Progetto per il corso di Progetto Automatico di Sistemi Digitali

Filippo Landi

2 agosto 2021

Sommario

Il mio progetto per il corso di Progetto Automatico di Sistemi Digitali (PASD in breve) consiste nello studio statistico del comportamento di un circuito "multiply and accumulate" (o "mac" in breve) in presenza di alcuni difetti di produzione.

1 Introduzione al progetto

Il progetto riguarda il collaudo dei sistemi digitali, argomento del corso di PASD.

Al fine di simulare i guasti del circuito userò "HOPE" un simulatore di guasto per circuiti digitali sequenziali sviluppato dall'università VirginiaTech, anche questo proposto durante il corso.

HOPE legge i circuiti attraverso delle descrizioni della rete (netlist) in formato .bench, quindi il primo punto del progetto sarà studiare la struttura del circuito per implementarlo in questo formato.

Ho deciso di usare Python per aiutarmi con i diversi passaggi del progetto.

Partiamo quindi dalla descrizione del circuito da studiare, passeremo poi alla sua implementazione e in fine ad alcuni studi statistici.

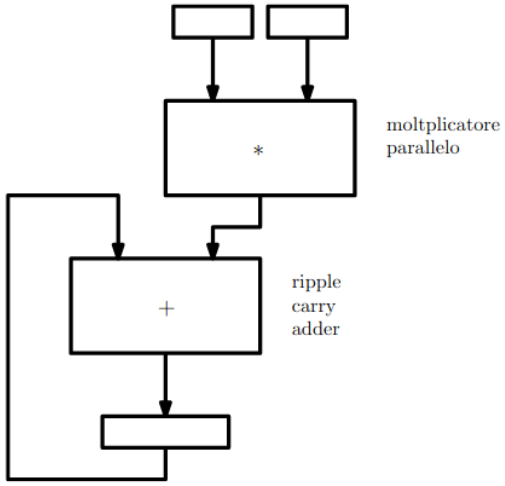
2 Circuito multiply and accumulate

Mi è stato richiesto di realizzare un circuito multiply and accumulate (mac) a 8 bit, esso è composto da un moltiplicatore con un sommatore in cascata e un registro per retroazionare le uscite in modo da accumulare i risultati.

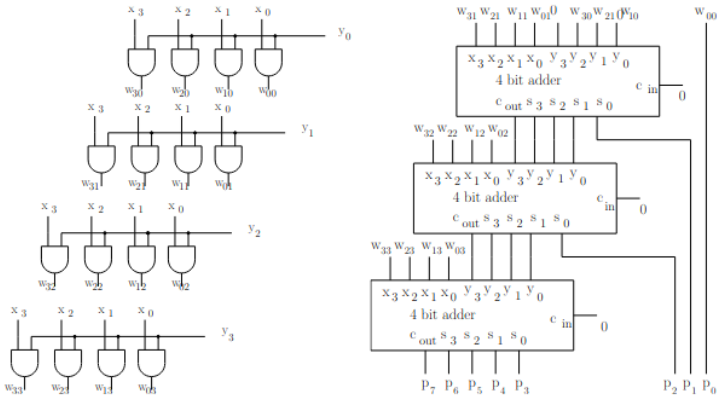
Riporto un documento inviatomi dal professore che illustra gli schemi di un circuito di questo tipo a 4 bit.

Multiply and accumulate

É un unitá ampiamente utilizzata nei DSP e nel machine learning per realizzare funzioni del tipo $s = \sum x_i y_i$. La dimensione degli ingressi sia n, per cui quella dell'uscita del moltiplicatore sia $2n$.



Qui c'é lo schema di un moltiplicatore parallelo a 4 bit che puó essere esteso per valori piú grandi di n



2.1 Moltiplicatore

La struttura del moltiplicatore è ben illustrata nella documentazione:

- Ogni segnale X è messo in AND con ogni Y, generando i segnali W.
- I segnali W vengono usati da degli adder strutturati su più livelli: si può notare che il numero di livelli è n-1 in quanto al primo livello vengono usati WX1 e WX0, poi i rimanenti WXY nei livelli successivi.

2.2 Sommatore

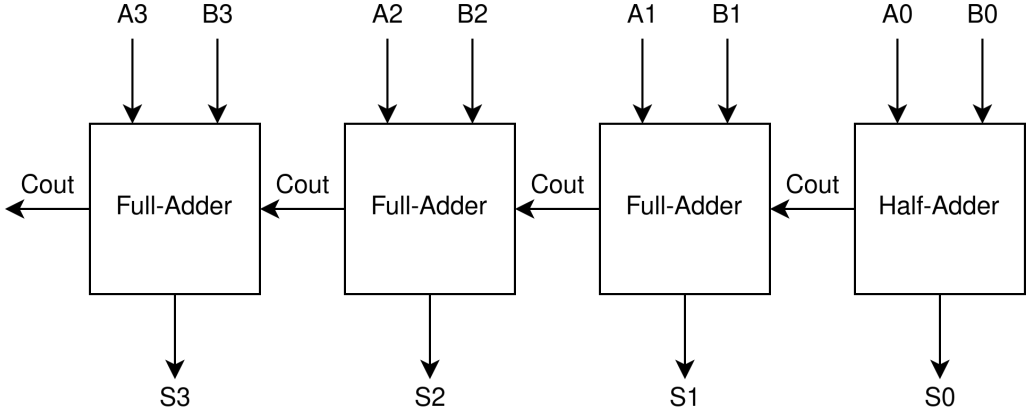
È un ripple carry adder con dimensione degli ingressi 2n: un ingresso è dato dall'uscita del moltiplicatore mentre l'altra è data dalle uscite stesse del sommatore retroazionate attraverso dei flip flop tipo D (circuito già integrato in HOPE).

2.2.1 Ripple Carry Adder

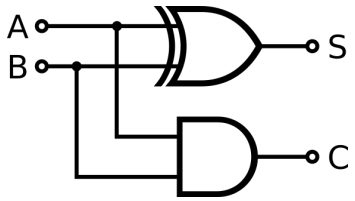
Usiamo dei ripple carry adder (rca) sia per la cascata di adder del moltiplicatore oltre che per il ripple carry adder successivo con dimensione degli ingressi 2n.

I ripple carry adder nella precedente documentazione erano illustrati a livello register transfer level (RTL) per ovvie ragioni di chiarezza, però per l'implementazione del circuito ci serve vedere come sono fatti a livello di porte logiche.

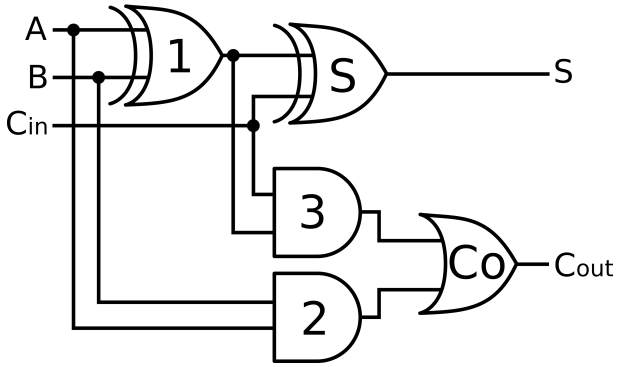
I ripple carry adder sono circuiti formati da una serie di adder in cascata:



Questo è lo schema che uso del half-adder:



Questo è lo schema che uso del full-adder:



Opportunamente collegando le varie porte logiche quindi si ottiene un ripple carry adder.

3 Realizzazione del circuito .bench

La stesura manuale del file .bench del circuito ad 8 bit non mi sembrava un approccio furbo vista l'architettura piuttosto complessa da rappresentare. Probabilmente proseguire con tale metologia avrebbe portato a vari errori: sia banalmente di battitura, sia dovuti alla complessità e quindi errori nel collegamento dei vari segnali. Per questo ho pensato ad una metodologia stile "divide et impera".

In partenza ho scritto alcuni circuiti di prova per esplorare i vari adder e la prima parte del moltiplicatore. Li trovate nella cartella "circuti_prova", non sono fondamentali per il progetto però possono aiutare alla comprensione della metodologia usata.

Appurata la struttura dei circuiti base ho creato un singolo file che li mettesse insieme in maniera opportuna (vari cicli for più certe condizioni). Questo approccio ha portato allo script "mac_generator.py" che permette di generare il circuito con una dimensione arbitraria degli ingressi da passare

come argomento: per generare un mac 4 bit si può scrivere da terminale questo comando Python: "python3 mac_generator.py 4".

Il passo successivo sarebbe un approccio più modulare, che otterrebbe lo stesso risultato chiamando sottofunzioni per la varie parti del circuito: per i miei scopi uno script monolitico è più che sufficiente quindi mi fermo con questa versione.

Il codice è ampiamente commentato quindi consiglio di leggerlo per comprenderne il funzionamento.

L'unico punto che potrebbe essere difficoltoso è la comprensione dei vari segnali, proprio per questo un approccio manuale a mio avviso avrebbe portato ad errori di battitura o logici dovuti dalla complessità del circuito.

Descrivo la struttura dei vari segnali sperando di fare un po' di chiarezza:

- Gli input sono facilmente individuabili e sono X e Y seguiti dal rispettivo peso del bit, es. X0,Y0,X1,Y1 etc.
- I segnali W ottenuti dagli AND di X e Y sono scritti come "W_XnYn" (n indica il peso del bit), es. W_X0Y0.
- I ripple carry adder contengono half-adder e full-adder:
  - Gli half-adder, come dagli schemi, hanno uscita S e Co (il carry out) e sono usati in generale (con qualche eccezione) per il bit di minor peso (bit 0):
    - Nel moltiplicatore gli rca sono strutturati in livelli:
      - SL0D0 indica l'uscita dell'half-adder al livello (L) 0 del bit di peso (D) 0.
      - CoL0D0 indica il carry out dell'half-adder al livello (L) 0 del bit di peso (D) 0, questo carry out sarà il carry in del full-adder successivo.
    - Nel sommatore invece non ci sono più i livelli:
      - S0 indica l'uscita dell'half-adder per il bit di peso 0.
      - C0 il suo carry out.
  - I full-adder, come dagli schemi, hanno uscita S e Co con l'aggiunta rispetto gli half-adder dei segnali interni 1,2,3:
    - Nel moltiplicatore gli rca sono strutturati in livelli:

        - SL0D1 indica l'uscita del full-adder al livello (L) 0 del bit di peso (D) 1
        - lo stesso full-adder avrà CoL0D1, 1L0D1, 2L0D1 e 3L0D1 (carry out e segnali interni).
    * Nel sommatore invece non ci sono più i livelli:
        - S1 indica l'uscita del full-adder per il bit di peso 1
        - Lo stesso full-adder avrà Co1, 11, 21 e 31 (carry out e segnali interni) sempre legati al primo bit.

Probabilmente i segnali interni dei full-adder sono i più confusionari, nella mia prima analisi li avevo assegnati numerici e li ho mantenuti così, in caso basta cambiarli con una qualche lettera non assegnata.

Spero che questa spiegazione dei segnali sia esaustiva alla comprensione della struttura del circuito risultante da questo primo script Python.

Gli script originali sono stati scritti con Python 3.8.10 e dovrebbero girare con ogni versione superiore alla 3.6.

Ho anche fatto una conversione per rendere gli script compatibili con Python 2.7, le modifiche non comportano differenze nei file di output.

Se interessa, a livello di sorgenti ci sono state queste modifiche:

- Python 3.6 aveva introdotto le cosidette "f-strings" molto più comode della metodologia precedente per formattare le stringhe: Python 2.7 ovviamente usa la vecchia formattazione quindi ho dovuto sostituire le f-strings con stringhe formattate con .format().

- Python 2 suppone che tutti i caratteri siano ascii e quindi se trova lettere accentate da' errore quindi ho forzato l'encoding utf-8 con un "commento magico" nella prima riga di codice.

- La concatenzione delle stringhe avviene in maniera diversa tra le due versioni per le differenze dovute alla formattazione: con il nuovo metodo le stringhe si concatenano automaticamente, con il vecchio bisogna aggiungere un "+".

4 Studio statistico del circuito

Ottenuto il file .bench contenente la descrizione del circuito possiamo passare ad usare HOPE.

Scrivendo "./hope -h g" si apre la guida utente in cui si possono vedere i vari comandi disponibili.

Si è deciso di sfruttare il circuito a 4 bit per le prime operazioni.

Con il comando "./hope -0 mac_4.bench" possiamo testare il nostro mac e scoprire la copertura di guasto, "-0" serve per settare i flip flop a 0.

Con il comando "./hope -F file_name -r 16 -0" possiamo:

- Grazie a "-F file_name" scrivere su un file per ogni guasto le uscite che saranno buone o guaste. Se rivelano il guasto avranno un asterisco nella loro linea.
- Grazie a "-r 16", testiamo con un pattern random di 16 vettori di test.
- "-0" come prima setta i flip flop a 0.

Prendendo il file di output poi lo possiamo analizzare per uno studio statistico.

Realizzo questa funzione con lo script Python "getvectors.py". Questo script apre il file ottenuto e conta per ogni test il numero di guasti iniettati e quante volte rivelano un errore e calcola quindi una copertura di guasto relativa a quel test particolare.

Per salvare l'output del programma basta ridirigere l'output da terminale a un file a piacimento: es. "getvectors.py mac > output_mac"

Il programma inoltre salva i vari vettori in file di testo sotto forma di array "vectors_mac".