Sintésis Lógica

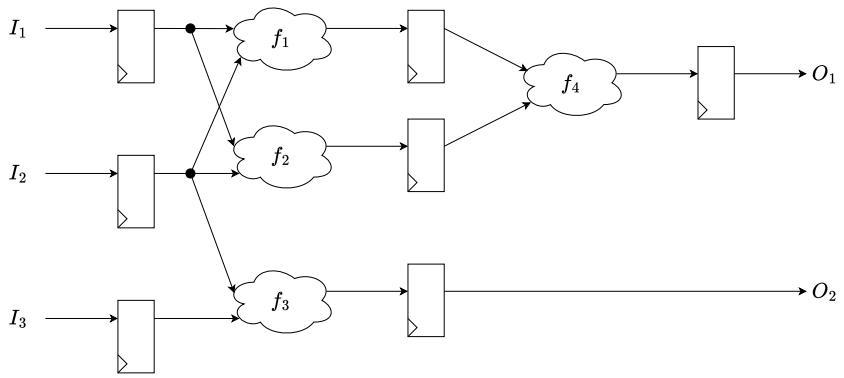
31 de octubre de 2024

1. Introducción

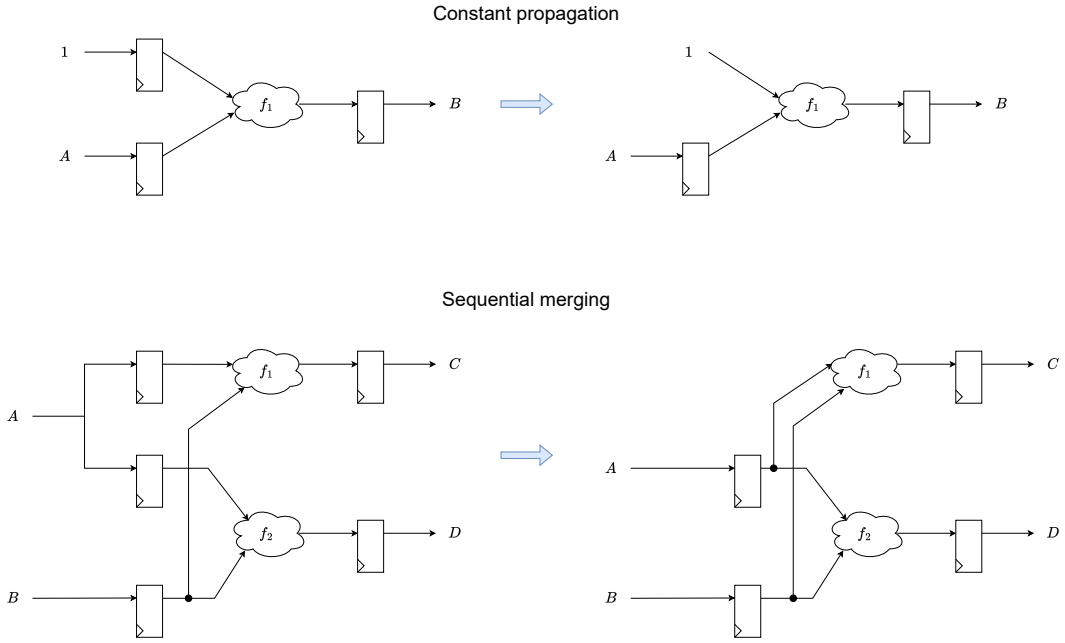
La síntesis lógica corresponde al proceso automático de convertir una descripción funcional RTL en un netlist (diagrama de conexiones) de compuertas lógicas.

Los pasos del proceso de síntesis son los siguientes:

- **Parsing:** Análisis sintáctico del código RTL en búsqueda de errores.
- **Elaboration:** Análisis jerárquico en búsqueda de todos los componenentes definidos que conforman el diseño final verificando que los mismos estén correctamente instanciados.
- **Extracción de DFFs:** se determinan los DFF del diseño.
- **Extracción de funciones lógicas:** Se identifican las funciones lógicas que relacionan todos los DFFs del diseño.

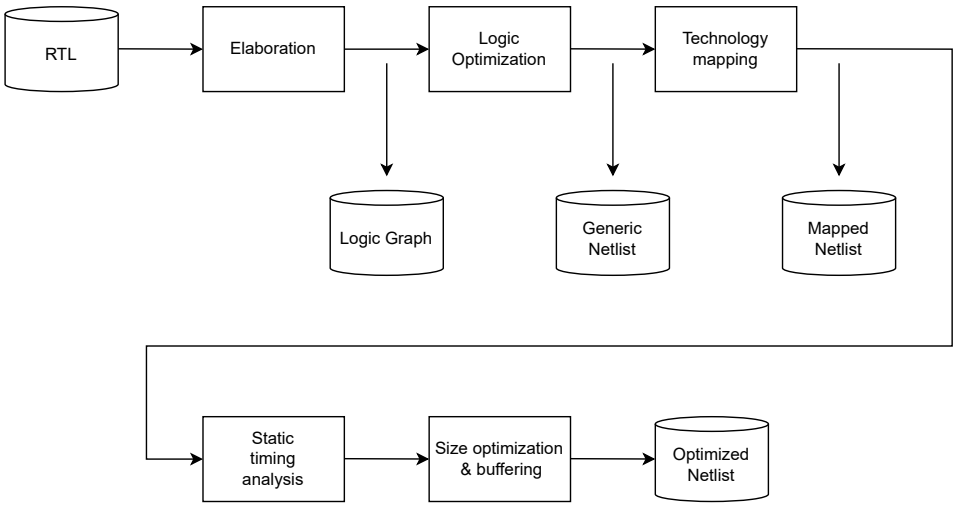


- **Minimización secuencial:** Se realizan únicamente dos minimizaciones de DFF: constant propagation, sequential merging.



- **Minimización combinacional:** Todas las funciones booleanas que relacionan los distintos DFFs del diseño son minimizadas. En general, se emplean transformaciones lógicas y minimización de multifunciones.
- **Generic mapping:** Las funciones lógicas son implementadas mediante compuertas genéricas que no necesariamente existen en el conjunto de celdas estándar.
- **Technology mapping:** Las celdas genéricas son convertidas a celdas estándar.
- **Setup slack timing analysis:** Considerando la señal de clock como ideal (retardo nulo, driving infinito), se analizan todos los caminos combinacionales verificando que se cumpla con el setup time de cada DFF del diseño. No se efectúa hold slack ya que se considera al clock como ideal.
- **Setup slack optimization:** Para aquellos DFF que no se obtiene un setup slack positivo en el paso anterior, se optimiza el camino combinacional de forma de reducir el retardo.

Nota: Cuando se detectan operadores aritméticos en el RTL, se extraen genéricamente en el paso de generic map de forma de no emplear las técnicas de minimización booleana sobre ellos.



2. Minimización lógica combinacional

2.1. Logic transformations

2.1.1. Decomposition

Es el proceso de escribir una función lógica como composición de otras funciones.

Ejemplo: $F(a,b,c,d) = abc + abd + \overline{a}c\overline{d} + \overline{b}\overline{c}\overline{d}$

Sean $X = ab$, $Y = c + d$, entonces

$F(a,b,c,d) = ab(c+d) + (\overline{a} + \overline{b})(\overline{c}\overline{d}) = ab(c+d) + \overline{ab}(\overline{c+d}) = XY + \overline{XY}$

Luego, $F = XY + \overline{XY}$, $X = ab$, $Y = c + d$.

2.1.2. Extraction

Es el proceso de identificar subexpresiones comunes entre distintas funciones lógicas.

Ejemplo: $F_1 = acd + bcd + e$, $F_2 = a\overline{e} + b\overline{e}$, $F_3 = ade$

Sean $X = a + b$, $Y = cd$. Luego, $F_1 = XY + e$, $F_2 = X\overline{e}$, $F_3 = Ye$.

2.1.3. Factoring

Es el proceso de identificar factores comunes en una representación suma de productos (SOP; *sum of products*) de una función lógica.

Ejemplo:

$F = ac + ad + bc + bd + e$, entonces puede ser factorizada como $F = (A + b)(c + d) + e$.

2.1.4. Substitution

Es el proceso de reescribir una función lógica en función de otra.

Ejemplo:

Sea $F = a + bc$ y sea $G = a + b$. Queremos sustituir G dentro de F, entonces:

$$\begin{aligned} F &= G(a + c) \\ &= (a + b)(a + c) = aa + ac + ba + bc \\ &= a + ac + ba + bc \\ &= a(1 + c + b) + bc \\ &= a + bc \end{aligned}$$

Finalmente, $F = G(a + c)$.

2.1.5. Collapsing

Es el proceso inverso a substitution, es decir se reemplaza G por su expresión en la ecuación de F:

Ejemplo:

Sea $F = Ga + \overline{G}b$, y sea $G = c + d$ entonces

$$F = (c + d)a + (\overline{c + d})b = ac + ad + b\overline{c}\overline{d}$$

2.2. División y divisores comunes

Sean f y p dos funciones lógicas, se quiere encontrar dos funciones lógicas q y r que satisfagan $f = pq + r$. Esta expresión se dice que es *la división booleana de f por p con un cociente q y resto r*. La función p se llama divisor de f si $r \neq 0$ o se llama factor de f cuando $r = 0$.

Proposición 1. *Una función p es un factor booleano de la función lógica f si y sólo si $f.\overline{p} = 0$.*

(Esto significa que el conjunto de maxitérminos de f está contenido en el conjunto de maxitérminos de p).

Proposición 2. *Si $f.\overline{p} \neq 0$, entonces p es divisor booleano de f.*

Definición 1. *El soporte de una función booleana f es el conjunto de todas las variables que sintácticamente ocurren en f y se lo nota $sop(f)$.*

Ejemplo: $f = a + \overline{a} + bc$, $sop(f) = \{a, b, c\}$

Definición 2. *Se dice que una función booleana f es ortogonal a otra función g, $f \perp g$, si $sop(f) \cap sop(g) = \emptyset$.*

Ejemplo: $f = a + b$, $g = c\overline{d}$ entonces $sop(f) = \{a, b\}$, $sop(g) = \{c, d\}$. Luego $sop(f) \cap sop(g) = \{a, b\} \cap \{c, d\} = \emptyset$ y por lo tanto $f \perp g$.

Definición 3. *Un cubo es toda conjunción de variables (es decir un producto de variables).*

Definición 4. *La función lógica g es divisor algebraico de f si existen dos funciones lógicas h y r tales que $f = g.h + r$ donde $h \neq 0$, $g \perp h$ y r es mínima (tiene la cantidad mínima de cubos).*

Nota: Puede demostrarse que con la condición de exigir que r sea mínima, el cociente h es único. A este conciente se lo nota f/g.

Definición 5. *Se dice que g divide a f de forma par si $f = g.h$, $h \neq 0$, $g \perp h$, $r = 0$.*

2.2.1. Cálculo de cocientes h

Sea $|f|$ la cantidad de cubos que posee la expresión SOP de f.

Sea $|g|$ la cantidad de cubos que posee la expresión SOP de g.

Sea el conjunto de cubos de la función f: $M_f = \{a_1, a_2, \ldots, a_{|f|}\}$.

Sea el conjunto de cubos de la función g: $M_g = \{b_1, b_2, \ldots, b_{|g|}\}$.

Se define $h_i = \{c_j : b_i.c_j \in M_f\}$ para todo $i = 1, 2, \ldots, |g|$. Es decir, hi corresponde a todos los factores de c_j que hacen que el i-ésimo cubo de g sea cubo de f.

Por lo tanto,

$$h = \bigcap_{i=1}^{|g|} h_i = h_1 \cap h_2 \cap \ldots \cap h_{|g|}$$

Ejemplo:

$f = abc + abd + de$, $|f| = 3$

$g = ab + e$, $|g| = 2$

$b_1 = ab$, $b_2 = e$ entonces

$b_1 c = a_1 = abc$

$b_1 d = a_2 = abd$

$b_2 d = a_3 = ed$

$h_1 = \{c, d\}$, $h_2 = \{d\}$ entonces $h = h_1 \cap h_2 = d$.

Luego, $f = gh + r = (ab + e)d + r$

donde $(ab + e)d = abd + ed$ y resulta $r = f - gh = abc$. Finalmente, $g = (ab + e)$, $h = d$, $r = abc$.

2.2.2. Kernels y divisores algebraicos

Definición 6. *El conjunto de divisores algebraicos de una función lógica f se define como $D(f) = \{g : h = f/g \neq 0\}$.*

Definición 7. *El divisor primario de f se define como $P(f) = \{f/c : c$ es un cubo$\}$.*

Ejemplo: $f = abc + abde$, entonces $f/a = bc + bde$. Como a es un cubo, entonces $bc + bde$ es un divisor primario.

Proposición 3. *Todo divisor de f está contenido en un divisor primario, es decir si g divide a f entonces $g \subset p \in P(f)$.*

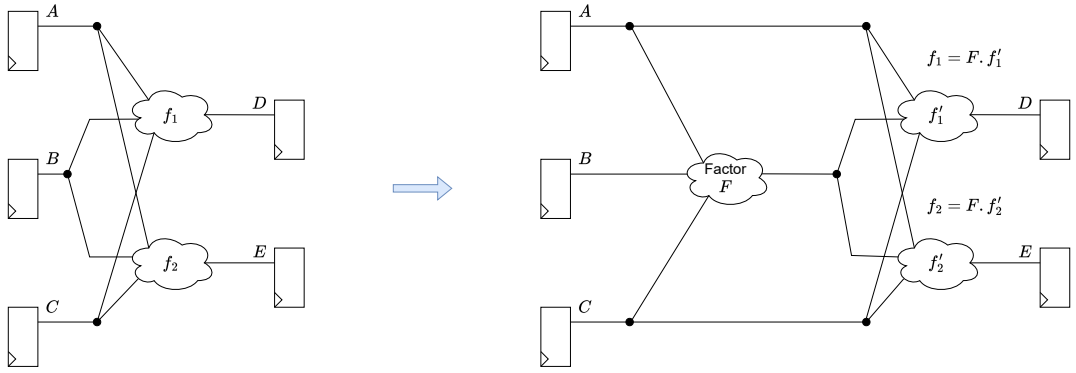
Definición 8. *Un función f se dice libre de cubos si el único cubo que la divide de forma par es 1.*

Definición 9. *Los kernels de f se definen como el conjunto $K(f) = \{k : k \in P(f)$ y k es libre de cubos$\}$. Dado un kernel $k \in K(f)$, su cokernel es el cubo k_c tal que $k = f/k_c$.*

Ejemplo: $f = abc + abde$, $f/a = bc + bde$, entonces $bc + bde$ es un divisor primario ya que a es un cubo pero no es libre de cubos ya que b es un factor de $f/a = b(c + de)$. Sin embargo, $f/(ab) = c + de$ es un kernel y su cokernel es ab.

Teorema 4. *Dos expresiones f_1 y f_2 tienen un divisor común d que no es un cubo, si y sólo si existen dos kernels $k_{f_1} \in K(f_1)$ y k_{f_2} tales que $k_{f_1} \cap k_{f_2}$ tienen dos o más términos (es decir $k_{f_1} \cap k_{f_2}$ no es un cubo).*

Observación: Podemos usar los kernels de f_1 y f_2 para buscar divisores comunes entre las dos funciones lógicas. Es decir, se buscan los conjuntos $K(f_1)$ y $K(f_2)$ y formamos su intersección. Si dicha intersección no contiene elementos que no sean cubos, por el teorema anterior sólo necesitamos buscar divisores que sean cubos. Caso contrario, se ha encontrado un divisor común entre f_1 y f_2.

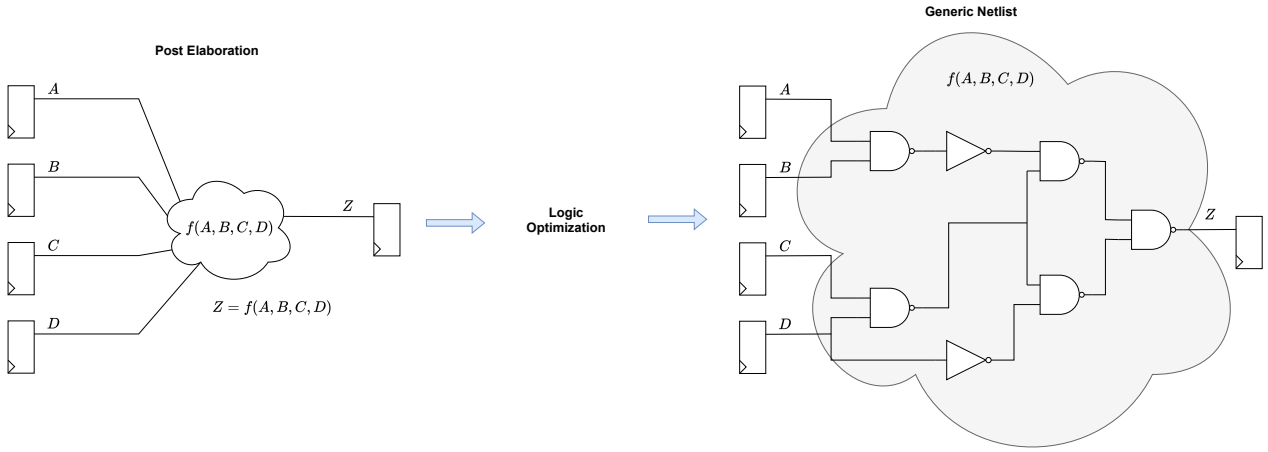


3. Technology mapping

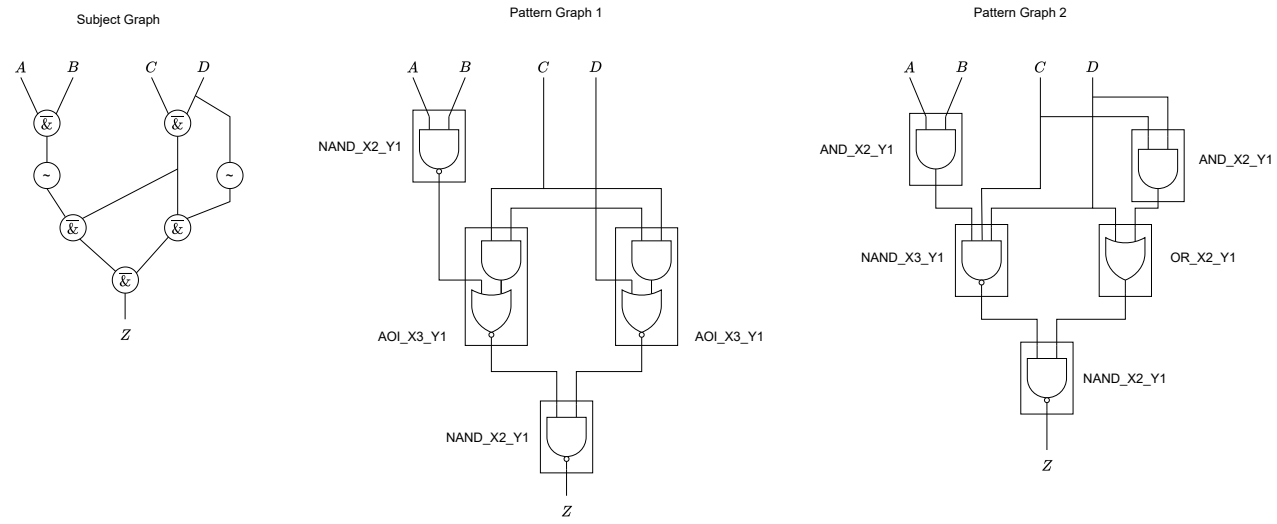
Technology mapping es el proceso de transformar el generic netlist en un technology netlist.

Post elaboration, se tiene un grafo de relaciones o funciones booleanas entre registros.

Post logic optimization, el generic nestlit puede representarse por un AIG (AND-INV Graph).



El proceso de technology mapping puede pensarse como el problema de cubrir el *subject graph* (AIG) por un *pattern graph* (celdas del technology library).



Para un mismo subject graph pueden existir más de un pattern graph que lo cubre. De estar forma, el problema de technology mapping radica en elegir de entre todos los pattern graphs que cubren el subject graph aquel que **minimiza el costo de cubrimiento**.

Es costo de un pattern graph está definido como la suma de los costos individuales de cada celda tecnológica que lo implementa:

$$PGC_j = \sum_{i_j} C_{i_j}$$

donde j indica cada uno de los posibles pattern graphs.

Cuando se tiene la colección de todos los posibles pattern graphs que cubren un específico subject graph, entonces se elige el óptimo:

$$OPG = arg \left[\min_j \{PCG_j\} \right]$$

Esta operación de elegir el óptimo pattern graph se repite para todos los subject graphs del circuito (recordar que hay uno por cada función lógica que relacione los registros).

La optimización tiene tres opciones posibles: area, potencia, retardo. Cada celda de la tecnología tiene entonces 3 costos asociaddos (el área que ocupa, la potencia que consume

en una transición, el tiempo de propagación). Elegir el pattern graph óptimo se hace en función de cuál de estas métricas quiere minimizarse y se toma entonces sólamente el costo de cada celda visto desde dicha métrica.

Nota: Tícicamente, los sintetizadores obtienen un netlist tecnológico para mínima área, y luego del timing analysis modifican los pattern graphs por otros que obviamente tienen un mayor costo en área pero logran cumplir las restricciones de setup time en los flip flop destino.

4. Wireload model

Para realizar el cálculo de setup slack, la herramienta de timing analysis conocer los parásitos del circuito. Sin embargo, esto no es posible ya que no se posee el layout del circuito aún. Solución: se utiliza un wireload model.

Definición: Un wireload model es una función $w : \mathbb{N} \to \mathbb{R}_{>0}$ tal que asocia

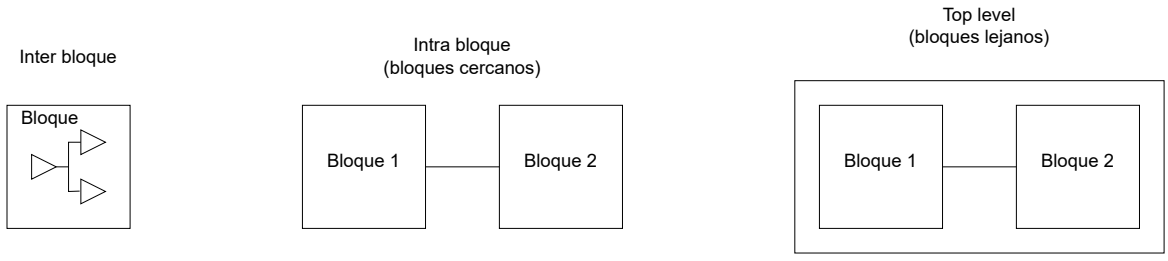
$$L = w(\text{FANOUT})$$

. Es decir si un nodo tiene un fanout N, se le asocia a la conexión una longitud que es función del fanout. Luego con esta longitud se obtienen las capacidades y resistencias parásitas utilizadas para timing analysis durante el proceso de síntesis.

Ejemplo:

FANOUT	Length
1	2.6μm
2	2.9μm
3	3.2μm
4	4.5μm

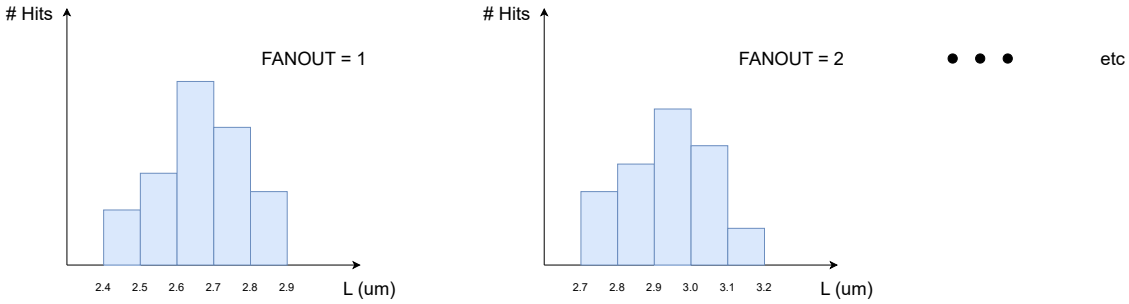
Generalmente se utilizan 3 wire models: inter bloque, intra bloque, top level:



4.1. Cómo se construye un wireload model

Se realiza una síntesis para una tecnología específica sin considerar parásitos, luego se hace un placement y un routing, y a partir de esta implementación se construye la estadística de la longitud de las conexiones en función del fanout. Se toman los valores promedio para extraer el wireload model:

Se utiliza esta tabla (la cual se incluye como parte de los Liberty files *.lib) para realizar un síntesis futura en la misma tecnología en la cual se estimen los valores promedio de los parásitos a partir del wireload model.



5. Physical layout estimation

En el procedimiento de estimación de parásitos mediante un wireload model existe el problema de que L es el valor promedio de la longitud para un fanout específico pero la varianza puede ser muy importante. Se ha ido observando que a medida que el proceso tecnológico disminuye, la varianza tiende a aumentar. Esto hace que la estimación de parásitos durante el proceso de síntesis tenga mucho error respecto a los valores que estos adopten durante la implementación final.

Una solución a este problema podría ser realizar una iteración entre síntesis y PnR, de forma tal de empezar con una síntesis basada en un wireload model histórico para la tecnología de implementación, implementar el layout del circuito, si hay problemas de timing, extraer el wireload model y con dicho wireload model realizar nuevamente una síntesis que ajuste mejor el tamaño de las compuertas en función de una mejor estimación de parásitos. Este proceso iterativo, sin embargo, no asegura la convergencia ya que los valores extremos de los parásitos en función del fanout pueden estar muy lejos de los promedios.

Las herramientas modernas de síntesis resuelven este problema mediante lo que se conoce como *physical synthesis* (Cadence) o *topographycal synthesis* (Synopsys). Esta metodología implenenta la iteración anterior de forma transparente para el usuario: se realiza una

síntesis basada en un wireload model donde en la misma se dimensiona preliminarmente el tamaño de las compuertas, luego se realiza un placement basado en la información de floorplanning deseado y a partir de este se realiza un *tryial routing* con el cual se extraen parásitos y se redimensionan las compuertas. Si bien no estamos hablando del routing final, se aproxima mucho mejor a este en cuanto a los parásitos estimados para dimensionar las compuertas que el inicial basado en un wireload model.