Appunti di Progettazione Hardware 1

Matteo Gianello

23 settembre 2013

Indice

1 Introduzione

1.1 Progettazione Hardware 1

Nel corso di progettazione hardware 1 si tratterà la progettazione di sistemi complessi, non è solo a livello di VHDL, ma in realtà si tratteranno le metodologie di progettazione hardware e gli algoritmi per tale progettazione.
Si tratteranno diversi aspetti del flusso di progettazione ovvero:

- Sintesi dei sistemi digitali: progettazione e costruzione del sistema
- Collaudo: fase tipica della progettazione hardware che non coincide con la verifica della corretta funzionalità del sistema rispetto alle specifiche.
- Verifica dei sistemi: questa è la vera fase di debug e di validazione rispetto alle specifiche.

I prerequisiti del corso sono la conoscienza degli argomenti trattati durante il corso di Reti Logiche. L'esame si compone di una prova scritta della durata di un'ora e un quarto, composta da 4/5 esercizi per un totale di 32 punti.

1.2 Progettazione Hardware 2

Nato da un corso tenuto da docenti di architetture, tenta di presentare lo sviluppo di hardware e software partendo da una specifica che non definisce la distribuzione dei compiti tra hardware e software.
Dove per software si parla di sistema dedicato. Mentre per hardware si intende una periferica (acceleratori hardware). Gli argomenti trattati nel corso sono:

- Fase di specifica e modellazione di un sistema: modellazione di un sistema fisico di tipo digitale (che cos'è un modello di computazione e comunicazione?) prestanto particolare attenzione ai problemi di determinismo, parallelismo e concorrenza.
- Progettazione e co-design: progettazione contemporanea di hardware e software (Problematiche di partizionamento, mapping delle risorse da usare).
- Co-simulazione a livello di sistema (dipende dal tempo a disposizine del corso).

Preriquisiti di progettazione hardware 2 sono gli argomenti trattati in architetture avanzate dei calcolatori e reti logiche. L'esame è composto da un test a risposta multipla che genera metà della valutazione finale mentre la seconda metà sarà assegnata previa realizzazione di un progetto che può essere di due tipi:

- Sviluppo di uno componente da integrare in un software per la progettazione assistita (CAD) da sviluppare in cpp.
- Ricerca bibliografica su un argomento assegnato dal docente.

2 Introduzione PHW1

Negli ultimi anni la complessità dei progetti hardware è di molto incrementata, mentre i tempi per la loro realizzazione si sono ridotti drasticamente. Per sopperire a queste mancanze la grandezza dei gruppi di lavoro è aumentata in maniera esponenziale, con progettisti con diverse capacità che lavorano in parallelo a diversi livelli di astrazione. La gestione delle complessità e delle comunicazioni di progetto è arrivata ad un punto critico e gli strumenti di progettazione e di sintesi anche se molto evoluti sono inadeguati allo sviluppo di tali progetti costringendo il progettista ad usare fino a 50 strumenti differenti. La sfida principale nella progettazione di un nuovo hardware si focalizzano principalmente sull'individuare il giusto compromesso tra time-to-market e costo/prestazioni. La simulazione è ancora il principale mezzo per la verifica funzionale del progetto ma molto spesso è inadeguata rispetto alle dimensioni dello stesso.
Vi è un thrade of per quanto riguarda i modelli ad alto livello e quelli dettagliati; i modelli ad alto livello sono facili da mantenere ma omettono una grande quantità di informazioni rendendo così le simulazioni molto veloci. I modelli più dettagliati richiedono un partizionamento maggiore aumentando i costi di comunicazione, inoltre, pur essendo più facili da maneggiare e ottimizzare richiedono tempi di simulazione molto lunghi. Per questo i progettisti sono soliti dividere il problema in sottoproblemi più semplici e facili da controllare, dal sottoproblema individuano una soluzione tenedo conto delle condizioni di contorno e costruendo delle interfacce per lo scambio dei dati tra i componenti. Una buona integrazione tra gli strumenti di progettazione e metodologie di progetto comporta comporta indirizzare la complessità algoritmica mediante la suddivisione da parte dei progettisti del problema e l'indirizzamento degli strumenti automatici in base all'esperienza dei progettisti.

3 Metodologie di progetto HW

Nella progettazione hardware si segue un flusso di progettazione che sfrutta diversi livelli di astrazione. Questo flusso permette di partire da una specifica ad alto livello fino ad arrivare alla vera progettazione del sistema. I progettisti non si occupano dello sviluppo del sistema completo ma si innestano a un certo livello per progettare un singolo componente/funzione.
Partiamo perciò dal contesto generale per spingerci poi a livello più piccolo. Ma prima di definire quali sono i flussi di progetto devo definire quali sono gli elementi che compongono questo flusso.

3.1 Dominio di rappresentazione dei circuiti

Dal 1986/87 si è passati dalla descrizione di un sistema come una rete di componenti, alla descrizione di tale sistema attraverso un linguaggio di descrizione. Domini di rappresentazione servono a identificare la complessità della descrizione nei diversi domini. Il diagramma a Y ripartisce lo spazio di progetto in tre parti:

- Dominio fisico: moduli posizionamento piastre cabinet un buon modo per descrivere le fasi del progetto per individuare il punto ne quale ci si trova.
- Dominio funzionale: è il dominio più vicino alle specifiche nel quale io specifico il comportamento del sistema. Si parte da una descrizione a livello di sistema che da una specifica ad altissimo livello descrive il comportamento del sistema (il tempo è trascurato); fino al livello di espressione logiche.
- Dominio strutturale: Uno schema logico del sistema che specifica l'interconnesione dei vari componenti per formare il sistema completo. Si considerano soltanto gli aspetti di interconnesione nel sistema.

I cerchi concentrici servono per individuare i vari livelli di progettazione da quello più esterno che è quello più astratto a quello più interno che è quello più specifico fisico.
Man mano che scendo nei livelli di astrazione il mio progetto diventa sempre più complesso ma comunque gestibili in quanto il sistema ad alto livello non è cambiato.
Il passaggio da una vista ad un'altra è chiamato sintesi; è il passaggio da il livello funzionale a livello strutturale a parte nel caso del passaggio dal dominio funzionale a quello fisico chiamata sintesi circuitale.

3.1.1 Sintesi

Cosa vuol dire fare sintesi? Vuol dire passare dalla descrizione del sistema ad un livello che è il più astratto possibile e via via dettagliarlo scendendo di livello. Partendo dal dominio funzionale a quello strutturale a quello fisico per ogni livello di astrazione. Per la verifica (collaudo) si cerca di anticipare sempre più il momento della verifica in modo da individuare sempre prima la presenza di errori.

System Level : Livello di descrizione del sistema in base alla sue funzionalità, descrizione molto astratta simile a quello utilizzato nei linguaggi di programmazione.

RT Level : modello accurato vicino all'implementazione hardware con costrutti sequenziali per modellizzare costrutti più complessi come il *while*. Manca ancora però l'idea di tempo.

Livello Logico :livello di definizione nel quale vengono descritte le funzionalità del sistema a livello di porte logiche e registri

Transitor Level :Applicazione a livello di transistor (solitamente CMOS); in questo caso possiamo avere diversi modelli in base alle applicazioni, functional equivalent checking o analisi timing accurata (tramite equazioni differenziali).

Layout level : Siamo al livello più basso dove i transistor sono visti come dei poligoni disposti sui diversi strati. Qui sono presenti le metallizzazioni e le diffusioni.

3.2 Fasi di progetto

Le fasi di progetto non sono isolate ma sono in parte sovrapposte o comunque esistono dei circoli tra le fasi. Per quanto riguarda i costi le prime fasi tendono a essere meno costose in quanto le simulazioni sono veloci e semplici; già a livello RTL però i costi aumentano in quanto si aggiungono molte complessità come la temporizzazione. Il flusso di progetto si può dividere in due macrofasi in quanto la seconda fase è già ben definita: front-end che comprende il System level, il Register transfert level e il Logic level; e il back-end che comprende il Transistor, il Layout ed il Mask level.

4 Analisi temporale

Durante il corso di reti logiche abbiamo visto metodi di ottimizzazione che riguardano la minimizzazione dell'occupazione spaziale del circuito; nel caso in cui dovessimo ottimiziare il tempo di esecuzione però le tecniche studiate non sono adatte.
Introduciamo alcune definizioni:

A_k**:** tempo di arrivo (Arrival time)

R_k**:** tempo richiesto (Required time) è il tempo che si richiede per completare la computazione ed avere un risultato stabile all'uscita.

S_k**:** scarto (Slack) ovvero differenza tra tempo richiesto e tempo di arrivo; se questa quantità è negativa allora il progetto non soddisfa i vincoli temporali.

Introduciamo ora due modelli che rappresentano in diversa maniera il ritado temorale. Il primo presenta un grafo con n nodi ai quali viene associato un certo ritardo e nel quale entrano un certo numero di ingressi. In questo caso il massimo tempo di uscita è dato dal massimo tempo di ingresso al nodo più il tempo di attraversamento del nodo $A_k = max(A_1, A_2, A_3) + D_k$.
Nel secondo modello, invece, il tempo di esecuzione è spostato sugli archi di ingresso; anche in questo caso da il tempo di uscita è dato dal massimo tra i tempi di ingressi $A_k = max(A_1 + D_k, A_2 + D_k, A_3 + D_k)$.

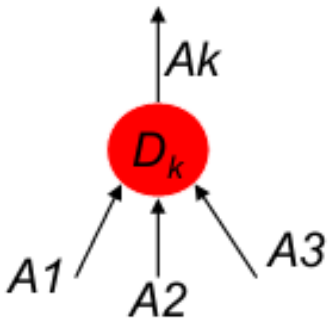

Figura 1: Esempio di ritardo di computazione

4.1 Analisi statica

Per calcolare il percorso critico di un circuito devo fare prima un preprocesso che calcola la massima distanza di ogni nodo dagli ingressi primari tramite l'algoritmo **LEVEL** che parte dai nodi di uscita e calcola in maniera ricorsiva che restituisce il distanza di ciascun nodo nel caso questo sia stato calcolato altrimenti richiama l'algoritmo per ogni nodo di ingresso. Dopodichè si calcola il tempo di arrivo (*arrival time*) tramite l'algoritmo **ARRIVAL** che, sfruttando l'ordinamento topologico del primo algoritmo, calcola successivamente il tempo di arrivo al nodo di uscita partendo da quelli di ingresso.

```
// level of PI nodes initialized to 0,
// the others are set to -1.
// Invoke LEVEL from PO
Algorithm LEVEL(k) { // levelize nodes
if( k.level != -1)
return(k.level)
else
k.level = 1+max{LEVEL(k_i)|k_i fanin(k)}
return(k.level)
}
// Compute arrival times:
// Given arrival times on PI
Algorithm ARRIVAL() {
for L = 0 to MAXLEVEL
for {k|k.level = L}
A_k = MAX{A_ki} + D_k
}
```

4.2 Required Times

Una volta calcolati i tempi di ingresso si procede ad analizzare il require time, ovvero i tempi ai quali è necessario avere i valori in uscita. I require time è una specifica del progetto. A questo punto vorrei sapere anche quali sono i require time all'interno della rete per individuare i punti critici del sistema. Per fare ciò si procede calcolando il require time ad un nodo come require time all'uscita del nodo successivo meno tempo di attraversamento del nodo successivo e tra tutte le uscite prendo il minimo. Il seguente sistema specifica in modo matematico le specifiche del sistema tramite lo slack ovvero la differenze tra require time ed arrival time. Nel caso questo sia negativo ho trovato un percorso critico del sistema.

$$\begin{aligned} S_k &= R_k - A_k \\ Rm,k &= R_k - A_k \\ A_k &= max\{A_{ki}\} + D_k \qquad k_i, m \in fanin(k) \\ S_{m,k} &= R_{m,k} - A_m \qquad k_r \in fanout(m) \\ S_{m,k} &= S_k + max\{A_k i\} - A_m \\ S_m &= min\{S_{m,kr}\} \end{aligned}$$

Un percorso critico è un percorso $P = \{i_1, i_2, \ldots, i_p$ dove $S_{i_k,i_{k+1}} < 0$. Il problema di ottimizzazione della static time analysis si riduce a trovare il percoso con slack negativo più grande in valore assoluto, ovvero minimizzare $max\{-S_i, 0\}$. Veniamo ora al perchè vi è la necessità di effettuare un'analisi temporale del circuito. Lo scopo principale è quello di stimare quando l'uscita del circuito diventa stabile e perciò verificare la correttezza del mio dispositivo. Inoltre l'analisi temporale permette di individuare dove è più efficace effettuare delle ottimizzazioni e vedere dove il sistema non rispetta i vincoli temporali.
Esistono diversi metodi per effettuare time analysis il metodo più grezzo per effettuare l'analisi temporale è effettuare una simulazione con tutti i vettori di input; ma questa tecnica risulta essere molto dispendiosa. Perciò il metodo che utilizzeremo sarà l'analisi temporale a livello di gate.
Prima di procedere però dobbiamo effettuare alcune ipostesi; la prima è quella

Porta	Valore controllante	Valore non controllante
AND	0	1
OR	0	1

Tabella 1: Tabella dei valori controllanti e non controllanti per le porte AND e OR.

di avere una caratterizzazione predefinita dei ritardi delle porte logiche e la seconda quello di effettuare la simulazione a livello logico. I problemi principali di questo tipo di analisi sono l'individuazione dei *false path* ovvero di quei percorsi che non vengono attivati qualsiasi siano i vettori di input, in modo da eliminarli in quanto non vengono mai attivati e concentrarsi su quelli attivi. Altrimenti avremmo una stima per eccesso del ritardo.

4.3 Analisi statica

4.3.1 Analisi dei false path

L'analisi dei false path serve ad individuare quei percorsi che non vengono attivati durante l'esecuzione dei programmi in modo da non doverli conteggiare nel calcolo dei ritardi nei percorsi critici. Questa analisi è complessa ma permette, almeno, di non sottostimare il tempo di ritardo e di sovrastimarli leggermente. Per effettuare l'analisi dei false path devo andare ad individuare le condizioni che mi permettono di tenere attivo tutto il percorso; qlcune di queste condizioni sono molto semplici da individuare se teniamo conto che il nostro circuito è creato da sole porte AND e OR. Introduciamo il concetto di valore controllante per una porta logica; per valore controllante si intende un valore che applicato ad una porta logica permette di conoscere il risultato in uscita senza conoscere il valore dell'altro ingresso. Nel caso di una porta AND il valore controllante è 0 mentre per una porta OR è 1 come mostrato in tabella 1

4.3.2 Statically-sensitizable

Ora però sorge la domanda se il false path individuato non influisce realmente sul ritado di computazione. Per individuare ciò si usa la *statically-sensitization analysis*.
Un percorso si definisce *statically-sensitizable* se esiste un vettore di input che imposta a tutte le porte, che ricevono in ingresso un input, un valore non controllante indipendentemente dal ritardo delle porte. Il percorso più lungo individuato è il *il percorso più lungo attivabile*.
Questa affermazione è vera finche non introduciamo i ritardi delle porte. Infatti è possibile che una porta abbia un valore controllante ad un certo istante di tempo e un valore non controllante in un altro istante di tempo come nell'esempio di fig.2

4.3.3 Timing Simulation

Nella timing simulation si imposta un vettore di ingresso e si simula il circuito tenendo conto dei ritardi delle porte come in figura 3 Questo meccanismo è un po impreciso infatti il ritardo delle porte è una quantità variabile $[0, d]$ dove d è il

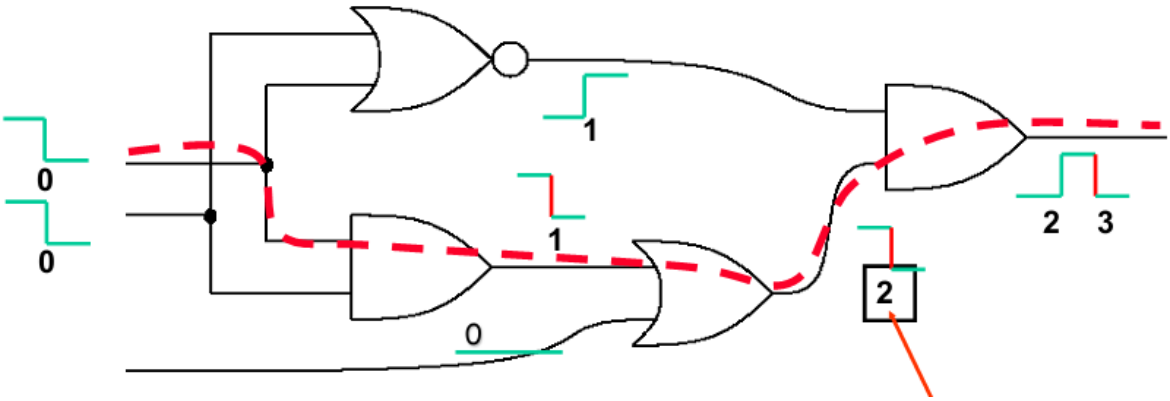

Figura 2: Esempio di valore non controllante ad un certo istante di tempo

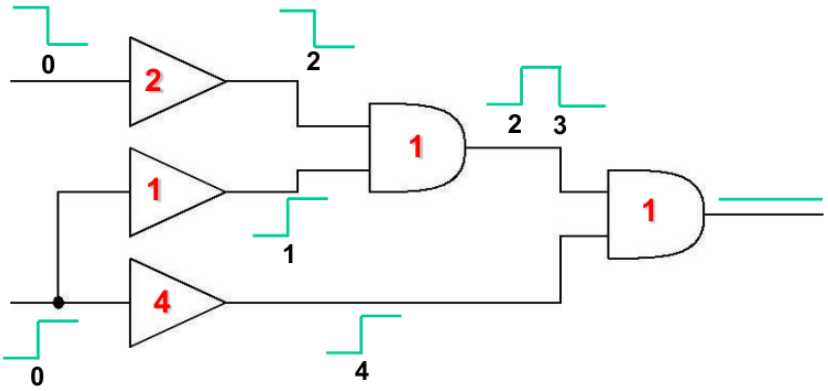

Figura 3: Esempio di timing simulation

ritardo massimo. Come si vede dalla figura 4 diminuire il ritardo delle porte può portare ad aumentare il ritardo complessivo del percorso. Questo aspetto viola la regola dello speedup monotono. Tale regola afferma che dato un circuito C e un circuito Cóttenuto da C riducendo il ritardo di alcune porte allora il ritardo complessivo deve essere minore di quello di C.

4.3.4 Simulazione X-Valued

La simulazione X-Valued assomiglia molto alla simulazione timing in questo caso però si trascurano i valori antecedenti all'istante di inizio simulazione; tale concetto è espresso da una notazione particolare mostrata in figura 5 In questo caso sappiamo che dall'istante successivo alla X il valore è stabile. Questo tipo di simulazione è la più accurata per circuiti logici con porte semplici.
Per conoscere il vero valore del ritardo massimo devo però provare tutti i diversi vettori di ingresso del circuito.

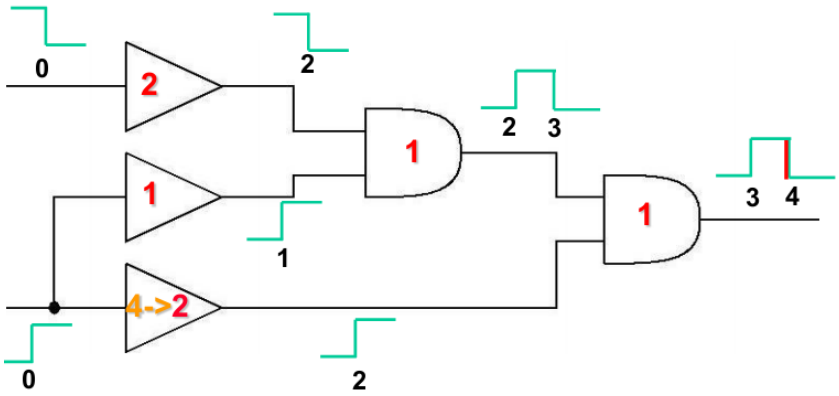

Figura 4: Esempio di timing simulation con riduzione dei ritardi

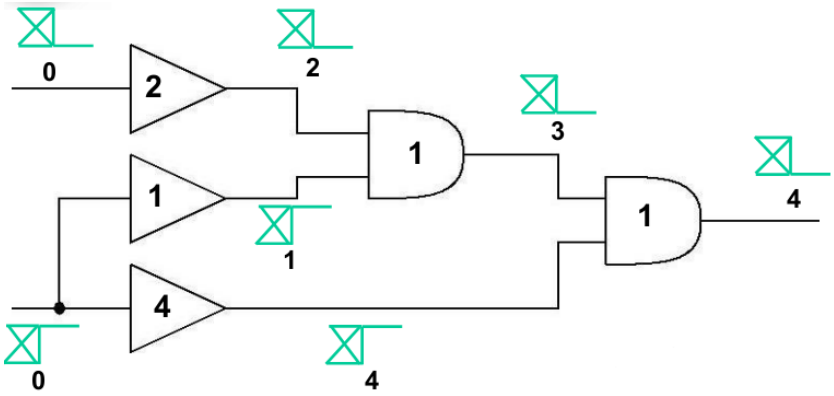

Figura 5: Esempio di X-Value simulation

4.3.5 Algoritmi di analisi dei percorsi critici

Verificare la criticità di ogni percorso di un circuito logico risulta essere troppo costoso; per fare un esempio analizziamo quale sarebbe il procedimento (ricerca per bisezione).

```
Start: set L = L_top /* = topological longest path delay */
L_old= 0
2. Binary search:
If (Delay(L))
!L = |L-L_old|/2, L_old = L, L = L-!L
Else,
!L = |L-L_old|/2, L_old = L, L = L +!L
If (L > L_top or !L < threshold), L = L_old
```

La funzione *Delay(L)* è uguale a 1 se esiste un vettore di ingresso tale che fa si che l'uscita sia stabile solo dall'istante t tale che $L \leq t$. Questo problema può essere risolto con due metodi:

- SAT problem

- timed-ATPG

4.4 SAT based False path analysis

L'analisi dei percorsi critici attraverso il metodo SAT permette di determinare se esistono dei vettori di ingresso per cui l'uscita è stabile al tempo t=T e se questo insieme di vettori comprende l'insieme dei vettori positivi che posso applicare al circuito. Per fare ciò basta prendere la funzione caratteristica di S(insieme dei vettori che soddisfa la condizione) e di S(T) (insieme dei vettori che non soddisfa la condizione) allora se $F \wedge !F(T) = \emptyset$ allora non esiste nessun vettore che rende non stabile l'uscita dopo T (F e F(T) funzioni caratteristiche rispettivamente di S e S(T)). L'idea è quella di verificare se esiste un certo istante T tale per cui $S(T) = S$ dove S è l'insieme di tutti i vettori di ingresso.

Esempio Prendiamo in considerazione il circuito di figura 6 Vogliamo verificare se esiste un qualche vettore di ingresso per cui l'uscita non è stabile ad un tempo $t > 2$. Dividiamo il problema in due sottoproblemi, il primo nel caso

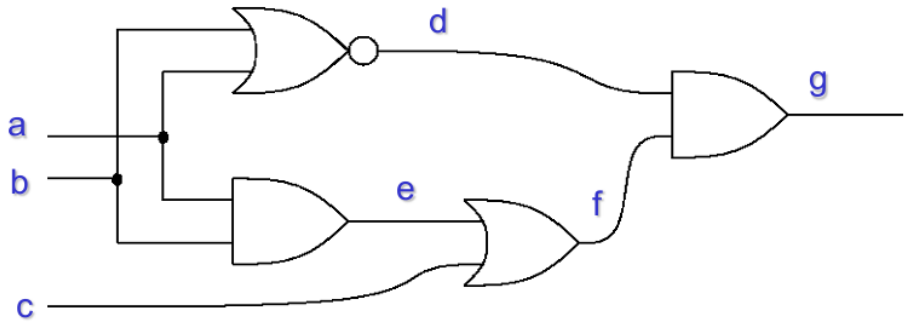

Figura 6: Esempio di analisi con il metodo SAT

l'uscita sia uguale a 1 e il secondo nel caso l'uscita sia uguale a 0 Verifichiamo ora che l'uscita è uguale a 1 al tempo t=2. Per fare ciò calcoliamo la funzione caratteristica $g(1, t = 2)$ che calcola i vettori di ingresso che rendono stabile l'uscita all'istante t=2.

$$\begin{aligned} g(1, t = 2) &= d(1, t = 1) \cap f(1, t = 1) \\ &= (a(0, t = 0) \cap b(0, t = 0)) \cap (c(1, t = 0) \cup e(1, t = 0)) \\ &= !a!b(c \cup \emptyset) = !a!bc = S_1(t = 2) \end{aligned}$$

La funzione caratteristica è una funzione utilizzata per rappresentare in modo insiemistico le funzioni logiche nel quale l'OR è l'unione l'AND l'intersezione. Calcoliamo ora l'onset del circuito ovvero quali valori rendono stabile l'uscita in qualsiasi istante di tempo. la funzione caratteristica dell'onset è indicata da $g(1, t = \infty)$. In questo esempio è:

$$g(1, t = \infty) = !a!bc = g(1, t = 2) = S_1$$

Il fatto che i due insiemi siano uguali fa si che per l'uscita uguale a 1 non esistono vettori di ingresso che rendono l'uscita stabile a un tempo $t > 2$.
Ora dobbiamo applicare i ragionamenti precedenti anche per l'uscita stabile al

valore 0. Quindi calcoliamo la funzione caratteristica all'istante 2 e all'istante infinito.

$$\begin{aligned} g(0,t=2) &= d(0,t=1) \cup f(0,t=1) \\ &= (a(1,t=0) \cup b(1,t=0)) \cup (c(0,t=0) \cap e(0,t=0)) \\ &= a+b+(!c \cap \emptyset) = a+b = S_0(t=2) \end{aligned}$$

Mentre l'offset del circuito è:

$$g(0,t=\infty) = offset = a+b+!c = S_0$$

$$g(0,t=\infty)/g(0,t=2) = (a+b+!c)!(a+b) = !a!b!c$$

Questo significa che per il vettore di ingresso [0,0,0] l'uscita non è stabile al tempo 2.

4.5 Ottimizzazione Combinatoria

I fattori che determinano il ritardo in un circuito si possono suddividere in 3 categorie a loro volta suddivisibili in sottocategorie:

- fattori che riguardano la tecnologia del circuito:
  - Tipo di tecnologia utilizzata per la costruzione del circuito
  - Tipo di gate
  - La dimensione dei gate più grandi danno prestazioni migliori
- fattori che riguardano la struttura logica del circuito:
  - La lunghezza dei percorsi di computazione.
  - I false path.
  - Meccanismi di buffering.
- fattori parassitari:
  - Layout del circuito.
  - Capacità parassite.

Lo scopo dell'ottimizzazione combinatoria è quello di partire dalla descrizione iniziale del circuito con i suoi vincoli, dalla lista delle specifiche e da una serie di librerie e funzioni primitive per arrivare a un'implementazione del circuito con queste librerie che rispetti i vincoli di prestazioni e che sia minimizzata in termini di area.
Esistono diversi approcci per ottenere i risultati richiesti; alcuni approcci sono locali ovvero ottimizzano solo una parte del circuito mentre altri approcci sono globali. I principali approcci locali sono:

- il trasporto in avanti (THR)
- la somma condizzionale (GST)
- il trasporto sul bypass (GBX e KMS)

I primi due puntano sull'ottimizzazione dei livelli e quindi più improntati ad una ottimizzazione dell'area mentre il terzo è improntato sull'analisi dei false path.

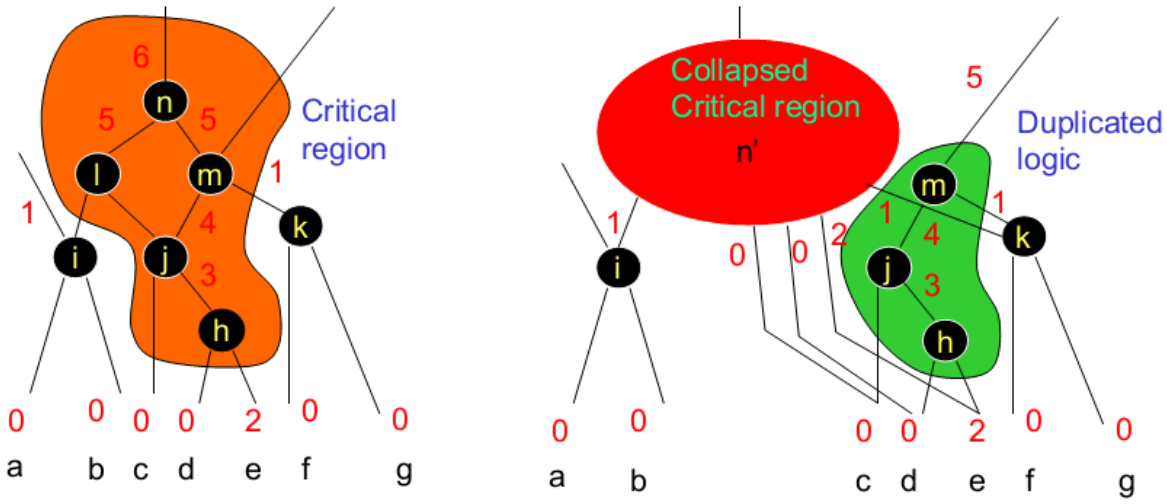

Figura 7: Esempio di THR

4.5.1 Tree-Height Reduction - THR

La prima tecnica che analiziamo ha come principio quello di indivuduare i punti del circuito che violano i vincoli e ridurre questa parte.
Dati gli arrival time all'ingresso e i required time alle uscite individuo la zona critica che non rispetta i tempi; nel caso di fig. 7 in cui la parte critica concorre al calcolo dell'uscita m trovo una sottoparte non critica che può essere duplicata e collaso il percorso che viola i vincoli risintetizzandolo. La risintetizzazione avviene solo su una parte del circuito altrimenti dovrei riprogettare tutto con conseguenti costi di riprogettazione, aumento di area ecc.

4.5.2 GBX e KMS

L'idea del GBX è quella di bypassare il percorso critico inserendo a livello dell'eventuale porta di attivazione del percorso critico un uscita collegata ad un multiplexer e comandata dal valore che attiverebbe il percorso critico rendendo così il circuito più veloce.
Il GBX comporta un piccolo aumento nell'area del circuito ed inoltre introduce un punto di non testabilità sul valore controllante del multiplexer.
Si è così introdotto il meccanismo KMS che rimuove il percorso critico senza incrementare il ritardo. Per fare ciò si duplica il percorso critico fino all'ultimo nodo che contiene un *fanout*; si inserisce inoltre un ingresso controllante sul primo nodo del percorso critico. Si ottiene così che le funzioni fino all'ultimo nodo con fanout non sono cambiate aggiungendo solo una piccola area al circuito.

4.5.3 Generalized Select Transform

Dato il percorso critico con due linee di ingresso delle quali una sola genera il percorso critico, in quanto il segnale arriva con un ritardo maggiore rispetto all'altro ingresso, allora si duplica tale percorso e si impongono all'ingresso dei due percorsi in un caso il valore 0 e nell'altro il valore 1 sulla linea che genera

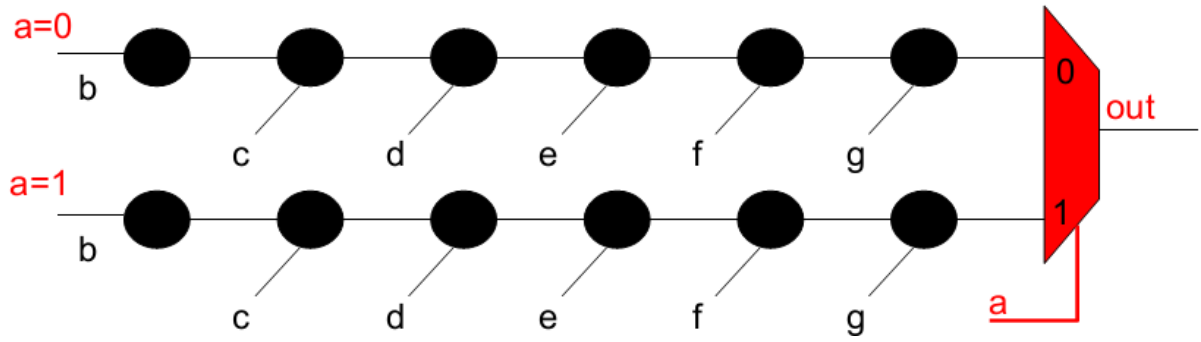

Figura 8: Esmpio di GST

il percorso critico; si inserisce alla fine del percorso un multiplexer controllato dal valore della linea duplicata. Questo meccanismo permette di anticipare il calcolo del percorso critico (vedi fig. 8).

4.6 Ottimizzazione sequenziale

L'ottimizzazione sequenziale tiene conto del tempo del circuito ed implica l'introduzione nel circuito di componenti atti a memorizzare lo stato passato del circuito.
Questo tipo di sintesi può essere applicata a diversi livelli di astrazione allontanandoci sempre più dal livello di sistema e avvicinandoci a quello fisico.
Questo implica un ottimizzazione migliore ma anche una maggior difficoltà di verifica.
Le due tecniche principali di ottimizzazione sono:

- **Clock Skew Scheduling** si tratta di una tecnica che mira a bilanciare il ritardi delle porte attraverso dei registri individuali.
- **Retiming** tecnica che prevede di spostare i registri all'interno del circuito attuando inoltre delle ottimizzazioni combinatorie.

4.6.1 Clock Skew Schedulig

Il clock skew scheduling si basa sul presupposto che la parte combinatoria tra due registri deve terminare la sua esecuzione entro il tempo di clock per permettere il campionamento del segnale di uscita. Più alcune costanti. Un circuito sequenziale sincrono può funzionare correttamente solo se vengono rispettati alcuni vincoli:

$$S_u + D_{max}(u.v) + SETUP_v \leq S_v + T$$

$$S_u + D_{min}(u.v) \geq S_v + HOLD_v$$

Il primo vincolo è detto di *zero-clocking* ed afferma che il segnale deve raggiunge il registro v prima del segnale di clock successivo; il tempo per raggiungere il registro successivo è dato dal tempo di setup del registro di arrimo più il ritardo massimo della parte combinatoria.
Il secondo vincolo, invece, chiamato di *double-clocking* specifica che il segnale in

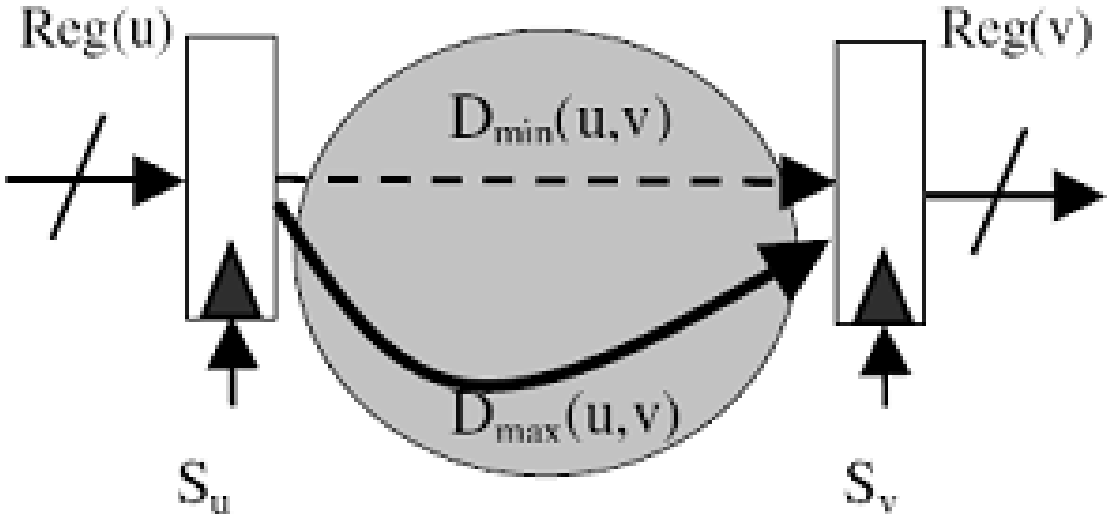

Figura 9: Porta con meccanismo di clock skew scheduling

ingresso e in uscita è sicuro fino all'istante di clock overo che il segnale precedente duri al massimo fino al tempo di clock.
In un circuito sincrono con un clock *zero-skew* S_u e S_v sono uguali e questo implica che il minimo periodo di clock T è uguale al massimo valore di $D_{max}(u, v)+$ $SETUP_v$ presente nel circuito. Se il valore $S_u - S_v \leq 0$ il periodo di clock può essere minore del massimo ritardo del circuito.
Lo scopo dell'ottimizzazione è quello di determinare il valore appropriato di S_u e S_v per tutte le coppie di registri per ottenere il minimo periodo di clock.
I vantaggi di questa tecnica sono una meccanismo di post-sintesi per ridurre il periodo di clock praticamente gratuita e si mantiene il design del circuito. I problemi principali invece sono che i vincoli devono essere rispettati, è impossibile mescolare questa tecnica con quelle di ottimizzazione combinatoria ed infine si tende ad avere una replicazione degli alberi di clock.

4.6.2 Retiming

Il retiming è una tecnica che consiste nello spostare prima o dopo il nodo alcuni buffer gia presenti nel circuito magari accorpando alcuni flip-flop. Questa tecnica risulta essere più semplice da attuare in quanto bisogna tenere in considerazione solo il vincolo di setup del buffer. Inoltre permette un'integrazione con altri metodi di integrazione sia sequenziale sia combinatoria per ottenere un ottimo globale. I punti a sfavore di questa tecnica sono il fatto che rendono più difficile la verifica e comportano un calcolo approssimativo del ritardo del circuito.
Il retiming può avere due obiettivi, il primo quello di minimizzare l'area accorpando i filip-flop e il secondo è quello di ridurre il tempo di clock; questi due obiettivi possono essere ottimizzati insieme e a quel punto si parla di *Retiming*

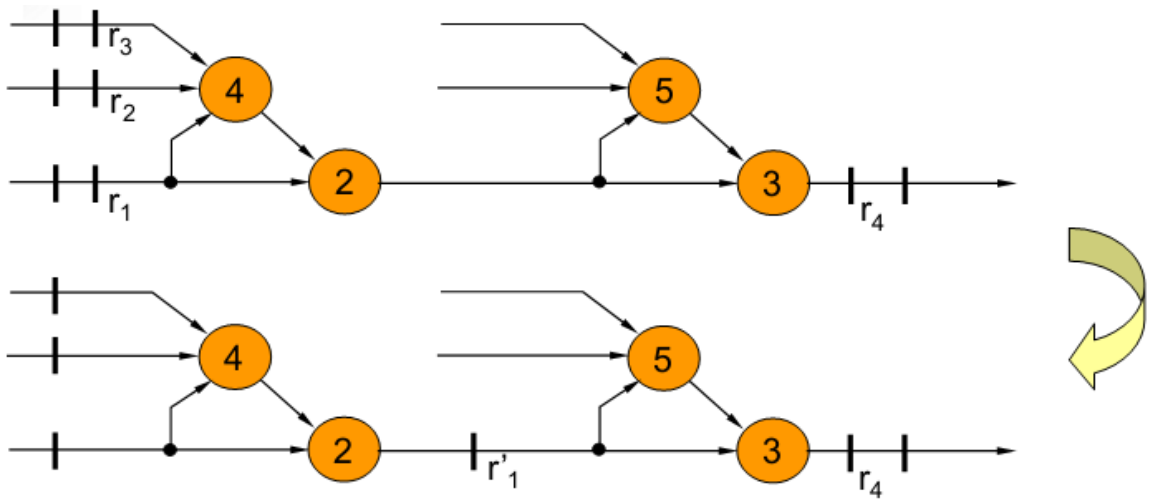

Figura 10: Esempio di retiming

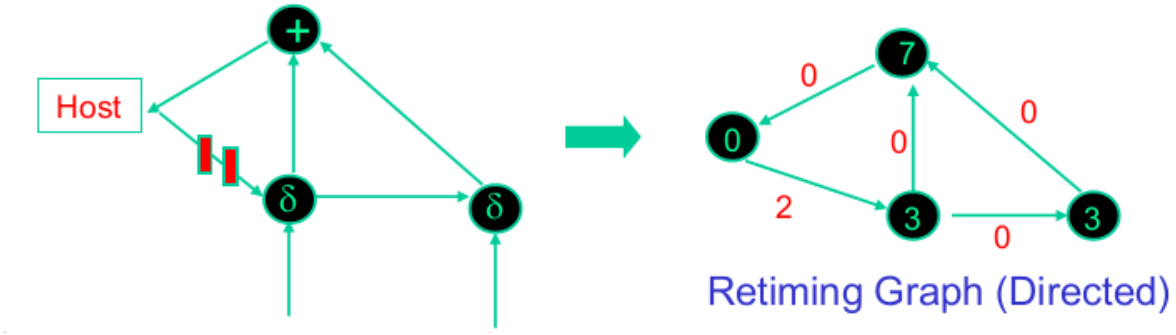

Figura 11: Esempio di creazione di un grafo per retiming

con ottimizzazione combinatoria. Vediamo ora un algoritmo per il calcolo del retiming; rappresentiamo il circuito come un grafo $G(V, E, d, w)$ dove:

- $V \rightarrow$ è l'insieme delle porte
- $E \rightarrow$ è l'insieme delle interconnessioni
- $d(v) =$ ritardo dei gate
- $w(e) =$ numero di registri sull'interconnessione

Vediamo l'esepio in fig. 11 Dove δ ha un ritardo di tre e + ha un ritardo di 7 unità di tempo.Definiamo ora che cosa è un *Percorso*, esso è un insieme di nodi ordinati del grafo con un nodo iniziale e uno finale. Il *ritardo* di un percorso è la somma dei ritadi di tutti i nodi estremi inclusi. Il *peso* di un percorso è la somma dei pesi degli archi ovvero la somma dei flip-flop presenti sul percorso. Dato il percorso:

$$v_0 \rightarrow^{e_0} v_1 \rightarrow^{e_1} \dots v_{k-1} \rightarrow^{e_{k-1}} v_k$$

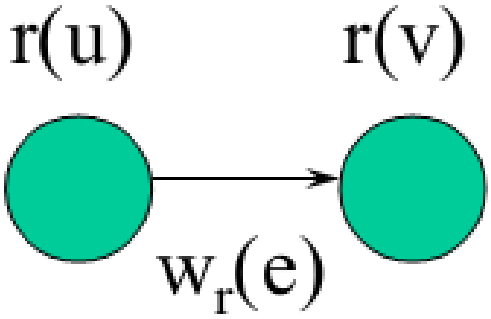

Figura 12: Funzione di retiming

Il suo ritardo è:

$$d(p) = \sum_{i=0}^{k} d(v_i)$$

Ed il suo peso è:

$$w(p) = \sum_{i=0}^{k-1} w(e_i)$$

A questo punto possiamo definire il periodo di clock del circuito come:

$$c = \max_{p:w(p)=0} \{d(p)\}$$

Significa che il massimo dei ritardi di tutti i percorsi con peso 0 definisce il ritardo massimo del circuito e quindi del periodo di clock.
Definiamo ora un operazioni di retimind uguale a -1 se sposto dei registri dagli ingressi alle uscite, un retiming uguale a $+1$ se sposto i registri dalle uscite agli ingressi. Dati due nodi e l'arco ad esso associati definiamo il peso dell'arco dopo l'operazione di retiming come (fig. 12):

$$w_r(e) = w(e) + r(v) - r(u)$$

dove $r(v)$ e $r(u)$ sono rispettivamente le operazioni di retiming sui nodi di uscita e quello di ingresso all'arco in esame. Definiamo ora il problema di minimizzazione del ciclo di clock tramite retiming. La funzione obiettivo è la minimizzazione del ritado massimo

$$\text{minimize } c = \max_{P:w_r(p)=0} \{d(p)\}$$

Definiamo ora due matrici, la matrice dei pesi e la matrice dei ritardi

$$W(u,v) = \min_{p} w(p) : u \rightarrow^{p} v$$

$$D(u,v) = \max_{p} d(p) : u \rightarrow^{p} v, w(p) = w(u,v)$$

Nel caso in esempio abbiamo a che le due matrici sono così composte:

$$W = \begin{matrix} 0 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 2 & 2 & 0 \end{matrix}$$

$$D = \begin{matrix} 0 & 3 & 6 & 13 \\ 13 & 3 & 6 & 13 \\ 10 & 13 & 3 & 10 \\ 7 & 10 & 13 & 7 \end{matrix}$$

La matrice W mi identifica i percorsi critici mentre mentre la matrice D, in corrispondenza dei percorsi nulli della matrice W mi identifica il periodo di clock; mi basta prendere il massimo tra tutti i numeri che hanno come peso il valore 0.
Tutto questo mi aiuta ad ottimizzare il circuito tramite retiming; supponiamo di volere un determinato periodo di clock T a questo punto dobbiamo applicare il retiming su quei percorsi che hanno un valore maggiore di T. Per minimizzare il numero dei registri invece mi basta minimizzare la somma dei pesi degli archi.

5 Tecnology Mapping

Il tecnology mapping è una fase di ottimizzazione che si colloca sotto l'ottimizzazione tecnology indipendent. Questa operazione consiste nell'assegnare alle diverse funzioni logiche del circuito una serie di porte in base a delle librerie che contengono una serie di porte. Queste librerie vengono chiamate *cell library*. Esistono due approcci per effettuare il tecnology mapping il primo è abbastanza euristico basato su regole di sostituzione di tipo tecnology indipendent molto efficente per piccoli circuiti ma richiede un tempo eccessivo di computazione.
Il secondo approccio invece si basa su algoritmi; si divide in diversi passi primo dei quali è trasformare la rete attraverso una serie di funzioni base. Il grafo così formato viene detto *subject graph* tipicamente formato solo da porte NAND a due ingressi e da inverter. Il secondo passo dell'algoritmo è rappresentare anche le funzioni di libreria mediante l'utilizzo di porte NAND e inverter, questo genera il cosidetto *pattern graphs*

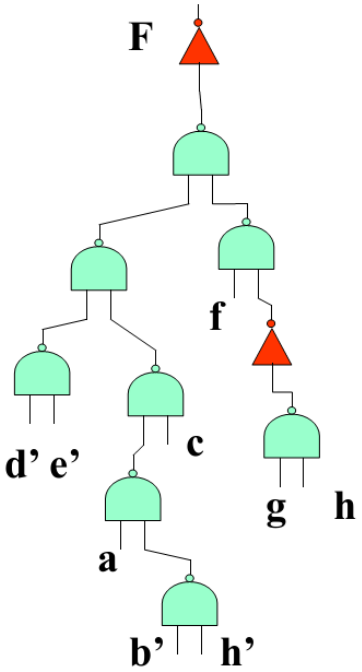

Figura 13: Esempio di subject graph

5.1 algoritmo

Terminata la fase di preparazione si passa alla fase in più propriamente algoritmica. Lo scopo è quello di effettuare la miglior *copertura* del grafo, ovvero, fare in modo che ogni nodo del subject graph sia contentuto in uno o più pattern graph. inoltre ogni input di un pattern graph deve essere l'output di qualche altro grafo.
I diversi algoritmi devono trovare la copertura di costo minimo migliore per il subject graph sottoposto all'analisi.

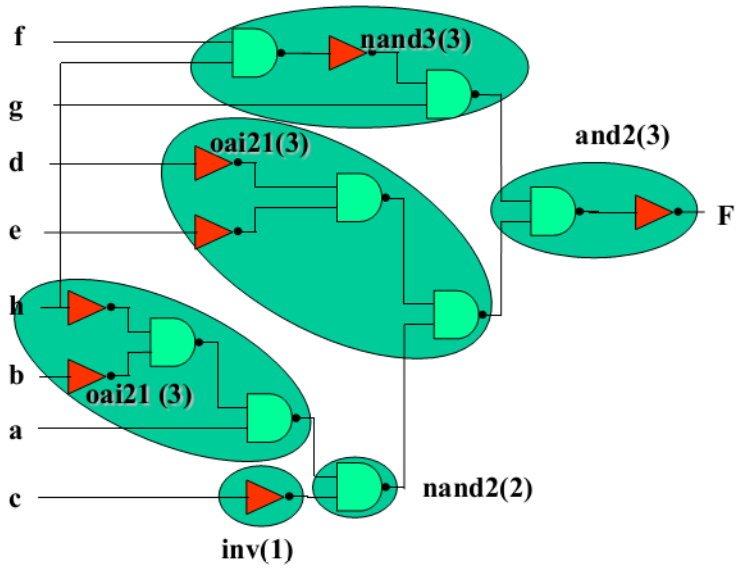

Figura 14: Esempio di copertura con porte AND OR-AND a 2+1 ingressi e NAND a 3 ingressi

5.2 Copertura DAG

Il tecnology mapping utilizzando il metodo DAG richiiede come ingresso la rete ottimizzata e descritta in modo indipendente dalla tecnologia e la libreria di porte da utilizzare. Esso restituisce la netlist di porte che minimizza il costo totale.
Il meccanismo è quello di verificare tutte le possibili combinazioni $\{m_k\}$ per ogni nodo e impostando $(m_i = 1)$ tutte le variabili per il quale il pattern graph è scelto. Dopodiche si scrivono una serie di clausule per ogni nodo del subject graph indicando quali match sono stati effettuati. Ripetendo tale operazione per tutti i nodi e moltiplicando le varie clausule si ottiene la soluzione cercata

	m_1	m_2	$\dots$	m_k
n_1				
n_2				
$\vdots$				
n_l				

Tabella 2: Esempio di tabella di copertura

5.3 Optimal tree covering

Nel caso particolare in cui la nostra rete logica sia ad albero, ovvero non vi è la presenza di fanout esiste un algoritmo più efficente per trovare la copertura migliore.
Il principo è quello di coprire gli alberi in modo ottimale usando la program-

mazione dinamica. L'assunzione che si fa è che per ogni figlio dell'albero si conosce in modo ricorsivo il costo migliore per la copertura e si assume che gli ingressi abbiano costo 0. A questo punto il costo totale è dato dalla somma di tutti i costi.

```
Algorithm OPTIMAL_AREA_COVER(node) {
foreach input of node {
OPTIMAL_AREA_COVER(input);// satisfies recurs. assumption
}
// Using these, find the best cover at node
node->area = INFINITY;
node->match = 0;
foreach match at node {
area = match->area;
foreach pin of match {
area = area + pin->area;
}
if (area < node->area) {
node->owarea = area;
node->match = match;
}
}
}
```

6 Sintesi ad alto livello

I primi processi di automazione del desidn di cicruiti digitali tendevano a dare troppa enfasi agli aspetti incrementali degli algoritmi facendo assunzioni su componenti e ritardi non realistici generando circuiti di basso livello di scarsa qualità La sintesi architeturale di alto livello permette un innalzamento del livello di astrazione delle specifiche portando così a specifiche più snelle e più facilmente modificabili. Inoltre permette di ridurre il tempo di design e un analisi e ottimizzazione del circuito più efficace.
Dati in ingresso una rappresentazione intermedia delle specifiche, un set di funzioni e i vincoli su area e tempi di esecuzione otteniamo in uscita dalla sintesi un data-path che include, *functional resource* ovvero un insieme di risorse che operano sui dati, *memori resources* cioè i meccanismi nei quali immagazzinare le informazioni, ed inifine *interconnection resources.* Inoltre in uscita alla sintesi è presente anche un inisieme di meccanismi di controllo del circuito ovvero una macchina a stati finiti (FSM) un controller con microprogramma e uno schema di sincronizzazione.
Gli scopi principali della sintesi ad alto livello sono, oltre a quelli tradizionali di minimizzazione di aria e ritardo e massimizzazione di velocità di clock, una migliore stima del comportamento del circuito, la testabilità e la sicurezza con la tolleranza ai guasti e l'autotest. I passi per la sintesi a livello architetturale sono:

- Trasformazione di un modello descritto in HDL in un modello IR (rappresentazione intermedia)
- Ottimizzazione del modello in maniera indipendente dalla sua futura implementazione
- Sintesi ed ottimizzazione architetturale.

6.1 Sintesi ad alto livello

è utile rappresentare un circuito con un linguaggio intermedio in qunto questo porta diversi vantaggi; primo fra tutti permette di analizzare il problema prendendo in considerazione sotto problemi più piccoli. Inoltre permette di isolare il front-end dal back-end, ed infine permette di effettuare ottimizzazioni indipendenti dalla tecnologia utilizzata.
Esistono diversi tipi di rappresentazioni intermedie:

- Abstract syntax trees (AST) da una descrizione compatta della specifica in un linguaggio più adatto ad un calcolatore, ma questa rappresentazione è troppo legata al modo di scrivere codice e manca di completezza per quanto riguarda la sintesi.
- Linear operator form of tree (e.g., postfix notation)
- Directed acyclic graphs (DAG)
- Control flow graphs (CFG)
- Program dependence graphs (PDG)

- Static single assignment form (SSA)
- 3-address code
- Hybrid combinations

Diamo ora alcune definizioni per capire meglio la sintesi ad alto livello. Si definisce *blocco base* un insieme di istruzioni che al loro interno non presentano alcun punto di ingresso (eccetto nella prima istruzione), e non presentano salti (eccetto nell'ultima). L'idea è quella che non si può entrare in un blocco base eccetto che dalla prima istruzione e non si può uscirne se non dall'ultima ed ogni operazione al suo interno viene eseguita se e solo se tutte le istruzioni precedenti sono state eseguite. Tutto questo semplifica l'identificazione del controllo.
Il *leader* è quell'istruzione di ingresso in un blocco base ovvero tutte le istruzioni di destinazione di un salto o quelle che si trovano subito dopo un salto condizionale.
Una volta identificati tutti i punti di partenta posso identificare i *basic-block* aggiungendo al blocco tutte le istruzioni che incontro partendo da un leader e fino a quando non incontro un altro leader.

6.1.1 Control flow graph - CFG

Una volta identificati i blocchi base possiamo collegarli con degli archi che rappresentano i salti tra i blocchi; otteniamo un *Control Flow Graph*.
Un *control flow graph* è un grafico nel quale ogni nodo contiene un'istruzione o una sequenza di istruzioni(blocco base) e gli archi orientati indicano un possibile flusso di controllo.
Dato un CFG si dice che un nodo x domina un nodo y se e solo se ogni percorso che da *Entry* arriva ad y contiene x, in questo caso x viene definito *dominatore*. Intuitivamente possiamo affermare che se un blocco domina un altro blocco, nell'esecuzione verrà eseguito prima il blocco dominante.
Questi grafi possono presentare alcuni cicli (*natural loop*) essi hanno alcune proprietà; hanno un singolo blocco di accesso chiamato *header* ed esiste un unico percorso che crea il loop che dall'ultimo blocco base si ricongiunge al primo chiamato*backedge*, e l'unico punto di ingresso del ciclo è lo header. Per individuare il backedge bisogna trovare quel percorso $x \rightarrow y$ nel quale y domina x. Per individuare i loop e semplificare il grafo si usa un meccanismo molto semplice; si individuano tutti i backedge presenti nel grafo, per ogni backedge si caratterizza il loop individuando l'header e i basic block che lo compongono, si uniscono i loop che hanno lo stesso header sommando i rami di backedge e i basic block dei due loop.

6.1.2 Static Single Assigment Form

Questa tecnica ha lo scopo di semplificare la procedura di ottimizzazione globale. Essa afferma che un programma è nella forma SSA se ogni variabile è assegnata al massimo una volta.
Questa analisi si definisce statica in quanto può essere fatto a *compile-time*.

Vediamo ora un esempio di codice normale e il suo equivalente SSA

$$\begin{array}{rcrcr} a & := & b & + & c \\ b & := & c & + & 1 \\ d & := & b & + & c \\ a & := & a & + & 1 \\ e & := & a & + & b \end{array}$$

$$\begin{array}{rcrcr} a_1 & := & b_1 & + & c_1 \\ b_2 & := & c_1 & + & 1 \\ d_1 & := & b_2 & + & c_1 \\ a_2 & := & a_1 & + & 1 \\ e_1 & := & a_2 & + & b_2 \end{array}$$

Tutto questo perchè altrimenti imporrei dei vincoli che in realtà non esistono. Come nel caso del primo pezzo di codice in cui le variabili a devono usare lo stesso registro anche se, come visto nel secondo esempio, sono due variabili diverse.
Nel caso di un costrutto condizionale nel quale non si sa di preciso quale variabile viene assegnata si identifica nel seguente modo:

```
if B then
a1:= b
else
a2:= c
End
a3:= PHI(a1,a2);
```

La funzione Φ si trova normalmente all'inizio di un blocco base e permette di selezionare tra due valori derivanti da un blocco di controllo.
Un'altra cosa da controllare è se le funzioni all'interno del basic block sono nella forma 3-address ovvero nella forma una variabile scritta due variabili lette. L'SSA viene spesso associato al CFG ed esso viene implementato conìme nell'esempio di figura 15

6.1.3 Data flow graph

Il data flow graph mostra il comportamento del modello a livello di operazioni, ed è utile per mostrare il percorso dei dati e per identificare quali operazioni possono essere parallelizzate e quali invece devono per forza essere eseguite in sequenza.
Ad ogni blocco base viene associato un DFG nel quale le operazioni sono i nodi e i dati sono i vertici e gli archi esprimono le dipendenze. Esistono diversi tipi di dipendenze tra i dati:

- **Flow** read-after-write
- **Anti** write-after-read
- **Output** write-after-write
- **Input** read-after-read

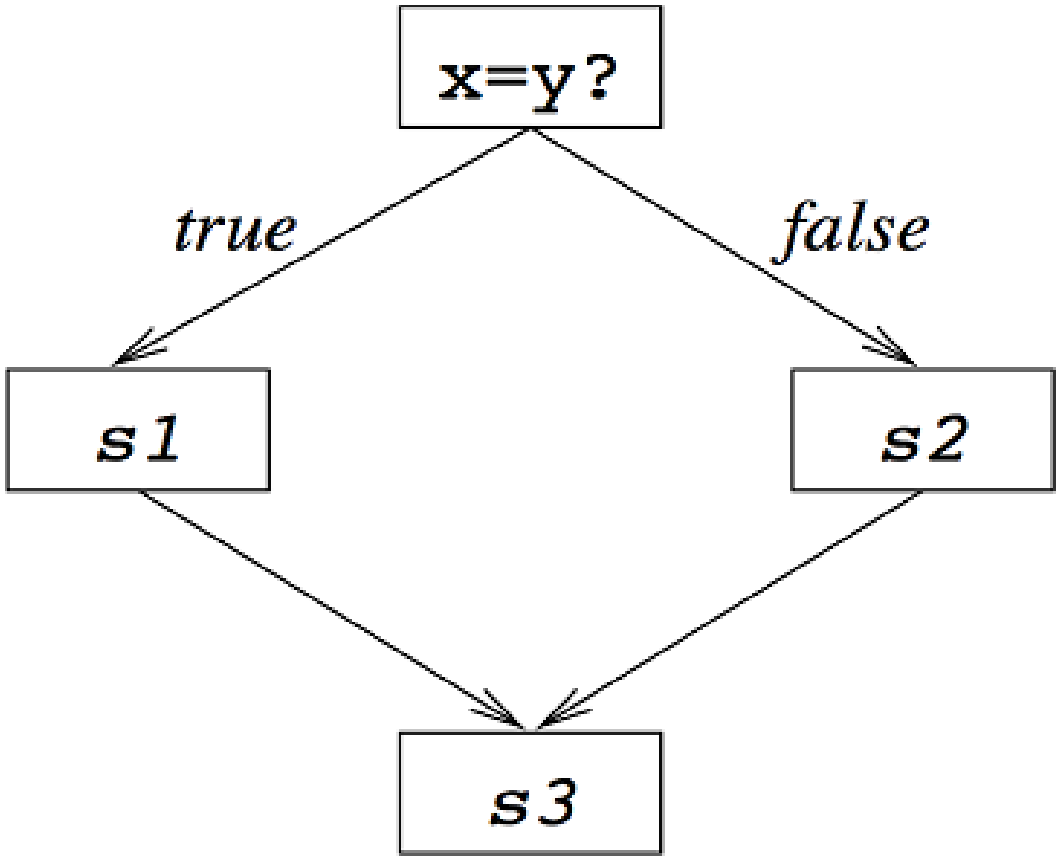

Figura 15: Esempio di applicazione dei meccanismi SSA e CFG ad un programma

Le dipendenze di tipo *Input* non creano vere dipendenze, mentre le dipendenze di tipo anti e output possono essere rimosse mediante l'utilizzo della tecnica di rinomina dei registri (SSA). A questo punto il DFG individua solamente le dipendenze di tipo *Flow*. Un esempio di DFG è mostrato riferito al codice di seguto è quello in fig. 16.

```
xl = x+dx
ul = u-(3*x*u*))-3*y*dx
yl=y+u*dx
c=xl<a
```

In questo caso noi stiamo ragionando a livello di basic block e quindi le precedenze sono riferite solo al loro interno. Combinando però il data flow graph e il control flow graph possiamo individuare le precedenze a livello di intero sistema. Un esempio di CDFG è mostrato in fig. 17

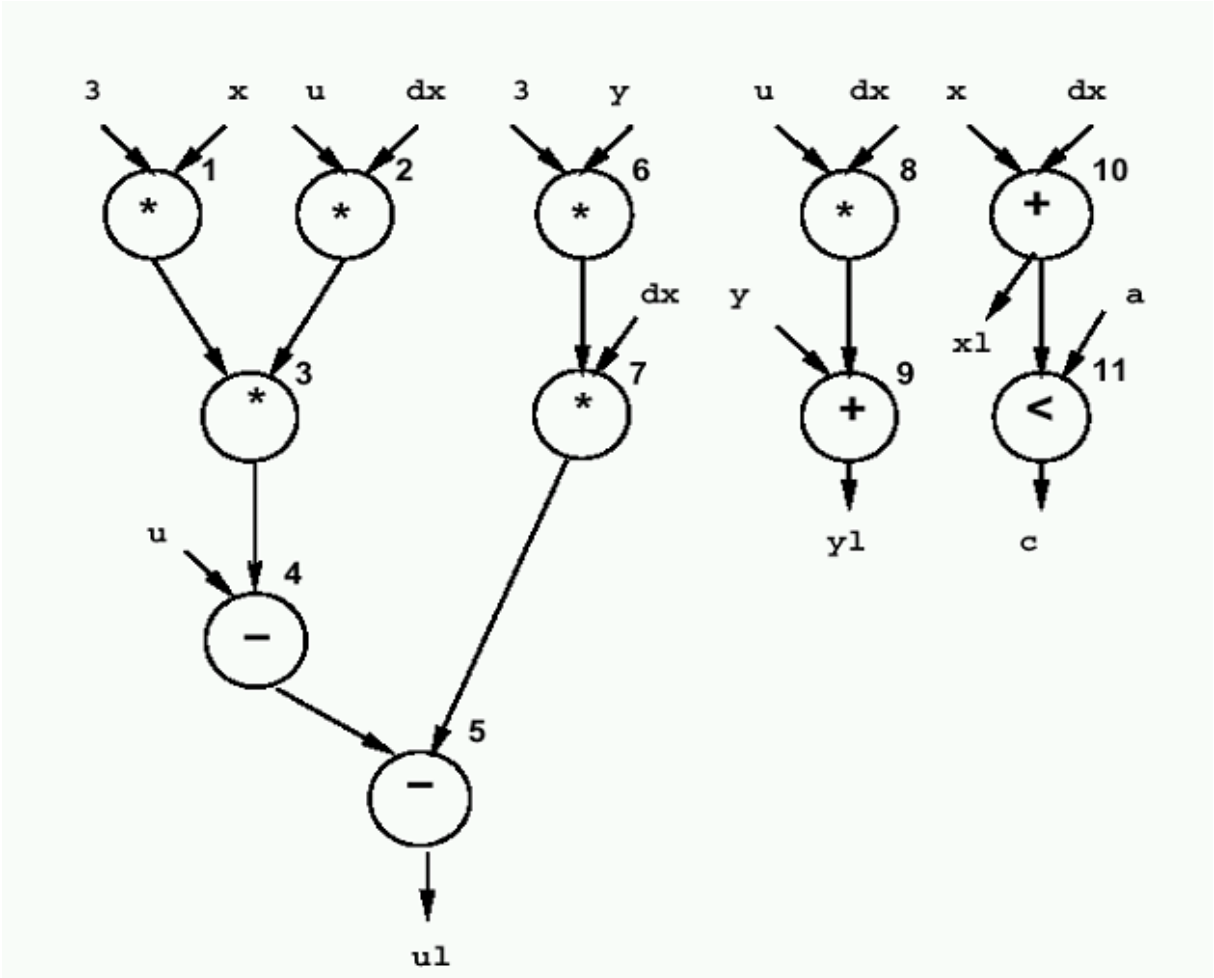

Figura 16: Esempio di Data Flow Graph

6.1.4 Hierarchical Task Graph (HTG)

A questo bunto risulta complesso però gestire le ristrutturazioni del sistema; questo non è un problema per grafi di tipo gerarchico *Hierarchical Task Graph* (fig. 18). I grafi di tipo gerarchico mi permettono di individuare quali operazioni potrebbero essere eseguite in parallelo in modo da aumentare le prestazioni del mio sistema.

6.1.5 Program dependency graph e system dependency graph

Il *PDG* (Program Dependency Graph) rappresenta il funzinamento di una singola procedura in cui i nodi rappresentano operazioni o predicati di controllo mentre gli archi rappresentano le dipendenze dei dati e di controllo.
L'*SDG* (System Dapendency Graph) è una collezione di PDG connessi mediante archi che rappresentano chiamate e passaggi di parametri; esso rappresenta

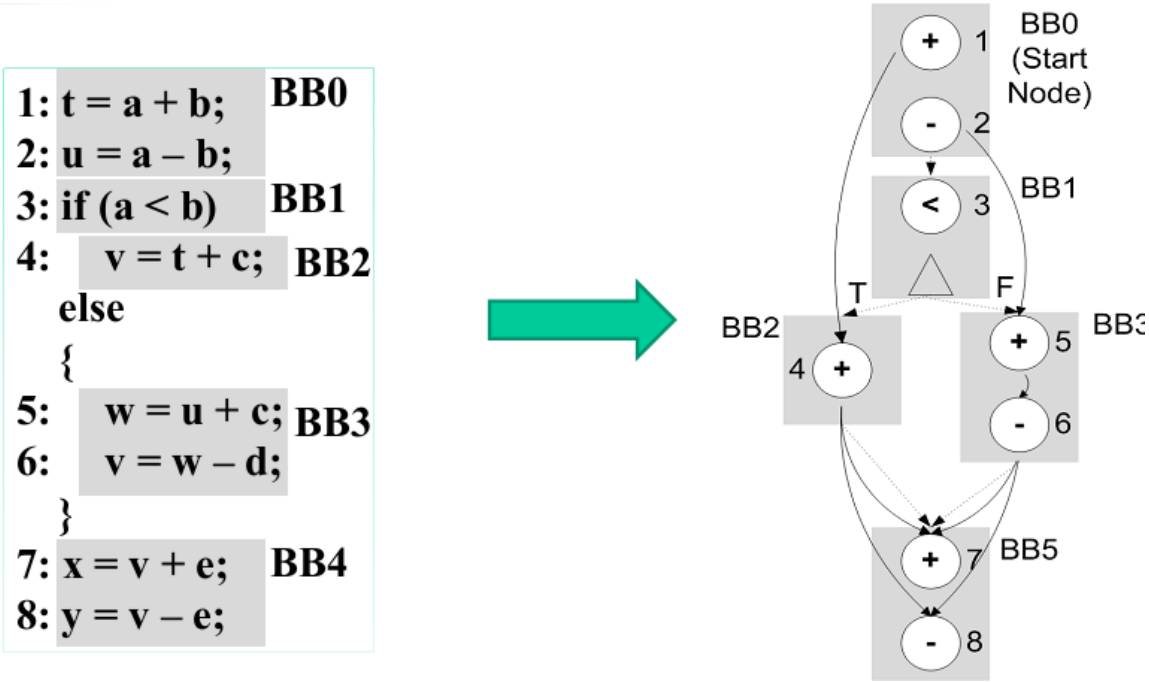

Figura 17: Esempio di CDFG con relativo codice

un astrazione del codice nel quale vengono esplicitate le dipendenze e permette un'individuazione più facile delle parti di codice parallelizzabili

6.1.6 Controllo delle dipendenze

Un nodo A e un dipende da un nodo B se un cambiamento diB fa eseguire o no il nodo A.
Una dipendenza di controllo si definisce se Y è controllato dipendentemente da X se e solo se esiste un percorso P da X a Y nel CFG nel quale non esiste un nodo Z in P che è post-domina Y e Y non è post dominato da X. Tramite questa definizione possiamo distinguere tra le dipendenze fittizie da quelle reali.

6.2 Tecniche di trasformazione

Esistono diverse tecniche di trasformazione che servono a modificare il codice in modo da evitare le dipendenze o prepararlo per una fase di ottimizzazione. Le tecniche principali sono:

- loop pipelining
- dynamic renaming
- copy propagation
- common subexpression elimination
- speculative code motion

- dynamic loop unrolling

Partiamo da un codice di esempio e analiziamo alcune tecniche:

```
while(k<10)
sum+=++k;
```

Loop Pipelining Nel caso di loop pipelining si cerca di eseguire le operazioni all'interno del loop in modo più possibile parallelo o comunque cercando di anticipare alcune operazioni. Dalla figura 20 vediamo come le ultime due operazioni del ramo di sinistra possano essere eseguite in parallelo.

Copy propagation In questo caso si crea una copia di un oggetto che viene propagata in modo da invertire eventuali dipendenze RAW come possiamo vedere in figura 21

Speculative Code Motion La speculazione è una tecnica che modifica il codice muovendo alcuni blocchi di codice dentro alcuni blocchi base per lo più all'interno di percorsi condizionali. Questo permette di aumentare il parallelismo e massimizzare le risorse utilizzate. Esistono diversi tipi di speculazione:

- **Condizionale** nel quale le operazioni eseguite dopo una scelta, ove possibile, vengono portate all'interno delle scelte stesse e la decisione viene presa alla fine dei due rami di scelta.
- **Speculation** Le operazioni che si trovano dentro una decisione e che possono essere eseguite in parallelo vengono eseguite prima della decisione stessa e alla fine viene scelto quale risultato utilizzare
- **Reverse Speculation** In questo caso i due rami di decisione vengono eseguiti contemporaneamente e le operazioni che si trovavano prima della decisione vengono duplicate all'interno dei due rami.
- **Across hierarchical blocks** Operazioni che si trovano dopo due rami di decisione vengono anticipati prima della decisione per aumentarne il parallelismo.

Uno schema riassuntivo è mostrato in fig. 22

Loop Unrolling La tecnica del loop unrolling consiste nel trasformare un normale loop come quello del codice seguente:

```
for(i=0; i<32; i++){
sum=sum+a[i]*b[i];
}
```

nel seguente ciclo:

```
for(i=0; i<31; i++){
sum=sum+a[i]*b[i];
sum=sum+a[i+1]*b[i+1];
}
```

In modo da effettuare una leggera pipelining all'interno del loop e rendere l'albero decisionale più bilanciato come mostrato in figura 23

6.3 Lo scheduling

Per scheduling si intende l'assegnamento dell'ordine di esecuzione alle operazioni nel tempo, possibilmente rispettando vincoli temporali e di hardware sfruttando potenziali parallelismi e cicli.
Per risolvere i problemi di scheduling è necessario avere a disposizione il modello del circuito a livello intermedio (soprattutto un CDFG), il tempo di ciclo e il ritardo delle operazioni, a livello di scheduling è necessario conoscere il tempo di inizio e i vincoli temporali tutto questo tenendo conto del trade-off tra area e ritardo.
Il problema di individuare lo scheduling migliore è un problema NP-difficile esistono così diverse tecniche euristiche per risolvere tale problema, le più comuni sono:

- ASAP (As soon as possible)
- ALAP
- List scheduling, Resource Constrained algorithms
- Force directed algorithms
- Path based
- Percolation algorithms
- Simulated annealing
- Tabu search and other heuristics
- Simulated evolution
- Linear Programming
- Integer Linear Programming

(ASAP) Partendo dal data flow graph in figura 24 pensiamo a come schedulare le operazioni, ovvero assegnare l'inizio dell'esecuzione delle operazioni a determinati istanti di tempo per. Nel caso in esempio v_1, v_2, v_3, v_4, v_10 possono partire subito in quanto ricevono degli ingressi primari, essendo i tempi di esecuzione unitari, all'istante **2** possiamo far partire le operazioni v_5, v_6, v_9 e v_11. Questo scheduling però tende a sprecare risorse come si può vedere nello schema dello scheduling in figura 25. Dato uno scheduling posso creare il controllore creando la macchina a stati che implementa quello scheduling per il singolo basic block, componendo tutte le macchine a stati si crea il controllore di tutto il sistema.

ALAP Duale rispetto all'algoritmo ASAP l'ALAP risolve i problemi dei vincoli temporali. Il tempo di latenza massimo viene impostato nel tempo di latenza calcolato dall'ASAP e l'ALAP sicuramente rispetterà questo vincolo.
Definiamo ora cos'è la mobilità; essa è riferita ad ogni operazione ed è la differenza tra l'istante di inizio dell'operazione in esame nei due scheduling. Nel caso una operazione abbia mobilità uguale a 0 allora significa che lo scheduling di quella operazione in ASAP e in ALAP è uguale. In caso non abbia risorse

necessarie per schedulare le operazioni parallelamente, la mobilità può darmi un indice per valutare quali operazioni schedulare per prime per non incrementare la latenza.

6.3.1 Scheduling complesso

Fino ad ora abbiamo visto lo scheduling applicato ad un singolo basic block ora vediamo dato un CDFG completo come costruire lo scheduling completo del sistema. Come detto prendendo le singole macchine a stati dei singoli basic block e componendole si ottiene la macchina a stati del sistema. Inoltre analizzando le risorse utilizzate dai basic block si ottiene il numero di risorse necessarie per l'esecuzione dell'intero sistema. Si può fare ciò perchè i singoli blocchi sono in mutua esclusione.

6.3.2 Scheduling con vincoli

In molti casi dobbiamo rispettare alcuni vincoli temporali per quanto riguarda lo scheduling; questi vincoli possono essere sia di tipo massimo ovvero non posso superare un certo limite temporale, ma anche di tipo minimo ovvero non posso completare l'esecuzione prima di un determinato tempo.
A questo punto io posso rappresentare questi vincoli nel DFG (fig. 27) questo permette, tramite opportuni algoritmi, di risolvere il problema dello scheduling.

6.4 Resource Binding

Lo scheduling trascura un componente fondamentale della progettazione, ovvero la distribuzione delle risorse. Il *Resource Binding* può essere effettuato in maniera molto banale assegnando alla prima risorsa libera l'operazione da eseguire, ma più efficentemente si può applicare alcuni meccanismi per l'individuazione dell'allocazione delle risorse e la costruzione del Data Path.
Bisogna anche stabilire se il nostro datapath possiede *resource sharing* che non è così ovvio nel caso di progetti molto grandi e con costi di interconnessione molto alto.

6.5 Register allocation

A questo punto dell'analisi abbiamo costruito il controllore del sistema e le risorse necessarie per eseguire le operazioni. Ora però dobbiamo rendere disponibile i dati alle risorse; per fare ciò dobbiamo effettuare la *Register Allocation* ovvero stimare quali sono i valori che devono essere memorizzati e individuare quali sono i registri compatibili per memorizzare i dati.

6.5.1 Scheduled data flow graph

Lo *Scheduled Data Flow Graph* non è altro che un Data Flow Graph nel quale vengono evidenziati i confini delle operazioni come si vede in figura 28 In questo schema le linee tratteggiate rappresentano il tempo di inizio e di fine del ciclo di clock e tra un ciclo di clock e l'altro i valori devono essere memorizzati. Perciò per ogni operazione possiamo rifarci allo schema in figura 29 dove possiamo distinguere i registri di lettura e quello di scrittura. Questo è il caso generale

per individuare quali variabili debbono essere memorizzate ma esistono altri metodi per individuare i registri, ad esempio:

Variabili: in un codice (possibilmente scritto in 3-address form) tutte le variabili rappresentano un registro.

Registri dichiarati: ovvero i registri vengono dichiarati dai progettisti

Valori: è quello che abbiamo già visto ovvero che ogni arco rappresenta un valore da memorizzare e quindi un registro.

Istanze di valori: Aggiungere dei valori identici in caso di archi che attraversano più cicli di clock. Questo aumenta il grado di libertà sui registri

Istanze limitate di valori: Simile all'istanze di valori quelle limitate applicano la copia dei valori un limitato numero di volte.

A questo punto abbiamo capito quali sono i possibili registri che dobbiamo allocare come si vede nello schema di figura 30

6.5.2 Il problema dell'allocazione dei registri

Il problema che ci poniamo, una volta individuati quali sono i valori da memorizzare, è quello di assegnare ad ogni valore un registro fisico minimizzando il numero dei registri.
Nel caso di rappresentazione in SSA automaticamente abbiamo l'insieme degli storage value.
Per risolvere il problemanegli anni si è visto che il metodo più efficave è l'algoritmo di colorazione dei grafi.
Vediamo ora un esempio

```
a := c+d;
e := a+b;
f := e-1;
```

Come si può vedere la variabile *a* viene scritta nella prima istruzione e viene letta l'ultima volta nella seconda istruzione, si dice che il suo tempo di vita è di una istruzione; per vedere se due storage value sono in conflitto bisogna vedere se i loro tempi di vita si intersecano. A questo punto riusciamo a costruiro un grafo di interferenza tra gli storage value come quello in figura 31 e 32 A questo punto se due storage value hanno archi che li uniscono allora non possono utilizzare lo stesso registro. In questo caso si può utilizzare l'algoritmo di un grafo per trovare in maniera ottima il numero di registri minimo.

6.6 Rilascio delle risorse semplificatrici

Fino ad ora abbiamo assunto alcune ipotesi per semplificarci le spiegazioni, ora però vediamo alcune di queste e come eliminarle.
La prima ipotesi è che tutte le operazioni avessero lo stesso ciclo di clock. Nella maggior parte dei casi però questo non è vero, un esempio è dato dal fatto che il moltiplicatore impiega più tempo di un sommatore come si vede in figura 33, esistono però alcune tecniche come il *multicycling* dove un operazione può impiegare più di un ciclo di clock questo mi permette di dimezzare il ciclo di

clock; il *chaing* consiste nel mantenere lo stesso ciclo di clock e di eseguire le operazioni più veloce nello stesso ciclo di clock; il *pipelining* consiste di eseguire due operazioni lente in due istanti di clock successivi senza che la prima sia conclusa.

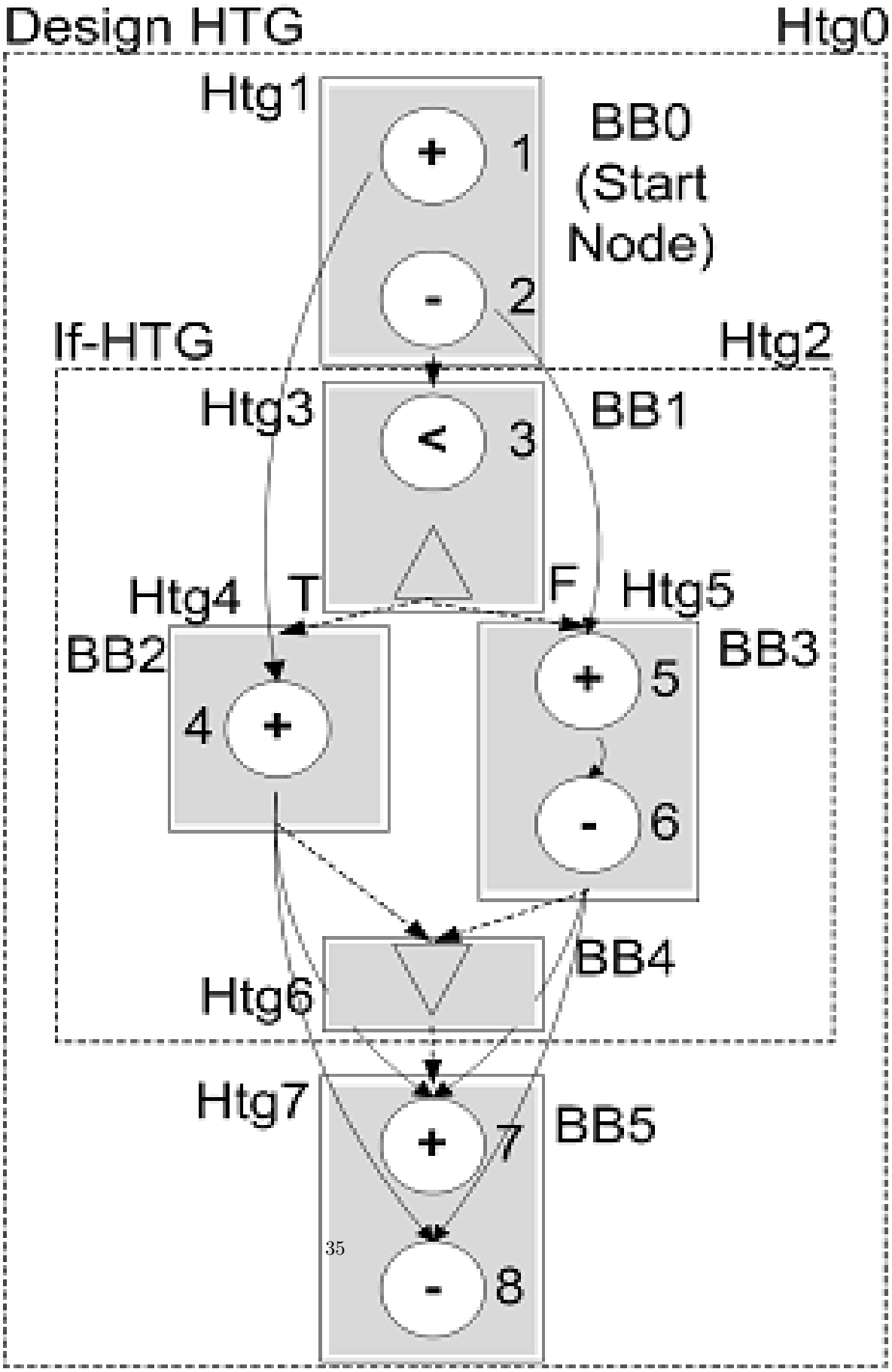

Figura 18: Esempio di grafo gerarchico

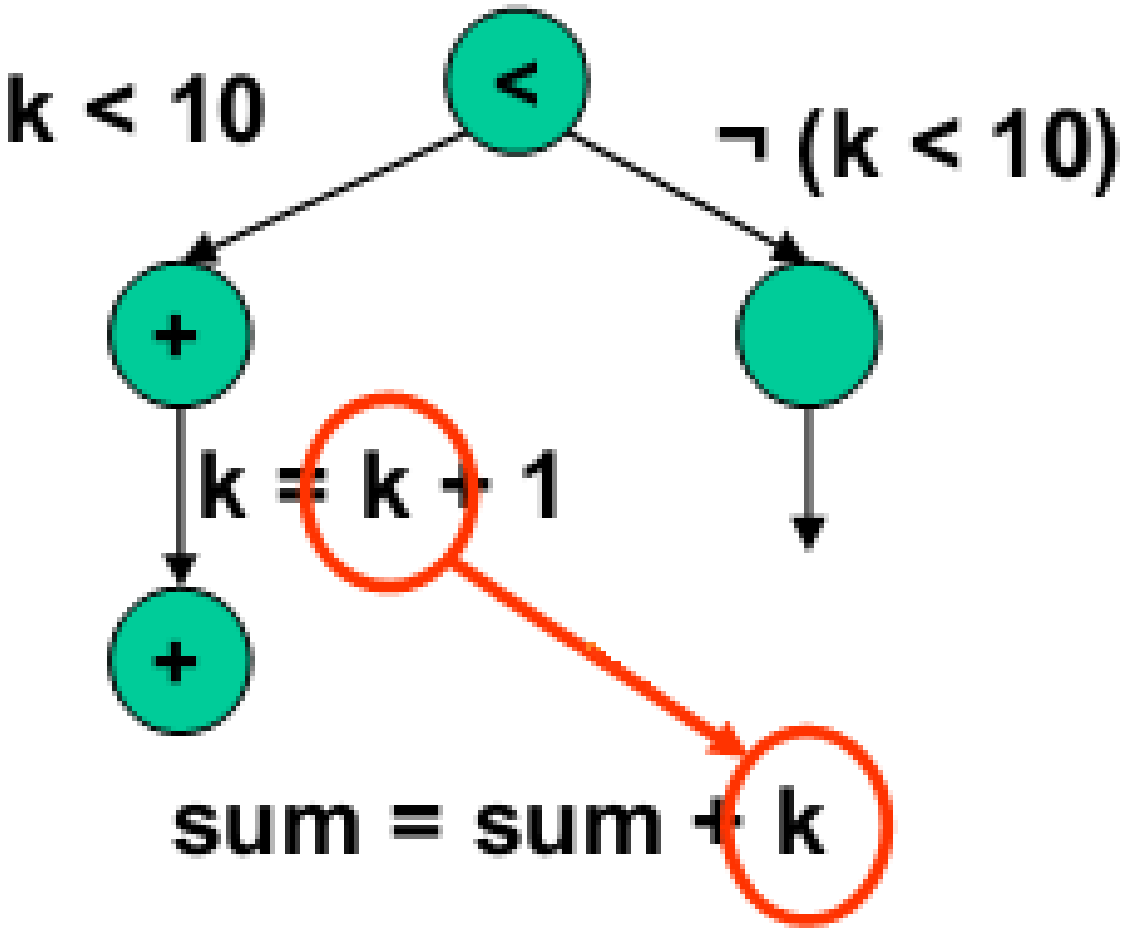

Figura 19: DFG del codice di esempio

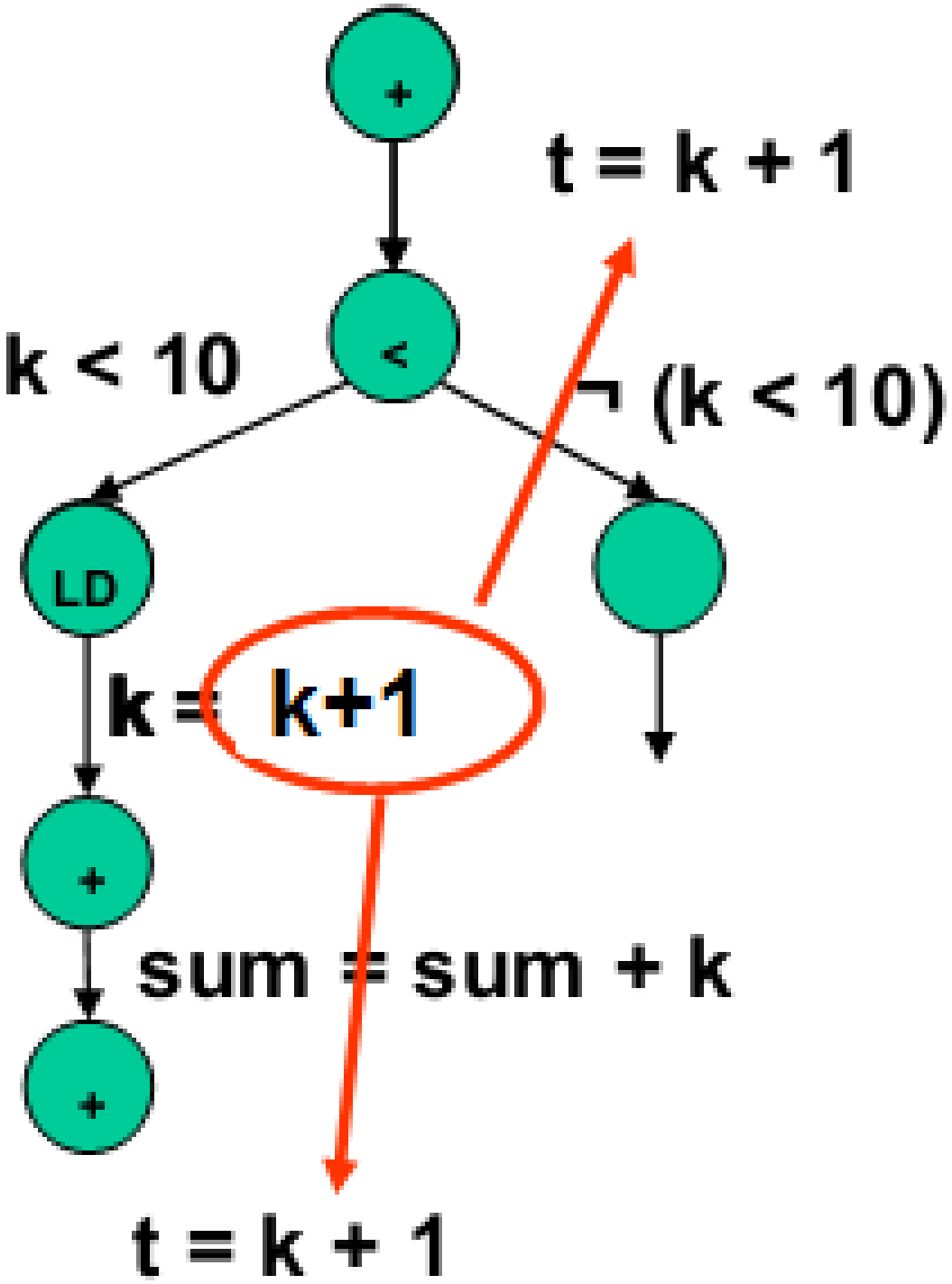

Figura 20: Esempio di loop pipelining

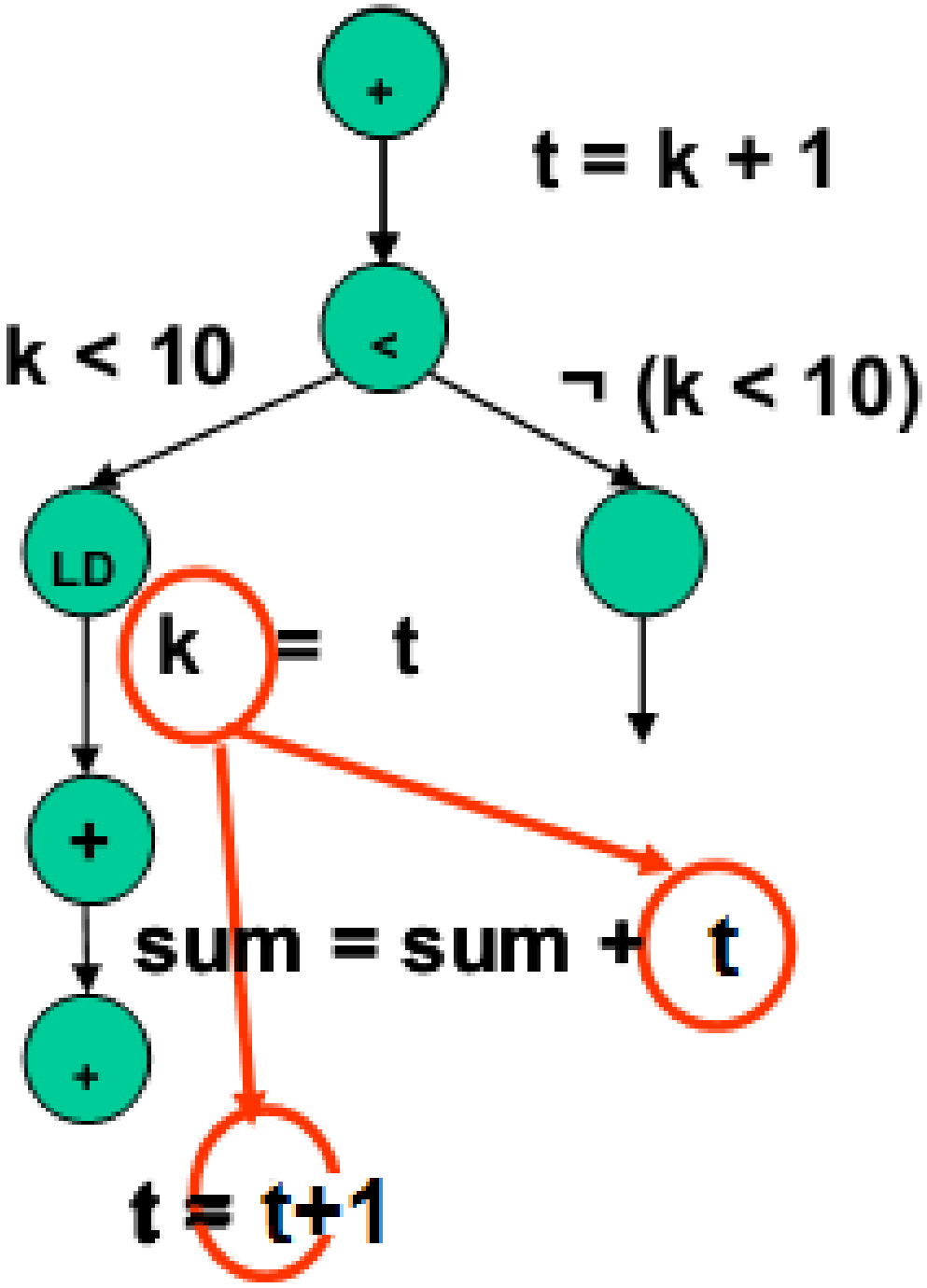

Figura 21: Esempio di copy pipelining

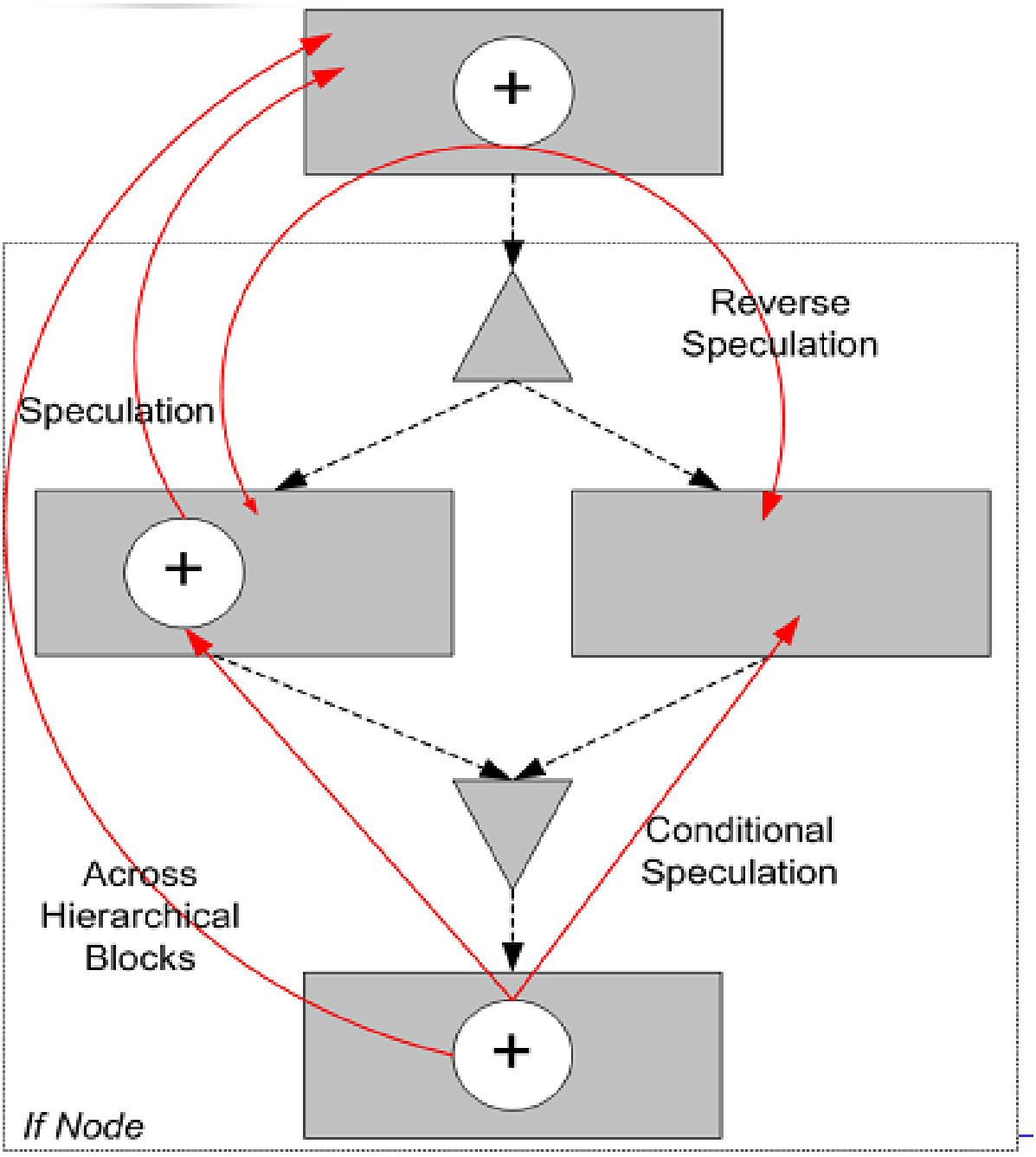

Figura 22: Schema riassuntivo sulla speculazione

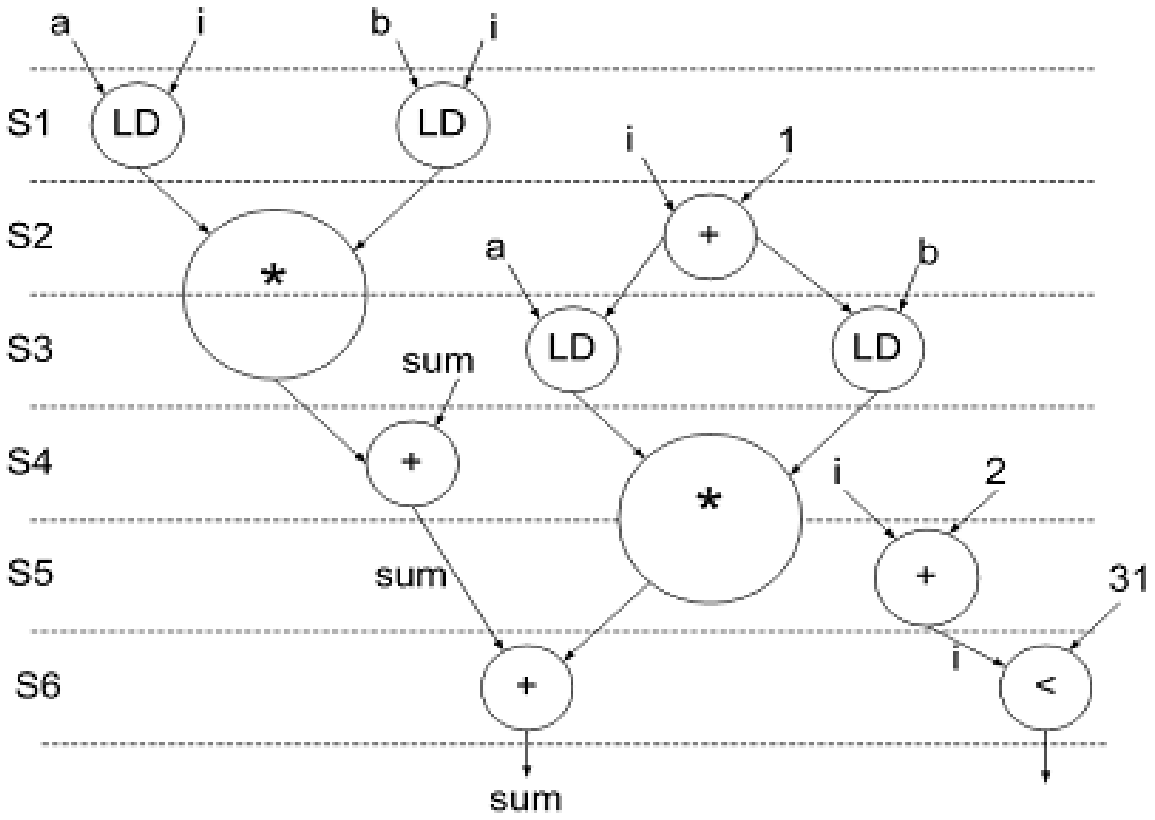

Figura 23: Schema di loop unrolling

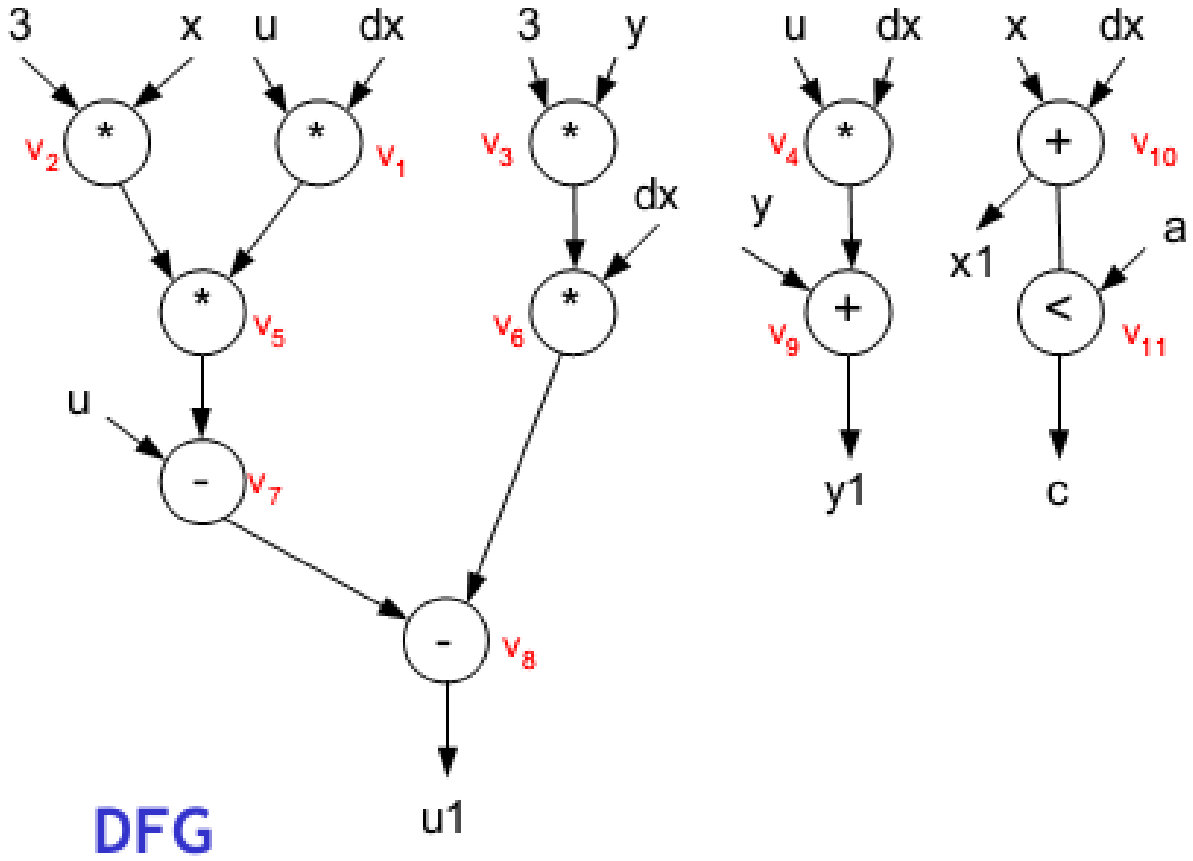

Figura 24: Control Data Flow Graph

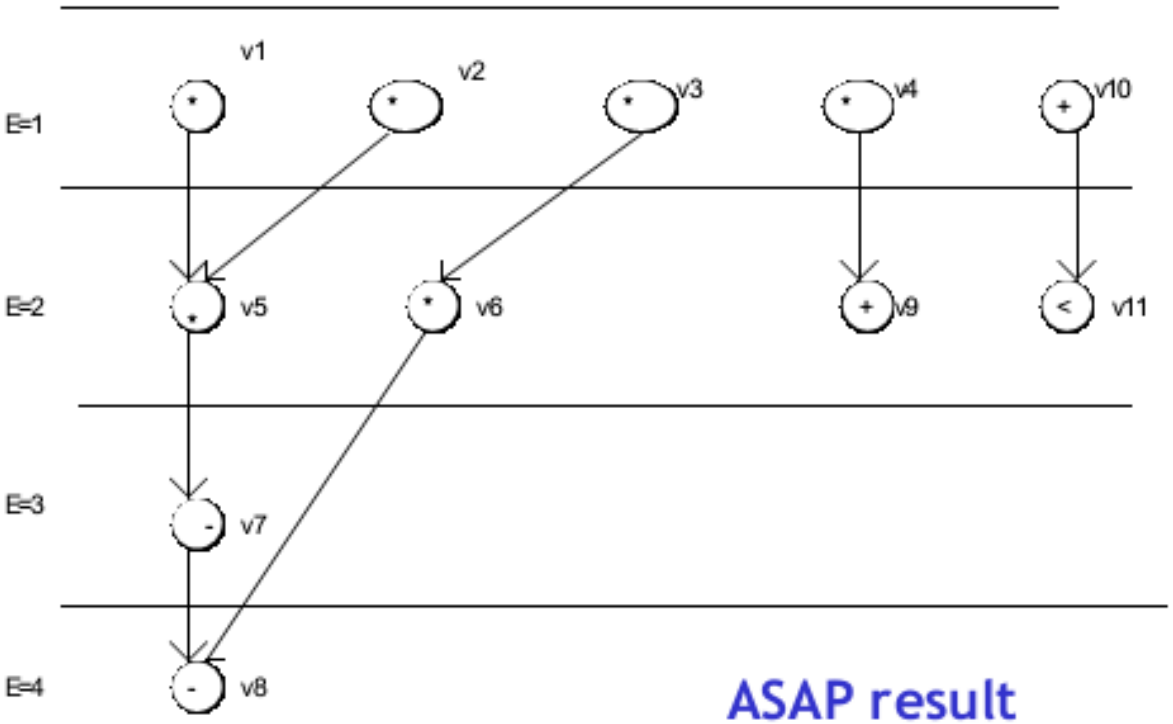

Figura 25: Scheduling ASAP

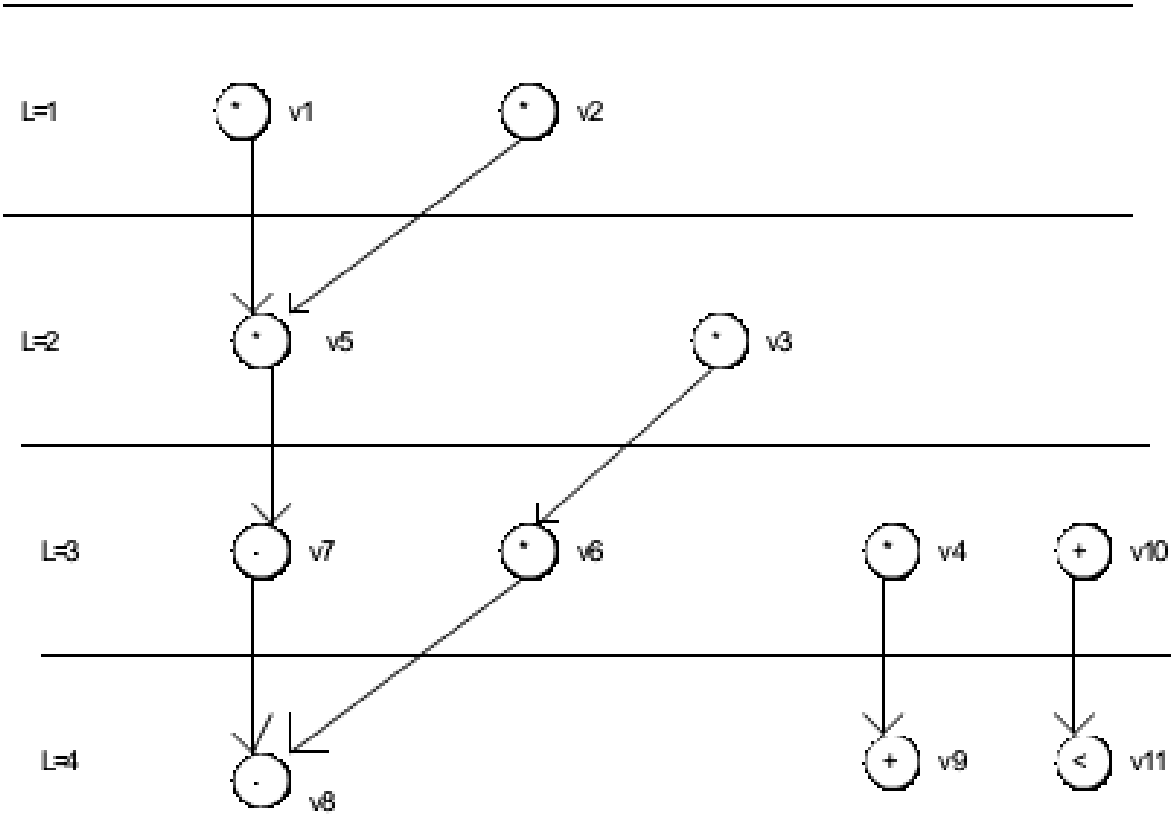

Figura 26: Scheduling ALAP

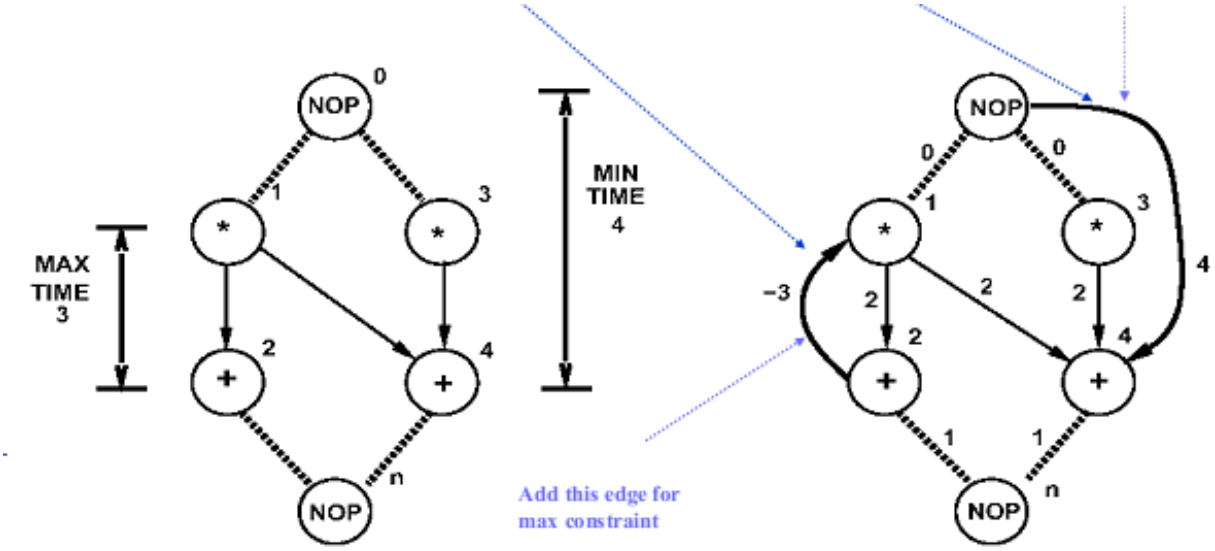

Figura 27: Esempio di DFG con vincoli temporali

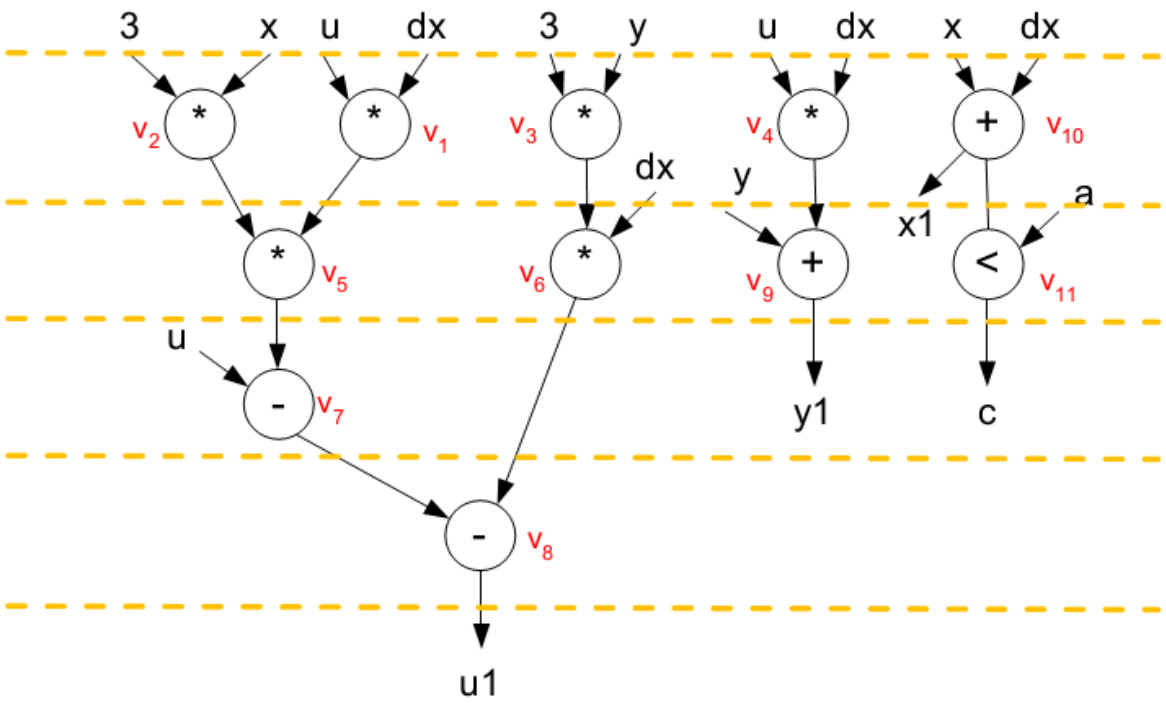

Figura 28: Scheduled Data Flow Graph

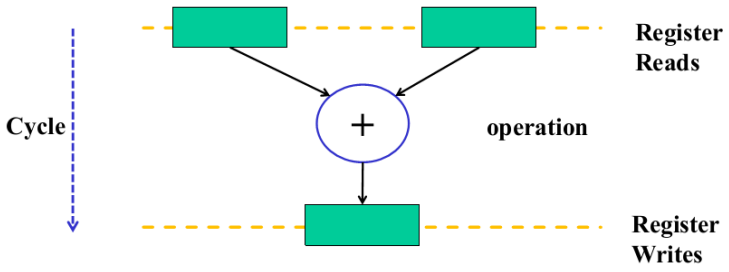

Figura 29: Schema dei registri

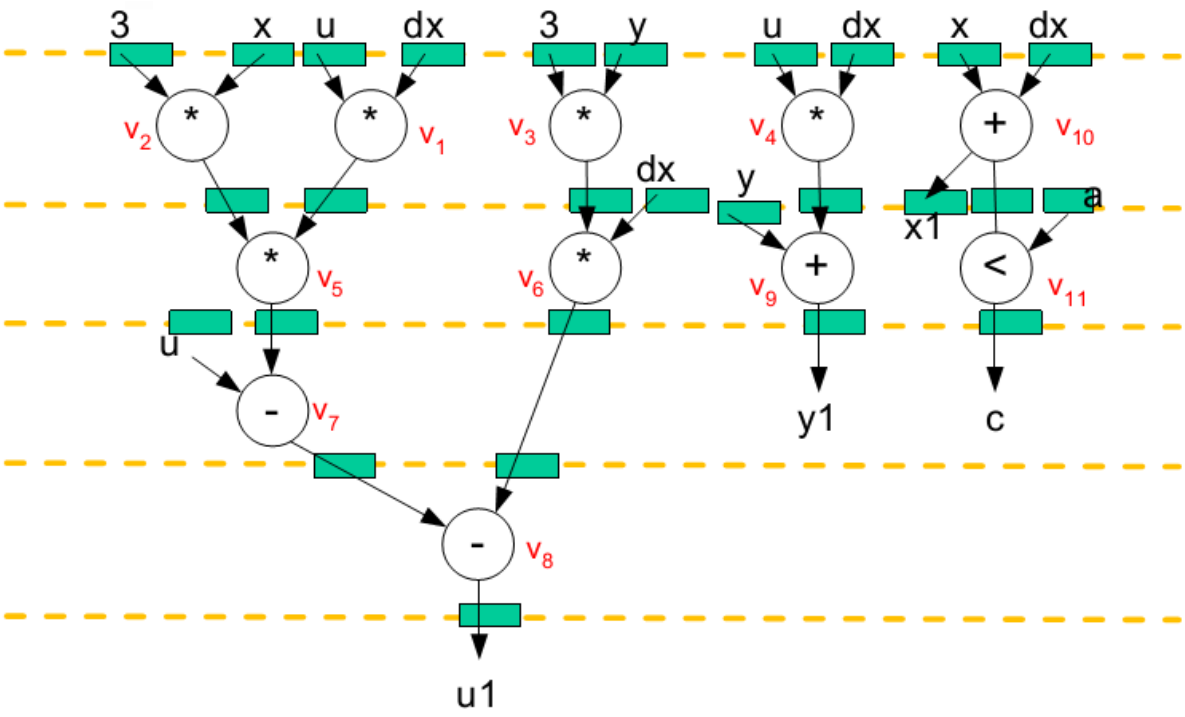

Figura 30: SDFG con registri

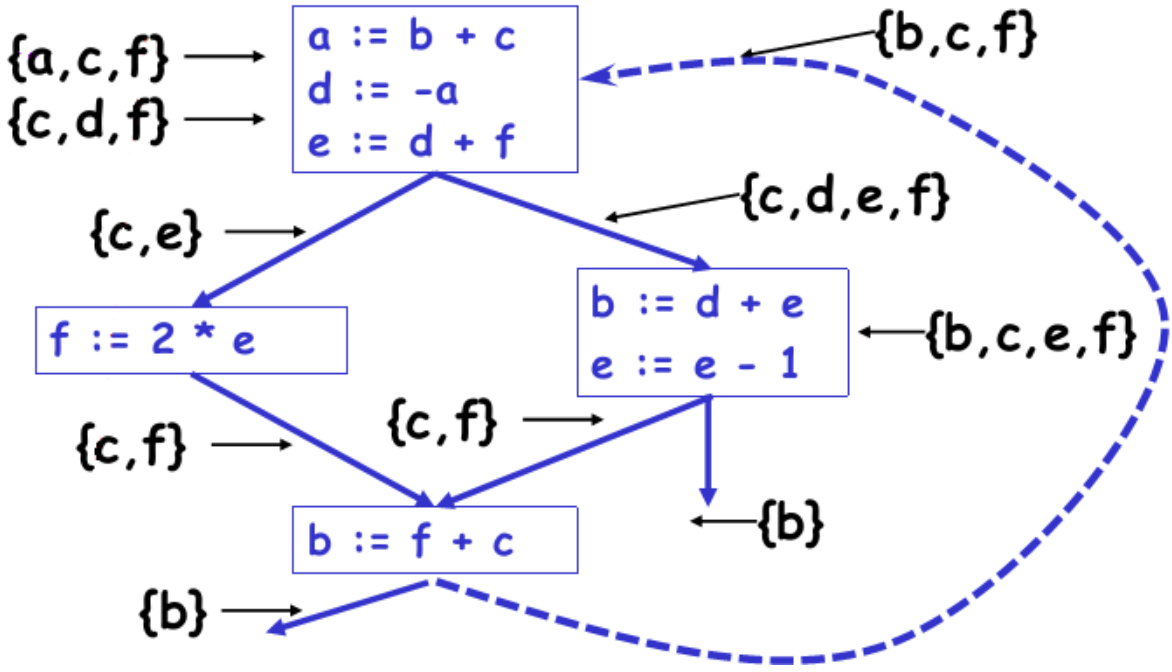

Figura 31: Esempio di grafo di interferenza

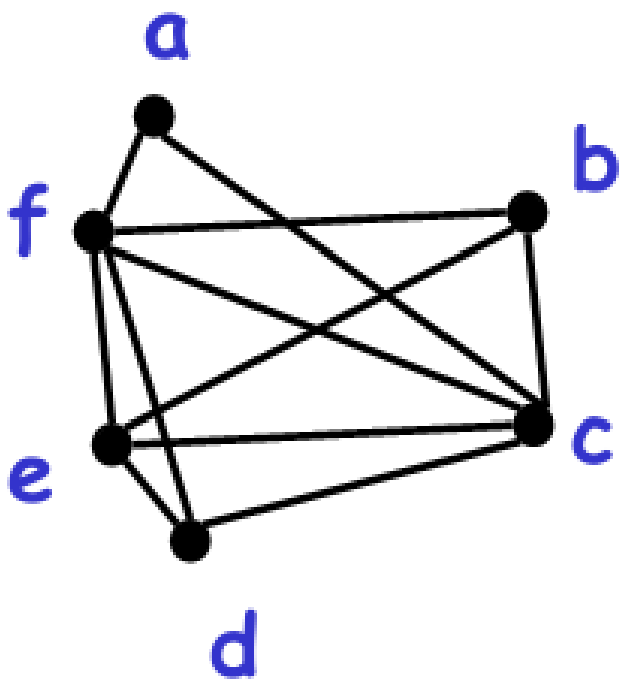

Figura 32: Esempio di grafo di interferenza

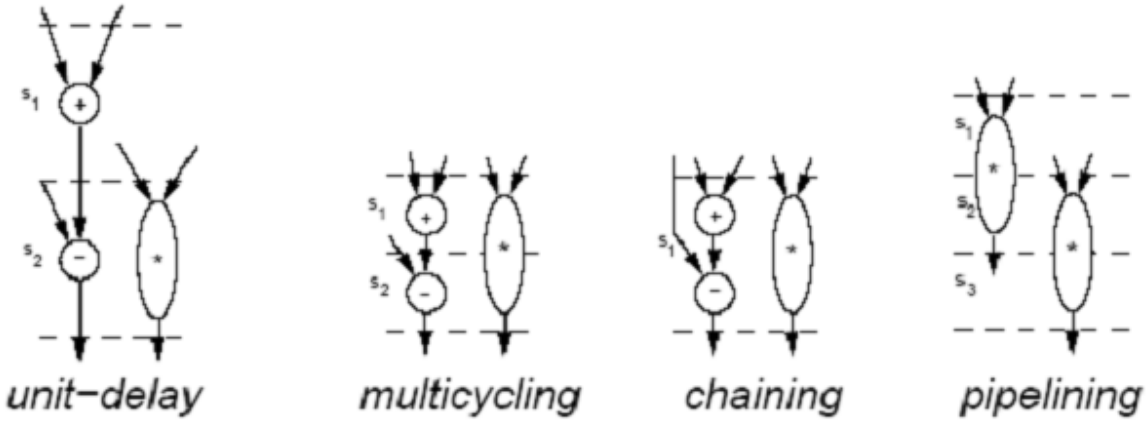

Figura 33: Ottimizzazioni per operazioni multiciclo