Sintesi ad alto livello dei sistemi digitali

Linguaggi di descrizione dell'hardware

M. Favalli

DE: Department of Engineering Ferrara

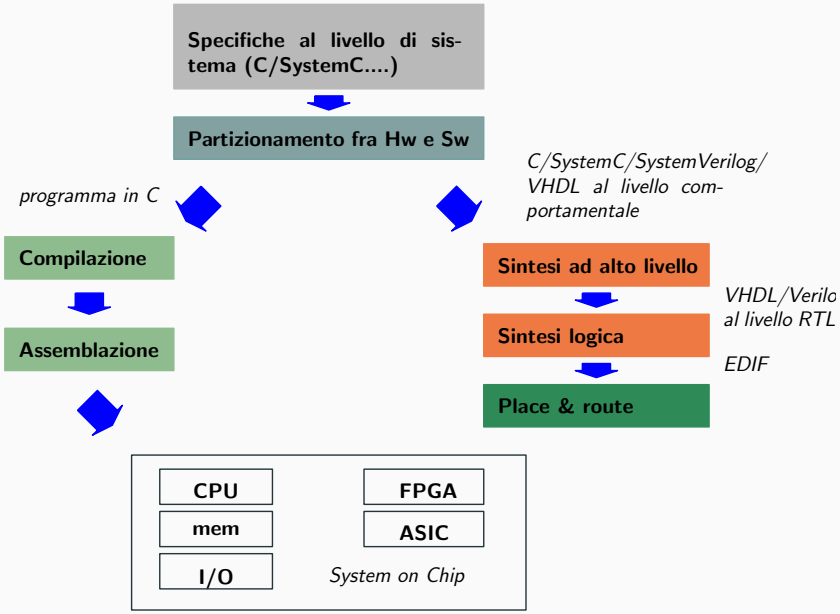
Motivazioni

- Presentare i concetti base della sintesi ad alto livello
- Introdurre le strutture dati utilizzate nella sintesi ad alto livello
- Introdurre i problemi relativi ad allocazione, scheduling e binding
- Descrivere alcuni algoritmi di sintesi ad alto livello
- Fornire alcuni esempi

Sommario

Introduzione

Sintesi di un sistema digitale



Introduzione

- **Realizzazione hardware di un algoritmo**
  - motivazioni e applicazioni
  - parallelo con il software
  - obbiettivi
- Ottimizzazione prima della sintesi
- Strutture dati per rappresentare un algoritmo
  - Control Data Flow Graph (CDFG)
- Applicazione alla valutazione di un espressione aritmetica
  - Data Flow Graph (DFG)
  - costo - risorse
  - allocazione e scheduling
  - prestazioni
  - controllo
- Istruzioni di selezione e di ciclo
- Flusso di progetto a basso livello

Motivazioni per l'HLS

- Algoritmo
  - indipendente dalla realizzazione hardware
  - non specifica le risorse da utilizzare
  - specifica solo in parte l'ordine delle operazioni
- CPU (semplice architettura di Von Neumann)
  - versatilitá
  - sequenzialitá
- Applicazioni (elaborazione dei segnali, AI)
  - specializzazione
  - specifiche:
    - prestazioni elevate
    - costi ridotti
- Obbiettivi non raggiungibili con una CPU che:
  - "spreca" risorse per garantire la versatilitá
    - una (o piú) unitá multifunzionali (ALU)
    - complessa circuiteria di controllo

Motivazioni

- L'architettura di Von Neumann per l'HLS é un caso particolare di sitema digitale
- Cambiamento di prospettive nella realizzazione di un algoritmo

Software		Hardware
Un numero limitato di unitá di elaborazione polifunzionali	⇔	**Eventualmente numerose unitá funzionali specializzate**
Sequenzialitá	⇔	**Parallelismo**

- Realizzazioni hardware specializzate richiedono la disponibilitá di:
  - tecnologia
  - linguaggi di descrizione degli algoritmi (HDL al livello behavioral)
  - strumenti EDA

Strumenti per la realizzazione di algoritmi al livello hardware

- **Tecnologie microelettroniche per ASICs (Application Specific Integrated Circuits)**
  - Gate-Array
  - FPGA
  - Standard-Cells
- **Linguaggi di descrizione hardware consentono di descrivere sia:**
  - algoritmi (livello behavioral)
  - architetture (livello Register Transfer Level - RTL)
- **Strumenti di EDA**
  - sintesi
  - analisi (verifica formale, simulazione)

Parallelo fra hardware dedicato e software

- L'algoritmo é una sequenza di passi (istruzioni) che porta al calcolo di un risultato
- Supponiamo che ciascuna istruzione possa:
  - calcolare un risultato parziale
  - decidere quale istruzione verrá eseguita di seguito
- Sia nel caso dell'hardware che in quello del software si ha un **data-path** e un **controllo**

Parallelo fra hardware e software

	Software	Hardware
Controllo	**Istruzioni binarie in memoria caricate in sequenza da una FSM con tre stati: fetch, decode, execute**	**Macchina a stati finiti**
	Istruzioni condizionali, salti (cicli)	**Mappati direttamente sulla FSM**
Data-path	**Valutazione sequenziali delle operazioni con una ALU**	**Operazioni mappate su adder, moltiplicatori, divisori .... con la possibilitá di parallelismo**
	Variabili contenute in un register file con l'indirizzo di risultati e operandi contenuto nelle istruzioni	**Registri non necessariamente organizzati in un register file e accessibili tramite una logica dedicata di multiplexing**

Costi

- L'ottimizzazione di costi e prestazioni é alla base della realizzazione hardware di algoritmi e in generale della sintesi ad alto livello
- A costi e prestazioni contribuiscono sia il controllo che il data-path
  - considereremo prevalentemente esempi in cui il data-path domina entrambi gli aspetti
- Area su silicio che non é nota prima della sintesi fisica
  - per esplorare diverse soluzioni, la sintesi ad alto livello necessita di stime approssimate
  - costo approssimato per i diversi componenti (c) blocchi funzionali, registri e multiplexer
  - costo stimato: $\sum_{\forall c} A(c)$

Prestazioni

- Anche queste sono valutabili accuratamente solo dopo la sintesi fisica e quindi la sintesi ad alto livello usa modelli approssimati
  - **prestazioni dinamiche (throughput e latenza)**
  - consumo di potenza
  - collaudabilitá, affidabilitá

Specifiche

- Le specifiche di un ASIC possono riguardare
  - la banda nel caso di sistemi che operano su flussi di dati continui
  - la latenza nel caso di sistemi reattivi
  - l'area del chip o di un modulo (se il sistema é parte di un SOC)
- Considerazioni di sistema dettano a volte la frequenza di clock per cui latenza e banda possono essere date in maniera relativa
- Le specifiche possono essere date come
  - vincoli su prestazioni e costi (es. D_{max} e A_{max})
  - un vincolo su una quantitá con la richiesta di minimizzare l'altra
    - minimo ritardo per una data area massima
    - minima area per un certo ritardo massimo

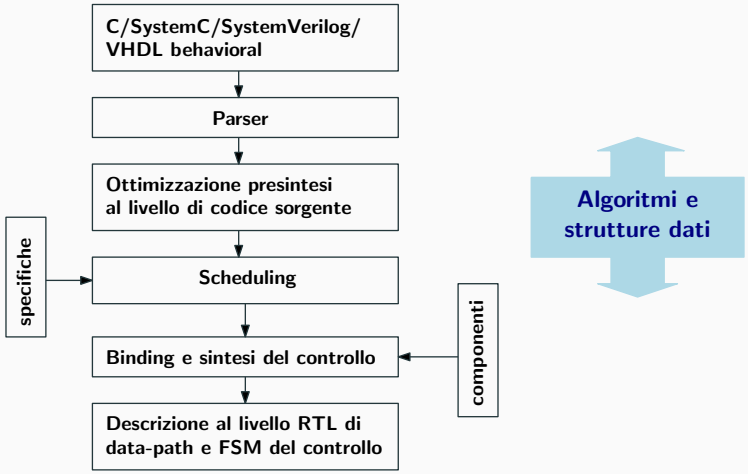
Risorse funzionali

- Blocchi funzionali ciascuno caratterizzato da
  - tipo di dato (interi e float di diverse dimensioni)
  - operazione svolta ($+$, $-$, $*$, $/$, $>$, $=$ ....)
  - stima dell'area occupata
  - stima del ritardo massimo e critical path
  - eventualmente alcuni blocchi possono realizzare piú funzioni (ALU)
- Registri, tipicamente con reset e write enable
- Steering logic (multiplexer, in alcuni casi realizzati con bus)
- Logica combinatoria e FF per realizzare la FSM di controllo

Sintesi e ottimizzazione

- L'insieme di realizzazioni funzionalmente corrette della descrizione comportamentale del componente costituisce lo spazio di progetto
  - ogni punto é caratterizzato da prestazioni e costo
  - i tool di EDA devono essere in grado di esplorarlo in maniera efficiente
- Tipicamente non esiste un ottimo assoluto che minimizzi area e latenza
- I punti di Pareto (soluzioni non dominate) danno un idea dello spazio di progetto
- Tassonomia di riferimento
  - rete sincrona
  - inizialmente senza pipelining

Sintesi ad alto livello



Strutture dati per HLS

Strutture dati per rappresentare un algoritmo

- Semplice algoritmo descritto mediante un comune linguaggio C/JAVA ....
- Interpretazione restrittiva di tipo imperativo e sequenziale riferita ad un'architettura di tipo Von Neumann
- In realtá il codice puó avere un interpretazione molto piú generale (parallelismo) che é sfruttata sia dalla sintesi dell'hardware che dai compilatori per le CPU attuali
- Sia i compilatori (software) che gli strumenti di sintesi (hardware) necessitano di una struttura dati astratta per rappresentare l'algoritmo

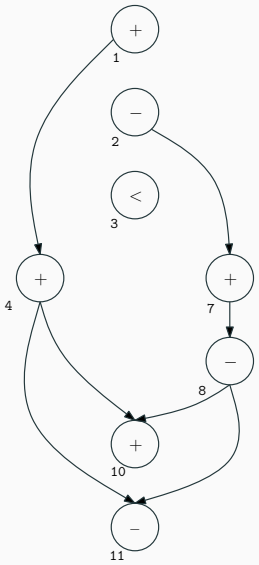
Data-Flow graph (DFG)

```
1  t=a+b;
2  u=a-b;
3  if (a<b)
4    v=t+c;
5  else
6    {
7      w=u+c;
8      v=w-d;
9    }
10 x=v+e;
11 y=v-e;
```

- Il significato di questo codice dal punto di vista di una CPU scalare é evidente
- In realtá l'algoritmo contiene delle informazioni piú generali
- Per esplicitarle é necessario ricorrere a un modello piú generale
- Si osserva che esistono istruzioni che si limitano ad alterare il flusso dei dati e altre che alterano il controllo cambiando l'ordine di esecuzione delle operazioni
  - data-flow
  - control-flow

Data-Flow graph (DFG)

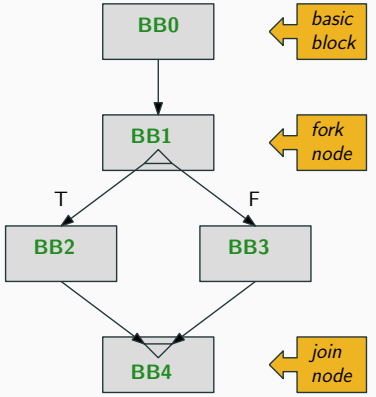
```
1  t=a+b;
2  u=a-b;
3  if (a<b)
4    v=t+c;
5  else
6    {
7      w=u+c;
8      v=w-d;
9    }
10 x=v+e;
11 y=v-e;
```



Struttura dati astratta che rappresenta le dipendenze che esistono fra i dati elaborati dalle varie operazioni supponendo che queste siano binarie

Control-Flow Graph (CFG)

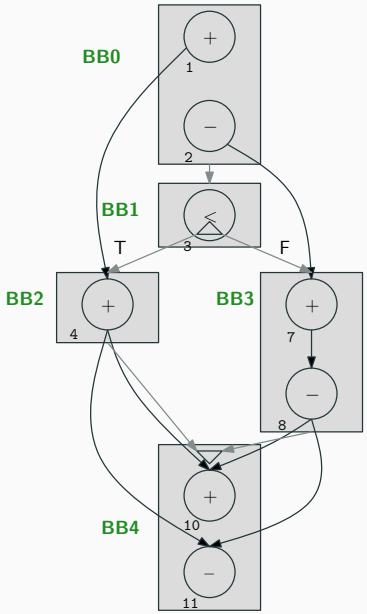
```
1  t=a+b;          BB0
2  u=a-b;
3  if (a<b)        BB1
4    v=t+c;        BB2
5  else
6    {
7      w=u+c;      BB3
8      v=w-d;
9    }
10 x=v+e;          BB4
11 y=v-e;
```



Struttura dati astratta che rappresenta le possibili alternative nel flusso di esecuzione dell'algoritmo

Control-Data-Flow Graph (CDFG)

```
1  t=a+b;
2  u=a-b;
3  if (a<b)
4    v=t+c;
5  else
6    {
7      w=u+c;
8      v=w-d;
9    }
10 x=v+e;
11 y=v-e;
```



Applicazione delle strutture dati alla sintesi ad alto livello

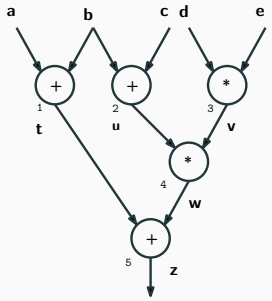
- Le strutture dati viste sono molto generali e possono essere utilizzate sia per HLS sia per la compilazione in CPU con parallelismo
- Si parte con il DFG avendo come obbiettivo architetture come i DSP il cui costo é dominato dal data-path
- Ipotesi
  - rete sincrona
  - blocchi funzionali di tipo combinatorio
- Si vedranno per prima cosa alcuni concetti base utilizzati per esplorare lo spazio di progetto cercando di ottimizzare costi e prestazioni (ritardo)
- In seguito vedremo anche alcuni euristici

Esempio: valutazione di un espressione

- Si consideri il seguente algoritmo corrispondente alla valutazione dell'espressione aritmetica
  $z = a + b + ((b + c) * (d * e))$

```
1  t=a+b;
2  u=b+c;
3  v=d*e;
4  w=u*v;
5  z=t+w;
```

Data Flow Graph



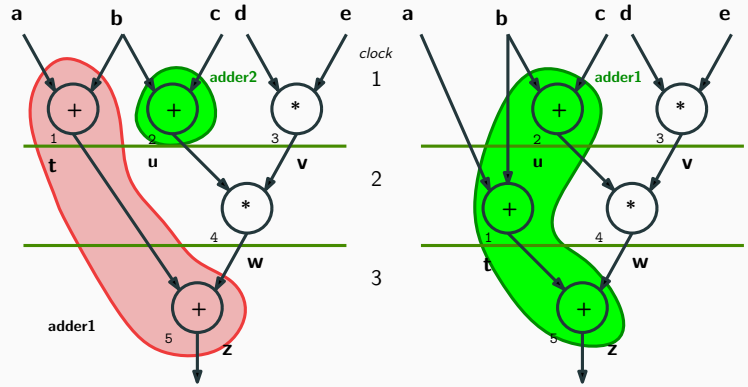
- Si noti l'ipotesi che tutti gli ingressi siano disponibili contemporaneamente

Data Flow Graph - I

- Il DFG ci dice:
  - l'ordine relativo di prioritá fra le operazioni
  - quali tipi di risorse sono necessari
- Il DFG non dice:
  - quali e quante risorse utilizzare (**allocazione**)
  - quale risorsa utilizzare per una data operazione (**binding**)
  - quando effettuare le operazioni (**scheduling**)
- Allocazione e scheduling sono legate fra loro e determinano costi e prestazioni
- Ottimizzazione

Data Flow Graph - II

- **Esempio di correlazione fra allocazione, scheduling e binding**
- **L'operazione `1` puó essere eseguita sia nel primo che nel secondo ciclo di clock**
- **Nel primo caso sono necessari due addizionatori**
- **Nel secondo é sufficiente un singolo addizionatore: riduzione di area a paritá di prestazioni**



Data Flow Graph - III

- Nell'esempio precedente si utilizza l'ipotesi di *ciclo singolo*: ciascun blocco funzionale esegue l'operazione associata in un ciclo di clock
- Sotto questa ipotesi, il periodo di clock é principalmente determinato dal blocco piú lento (il moltiplicatore in questo caso)
- Questo fa si che alcuni cicli di clock siano sovradimensionati
- Per risolvere questo problema si possono utilizzare due tecniche dette **multicycling** e **chaining**

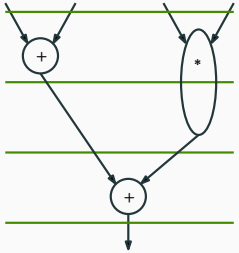
Temporizzazioni

Cammino critico

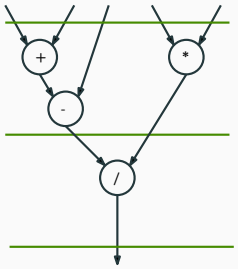
- Sotto l'ipotesi di ciclo singolo, il **cammino critico** di un DFG é il percorso che va da un ingresso a un uscita con il maggior numero di nodi
- Si noti che é un concetto diverso da quello di cammino critico in una rete combinatoria
  - il cammino critico in una rete combinatoria determina il periodo di clock minimo
  - il cammino critico di un DFG determina la latenza relativa minima del DFG

Multiycling e chaining

multicycling: se il ritardo in un componente é maggiore del periodo di clock si puó fare un operazione multiciclo agendo sul comando di WE del registro ove viene scritta la sua uscita

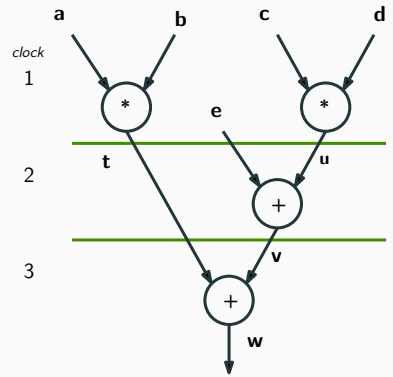


chaining: se la somma dei ritardi di cammino critico di due componenti é minore del periodo di clock, le due operazioni corrispondenti possono essere concatenate



Esempi su chaining e multicycling

- Algoritmo:
  `u=a*b; v=c*d; w=e+v; z=w+u;`
- Ritardi
  - adder 3.4ns
  - moltiplicatore 11ns
  - FF e MPX 0.5ns
- Specifiche: latenza minima non considerando il costo
- Realizzazione combinatoria $D = 17.8ns$
- Realizzazione a ciclo singolo: $T > 11.5ns$
- Latenza: $D = 3T$



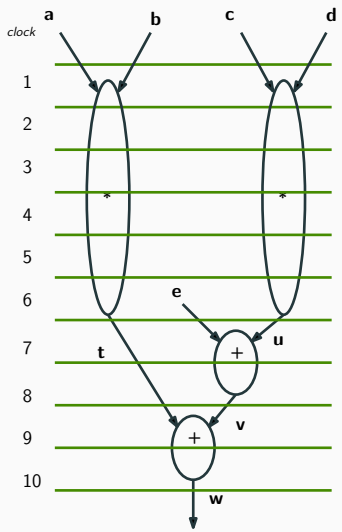
Soluzioni per diversi periodi di clock

- Si vuole esplorare lo spazio dei possibili periodi di clock dal punto di vista della latenza
- Chiaramente c'é un impatto sul consumo di potenza che qui non viene considerato

T (ns)	latenza (ns)	technique
2	20 (10T)	multicycling
3	21 (7T)	multicycling & chaining
4	20 (5T)	multicycling
5	25 (5T)	multicycling
6	24 (4T)	multicycling
....		
12	24 (2T)	chaining
12	36 (3T)	single cycle

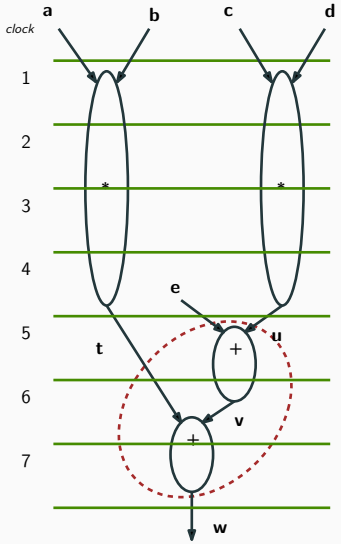
- Si osserva una dipendenza non lineare e non monotona
- Tutte le realizzazioni richiedono 2 moltiplicatori e un adder, a parte l'ultima che richiede due adder

Esempio di multicycling



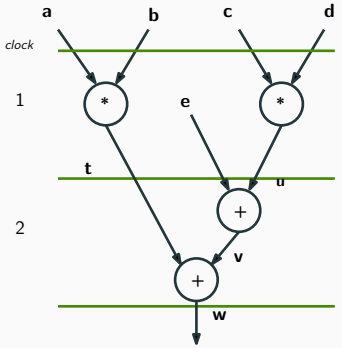
periodo di clock $T = 2ns$
latenza $D = 10T$
risorse 2 moltiplicatori, 1 adder

Esempio di multicycling e chaining



- Periodo di clock $T = 3ns$
- Per le moltiplicazioni sono richiesti 4 periodi di clock
- Le somme richiedono 3.9ns e quindi sarebbe necessario un multicycling di due periodi di clock per ciascuna di esse
- Concatenando due sommatori come un sommatore a 3 operandi (tratteggiato), si puó applicare il multicycling e risparmiare un periodo di clok
  - latenza $D = 7T$
  - risorse 2 moltiplicatori, 2 adder

Esempio di chaining



- Periodo di clock $T = 12ns$
- Latenza $D = 2T$
- Risorse 2 moltiplicatori e 2 adder

Ottimizzazione

Ottimizzazione

- Si é vista la dipendenza di costi e prestazioni da allocazione e scheduling
- Si ha uno spazio di possibili soluzioni che va esplorato alla ricerca di una soluzione che soddisfi le specifiche di progetto
- Si utilizzano le seguenti ipotesi
  - solo operazioni di tipo data-path
  - ciclo singolo
- E le seguenti specifiche
  - area minima per una data latenza
  - latenza minima per una data area
- Il problema ha un costo esponenziale nel numero di operazioni

Spazio delle possibili soluzioni

- Esistono dei punti particolarmente interessanti
  - soluzione con area minima (determinata dal minimo numero di blocchi funzionali che possono svolgere le operazioni del DFG)
  - soluzione con latenza minima (determinata dal cammino critico del DFG)
- L'esplorazione dello spazio di possibili soluzioni puó essere fatta in maniera esatta o mediante metodi euristici se il numero di operazioni é molto grande
- Si noti che per semplicitá stiamo trascurando costi e ritardi legati a multiplexer e flip-flop: anche questi possono essere soggetto di ottimizzazioni

Allocazione

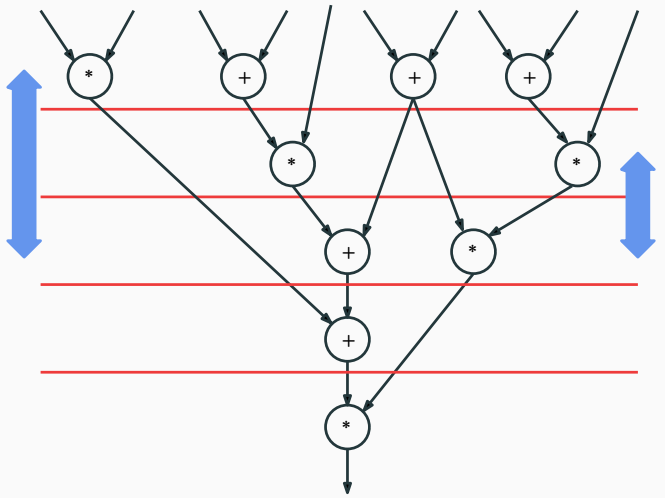
- Il numero di risorse da considerare per ogni tipo di blocco funzionale puó
  - essere fissato come specifica iniziale (si noti che questa é una semplificazione perché potrebbe essere specificato il costo senza distinguere fra i diversi tipi di risorse)
  - essere calcolato per una certa allocazione
- **Dato uno scheduling, il numero di risorse di un certo tipo ($+, *, \ldots.$) puó essere calcolato come il massimo numero di quel tipo di risorse per ogni ciclo di clock**

Scheduling

- É l'operazione principale nell'ottimizzazione del DFG di un algoritmo
- Per prima cosa individuiamo i gradi di libertá che abbiamo a disposizione con l'ipotesi di fare uno scheduling a latenza minima ottimizzando le risorse
- A ogni operazione del DFG puó essere associata una **mobilitá** che descrive i suoi possibili scheduling
  - le operazioni sul critical path non hanno mobilitá
  - il concetto é relativo, la mobilitá di un operazione dipende da quella delle altre
- Introduciamo due tipi di scheduling che ci aiutano a valutare quantitativamente la mobilitá

Scheduling ASAP (As Soon As Possible)

Ogni operazione viene programmata non appena sono disponibili i suoi operandi

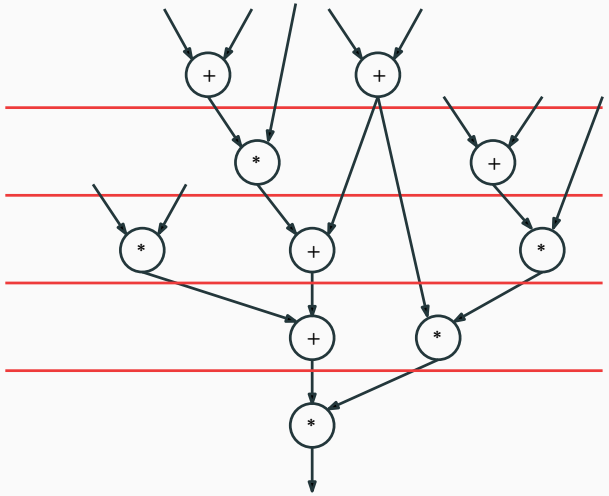


latenza = 5T

allocazione
2 adder
2 moltiplicatori

Scheduling ALAP (As Late As Possible)

La programmazione di ogni operazione viene dilazionata fino a quando un ulteriore ritardo risulterebbe in un aumento della latenza

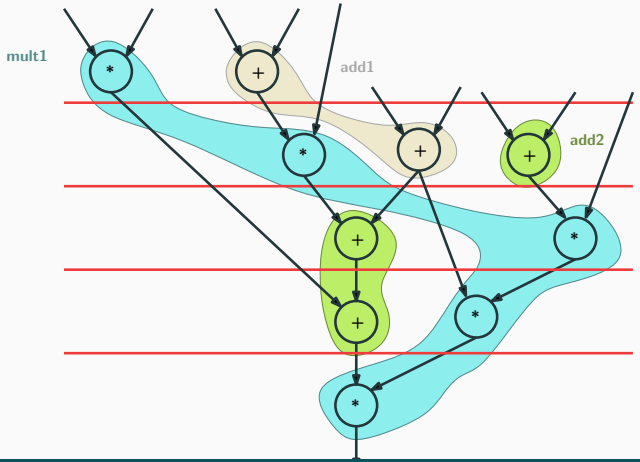


latenza = 5T

allocazione
2 adder
2 moltiplicatori

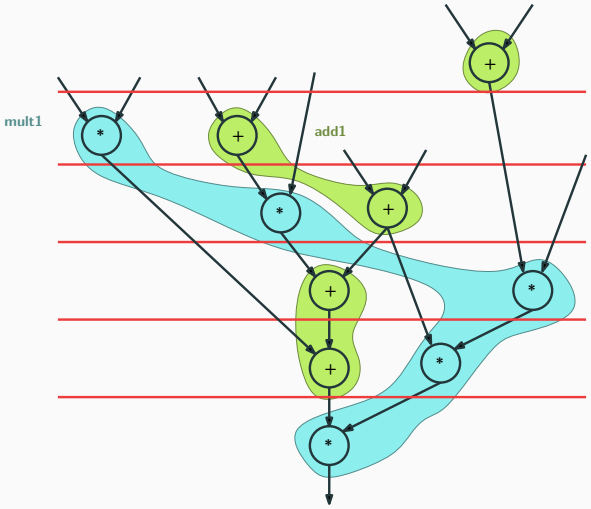
Scheduling ottimizzato

- Lo scheduling ASAP e ALAP tipicamente non presentano l'ottimo nell'allocazione delle risorse
- Infatti, questo scheduling consente di utilizzare un singolo moltiplicatore e due adder



Scheduling a risorse fissate con ottimizzazione della latenza

- Allocazione di 1 adder e 1 moltiplicatore
- Si osserva che é necessario aggiungere un periodo di clock



Prossimi passi

- Abbiamo visto come scheduling e allocazione siano fortemente correlati e come il binding sia una conseguenza di questi passi di progetto
- Per arrivare a capire il ruolo della HLS ci manca ancora
  - la descrizione di approcci sistematici allo scheduling
  - una strategia per definire i dettagli implementativi corrispondenti a un certo scheduling e alla corrispondente allocazione
    - registri
    - multiplexer
    - input/output
    - controllo

Algoritmi

Algoritmi di scheduling

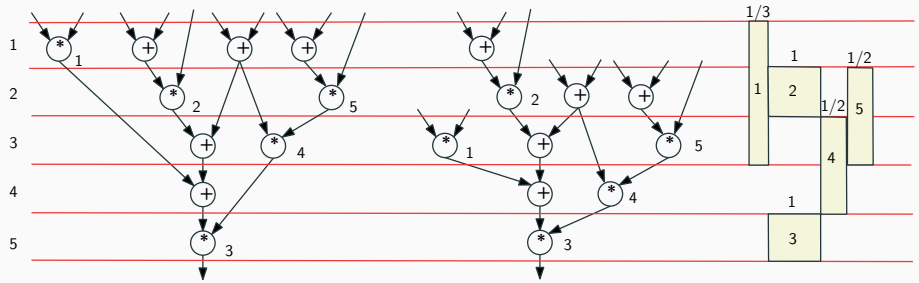
- Iniziamo con il vedere un paio di tecniche euristiche per l'ottimizzazione dello scheduling di un DFG
  - **Force Directed Scheduling** per la minimizzazione delle risorse per una data latenza
  - **List Based Scheduling** per la minimizzazione della latenza per una data allocazione
- Queste tecniche hanno una complessitá computazionale inferiore a quelle esatte
- Chiaramente possono produrre risultati corrispondenti a minimi locali di costo e prestazioni

Force directed scheduling

- Euristico per minimizzare l'area in presenza di vincoli sulla latenza (tipicamente dati dal cammino critico)
- In pratica, la densit'¡ operazioni dello stesso tipo in un certo ciclo di controllo viene vista come una forza elastica in grado di attrarre/respingere un operazione di quel tipo
- Tale forza viene minimizzata per ottenere una distribuzione bilanciata delle risorse
- L'algoritmo serve sia per unitá funzionali che per registri e logica di steering

Force directed scheduling

- Vengono fatti prima uno scheduling ASAP e uno ALAP
- Time frame di un operazione: intervallo fra lo scheduling ASAP e quello ALAP
- Si suppone che la probabilitá di programmare un operazione in tale intervallo sia distribuita uniformemente
- Considereremo le moltiplicazioni perché dominanti dal punto di vista del costo

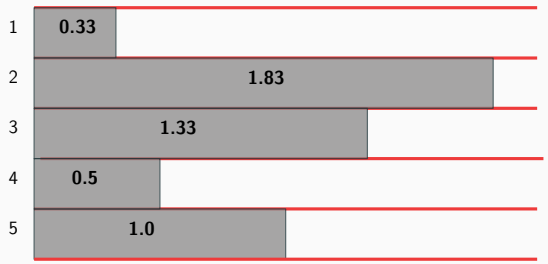


Grafo di distribuzione delle operazioni

- Concorrenza in ogni ciclo di controllo (i) delle operazioni dello stesso tipo (t)

$$DG(i) = \sum_{op=t} Prob(opn, i)$$

- Maggiore é $DG(i)$, maggiore sará la probabilitá che il ciclo i sia quello che definisce il numero di risorse per operazioni di tipo t



Calcolo della forza

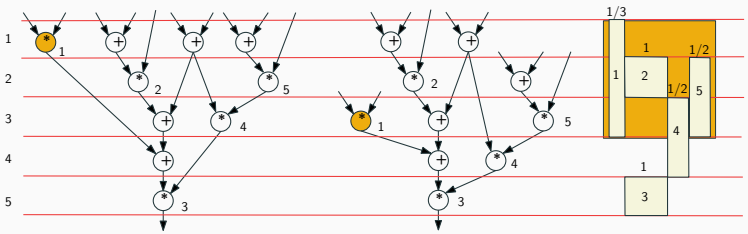
- Per ogni operazione si definisce una forza per ogni ciclo di controllo (j) corrispondente a uno scheduling fattibile per tale operazione

$$Force(j) = DG(j) - \sum_{i=t}^{b} \frac{DG(i)}{b - t + 1}$$

- Dove
  - b é il ciclo di controllo dell'operazione nello scheduling ALAP
  - t é il ciclo di controllo dell'operazione nello scheduling ASAP
  - DG é il grafo di distribuzione per il tipo dell'operazione considerata
- Una forza negativa indica che l'operazione é "attratta" dal ciclo j, mentre una forza positiva indica che é "respinta" dal ciclo j

Esempio

- Si consideri la moltiplicazione 1 nell'esempio di DFG
- Lo scheduling di tale operazione puó essere da $t = 1$ a $b = 3$
  $Force(1) = 0.33 - 1.16 = -0.83$
  $Force(2) = 1.83 - 1.16 = 0.33$
  $Force(3) = 1.33 - 1.16 = 0.17$
- Da cui si vede che risulta conveniente utilizzare il ciclo $j = 1$



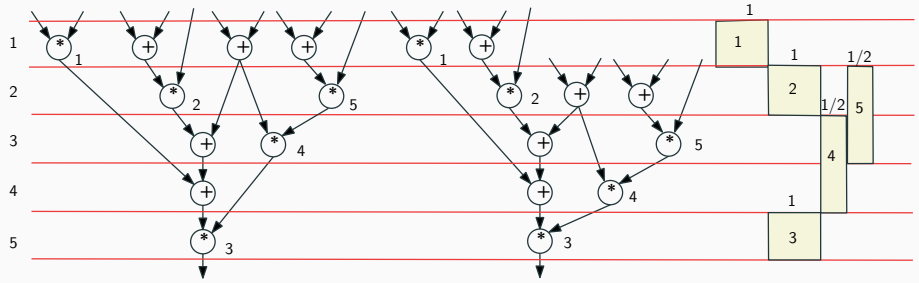
Algoritmo force directed scheduling

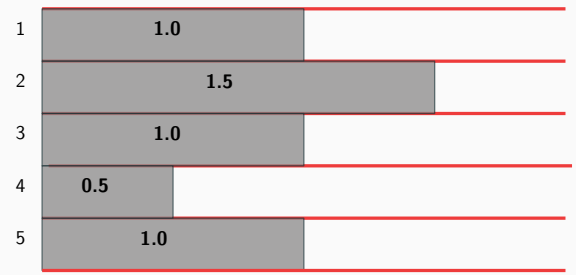
- A ogni passo l'algoritmo sceglie l'operazione con la minor forza e la programma
- Dopo questa operazione, viene aggiornato DG e vengono ricalcolate le forze e se lo scheduling non é completo, si torna al passo precedente
- In realtá il problema é piú complesso perché ogni scheduling cambia la mobilitá di alcune operazioni
- Bisogna aggiungere una self-force e una predecessor-successor force
  - ad esempio se si programma l'operazione 5 nel ciclo di controllo 3, l'operazione 4 é fissata nel ciclo 4 e questo viene tenuto in conto da una successor force

Esempio

- Per le operazioni 4 e 5 si hanno i seguenti risultati
  - operazione 4: $Force(3) = 1.83 - 0.91 = 0.92$,
    $Force(4) = 0.5 - 0.91 = -0.41$
  - operazione 5: $Force(2) = 1.83 - 1.58 = 0.25$,
    $Force(3) = 1.33 - 1.58 = -0.25$
- La moltiplicazione 1 ha il minor valore di forza ($Force(1) = -0.83$) e quindi viene programmata nel primo ciclo di controllo
- A questo punto, tenendo fissa l'operazione 1, si ricalcola il grafo di distribuzione e le forze

Esempio





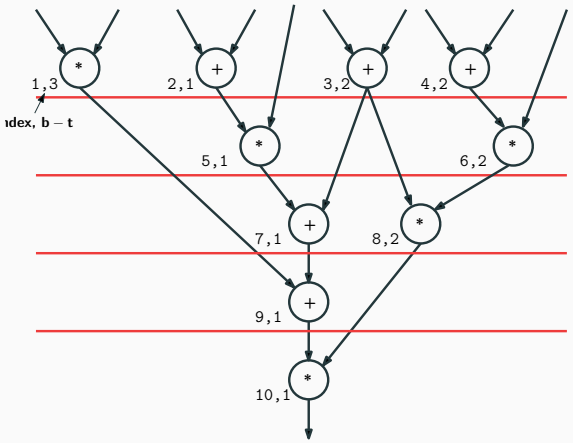
operazione 4
$Force(3) = 1.0 - 0.75 = 0.25$
$Force(4) = 0.5 - 0.75 = -0.25$
operazione 5
$Force(2) = 1.5 - 1.25 = 0.25$
$Force(3) = 1.0 - 1.25 = -0.25$
si puó programmare l'operazione 4 nel ciclo 4 o equivalentemente la 5 nel ciclo 3, in questo caso non serve la successor force e si ottiene lo scheduling ottimale

List based scheduling

- Diversamente dal force directed scheduling, il list based scheduling cerca di minimizzare la latenza per un ben definito ammontare delle risorse
- In pratica, é una generalizzazione del metodo ASAP
- Si mantiene una lista di operazioni pronte (i cui predecessori sono giá stati programmati)
- A ciascuna operazione pronta si associa una funzione di priorità
  - ad esempio l'inverso della mobilitá ($b - t$)
- A ciascuna iterazione, per un dato tipo di operazione, vengono programmati i nodi con la priorità piú alta dilazionando gli altri
- Gli operatori a piú alta mobilitá sono quelli programmati dopo (greedy)
- Sono state sviluppate diverse funzioni di costo

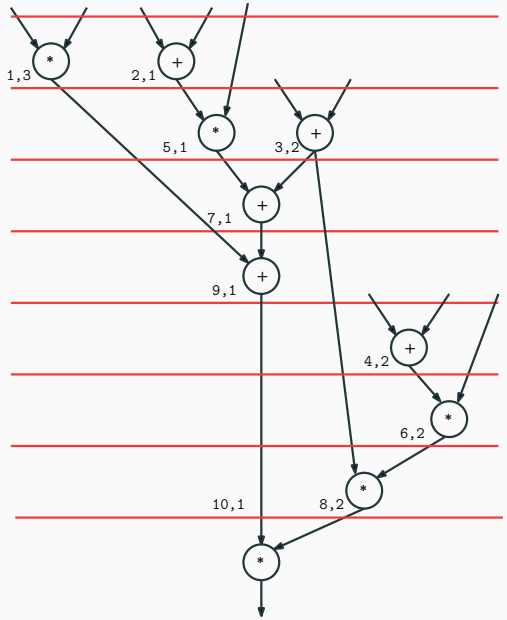
Esempio - I

- Allocazione: 1 adder e 1 moltiplicatore
- Si calcolano lo scheduling ASAP e quello ASAP annotando ciascun nodo con $b - t$ (inverso della mobilitá)



passo	lista di nodi pronti	scheduling
1	1, 2, 3, 4	1(*), 2(+)
2	3,4,5	5(*),3(+)
3	7,4	7(+)
4	9,4	9(+)
5	4	4(+)
6	6	6(*)
7	8	8(*)
8	10	10(*)

Esempio - II



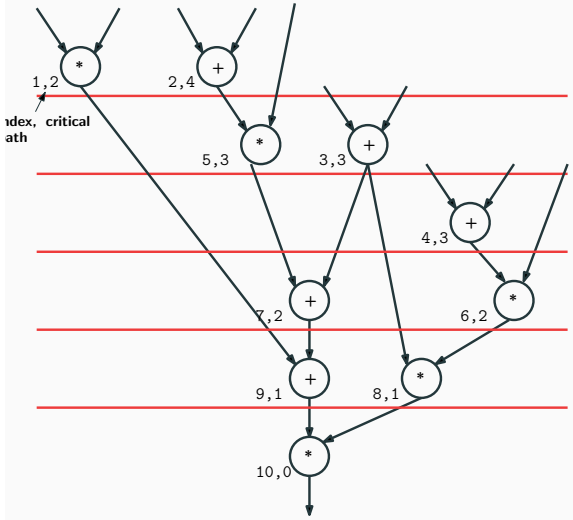
- latenza uguale a 8 cicli di controllo
- la scelta di programmare 3 invece di 4 al passo 2 é arbitraria
- una scelta diversa sarebbe risultata in una minore latenza
- trattandosi di un euristico é normale che l'algoritmo sia arrivato a un minimo locale

List based scheduling basato sul critical path

- Il critical path di un nodo é il numero massimo di nodi fra tale nodo e l'uscita
- La funzione di priorità é la lunghezza del cammino critico
- Ogni nodo é annotato con il suo indice e il suo cammino critico
- Anche in questo caso si tratta di un euristico
- Questa tecnica viene utilizzata anche dai compilatori nel caso di CPU superscalari

Esempio

- Allocazione: 1 adder e 1 moltiplicatore



passo	lista di nodi pronti	scheduling
1	1, 2, 3, 4	1(*), 2(+)
2	3,4,5	5(*),3(+)
3	7,4	4(+)
4	7,6	6(*), 7(+)
5	8,9	8(*),9(+)
6	10	10(*)

Approccio esatto al problema dello scheduling

- L'ottimizzazione dello scheduling puó essere formulata come un problema di Integer Linear Programming
- Si tratta di un insieme di disequazioni lineari con una funzione lineare di costo (area o latenza) da minimizzare
- Una volta descritto il problema in questo modo, si puó utilizzare un solver/optimizer ILP per avere la soluzione