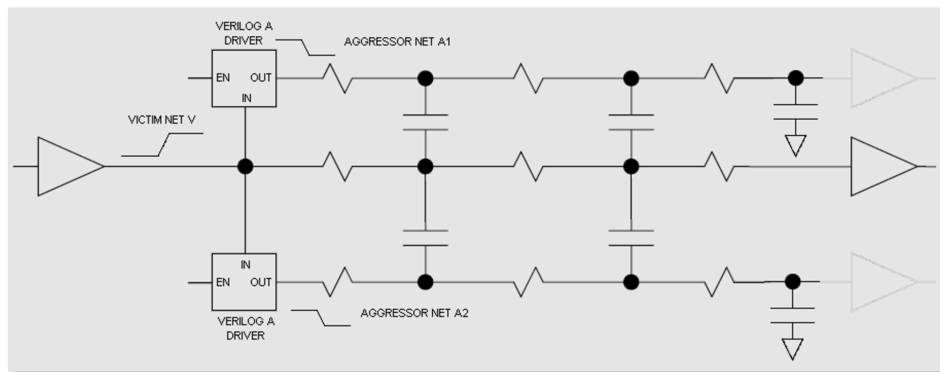


Figura 28. Esempio di connessione di due moduli Verilog-A per modellizzare il crosstalk

Alcuni aspetti sono degni di nota.

Tutti gli aggressori si allineano alla transizione vittima per avere il caso pessimo, ossia ritardare il più possibile la propagazione del dato in caso di analisi di setup e anticiparla per quella di hold.

La ripidezza di ciascun fronte viene selezionata con cura fra tutti i casi possibili per la specifica cella che pilota la net e che viene da noi sostituita con il modulo Verilog-A. Ad esempio, per un'analisi di setup (hold) viene selezionata la massima pendenza fra le

transizioni con segno opposto (concorde) alla transizione vittima. Il pessimismo introdotto può essere importante ma è solo quello strettamente necessario. Si pensi, infatti, alla situazione in cui un inverter asimmetrico abbia *rise time* (fra il 10% e il 90% della dinamica) di 10ps e *fall time* di 15ps, e la net che pilota aggredisca la net di un path. Se la vittima ha una transizione da basso a alto, durante un'analisi di setup verrà generato un fronte di discesa con durata 15ps e non 10ps. L'unico pessimismo introdotto è dunque dovuto all'allineamento temporale.

Quando disabilitati, i moduli Verilog-A mostrano una bassa impedenza rendendo la rete RC aggressori-vittima più realistica.

Le celle di carico delle net che fanno da aggressori sono sostituite da capacità equivalenti verso massa in quanto la perdita di accuratezza non è apprezzabile mentre il guadagno in termini di velocità di simulazione è consistente.

Aggressori da path a path

Nel caso in cui una net di un path è aggredita da una net appartenente ad un path anch'esso sotto analisi ci sono alcune considerazioni da fare.

Innanzitutto, se PrimeTime non ha scartato questo aggressore significa che è la situazione è logicamente possibile e l'aggressore potrebbe essere in grado di allinearsi con la vittima disturbandola.

Le condizioni imposte sul path contenente l'aggressore potrebbero portare ad un crosstalk troppo ottimistico per cui è stata fatta la scelta di duplicare l'aggressore creando una net fittizia per evitare conflitti.

Si ricade dunque nella situazione precedente di aggressori esterni e occorre istanziare gli opportuni moduli Verilog-A.

Aggressori da clock a net del path

Condizione in cui il dato che viaggia sul path può essere ritardato o accelerato in maniera indipendente dal tipo di analisi effettuata. L'aggressione avviene da parte di una net della clock-tree per cui non è necessario alcun modulo Verilog-A.

Aggressori da clock a clock

In questo caso si è nella condizione più favorevole in quanto le transizioni degli aggressori sono quelle reali. L'accuratezza è dunque massima e pari a quella del tool di estrazione dei parassiti da layout. Non necessita chiaramente di alcun modulo Verilog-A.

Aggressori esterni di net del clock

Il pessimismo massimo per ogni path si otterrebbe con l'allineamento sfavorevole di tutti gli aggressori "esterni" sia nella *launch-clock* che nella *capture-clock*. Ciò non è possibile per due motivi, entrambi inerenti la topologia della clock-tree. La Figura 29 mostra il caso frequente in cui clock di cattura e di lancio sono generati dalla stessa clock-root, hanno un percorso in comune e poi si diramano. In questo caso, la scelta del segno dei generatori di crosstalk nel tratto comune può essere arbitraria in quanto non comporta alcuna differenza in termini di slack.

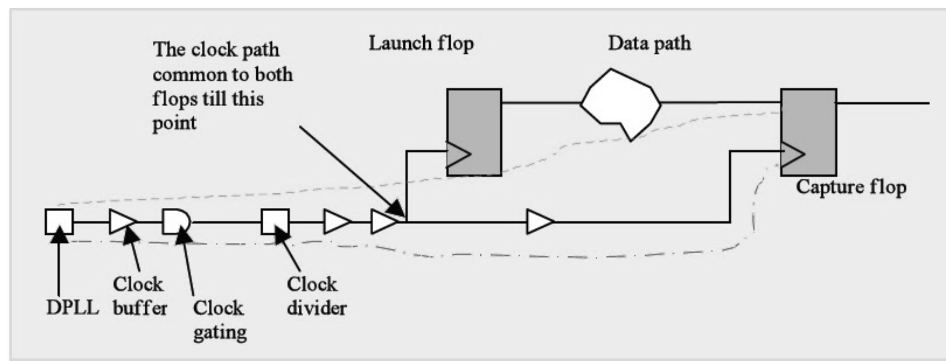


Figura 29. Clock reconvergence pessimism removal (CRPR)

Il secondo motivo è che un percorso di lancio di un path può avere una grossa parte in comune con il percorso di cattura di un altro path. La condizione pessima per il primo path sarebbe quella ottima per il secondo e viceversa. Questo è ovviamente intollerabile ed ha richiesto una soluzione. Per il momento la soluzione implementata prevede la separazione dei due path in due sottocircuiti separati con conseguente duplicazione di parte della clock-tree. Questo è stato necessario per proseguire con il supporto ai clienti, ma è già in fase di test una soluzione ottimizzata che prevede l'uso di una simulazione intelligente che a runtime modifica gli aggressori e cattura i ritardi necessari nei due casi possibili.

4.6.3 Aggressori in caso di clock-tree analysis

Considerando un circuito con un'unica clock-tree è immediato dedurre che essa aggredisce se stessa, quindi non c'è bisogno di modellizzare il crosstalk. Una piccola complicazione è l'aggressione da parte delle net dei path. Generalmente questo tipo di aggressione è poco probabile e poco determinante, ma se ne tiene conto lo stesso. Per ora i moduli Verilog-A sono istanziati solo in modo da ritardare i fronti, ma è in fase di implementazione anche il caso opposto in modo da fornire sia un limite inferiore che uno superiore. In questo modo è immediato valutare se questi aggressori sono importanti o meno e fare ulteriori analisi.

In caso di circuito con più clock-tree, visto che è possibile analizzarle una alla volta, si può scegliere se fare un'analisi con o senza crosstalk. Il caso più interessante è ovviamente l'analisi comparativa con/senza crosstalk.

4.7 Deviazioni dal Flusso APV

La semplicità dell'idea alla base del flusso APV di estrarre una netlist SPICE da un circuito digitale e l'impostazione generale data al flusso stesso e alle procedure che lo implementano a livello software permettono una buona flessibilità di utilizzo del tool. I tre esempi più significativi verranno riportati in questo paragrafo.

4.7.1 Clock-tree analysis

Il tool APV non si propone come analizzatore di clock-tree, esistono indubbiamente tool più adatti allo scopo, in primis i tool dedicati. Questi ultimi guidano il designer in tutte le fasi della CTS (*Clock-Tree Synthesis*) sia in pre che in post-layout. Ciò non toglie il fatto che APV possa essere un valido aiuto per una veloce analisi di Power e/o di EMI (*Electro-Magnetic Interference*) di una clock-tree o di diverse implementazioni della stessa.
Il flusso è quasi uguale a quello principale per cui ritengo opportuno ripercorrerlo brevemente mediante un *Case Study* nel prossimo capitolo (sezione 5.2 - Analisi della clock-tree: Power/EMI) evidenziandone le differenze da un punto di vista più pratico.

4.7.2 Simulazioni manuali: NetFlow

Come accennato nella sezione 4.1.3 - Tipi di analisi disponibili, questo flusso aggiunge flessibilità al tool permettendo di selezionare una porzione di circuito a partire da una lista di net, prescindendo dal concetto di path. La Figura 30 mostra uno schema del funzionamento del NetFlow. Le parti in viola sono i compiti che l'utente deve svolgere, in verde sono indicati i file di parassiti in ingresso e uscita da apvEasy mentre tutto ciò che ha colore rosa indica la parte di simulazione.

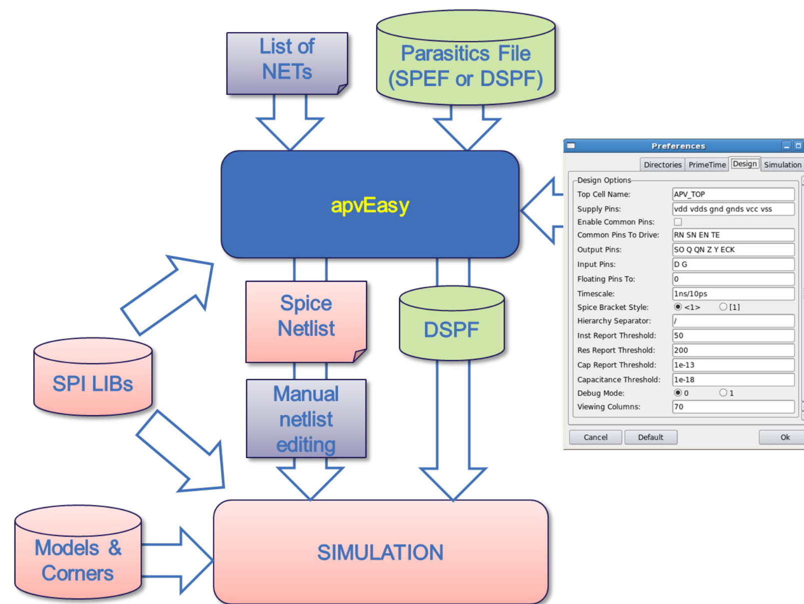


Figura 30. Schema di funzionamento del NetFlow

4.7.3 Simulazioni AMS

Una particolare richiesta mi è stata spesso rivolta durante le svariate occasioni in cui mi sono trovato a dover pubblicizzare il tool e il flusso APV e a formare le diverse figure professionali interne all'STMicroelectronics che ne fanno uso. Si tratta di poter utilizzare il tool per analizzare particolari path problematici al contorno di grosse celle di memoria o di altri circuiti analogici quali MEMS o *motor drivers* per HDD. Questi blocchi analogici possono essere descritti sia al *gate level* che a livello comportamentale. Nel primo caso la simulazione sarà puramente analogica, mentre nel secondo sarà necessario effettuare una simulazione mixed-signal (si veda la sezione 2.1 - Tipi di simulazione).

Le modifiche da apportare alla netlist SPICE sono più o meno estese a seconda della complessità della parte analogica. In linea di massima è necessario includere al top level (nel file Top.cir) il file con la descrizione del circuito analogico nel formato voluto. Per riferimento chiameremo questo file ***analogBlock.spi***. E' inoltre necessario promuovere al top level tutte quelle net del circuito digitale che devono interfacciarsi con la parte analogica. Questa operazione può essere compiuta facilmente modificando le definizioni e le istanziazioni dei sottocircuiti interessati. A tal proposito si veda l'Appendice A: SPICE.

Data la grossa richiesta in questa direzione e i numerosi vantaggi già ottenuti con la prima versione del tool APV, molto lavoro sarà dedicato in futuro all'ingegnerizzazione di una procedura automatica per l'importazione di blocchi analogici.

5 Case studies

Durante lo sviluppo del flusso APV sono stati utilizzati svariati circuiti reali, forniti dai gruppi di *Digital Design* di Agrate Brianza, Castelletto e Catania. I design sono serviti sia come *testcase* per validare l'algoritmo di creazione della netlist SPICE, ma soprattutto per verificare che i dati delle simulazioni con la modellizzazione del crosstalk fossero consistenti sul silicio. E' doveroso precisare questa ultima affermazione. La verifica su silicio non è stata fatta con lo scopo di validare i modelli di crosstalk generati dal tool APV, seguendo un flusso di "caratterizzazione sperimentale". Sono stati invece presi i dati delle caratterizzazioni sul silicio nei vari corners tecnologici per controllare se l'informazione binaria "path violato/non violato" fosse corretta. In seguito sono state simulate alcune configurazioni del circuito sotto esame per valutare se i ritardi globali con e senza crosstalk fossero in un certo range di tolleranza definito dal progettista del circuito.

Per brevità, saranno riportati in questa Tesi solo due fra gli esempi più significativi, selezionati per dare rilievo ad alcuni aspetti pratici del flusso.
Vista l'importanza data al caso in cui è necessaria l'integrazione di blocchi analogici nella sezione 4.7.3 - Simulazioni AMS, questo caso non sarà ulteriormente trattato.
Non sarà esplicitamente trattato neanche il caso in cui il designer voglia migrare il suo intero progetto verso una tecnologia scalata o comunque diversa da quella iniziale. Questo perché generalmente, nel caso in cui si voglia passare ad una tecnologia meno performante (caso più unico che raro) il circuito avrà violazioni di setup, mentre nel caso frequente in cui la tecnologia di arrivo è più avanzata il circuito avrà maggiormente violazioni di hold.

Il flusso APV non subisce dunque modifiche se non nei suoi file di input, più precisamente il designer cambierà le librerie tecnologiche.

Il primo *Case Study* è un circuito costituito da un ***Phase Locked Loop*** (PLL) e altra circuiteria in tecnologia 65nm, non certo l’avanguardia del processo CMOS, ma ostico per costruzione dal punto di vista della topologia della netlist.
Il secondo, un **microprocessore**, è stato principalmente usato per l’analisi della clock-tree. Anch’esso in tecnologia 65nm, ha permesso di avere tempi di simulazione non troppo lunghi durante lo sviluppo e il debugging del tool APV.

5.1 Verifica di “piccole” violazioni di setup

Le difficoltà di questo circuito derivano principalmente dalle caratteristiche del suo PLL. La Figura 31 mostra lo schema a blocchi del PLL. Si notino i “classici” blocchi dell’anello del secondo o terzo ordine a pompa di carica: il PFD (*Phase Frequency Detector*), la *Charge Pump*, il filtro d’anello (LF) e il VCO (*Voltage Controlled Oscillator*). Quest’ultimo è in grado di generare otto fasi del clock. Infine, al contorno, è possibile vedere altri blocchi che rappresentano tutta la circuiteria che si occupa della gestione del clock (*clock-gating* e più in generale *clock-management*) [23].
Gli anelli di retroazione e i contatori M, D e O hanno messo a dura prova l’algoritmo di esplorazione della netlist, basato su un metodo iterativo che agisce sui punti definiti nell’oggetto “path” di PrimeTime. Superata questa difficoltà, rimane quella legata alla disomogeneità dei path critici. E’ molto frequente in questa topologia di circuito, infatti, avere path con clock di lancio e di cattura diversi, path con cicli multipli, path che trasportano sia dati che clock, celle di *clock-gating* da sensitizzare, path con Flip-Flop sia *positive* che *negative-edge triggered* e anche path con Latch al posto di Flip-Flop [23].

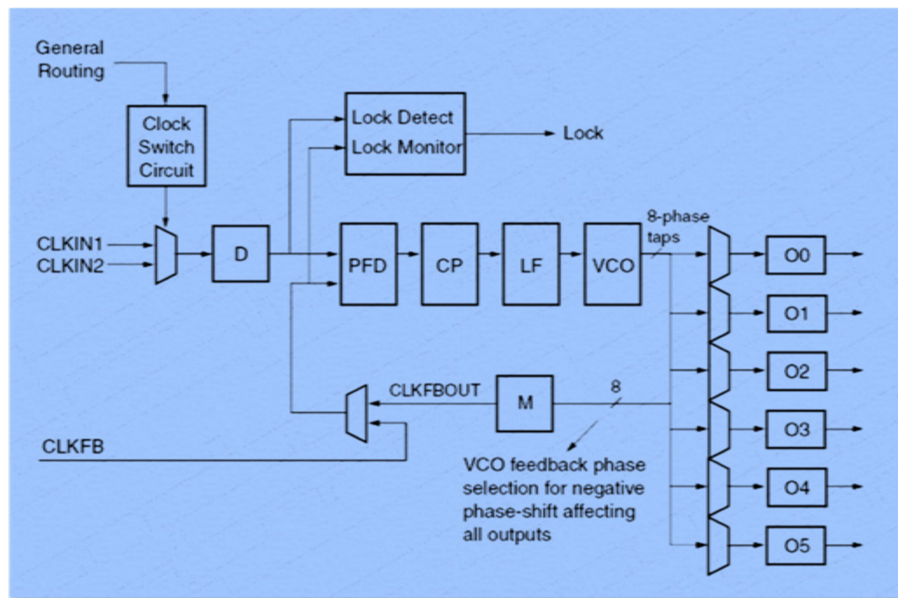


Figura 31. Schema a blocchi di un complesso Phase Locked Loop

Il principale scopo di questo *Case Study* è quello di mostrare quali sono le difficoltà del designer che ha il compito di dover decidere se mandare il suo progetto al *tape-out* rischiando fallimenti su silicio, oppure analizzare ulteriormente il circuito rimandando il tape-out e rischiando di allungare il *time-to-market*. Verrà mostrato come una scelta oculata dei path critici da analizzare possa fornire abbastanza accuratezza per ottenere in breve tempo la chiusura del progetto (in gergo *Design Closure*).

Le analisi statiche come la STA di PrimeTime introducono del pessimismo per evitare fallimenti su silicio. Questo comporta due tipi diversi di problemi: è difficile progettare circuiti che lavorino al limite delle potenziali prestazioni ottenibili dalla tecnologia; il pessimismo potrebbe non essere sufficiente a coprire fallimenti dovuti a condizioni critiche dal punto di vista del crosstalk. Un’ottima scelta è dunque quella di analizzare oltre ai path violati (se esistenti), anche quelli con poco margine.

Nel caso in esame, dopo un paio di iterazioni di STA e ECO, il designer si è trovato ad avere ancora **5 violazioni di setup** e circa 5 path non violati ma con poco margine di sicurezza. E’ doveroso precisare che con “poco margine” e “piccole violazioni” si intende che lo slack è dello stesso ordine dei ritardi che il dato subisce nell’attraversare ogni cella del path. In questi casi bastano piccole differenze fra l’analisi statica e quella dinamica

(anche per via del crosstalk) affinché si abbia un’informazione errata sulla violazione del path.

Vediamo dunque l’analisi che è stata compiuta per aiutare il designer in questo caso.
La cartella di lavoro è “APV_WORK_0”, ci tornerà utile in seguito.
La Figura 32 mostra la GUI, ancora agganciata a PrimeTime, riempita dei parametri usati per catturare i path critici e anche dei risultati della STA.

advUtils © [AMS Design Framework]

File Edit Simulate Help

pt_shell | Max | -max_paths 10 -slack_lesser_than 0.1 | Find | Netlist | Simulate | Results | Waves

	path_name	delay_type	slack	from	to	arrival	required	setup_time	hs
1	path1	max_fall	-0.012590	CORR_D_regx3x/CP	)UT_N_regx0x/D	0.488249	0.475659	0.056254	11
2	path2	max_rise	-0.012480	CORR_D_regx3x/CP	)UT_N_regx0x/D	0.488249	0.475659	0.056254	9
3	path3	max_rise	-0.012383	CORR_D_regx0x/CP	)UT_N_regx4x/D	0.494324	0.481941	0.049923	8
4	path4	max_rise	-0.010056	CORR_D_regx0x/CP	)UT_N_regx2x/D	0.489464	0.479408	0.052335	14
5	path5	max_fall	-0.011008	DACP_regx5x/CP	DACP[5]	0.311008	0.300000	1.450000	10
6	path6	max_fall	0.021483	:NT_RSY_regx1x/CP	DACP_regx3x/D	0.459393	0.480876	0.053056	15
7	path7	max_fall	0.021696	:NT_RSY_regx1x/CP	DACN_regx2x/D	0.459401	0.481097	0.053051	17
8	path8	max_fall	0.021979	:NT_RSY_regx1x/CP	DACP_regx1x/D	0.458653	0.480632	0.053200	17
9	path9	max_fall	0.022828	:NT_RSY_regx1x/CP	DACN_regx4x/D	0.458263	0.481091	0.053087	16
10	path10	max_fall	0.023189	:NT_RSY_regx1x/CP	DACP_regx5x/D	0.457862	0.481051	0.052909	9

ready 06:27:50 pm

Figura 32. Interfaccia grafica con i dati della Static Timing Analysis

I parametri usati sono i seguenti:

```
-delay_type max

-path_type full_clock_expanded

-slack_lesser_than 0.1
-max_paths 10
```

Il *delay_type* indica un’analisi di setup il *path_type* è forzato dal tool per catturare le informazioni sulla clock-tree di ciascun path e, infine, gli ultimi due parametri permettono di selezionare solo i primi path con slack minore di 100ps.

Come è possibile notare dalla ripetizione di alcuni *startpoint/endpoint* dei path, la parte critica del design è abbastanza confinata. Questo aumenta la probabilità di avere conflitti fra i path, infatti, l'algoritmo di generazione della netlist ha creato due sottocircuiti differenti.

Il file *APV.log*, alla fine della creazione della netlist, non ha riportato alcun messaggio di errore relativo a pin rimasti flottanti. Un'analisi manuale dei due file *opcond* della netlist ne è stata la conferma, infatti, la ricerca di pin con tensione chiamata, come impostazione predefinita, "Why_am_I_floating" non ha prodotto alcun risultato.

A questo punto, con due soli click sui tasti "Simulate" e "Results" è stata lanciata la *Dynamic Timing Analysis* con le impostazioni di default, quindi col simulatore CustomSim-XA, e i risultati sono stati annotati nella tabella visibile in Figura 33.

advUtils © [AMS Design Framework]

File Edit Simulate Help

pt_shell | Max | -max_paths 10 -slack_lesser_than 0.1 | Find | Netlist | Simulate | Results | Waves

	path_name	data_arrival	data_capture	sim_slack	data_arrival_xtk	data_capture_xtk	sim_slack_xtk	pt_slack	setup_time
1	path1	0.473569	0.671761	-0.008063	0.480018	0.665561	-0.020711	-0.012590	0.056254
2	path2	0.483509	0.729749	0.039986	0.515478	0.710575	-0.011157	-0.012480	0.056254
3	path3	0.481891	0.670578	-0.011236	0.499054	0.670942	-0.028036	-0.012383	0.049923
4	path4	0.471062	0.694031	0.020633	0.522069	0.703015	-0.021389	-0.010056	0.052335
5	path5	0.915812	out_port	N/A	0.938002	out_port	N/A	-0.011008	1.450000
6	path6	0.463360	0.705995	0.039579	0.467922	0.694660	0.023683	0.021483	0.053056
7	path7	0.463361	0.706246	0.039834	0.465622	0.626224	-0.042449	0.021696	0.053051
8	path8	0.460366	0.705878	0.042311	0.464928	0.694543	0.026415	0.021979	0.053200
9	path9	0.459948	0.706280	0.043245	0.464510	0.705405	0.037808	0.022828	0.053087
10	path10	0.459375	0.706027	0.043742	0.462237	0.694692	0.029546	0.023189	0.052909

ready | 06:36:12 pm

Figura 33. Tabella dei risultati al termine della simulazione

Analizziamo i risultati della simulazione distinguendo cinque situazioni:

- **Path 5** – ha come endpoint una porta d'uscita, per cui APV non mostra nella tabella lo slack calcolato. Questo serve a ricordare al designer che i *timing constraint* sulla porta riguardano un pezzo di circuito esterno al design, quindi non simulato. La parte esterna

conterrà quasi sempre una net e un registro, quindi si consiglia di includerli e rilanciare la simulazione.

Questo compito è semplice da mettere in pratica. E’ sufficiente cambiare la cartella di lavoro, ad esempio in “APV_WORK_path5”, e usare la tabella dei risultati della simulazione come input per PrimeTime. Selezionando, infatti, il tipo di analisi “*Table*” il tool legge solo i campi *startpoint*, *endpoint*, *delay_type* e *group* creando una nuova collezione di critical path da analizzare. In questo caso si eliminano o disabilitano tutte le linee tranne quella del path5 e si procede alla creazione della netlist. Editare la netlist per inserire il registro esterno di cattura del dato è immediato. Editare il DSPF per introdurre i parassiti della net di collegamento un po’ meno (si veda l’Appendice B: SPEF/DSPF). La simulazione, che per un path dura all’incirca qualche secondo, dà ragione alla STA, indicando il path come violato. A questo punto il progettista ha una netlist completa e un ambiente che gli consente di avere più strumenti, come le forme d’onda di tutti i punti del path e della sua clock tree, e dati più precisi rispetto alla sola STA di PrimeTime. In poco tempo è stato individuato il problema e una possibile soluzione alla violazione di questo path. Si trattava di un eccessivo tempo di salita di una cella del path.

Il problema poteva essere indubbiamente essere risolto direttamente in PrimeTime, ma la soluzione di APV sarebbe indispensabile nel caso in cui si debba portare il path a lavorare al massimo delle sue prestazioni.

- **Path 1 e 3** – sono entrambi violati, sia per la STA che per la DTA con e senza crosstalk. Un’analisi mirata di tipo “*Table*”, eseguita nella directory di lavoro “APV_WORK_path1e3”, ha permesso di individuare facilmente i punti critici e correggere le due violazioni. La Figura 34 mostra lo schematico dei due path, senza la relativa clock-tree.

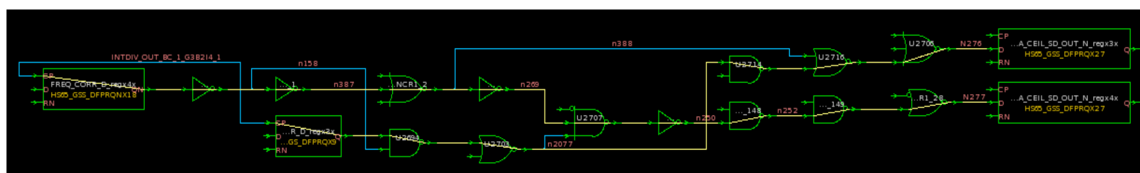


Figura 34. Schematico di due path registro-registro senza clock-tree

- **Path 2 e 4** – per la STA hanno uno slack negativo, ma di soli 12ps e 10ps rispettivamente. E’ necessario capire se la violazione è dovuta al pessimismo della STA o a una reale condizione del path.

  La simulazione analogica lanciata da APV mostra una situazione interessante. Senza il crosstalk il vincolo di setup non è violato, essendo lo slack compreso fra +20ps e +40ps. Includendo il crosstalk, invece, lo slack è negativo e circa uguale a quello dato da PrimeTime.

  Ancora una volta è stata effettuata un’analisi “*Table*” sui soli path in questione. Si mostra qui di seguito il CSV, con le sole colonne utili, che il tool vede dopo aver disabilitato le righe dei path estranei.

```
from, to, group, delay_type

FREQ_CORR_D_regx3x/CP, DELTA_CEIL_SD_OUT_N_regx0x/D, INTDIV_OUT, max_rise
FREQ_CORR_D_regx0x/CP, DELTA_CEIL_SD_OUT_N_regx2x/D, INTDIV_OUT, max_rise
```

  L’analisi ha evidenziato due contributi principali di crosstalk dannosi per il timing dei path. Per brevità chiameremo NET2 la vittima del path2, NET4 la vittima del path 4 e CNET2 e CNET4 gli aggressori appartenenti alla clock-tree. La CNET4 era in grado di indurre un delay di poco più di una decina di picosecondi sulla NET4, che, sommato agli altri minori contributi di crosstalk, era in grado di far violare il vincolo di setup. In Figura 35 è possibile vedere un esempio di aggressione.

  Il metodo utilizzato per risolvere questo problema è stato il seguente: sono state eseguire brevi simulazioni parametriche modificando solo la ripidezza del fronte (o *slope*) di aggressione della CNET4 direttamente dal file *opcond.spi* nella directory di lavoro corrente “APV_WORK_path2e4”. Il codice seguente mostra l’istanziazione dell’aggressore CNET4.

```
inst_name supply victim_net aggressor_net aggressor_cell [params ...]
XAGG_NET4 vdd NET4 CNET4 apvaggressor slope=27e-12 inv=0
```

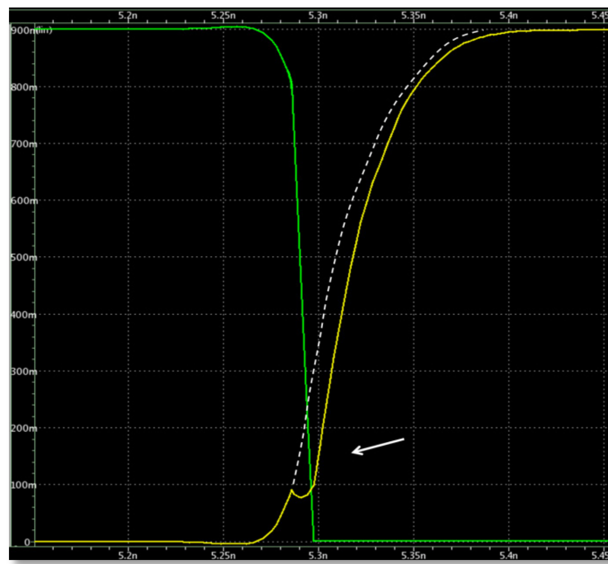


Figura 35. Forme d’onda simulate di un aggressore (verde) che induce un bump negativo sulla transizione rise di una net vittima (gialla)

Una minore ripidezza ci ha permesso di rientrare nei vincoli, ma non è facile da ottenere in quanto richiede il cambio del driver di CNET4 o l’inserimento di due invertitori. Un’altra soluzione consisterebbe nel modificare leggermente il layout relativo alle due net, allontanandole o schermandole con una *dummy* net.

Anche quest’ultima soluzione non è immediata. Ha richiesto di ri-estrarre i parassiti da layout per la sezione modificata e creare nuove sessioni di PrimeTime. A questo punto le varie sessioni di PrimeTime sono state connesse una ad una ad apvEasy, che fino ad ora non è mai stato chiuso. Le STA su tutto il design delle nuove sessioni non hanno evidenziato nuovi path critici, mentre le singole DTA effettuate da apvEasy hanno portato a slack positivi superiori ai 40ps.

Contestualmente alle modifiche di layout per le net NET4 e CNET4 si è agito anche su NET2 e CNET2 con considerazioni analoghe.

- **Path 6, 8, 9 e 10** – le analisi analogiche con crosstalk confermano che questi path non violano il vincolo di setup. Si notino gli slack di poco maggiori rispetto a quelli della STA nella Figura 33. Un esempio di tutte le forme d’onda caratteristiche di un path, con e senza crosstalk, è riportato in Figura 36. Aprendo una piccola parentesi, si noti

come il dato in ingresso del path abbia una transizione, questo comportamento è controllato dall’apposito modulo Verilog-A che genera un “dato sbagliato”, lo fa campionare al primo fronte attivo del clock e poi imposta il “dato corretto”. Ciò è necessario per poter avere le dovute transizioni lungo tutto il path e quindi poterne misurare gli istanti temporali.

Infine si notino le differenze fra le forme d’onda con e senza crosstalk.

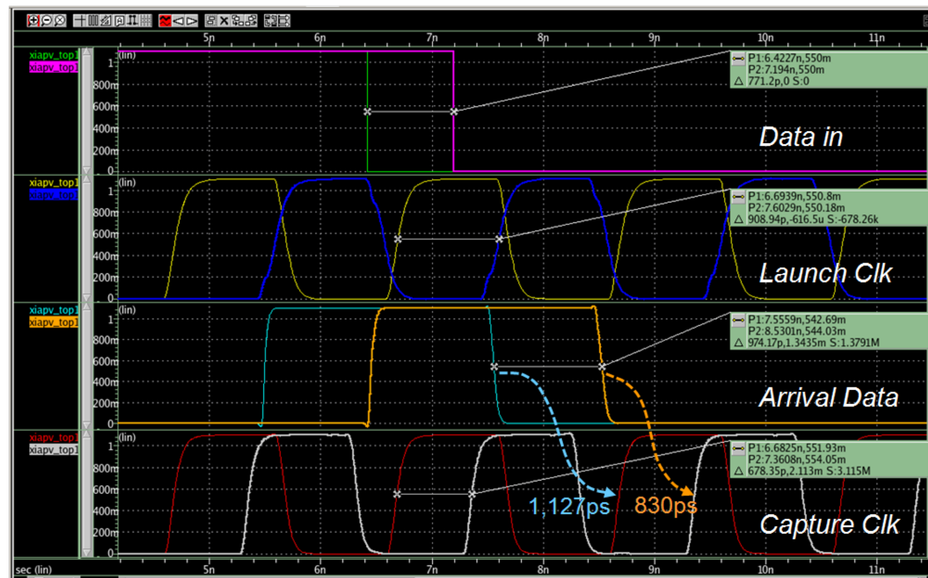


Figura 36. Principali forme d’onda per un path (con e senza crosstalk)

- **Path 7** – la CNET4 che dava problemi al path 4 aggredisce anche una net del path 7. Questo ha portato ad avere una violazione solo nel caso di DTA con crosstalk, facendo intuire un allineamento pessimistico aggressore/vittima irrealistico. Ciò è stato confermato da una successiva analisi personalizzata della clock-tree.

5.2 Analisi della clock-tree: Power/EMI

Oggigiorno i designer devono prestare un'attenzione sempre maggiore al consumo di potenza e alle interferenze elettromagnetiche, soprattutto per quanto riguarda applicazioni portatili e in campo *automotive*. Per questo motivo, l'estrazione dell'intera clock-tree, o di una sua porzione, per la simulazione analogica può essere molto utile in fase di *Physical Synthesis* del flusso di progettazione VLSI per SoC [24][25][26].

Durante la sintesi della clock-tree il designer è aiutato da prodotti CAD professionali, ma il tool apvEasy, abbinato ad un framework per la STA come PrimeTime può aiutare in situazioni particolari.

Ad esempio, è capitato di avere più implementazioni della stessa clock-tree di un microprocessore in tecnologia 65nm e di voler valutare quantitativamente i trend della potenza media assorbita, di quella di picco, o l'andamento delle componenti spettrali ad alta frequenza delle correnti del circuito. Le diverse implementazioni erano differenti in termini del rapporto fra la profondità (numero di stadi in serie fra la *clock-root* e gli *endpoints*) e il *fan-out* (numero di driver dello stadio successivo pilotati) [24]. La corrente istantanea fornita dall'alimentazione e il suo spettro di frequenza (visibili in Figura 37) così come la distribuzione degli *slew* dell'intera clock-tree (Figura 41) sono solo due esempi di strumenti utilizzati per prendere decisioni come quella di rilassare i fronti di clock e disperdere gli *skew* per ridurre le EMI e il consumo di potenza [26].

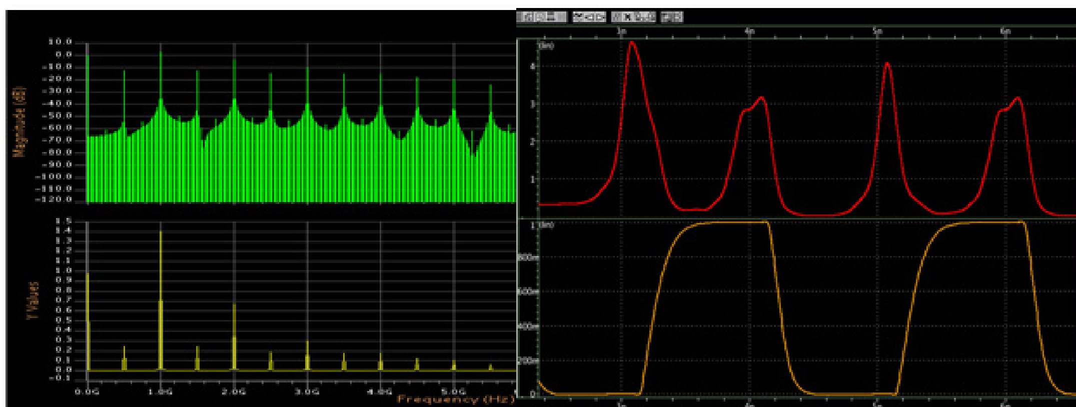


Figura 37. Corrente istantanea fornita dall'alimentazione (in rosso), rispettiva FFT (in giallo, scala lineare, in verde, scala in dB$_A$) e clock principale (in arancione)

Il flusso per l'analisi della clock-tree è molto simile al flusso principale di *Accurate Path Verification*. Una procedura, guidata dalle opzioni del designer, colleziona le informazioni sulla clock-tree da analizzare. Poi si crea la netlist SPICE, che conterrà solo un sottocircuito, pronta per essere simulata.

Il codice seguente mostra le istruzioni principali utilizzate per catturare le porzioni di clock-tree il cui nome inizia per "ACLK", come indicato dal designer attraverso la GUI visibile in Figura 38.

```
set clock_list [get_clocks ACLK*]
set clock_nets [get_clock_network_objects $clock_list \
                   -type net -include_clock_gating_network]
```

La variabile *clock_nets* viene trattata da un algoritmo che dagli oggetti di tipo net si espande catturando le celle e gli aggressori, e che genera le tensioni necessarie a generare i clock e a sensitizzare i pin flottanti di *clock-gating*.

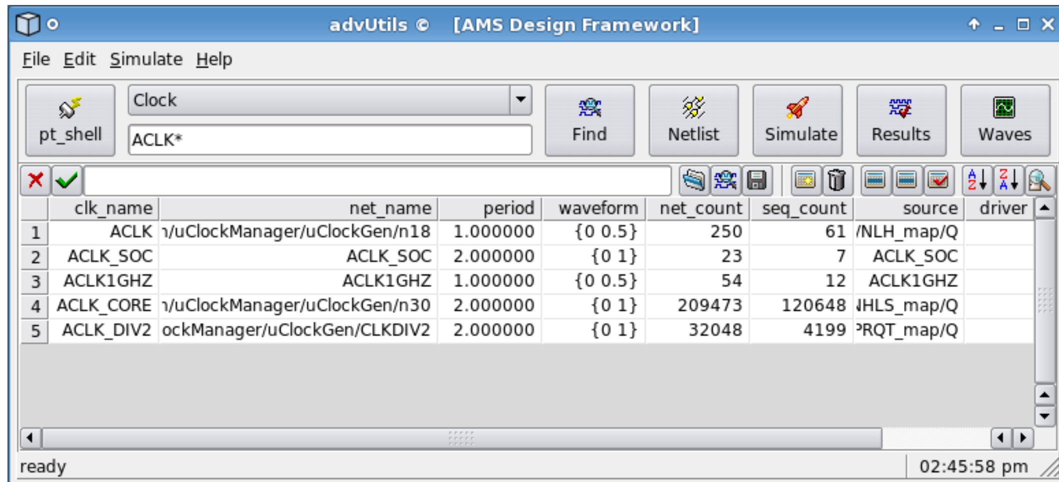


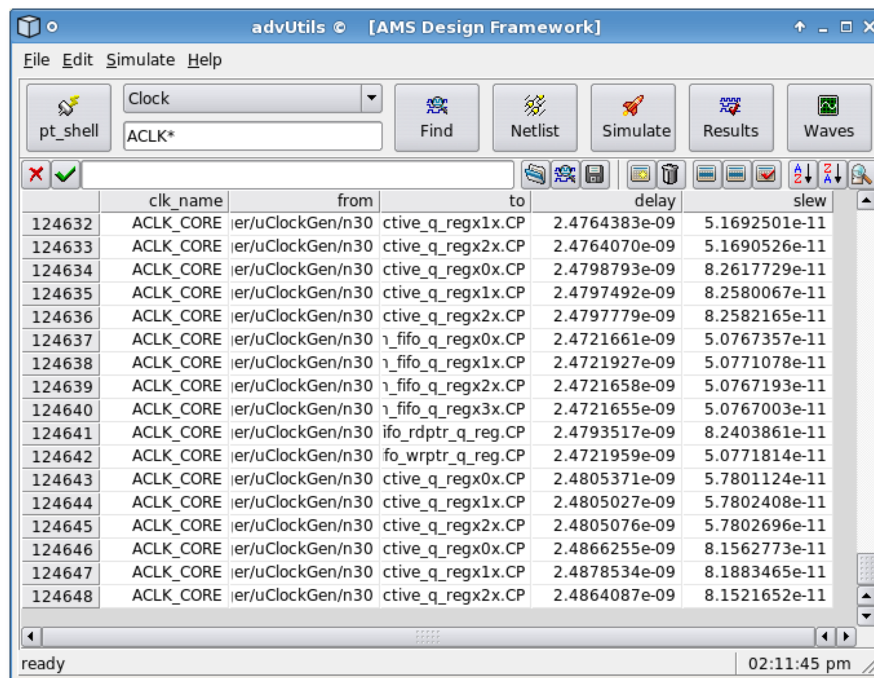
Figura 38. Snapshot durante un'analisi di clock-tree

Nella GUI si nota la tabella dei risultati riempita di cinque clock-tree. La colonna source indica la *clock-root*, mentre la fase e il *duty-cycle* sono condensati nella colonna *waveform*. I clock a 500MHz e 250MHz sono generati a partire da quello principale a 1GHz.

La simulazione è durata circa nove ore, che non è tanto considerando una netlist con circa 127'000 celle e 125'000'000 di componenti parassiti RC. La seguente tabella mostra altri dai sulla simulazione.

Tecnologia	CMOS a 65nm
Frequenze di clock	1GHz, 500MHz e 250MHz
Celle	circa 127 000
Net	circa 242 000
Endpoint (FF e Latch)	124 648
MOS	circa 4 900 000
Parassiti RC	circa 125 000 000
Simulatore	CustomSim-XA (con accuratezza molto elevata)
Durata analisi transitorio	10ns (per garantire due periodi su ogni endpoint)
CPU time	9h22m (con singolo processore)

La Figura 39 mostra la tabella dei risultati riempita con le migliaia di misure di *delay* e *slew* sugli *endpoint*.



	clk_name	from	to	delay	slew
124632	ACLK_CORE	ıer/uClockGen/n30	ctive_q_regx1x.CP	2.4764383e-09	5.1692501e-11
124633	ACLK_CORE	ıer/uClockGen/n30	ctive_q_regx2x.CP	2.4764070e-09	5.1690526e-11
124634	ACLK_CORE	ıer/uClockGen/n30	ctive_q_regx0x.CP	2.4798793e-09	8.2617729e-11
124635	ACLK_CORE	ıer/uClockGen/n30	ctive_q_regx1x.CP	2.4797492e-09	8.2580067e-11
124636	ACLK_CORE	ıer/uClockGen/n30	ctive_q_regx2x.CP	2.4797779e-09	8.2582165e-11
124637	ACLK_CORE	ıer/uClockGen/n30	ı_fifo_q_regx0x.CP	2.4721661e-09	5.0767357e-11
124638	ACLK_CORE	ıer/uClockGen/n30	ı_fifo_q_regx1x.CP	2.4721927e-09	5.0771078e-11
124639	ACLK_CORE	ıer/uClockGen/n30	ı_fifo_q_regx2x.CP	2.4721658e-09	5.0767193e-11
124640	ACLK_CORE	ıer/uClockGen/n30	ı_fifo_q_regx3x.CP	2.4721655e-09	5.0767003e-11
124641	ACLK_CORE	ıer/uClockGen/n30	ifo_rdptr_q_reg.CP	2.4793517e-09	8.2403861e-11
124642	ACLK_CORE	ıer/uClockGen/n30	fo_wrptr_q_reg.CP	2.4721959e-09	5.0771814e-11
124643	ACLK_CORE	ıer/uClockGen/n30	ctive_q_regx0x.CP	2.4805371e-09	5.7801124e-11
124644	ACLK_CORE	ıer/uClockGen/n30	ctive_q_regx1x.CP	2.4805027e-09	5.7802408e-11
124645	ACLK_CORE	ıer/uClockGen/n30	ctive_q_regx2x.CP	2.4805076e-09	5.7802696e-11
124646	ACLK_CORE	ıer/uClockGen/n30	ctive_q_regx0x.CP	2.4866255e-09	8.1562773e-11
124647	ACLK_CORE	ıer/uClockGen/n30	ctive_q_regx1x.CP	2.4878534e-09	8.1883465e-11
124648	ACLK_CORE	ıer/uClockGen/n30	ctive_q_regx2x.CP	2.4864087e-09	8.1521652e-11

Figura 39. Risultati dell'analisi di clock

Un esempio di prima analisi di Power e EMI può essere ora fatta attraverso le forme d'onda di corrente delle net di alimentazione o in qualsiasi altro punto del circuito nel dominio del tempo o della frequenza (di cui si è già parlato, si veda la Figura 37) e attraverso la distribuzione degli *slew*. La Figura 40 mostra come lanciare **gnuplot** (fra le altre cose) per generare il grafico della distribuzione in Figura 41.

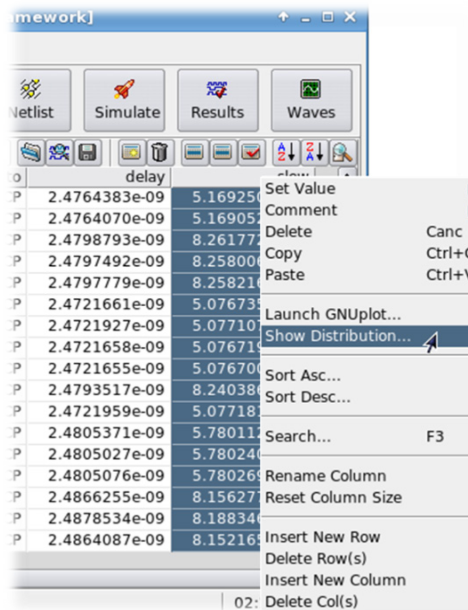


Figura 40. Menu tasto destro

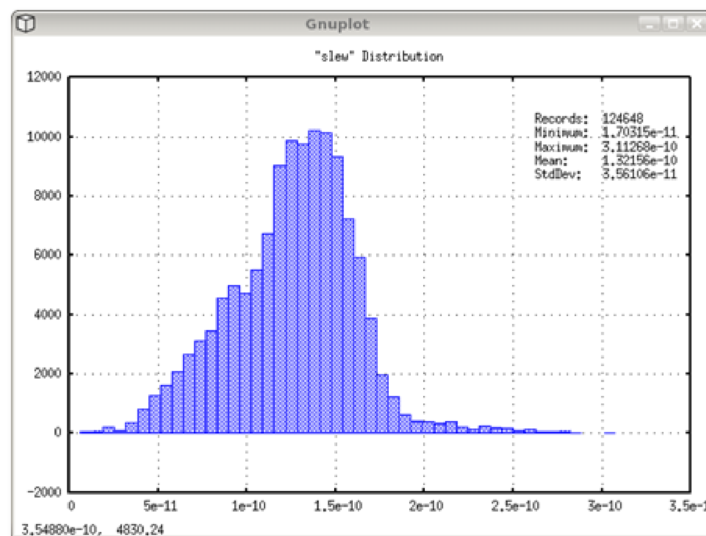


Figura 41. Distribuzione degli slew della clock-tree

Conclusioni e Sviluppi Futuri

In questa Tesi è stata più volte evidenziata l'importanza, per le aziende che producono circuiti integrati, di progettare ***System-on-Chip*** sempre più complessi e al contempo **ridurne i costi e i tempi di realizzazione**. Lo sviluppo di applicativi CAD che aiutino il progettista risulta essere in ritardo rispetto alla crescita della complessità dei circuiti, con ripercussioni sull'**affidabilità** e sul **time-to-market**.
Questi motivi ci hanno spinto a sviluppare una **metodologia di verifica** che ben si presta ad essere usata durante tutta la *Physical Synthesis* del flusso di progettazione VLSI, ma che è particolarmente studiata per la *Timing Closure post-layout*, ossia la certificazione che tutti i vincoli di setup/hold saranno rispettati su silicio. Infatti, come mostrato nel *Case Study* del Capitolo 5.1, il flusso di verifica principale, chiamato ***Accurate Path Verification***, permette di ridurre i tempi di verifica, debugging e correzione delle violazioni per arrivare precocemente e in modo affidabile al ***tape-out*** su silicio.

Il nucleo computazionale scritto in linguaggio TCL è stato alla fine corredato di una **interfaccia grafica** (in TK) per permettere una facile esecuzione del flusso, con strumenti per la gestione dei dati e dei risultati. E' stato creato dunque un **tool completo**, chiamato **apvEasy**, che integra la piattaforma CAD "**advUtils**" per l'***Accurate Design Verification***.

La struttura base del flusso principale è costituita da due semplici idee: **estrazione di informazioni topologiche e operative** su una porzione del design e successiva **creazione di una netlist SPICE** modulare da simulare. Questa generalità abilita una infinità di

scenari di analisi standard e custom, alcuni dei quali sono stati automatizzati in maniera analoga al flusso APV. Si tratta dell'analisi di *scan-chains*, dell'analisi di porzioni di circuito non necessariamente legate a questioni di vincoli di setup/hold e soprattutto **all'analisi di *clock-tree***. Si rivedano le sezioni 4.1.3 - Tipi di analisi disponibili, 5.2 - Analisi della clock-tree: Power/EMI e 4.7 - Deviazioni dal Flusso APV.
In quest'ultima si è parlato anche di una importante estensione del flusso APV verso l'AMS, ossia della possibilità di **includere interi blocchi analogici** (ad esempio memorie o MEMS) per simulare con più accuratezza e debuggare meglio il circuito. L'operazione non è ancora automatizzata, ma sarà uno dei principali **sviluppi futuri** in quanto è stata richiesta da molti gruppi di design interni alla STMicroelectronics. Bisogna dire però che, allo stato attuale, la struttura della netlist facilita molto questo compito.
Uno dei principali meriti del tool apvEasy è, infatti, la **struttura della netlist**, tantoché esso è utilizzato anche solo come estrattore di netlist. Il Capitolo 4.4 - Generazione della Netlist SPICE ha descritto i vari file che la compongono, divisi per scopo e scritti in modo ordinato e commentato. La netlist diventa in questo modo un potente strumento per il debugging. Un designer con buone conoscenze di simulazioni analogiche è estremamente facilitato nella modifica della netlist ai fini di effettuare analisi approfondite come, ad esempio, simulazioni *Monte Carlo*, simulazioni al di fuori del dominio di caratterizzazione delle celle o simulazioni comparative di diverse implementazioni di clock-tree.

Durante tutta l'ingegnerizzazione della metodologia si è tenuto conto di due aspetti: **standardizzazione** e **certificazione**. Il primo non è fondamentale al funzionamento dei flussi di verifica, ma è ugualmente importante. Consiste nella scelta dei formati di file per la rappresentazione dei parassiti e dei costrutti della netlist SPICE e Verilog-A in grado di essere capiti dai principali simulatori analogici. L'intera impostazione del database delle informazioni è indubbiamente legata al tool di simulazione statica PrimeTime ma è stata concepita in modo da poter essere adattata ad un tool di una *software house* diversa da Synopsys. Il flusso di *Verifica e Signoff* (che fra tutte le certificazioni per procedere al *tape-out* su silicio contiene anche la *Timing Closure*) usato dalle più grandi compagnie come STMicroelectronics è basato su PrimeTime. Tuttavia è sempre bene mantenere un approccio il più standard possibile. La **certificazione** che un tool riesce a dare ai suoi

risultati è ancora più importante. Nella sezione 4.2.1 si sono viste le accortezze adottate e quelle da migliorare al fine di fare, nel prossimo futuro, di apvEasy uno **strumento del Signoff**.

A parte gli eventuali miglioramenti per l'ottenimento del *Signoff* e l'automatizzazione del flusso per circuiti AMS ricordati fino ad ora, saranno oggetto di sviluppi futuri le **misure avanzate** e le **simulazioni iterative**. Per "misure avanzate" si intende la costruzione di una classe di *.measure* e moduli di controllo e di misura in Verilog-A. Queste possono servire al designer per fare misure sul circuito in esame, ma soprattutto per permettere le simulazioni iterative, un'idea nata in corso d'opera e descritta alla fine della sezione 4.5.1. Sostanzialmente consiste nello scambio di informazioni fra una simulazione e la successiva per modificare alcuni parametri della netlist o al più ambizioso progetto di gestire un *testbench* con monitor Verilog-A che controllano l'andamento della simulazione e il passaggio a iterazioni successive sulla base del risultato dell'attività di monitoraggio che svolgono.

Infine, intendo sottolineare ancora una volta l'**idea che sta alla base dell'implementazione della metodologia APV creata**. La sua forza sta nell'unire l'estrema **velocità delle analisi statiche** e la grande **precisione di quelle dinamiche**. Non è possibile eseguire simulazioni analogiche su grossi e complessi design, tantomeno farne molte per raggiungere una decente ***coverage*** delle possibili condizioni operative sia in termini di stati logici che di **corners tecnologici**. D'altra parte non è possibile fare simulazioni statiche senza introdurre del **pessimismo**, quindi non è facile verificare le violazioni dei vincoli temporali e soprattutto far lavorare un circuito al massimo delle sue potenziali **prestazioni**. A complicare tutto ci pensa il **crosstalk**, sempre più determinante con lo ***scaling* tecnologico**.

Appendice A: SPICE

SPICE

Il linguaggio SPICE si riferisce al simulatore SPICE: un programma utilizzato per simulare circuiti analogici elettronici; il suo intento originale era lo sviluppo di circuiti integrati, da cui ha dedotto il suo nome: *Simulation Program with Integrated Circuit Emphasis*.

SPICE analizza circuiti basati su una descrizione testuale dei componenti e delle connessioni del circuito facendo un'analisi matematica. Il complesso di file testuali descrittivi, dati in ingresso a SPICE, è chiamato **netlist**.

Ciascun file della netlist è composto da una serie di linee che possono essere di due tipi:

- **linee elemento**: descrivono la topologia del circuito e i valori dei componenti; contengono il nome dell'elemento, i nodi a cui è connesso ed i valori dei parametri che ne determinano le caratteristiche elettriche.
- **linee comando**: descrivono i tipi di analisi da effettuare, i parametri dei modelli o assegnano direttive al programma; hanno un punto in prima colonna.

La prima linea del file di ingresso deve essere una linea titolo, che può contenere qualunque informazione utile all'utente, mentre l'ultima deve essere una linea .END. L'ordine in cui si eseguono le altre linee è arbitrario, a parte casi particolari come, solo ad esempio, le righe che iniziano con il carattere "+" che sono continuazioni della riga precedente.

La sintassi SPICE è *case insensitive*, cioè non distingue lettere maiuscole e minuscole.

Ad ogni nodo del circuito è assegnato un identificativo non nullo, quello di riferimento (massa) deve essere sempre associato a 0. Ogni nodo deve avere, in continua, almeno una

connessione a massa e almeno due elementi ad esso connessi, tranne che per quelli di linee di trasmissione e di substrato dei MOSFET.

Tutti i componenti presenti in una netlist sono primariamente identificati dalla prima lettera presente nella rispettiva linea, i caratteri che seguono la lettera identificativa sono utilizzati per distinguere un componente di un certo tipo da un altro dello stesso tipo (per esempio r1, r2, r3 indicano tre resistori diversi).
Un aspetto da tener presente è che le eventuali lettere che seguono immediatamente un numero e che non sono fattori di scala vengono ignorate, così pure quelle che stanno subito dopo un fattore di scala. Quindi 10, 10V e 10HZ rappresentano tutti lo stesso numero; mentre M, MA e MSEC indicano il medesimo fattore di scala (M = 1E-3). I circuiti possono contenere diversi tipi di componenti, raggruppabili in tre categorie: passivi, attivi e generatori.
Di seguito sono descritti i principali componenti passivi e i loro modelli:

```
resistenze:    rnome nodo1 nodo2 valore
capacità:      cnome nodo1 nodo2 valore [ic=tensione_iniziale]
induttanze:    lnome nodo1 nodo2 valore [ic=corrente_iniziale]
trasformatori: knome lnome1 lnome2 fattore_di_accoppiamento_fra_0_e_1
```

Tutti i componenti attivi devono avere le loro caratteristiche elettriche descritte in un modello, vale a dire una linea che inizia con la parola .MODEL dove viene indicato il comportamento del dispositivo. La sintassi di un modello è la seguente:

```
.model nome_modello tipo_modello lista_parametri
```

I principali componenti attivi sono, con i rispettivi modelli:

```
diodi:              dnome anodo catodo modello
transistor BJT:     qnome collettore base emettitore modello
transistor JFET:    jnome drain gate source modello
transistor MOSFET:  mnome drain gate source substrato modello
```

In tutti i componenti indicati, nella parte del modello, i parametri che si possono indicare sono molteplici ed opzionali; inoltre è importante segnalare che i nomi dei modelli devono iniziare sempre con una lettera, non con un numero.

Fra i generatori occorre mostrare quelli a corrente continua (DC)

```
tensione: vnome nodo+ nodo- valore_dc
corrente: inome nodo+ nodo- valore_dc
```

e quelli ad impulsi

```
tensione: vnome nodo+ nodo- pulse(i p td tr tf pw pd)
corrente: inome nodo+ nodo- pulse(i p td tr tf pw pd)
```

I parametri rappresentano: i = valore iniziale, p = valore finale, td = ritardo, tr = tempo di salita, td = tempo di discesa, pw = larghezza impulso in secondi, pd = periodo in secondi.

In SPICE è possibile definire un sottocircuito e fare riferimento ad esso in modo similare a quanto sviene fatto per i modelli dei dispositivi a semiconduttore. Un sottocircuito è definito raggruppando un certo numero di linee elemento, successivamente viene automaticamente inserito il gruppo di elementi ogni volta che si fa riferimento ad esso. Non c'è limite alla dimensione e complessità dei sottocircuiti e un sottocircuito può contenerne altri. La sintassi per definire un sottocircuito è la seguente:

```
.subckt nome_sottocircuito lista_porte lista_parametri
 ...
.ends nome_sottocircuito
```

Come si può vedere la definizione del sottocircuito inizia con la linea comando .SUBCKT, dove sono specificati il nome identificativo, la lista delle porte e dei parametri; le linee elemento che vengono immediatamente dopo descrivono il sottocircuito, la cui definizione è terminata dalla linea comando .ENDS. All'interno della definizione non deve comparire alcuna linea comando, possono comunque essere presenti definizioni di altri sottocircuiti, modelli di dispositivi a semiconduttore e chiamate di altri sottocircuiti; ogni modello o definizione di sottocircuito incluso è strettamente locale, inoltre ogni nodo che non viene incluso nella linea .SUBCKT è prettamente locale.

La definizione di chiamata di un sottocircuito è:

```
xnome lista_nodi nome_sottocircuito
```

Il nome deve iniziare con una "**x**" e deve esserci lo stesso numero di nodi della definizione nello stesso ordine. Infine il nome del sottocircuito deve coincidere. con quello della definizione. Vediamo ora un esempio di sottocircuito:

```
.subckt INVstd A Z vdd gnd
.model m1 nmos level=70 vto=0.6v
.model p1 pmos level=70 vto=0.6v
m2 Z A gnd gnd m1 w=10u l=4u ad=100p pd=40u as=100p
m1 Z A vdd vdd p1 w=15u l=4u ad=100p pd=40u as=100p
.ends INVstd
```

La cui chiamata è la seguente:

```
xi99 IN OUT vdd gnd INVstd
```

Liberty file

I file .lib sono rappresentazioni in formato ASCII dei **parametri temporali e di power** associati a ciascuna cella di una particolare tecnologia. I parametri sono ottenuti attraverso simulazioni effettuate per molte condizioni operative. Le celle sono descritte in una moltitudine di aspetti, ad esempio la loro funzionalità e le proprietà delle singole porte che hanno. I liberty file possono avere dimensioni molto grandi e contengono anche le **funzioni logiche delle celle** e le molteplici **condizioni logiche** necessarie a far passare un dato da ciascun ingresso a ciascuna uscita (le cosiddette ***sensitizations***). Il codice seguente mostra una piccola porzione usata da PrimeTime per generare, appunto, le tensioni di sensitizzazione.

```
sensitization (2in_1out){
  pin_names (IN1, IN2, OUT);
  vector (0, "0 0 0") ;
  vector (1, "0 0 1") ;
  vector (2, "0 1 0") ;
  vector (3, "0 1 1") ;
  vector (4, "1 0 0") ;
  vector (5, "1 0 1") ;
  vector (6, "1 1 0") ;
  vector (7, "1 1 1") ;
  cell(my_cell){
    sensitization_master : 2in_1out;
    pin_name_map (A, B, Z);
    ...
  pin(Z) {
    ...
    timing() {
    related_pin : A;
    wave_rise (0, 4, 2, 6, 3);
    wave_fall (1, 5, 3, 6);
    wave_rise_sampling_index : 4;
    wave_fall_sampling_index : 2;
    }
  }
}
```

Deck file

Il deck file generato da PrimeTime non è altro che una netlist SPICE di un singolo path [2]. Si genera tramite l'apposito comando:

```
write_spice_deck -header header.spi \
  -analysis_type above_high \
  -output ../SIMUL_BEYOND_HIGH/new_general.spi \
  -logic_one_voltage 1.5 -logic_zero_voltage 0.0 \
  -sub_circuit_file ./SPICE/subckt.spi \
  [get_timing_path -to buf5/A]
```

Contiene i riferimenti delle celle del path e della sua clock-tree, i parassiti RC, i generatori costanti per la sensitizzazione dei pin flottanti, un PWL per il dato e uno (o due) per il clock, i PWL per gli aggressori e infine i .measure per i ritardi di ogni singolo stadio.

La netlist si divide in due file non molto semplici da navigare.

Appendice B: SPEF/DSPF

Questa appendice descrive i due formati più utilizzati per la rappresentazione dei parassiti R, C e L delle interconnessioni di un chip.

Il formato **SPEF** o ***Standard Parasitic Extraction Format*** è parte dello standard 1481 definito dalla IEEE per i circuiti integrati. La struttura dei file è mostrata nel seguente pseudo-codice. Saranno descritte solo le parti più importanti [9].

```
header_definition
[ name_map ]
[ power_definition ]
[ external_definition ]
[ define_definition ]
internal_definition
```

L'*header definition* contiene informazioni basilari come la versione dello SPEF, il nome del design e le unità di misura utilizzate.

La *name map* è la sezione in cui si assegna un identificativo numerico a ciascuna net, istanza e porta del design. Ad esempio, se una certa net gerarchica di nome "*top/hie_1/hie_2/SUB/n180*" viene mappata con "**123*", in tutto il resto del file comparirà questo numero invece del nome lungo. Questo permette di ridurre molto la ridondanza e quindi la dimensione del file SPEF.

L'*internal definition* è la parte più importante, quella che contiene i parassiti e la connettività di tutte le net. Ciascuna net può essere descritta sostanzialmente in due modi: con un modello distribuito o concentrato. Solitamente i software di estrazione forniscono un modello distribuito, più preciso. Un esempio di definizione di net distribuita è riportata nel codice seguente.

```
*D_NET *5426 0.899466
*CONN
*I *14212:D I *C 21.7150 79.2300
*I *14214:Q O *C 21.4950 76.6000 *D DFFQX1
*CAP
1 *5426:10278 *5290:8775 0.217446
2 *5426:10278 *16:3754 0.0105401
3 *5426:10278 *5266:9481 0.0278254
4 *5426:10278 *5116:9922 0.113918
5 *5426:10278 0.529736
*RES
1 *5426:10278 *14212:D 0.340000
2 *5426:10278 *5426:10142 0.916273
3 *5426:10142 *14214:Q 0.340000
*END
```

**D_NET* indica l'inizio della definizione della net **5426*, che ha una capacità complessiva di 0.899466 fF. La porzione **CONN* contiene l'indicazione della connessione al driver e a tutti i carichi. Il driver in questo caso è il pin Q dell'istanza **14214* della cella di libreria *DFFQX1*. La porzione **CAP* definisce tutte le capacità verso massa e di accoppiamento verso altre net. I numeri dopo il nome delle istanze indicano i nodi interni del modello distribuito della net. L'ultima porzione definisce le resistenze.

Il formato **DSPF** o ***Detailed Standard Parasitic Format*** descrive i parassiti con una sintassi SPICE, infatti, può anche essere dato direttamente in pasto ad un simulatore analogico. Un file DSPF ha un header e la definizione di sottocircuito (si veda l'Appendice A: SPICE). Quest'ultima contiene la sezione che descrive i parassiti di ciascuna net e la sezione dedicata alle istanziazioni delle celle (driver e carichi connessi alle net).

Come per lo SPEF esistono vari modelli per descrivere i parassiti di una net, tra cui i modelli distribuito e concentrato. Il codice seguente mostra il file DSPF con modello a parametri distribuiti per il semplice circuito costituito da un inverter visibile in Figura 42.

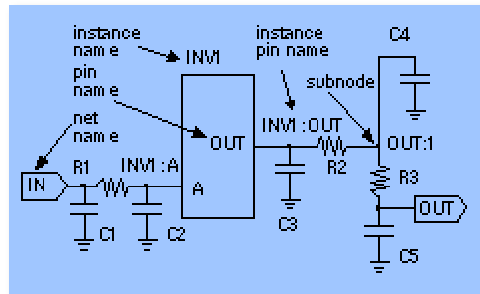


Figura 42. Esempio di parassiti per due net

Nell'header oltre a informazioni utili all'utente (le righe commentate con *) ci sono indicazioni utili al tool che leggerà il file, ad esempio il tipo e la versione del file.

I commenti seguiti dal carattere "|" sono direttive del formato DSPF. I principali sono (riferirsi alla Figura 42):

- ***|NET NetName NetCap** per dichiarare la net.
- ***|S(SubNodeName X Y)** per definire i sottonodi della net.
- ***|P(PinName PinType PinCap X Y)** per definire le porte connesse.
- ***|I(InstancePinName InstanceName PinName PinType PinCap X Y)** per definire le istanze connesse.

Il resto sono R e C dichiarate con sintassi SPICE classica.

```
* Design Name : EXAMPLE
* Date : 6 July 2013
* Time : 15:33:01
* Resistance Units : 1 ohms
* Capacitance Units : 1 pico farads
*| DSPF 1.0
*| DELIMITER "_"
.SUBCKT BUFFER OUT IN

* Net Section
```

```
*|GROUND_NET VSS

*|NET IN 3.8E-01PF
*|P (IN I 0.0 0.0 5.0)
*|I (INV1:A INV A I 0.0 10.0 5.0)
C1 IN VSS 1.1E-01PF
C2 INV1:A VSS 2.7E-01PF
R1 IN INV1:A 1.7E00

*|NET OUT 1.54E-01PF
*|S (OUT:1 30.0 10.0)
*|P (OUT O 0.0 30.0 0.0)
*|I (INV:OUT INV1 OUT O 0.0 20.0 10.0)
C3 INV1:OUT VSS 1.4E-01PF
C4 OUT:1 VSS 6.3E-03PF
C5 OUT VSS 7.7E-03PF
R2 INV1:OUT OUT:1 3.11E00
R3 OUT:1 OUT 3.03E00

*Instance Section
XINV1 INV1:A INV1:OUT INV

.ENDS
```

Appendice C: Verilog-A

Lo scopo del linguaggio Verilog-A HDL è quello di permettere al designer di sistemi analogici e circuiti integrati di creare moduli che descrivono i propri sistemi reali a livello strutturale e/o comportamentale [17]. Il **comportamento** di ciascun modulo può essere descritto tramite relazioni matematiche fra i vari terminali. Anche i parametri descrittivi di ciascun modulo entrano in gioco nelle formule. La **struttura** di ciascun modulo, invece, può essere descritta come interconnessione fra vari componenti. Le descrizioni possono spaziare in vari campi: **elettrico, meccanico, fluido-dinamico, termodinamico**. Ciò è possibile grazie all'uso di concetti molto generali quali quello di nodo, ramo e porta, che permettono l'uso delle leggi di conservazione dell'energia, ossia la generalizzazione delle **leggi di Kirchhoff**.

Un sistema è una collezione di componenti interconnessi. Ciascun componente riceve degli stimoli e produce delle risposte e può essere a sua volta un sistema, rendendo il sistema principale gerarchico. Se il componente non ha dei sotto-blocchi viene definito **componente primitivo** e il suo comportamento è descritto in termini di relazioni fra i suoi terminali.

Il Verilog-A può essere usato, ad esempio, per simulare ad alto livello un intero sistema, meccanico, elettrico o di altro tipo, oppure per sintetizzare dei blocchi già verificati all'interno di un circuito elettrico nuovo per velocizzarne la simulazione. Per permettere questa complessità, ovviamente, i costrutti del linguaggio sono molteplici. Questa appendice descriverà solo quelli principali, sufficienti per poter interpretare i moduli creati appositamente per la trattazione.

I moduli Verilog-A presentati nel Capitolo 4 servono alla sensitizzazione corretta del dato all'ingresso dei path, alla modellizzazione del crosstalk e all'implementazione di misure dinamiche (cioè per le simulazioni iterative).

Il codice Verilog-A, a differenza del codice SPICE, è *case sensitive*, quindi bisogna prestare attenzione quando si usano entrambi i linguaggi in una stessa netlist.

Il codice seguente verrà usato per descrivere le varie parti della definizione di un modulo Verilog-A. Si tratta di un monitor di fronti di salita/discesa ed è uno dei tanti monitor usati nel framework advUtils che contiene apvEasy.

```
module vP_vtransition(vin_p, vin_n, outrise, outfall);
   electrical  vin_p, vin_n, outrise, outfall;
   inout       vin_p, vin_n;
   output      outrise, outfall;

   parameter        enable           = 1;
   parameter real   voltage_max      = 1.8;
   parameter real   percentage_th_L  = 0.1, percentage_th_H = 0.9;

   real      start_rise, start_fall, stop_rise, stop_fall, supply_range, vthL, vthH, volt,
             trans_rise_signal, trans_fall_signal, time_rise, time_fall;
   integer   rise, fall, state_rise, state_fall, outfile;

   analog begin
     // SEE THE FOLLOWING PORTION OF CODE
   end

endmodule
```

La definizione di un modulo è contenuta fra le parole chiave "***module***" e "***endmodule***", necessita di un nome e può avere nessuna, una o più porte.

Ciascuna porta, e anche i nodi interni al modulo, possono essere dichiarati come "*electrical*" (o "*mechanical*" o altro), ereditando in questo modo una serie di caratteristiche e procedure definite dal linguaggio e/o dall'utente. Non è obbligatorio, ma è consigliata la dichiarazione di direzionalità delle porte, in modo da impedire assegnazioni/valutazioni indesiderate. Ad esempio una porta dichiarata come input può essere solo monitorata ma mai settata; una porta di tipo output invece può essere solo settata. Se non si dichiara esplicitamente la direzione la porta sarà di default una "*inout*" ossia senza vincoli.

Segue la sezione dei parametri. I valori di default devono essere specificati, non possono essere modificati durante la simulazione, ma solo all'atto dell'istanziazione del modulo. Come per le porte è possibile la dichiarazione multipla separando ogni assegnazione con una virgola. Il tipo, *integer* o *real*, è facoltativo.

Le variabili sono usate per l'elaborazione interna al modulo e richiedono la dichiarazione obbligatoria del tipo. Un utilizzo atipico che si è pensato di dare alle variabili, durante il lavoro su apvEasy e più in generale sull'intero framework advUtils, è quello di scambiare informazioni fra una simulazione e l'iterazione successiva in modo da renderle più intelligenti e veloci.

Fra le parole chiave "***analog begin***" e "***end***" è racchiusa la vera e propria parte che descrive il comportamento del modulo. Il codice esempio del monitor è riportato qui di seguito.

```
analog begin
   @(initial_step) begin
       start_rise     = 0.0;
       stop_rise      = 0.0;
       ...
       supply_range   = voltage_max - voltage_min;
       vthL           = voltage_min + supply_range*percentage_th_L;
       vthH           = voltage_min + supply_range*percentage_th_H;
       outfile        = $fopen("outVA.meas", "w");
       state_rise     = 0;
       state_fall     = 0;
   end
```

La "chiocciola" permette di catturare degli eventi che genera il simulatore. Ad esempio, l'*initial_step* si ha all'inizio di ogni analisi (tran, DC, AC…) e permette di eseguire le azioni definite fra i suoi "begin" e "end". Il *cross(expr,dir,…)*, invece, viene attivato quando si verifica il crossing dell'espressione *expr* con il valore zero nella direzione definita da *dir* (+1 in salita, -1 in discesa, 0 o non definito per entrambe le transizioni).

```
@(cross(V(vin_p) - vthL, 0)) begin
...
end;
```

Un altro evento è il *timer*, che permette azioni a tempi prefissati o a intervalli regolari. Una variabile d'ambiente molto utile è *$abstime* che contiene il tempo attuale della simulazione.

A parte le istruzioni condizionali e di ciclo presenti in ogni linguaggio di programmazione, esistono molte funzioni matematiche eseguite direttamente sulle variabili elettriche (o meccaniche ecc…): *idt* per l'integrale, *ddt* per la derivata, *transition* per creare dei fronti, *discontinuity* per definire delle discontinuità ecc…

```
// FINITE STATE MACHINE for rising edges
 case (state_rise)
    0: begin
      start_rise = 0.0;  stop_rise  = 0.0;
      if (enable == 1) state_rise = 1;
    end
    1: begin
      volt = V(vin_p);
      if (volt < vthL) state_rise = 2;
    end
    2: begin
       volt = V(vin_p);
       if (volt > vthL) begin
          start_rise = $abstime;
          trans_rise_signal = 1;
          state_rise = 3;
       end
  end
  3: begin
     volt = V(vin_p);
     if (volt > vthH) begin
        stop_rise = $abstime;
        time_rise = stop_rise - start_rise;
        trans_rise_signal = 0;
        state_rise = 1;
        $fwrite(outfile,"rise_time=%e from %e to %e\n",time_rise,start_rise,stop_rise);
      end
     if (volt < vthL) begin
       stop_rise = $abstime;
       trans_rise_signal = 0;
       $fwrite(outfile,"#rise_time=%e from %e to %e;FAILED",time_rise,start_rise,stop_rise);
       state_rise = 2;
      end
  end // end of 3: begin
 endcase
```

```
  // FINITE STATE MACHINE for falling edges
   case (state_fall)
      ...
   endcase
```

Il codice precedente implementa una macchina a stati grazie al costrutto condizionale *case*. Ci sono lo stato “disattivato” (*state_rise* = 0), lo stato 1 in cui si aspetta l’attraversamento della soglia di tensione inferiore, lo stato 2 che cattura il tempo in cui questa soglia viene attraversata, e infine lo stato 3 che aspetta la soglia superiore, cattura il tempo e scrive il risultato, con eventuale indicazione di fallimento della misura in un file.

La parte seguente, invece, mostra l’assegnazione dei livelli di tensione delle due porte d’uscita del modulo. La “*outrise*” avrà impulsi rettangolari durante fronti di salita del segnale d’ingresso, mentre la “*outfall*” durante i fronti di discesa. Non sono mostrate qui le due uscite che decretano la validità degli impulsi.

```
  // OUTPUTs assertions
   V(outrise) <+ voltage_min + trans_rise_signal*supply_range;
   V(outfall) <+ voltage_min - trans_fall_signal*supply_range;
end
```

Bibliografia

[1] **An Accurate Path Verification to Secure and Speed Up Nanometer Design Closure**
Salvatore Santapà, Alessandro Valerio, Pierluigi Daglio (STMicroelectronics), Andrea Barletta (Politecnico of Milan), Massimo Prando (Synopsys)
SNUG France 2013
http://www.synopsys.com/Community/SNUG/France/Pages/Abstracts.aspx?loc=france&locy=2013

[2] **PrimeTime and PrimeTime SI User Guide Version G-2012.06**
Synopsys

[3] **VLSI Physical Design: From Graph Partitioning to Timing Closure**
A. Kahng, J. Lienig, I. Markov, J. Hu
Springer (2011)

[4] **Critical Paths Verification and Debugging with PrimeTime Advanced Features**
Wei-Si Jiang
SNUG San Jose 2006

[5] **Mixed-Signal Methodology Guide**
Advanced Methodology for AMS IP and SOC Design, Verification and Implementation
Jess Chen (and contributes by Michael Henrie, Ph.D. Monte F. Mar, Mladen Nizic)
Cadence Design Systems

[6] **Have I Really Met Timing? Validating PrimeTime Timing Reports with Spice**
Tobias Thiel
IEEE

[7] **Static Timing Verification with Crosstalk Timing Effects: the CASTA approach**
B. Franzini, C. Forzan, D. Pandini, P. Scandolara, A. Dal Fabbro, M. Cavalli,
R. Zafalon (STMicroelectronics)
ESNUG 2000

[8] **Crosstalk Aware Static Timing Analysis Environment**
B. Franzini, C. Forzan – STMicroelectronics
ESNUG 2001

[9] **Static Timing Analysis for Nanometer Designs - A Practical Approach**
R. Chadha and J. Bhasker
Springer, 2009

[10] **What is the difference between single, bc_wc, and on_chip_variation analysis modes?**
https://solvnet.synopsys.com/retrieve/print/013762.html

[11] **Accurate Signoff Analysis with Path-Based Analysis in PrimeTime**
https://solvnet.synopsys.com/retrieve/print/012134.html

[12] **PrimeTime Advanced OCV Technology – Easy-to-Adopt, Variation-Aware Timing Analysis for 65-nm and below**
Sunil Walia
April 2009

[13] **PrimeTime Simultaneous Multivoltage Timing Analysis – Tecnology Validation Guide**
Synopsys
July 2012

[14] **What is the Appropriate Model for Crosstalk Control?**
Lou Scheffer – Cadence Design Systems

[15] **Frequently asked questions about PrimeTime SI's enhanced alignment modes**
https://solvnet.synopsys.com/retrieve/print/015943.html

[16] **Practical Programming in Tcl and Tk**
Brent B. Welch, Ken Jones, Jeffrey Hobbs
Prentice Hall Professional, 4th ed., 2003

[17] **Analog Behavioral Modeling with the Verilog-A Language**
Dan FitzPatrick, Ira Miller
Springer 1997

[18] **Custom WaveView UserGuide Version F-2011.09**

Synopsys

[19] **Eldo User's Manual Release 2011.03**

Mentor Graphics

[20] **CustomSim User Guide Version H-2013.03**

Synopsys

[21] **GNUplot**

http://www.gnuplot.info

[22] **Synopsys Expands Collaboration With STMicroelectronics in Timing Sign-Off**

http://news.synopsys.com/index.php?s=43&item=643

[23] **Working with PLLs in PrimeTime – avoiding the "phase locked *oops*"**
Paul Zimmer
SNUG San Jose 2005

[24] **Design Methodologies for Reliable Clock Networks**

Deokjin Joo, Minseok Kang, Taewhan Kim

JCSE Vol. 6, No. 4, December 2012, pp. 257-266

[25] **Clock Tree Optimization for Electromagnetic Compatibility (EMC)**

Xuchu Hu, Matthew R. Guthaus

University of California Santa Cruz

[26] **Clock-tree synthesis for low-EMI design**

Davide Pandini, Guido A. Repetto, Vincenzo Sinisi (STMicroelectronics)

J. Embedded Computing 3 (2009)

[27] **Digital Systems Testing and Testable Design**

M. Abramovici, M. A. Breuer, A. D. Friedman

IEEE Press (1990)

[28] **Functional scan chain testing**

D. Chang, K.T. Cheng, M.Sadowska

pp. 278-284 DATE 1998

[29] **The Semiconductor Wiki Project**

http://www.SemiWiki.com

Ringraziamenti

Il ringraziamento più grande va ai miei genitori e a mia sorella Erika per avermi consentito di intraprendere e completare la mia carriera universitaria e per aver creduto in me.

Ringrazio la mia ragazza Elena, senza di lei non ce l'avrei mai fatta e anche Anna e Riccardo per il loro enorme aiuto.

Non potrei non ringraziare: mia nonna Maria, che cerca sempre di farmi cambiare idea e farmi tornare a vivere giù in Calabria; mia cugina Annalisa per i suoi consigli e la sua abilità culinaria; mio zio Rocco per la sua straordinaria capacità di infondermi calma e positività.

Ringrazio per la loro vicinanza Davide e Marianna a cui vanno i miei più cari auguri per il futuro. A Barone, Spo, Alex e Piedone vanno, invece, le mie scuse. Siamo lontani e ci vediamo poco, ma vi penso sempre e vi auguro un mondo di bene.

Saluto tutti i miei compagni del Poli, soprattutto Mauro e Pietro, e tutte le persone che ho conosciuto in STMicroelectronics. Un ringraziamento speciale va ai miei colleghi, che hanno reso questa mia esperienza lavorativa estremamente significativa anche dal punto di vista umano. Sono tanti, ma meritano di essere nominati: Pier, Salvo, Ale, Paolo Va., Marco, Elena, Paolo Ve., Emmanuela, Simone, Francesco, Giuseppe, Davide, Gabriele. A Salvo rivolgo la mia gratitudine per aver sopportato la mia testardaggine ed per aver mantenuto il mio lavoro sempre stimolante. Il suo ruolo è stato non solo quello di un tutor aziendale, ma anche quello di un vero e proprio mentore, grazie alla sua grande esperienza professionale e di vita. Non sono mancati, ovviamente, accesi e costruttivi confronti, senza i quali questa Tesi non sarebbe così di rilievo.

Infine, ci tengo a fare i miei migliori auguri al Prof. Ripamonti che è stato in grado di trasmettermi egregiamente il suo sapere durante i corsi e durante il suo periodo come mio relatore.

Andrea