

Progetto per il corso di Progetto Automatico di Sistemi Digitali

Filippo Landi

5 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, un argomento trattato ampiamente nel corso.

Al fine di simulare i guasti del circuito userò *HOPE*: un simulatore di guasto per circuiti digitali sequenziali sviluppato dall'università VirginiaTech, uno strumento di Electronic Design Automation proposto durante il corso.

HOPE è un software per le università e la ricerca, per averne una copia bisogna entrare in contatto con l'università di VirginiaTech: io non lo condividerò in questo progetto.

HOPE legge i circuiti attraverso delle descrizioni della rete (dette anche *netlist*) in formato *.bench*, infatti il primo punto del progetto sarà l'implementazione del circuito in questo formato.

Ho usato *Python* per diversi passaggi del progetto:

- Non è un linguaggio efficiente come ad esempio il linguaggio C, però mette a disposizione funzioni built-in per la manipolazione di stringhe che si sono rivelate molto comode da usare.

- Ci sono molte librerie interessanti per Python come la libreria *matplotlib* che mi ha permesso in maniera comoda di realizzare dei grafici dei dati raccolti.

I prerequisiti software quindi sono:

1. *Python* in versione superiore alla 3.6.
2. Il software *HOPE*.
3. La libreria *matplotlib* (con le varie dipendenze).

Ho inoltre modificato gli script per renderli compatibili con la versione 2.7 di Python tranne per quanto riguarda la realizzazione automatica dei grafici: ho trovato qualche problema probabilmente perché *matplotlib* si è installato per la mia versione (versione 3.8.10), probabilmente creando degli ambienti virtuali si risolve ma non ho esplorato questa via.

L'esposizione del progetto seguirà questo flusso:

1. Partiremo dalla descrizione del circuito da studiare in termini di porte logiche.
2. Passeremo poi ad analizzare la sua implementazione attraverso uno script Python.
3. Vedremo come usare *HOPE* per i nostri scopi.
4. Estrapoleremo dei dati e calcoleremo delle statistiche attraverso un altro script.
5. Infine vedremo e commenteremo i risultati con alcuni grafici.

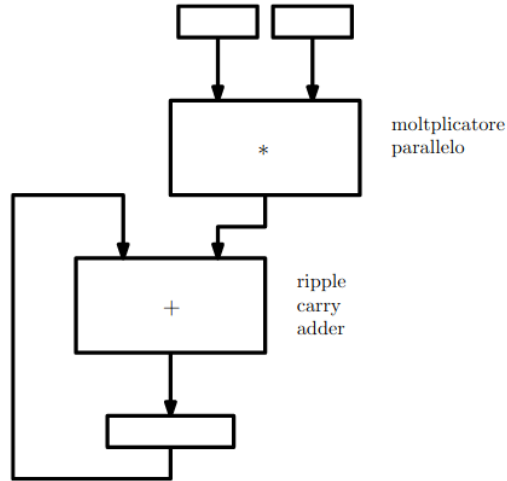
2 Circuito multiply and accumulate

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.

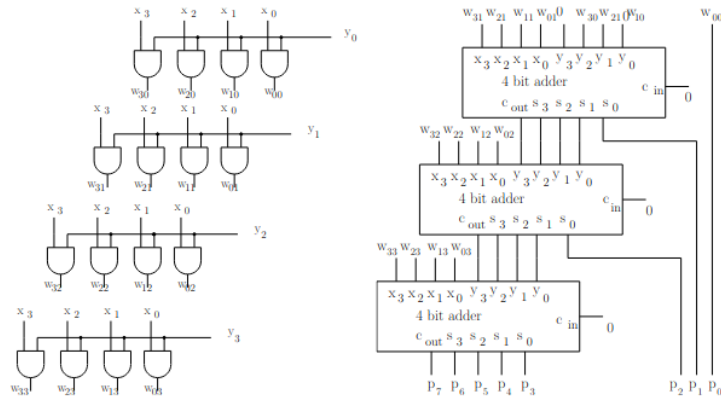
Riporto un documento inviatomi dal professore che illustra gli schemi di un circuito di questo tipo a 4 bit.

Multiply and accumulate

É un unità 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'è lo schema di un moltiplicatore parallelo a 4 bit che può essere esteso per valori più 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* (*rca* in breve) 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 (circuiti già integrati in HOPE).

2.2.1 Ripple Carry Adder

Usiamo dei circuiti ripple carry adder sia per la cascata di adder del moltiplicatore oltre che per il successivo sommatore.

Questi circuiti nella precedente documentazione erano illustrati a livello *register transfer level* (*RTL*) per ragioni di chiarezza: per l'implementazione ci serve vedere come sono fatti a livello di porte logiche.

Gli *rca* sono formati da una serie di adder opportunamente collegati: gli adder sono half-adder o full-adder a seconda del peso del bit.

Si possono vedere gli schemi alla pagina 5.

3 Script `mac_generator.py`

La stesura manuale del file `.bench` del mac ad 8 bit non mi sembrava un approccio furbo vista l'architettura piuttosto complessa da rappresentare. Probabilmente proseguire con tale metodologia 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 *circuiti_prova*:



Figura 1: Schema del ripple carry adder.



Figura 2: Schema dell'half-adder.



Figura 3: Schema del full-adder.

non sono fondamentali per il progetto però possono aiutare alla comprensione della metodologia usata.

Appurata la struttura dei circuiti base, ho creato uno script monolitico *mac_generator.py* che attraverso alcuni cicli e condizioni li collega in maniera consistente con gli schemi a mia disposizione. Questo script permette di generare mac con dimensione arbitraria degli ingressi da passare come argomento. Ad esempio 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 a 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 non sarebbe stato efficiente.

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.

3.1 Versione compatibile con Python 2.7

Lo script `mac_generator.py` funziona con Python 3.8.10 e dovrebbe funzionare con ogni versione superiore alla 3.6.

Ho realizzato una versione compatibile con Python 2.7, nella cartella *python2.7* sotto il nome di *mac_generator_python2.py*: 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 cosiddette "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 dà errore quindi ho forzato l'encoding utf-8 con un "commento magico" nella prima riga di codice.

- La concatenazione 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 Testing del circuito con HOPE

Ottenuto il file `.bench` utilizzando lo script precedente possiamo passare ad usare HOPE.

Spostandosi nella cartella di installazione di HOPE si può aprire la guida utente scrivendo a terminale:

```
./hope -h g
```

Per l'analisi dei nostri circuiti in particolare useremo questo comando:

```
./hope -F fn -r n -0 -N -s m mac_{bit}.bench
```

Per spiegare il comando riporto una mia traduzione dalla documentazione di HOPE dei vari argomenti:

- "-F fn": Vengono riportate nel file `fn` le uscite del circuito buono e difettoso per ogni errore. Con questa opzione, l'euristica di fault dropping di HOPE non viene eseguita, vale a dire che tutti i guasti vengono iniettati e simulati in parallelo. (default: l'uscita del circuito difettoso non viene segnalata.)
NDR: Le linee dove il guasto viene rivelato hanno un asterisco.
- "-r n": (Modalità a pattern random) I pattern di test sono generati in maniera random. La simulazione di guasto termina se se tutti i guasti sono stati rivelato oppure se sono stati applicati `n` pattern. (default: -r 224)
- "-0": tutti i flip-flop sono settati inizialmente al livello logico 0
- "-N": Modalità diagnostica. Il fault dropping non è eseguito. Quindi, tutti i guasti sono simulati per ogni pattern di test. (default: i guasti rivelati durante la simulazione di guasto vengono tolti dalla lista dei guasti.)

- "-s m": Il seme iniziale del generatore casuale di numeri è impostato da m. Se m=0, il seme casuale è generato usando l'orario del computer. (default: -s 0)

In breve: avendo dei flip-flop li settiamo a zero ed eseguiamo il test con m vettori di test, disabilitando il fault dropping, salvando il risultato in un file con tutti i vari guasti e fissando un seme per rendere i test ripetibili.

Vedremo degli esempi nel capitolo di analisi dei circuiti.

5 Script `getstats.py`

L'analisi del file salvato da HOPE sarà fatto da uno script Python che ho chiamato *gestats.py*.

Anche in questo caso il codice è ampiamente commentato e consiglio di leggerlo per comprenderne il funzionamento.

In breve questo programma esegue questo algoritmo:

- Apre il nostro file e con un ciclo lo legge riga per riga
- Se individua la parola "test" in una riga, vuol dire che sono all'inizio di un nuovo test: da questa riga posso ricavare i vettori di ingresso e l'uscita attuale del circuito.
- Ad ogni riga successiva ho l'iniezione di un guasto, conto queste righe per sapere il numero di guasti iniettati, inoltre:
 - Una riga con asterisco significa che ho un guasto rivelato in uscita: conto questa riga nel contatore dei guasti rivelati
 - Se non ho un guasto non c'è l'asterisco
- Se individuo di nuovo la parola test:
 - Vuol dire che è terminato il test precedente quindi devo calcolare la statistica che mi interessa che è la probabilità di errore in uscita, data dalla divisione tra i guasti rivelati e i guasti iniettati.
 - Azzero i contatori e continuo.
- Quando esco dal ciclo calcolo le statistiche per l'ultimo test.

Per salvare l'output del programma si può ridirigere l'output da terminale a un file a piacimento: es. `"getvectors.py mac > output_mac"`. Ridirigere l'output inoltre fa risparmiare molto tempo di esecuzione: mostrare a terminale tutto rallenta di molto l'esecuzione del codice.

Il programma inoltre salva i vari vettori in file di testo sotto forma di array `"vectors_mac"`, questo è un semplice dump, probabilmente non di grande utilità.

Alla fine di tutto questo lo script che gira su versioni di Python superiori alla 3.6 creerà un grafico a barre utilizzando matplotlib che illustra la probabilità di errore per ogni test.

Purtroppo non sono riuscito a far girare questa feature con Python 2.7, in quanto non sono riuscito ad installare matplotlib per Python 2.7, probabilmente per una questione di ambiente, bisognerebbe provare con degli ambienti virtuali, ma non sono pratico nel loro utilizzo anche se ne comprendo l'utilità.

6 Analisi dei circuiti mac

6.1 Analisi del mac 4 bit

Incominciamo con l'analisi di un mac a 4 bit.

1. Creiamo il file `mac_4.bench`.

```
python3 mac_generator.py 4
```

2. Copiamo il file nella cartella di *HOPE* e lanciamo il comando:

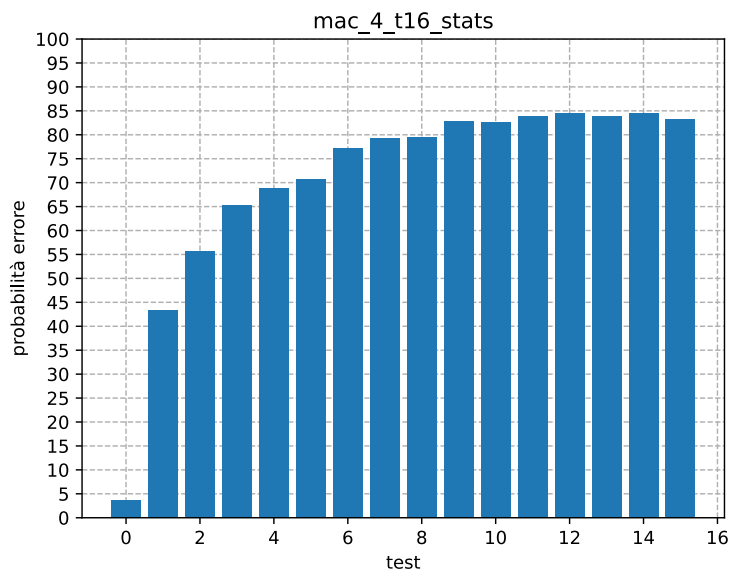
```
./hope -F mac_4_t16 -r 16 -0 -N -s 1 mac_4.bench
```

N.B.: stiamo testando il circuito con un pattern di 16 vettori.

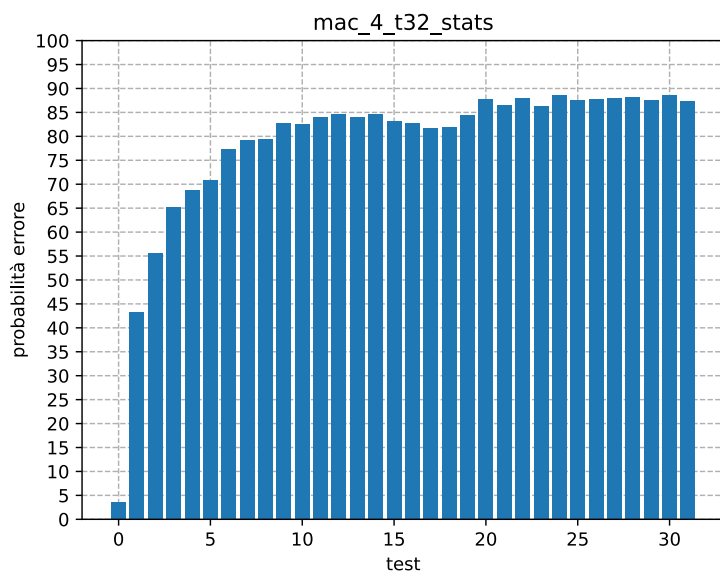
3. Copiamo il file `mac_4_t16` nella cartella con i gli script Python e lanciamo il comando:

```
python3 getstats.py mac_4_t16
```

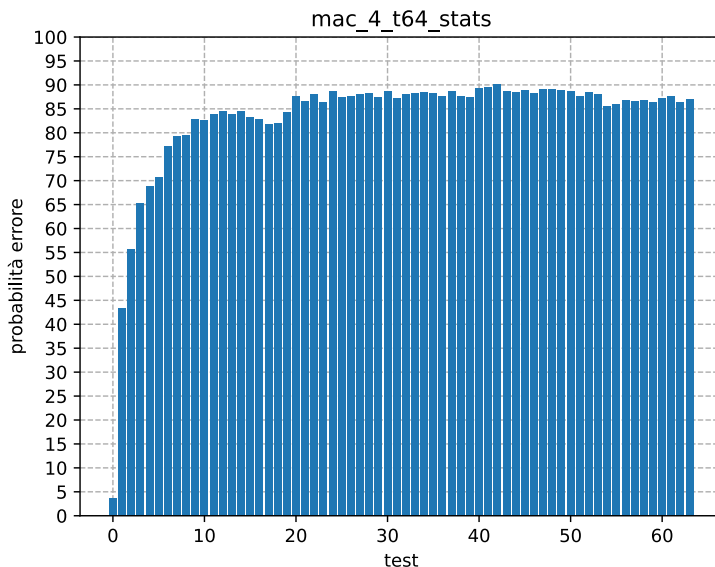
Questo è il grafico risultate:



Ripetendo lo stesso procedimento per con un pattern di 32 vettori di test abbiamo questo grafico:



Ripetendo lo stesso procedimento con un pattern di 64 vettori di test abbiamo questo grafico:



Notiamo quindi che raggiunti i 20 test, la probabilità di errore in uscita rimane intorno al 85

Sarei potuto partire direttamente dal grafico con 64 vettori, nel mentre che facevo i test ero partito da 16 e ho scalato e mi sembrava un peccato non condividere i test.

Passiamo ad analizzare nello stesso modo il circuito ad 8 bit.

6.2 Analisi del mac 8 bit

Il procedimento è sempre lo stesso, lo riporto nuovamente anche se non necessario. Qui partiamo subito con 64 vettori di test.

1. Creiamo il file *mac_8.bench*.

```
python3 mac_generator.py 8
```

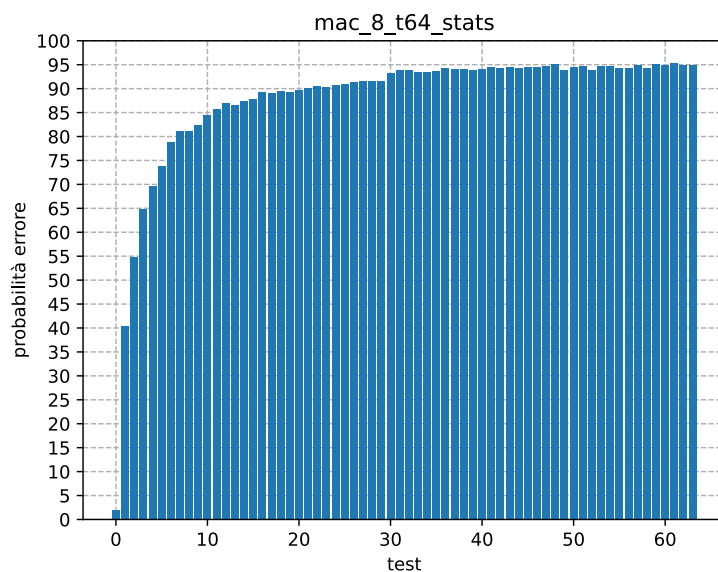
2. Copiamo il file nella cartella di *HOPE* e lanciamo il comando:

```
./hope -F mac_8_t64 -r 64 -O -N -s 1 mac_8.bench
```

3. Copiamo il file *mac_8_t64* nella cartella con i gli script Python e lanciamo il comando:

```
python3 getstats.py mac_8_t64
```

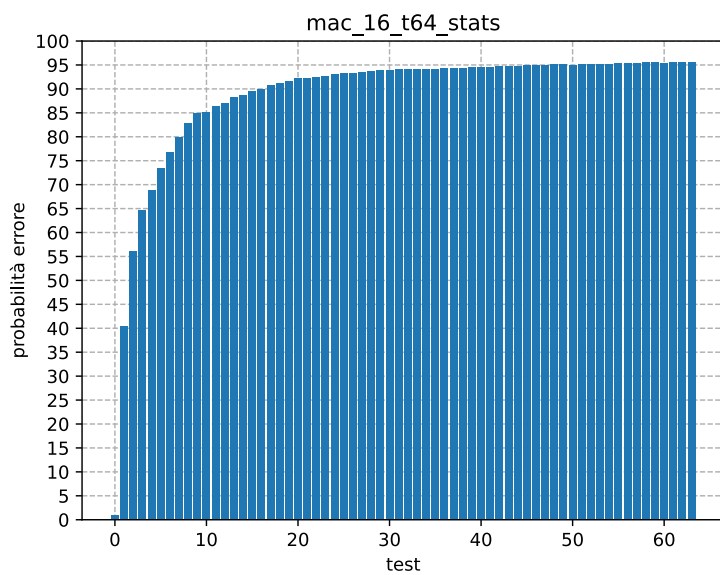
Il grafico risultante è questo:



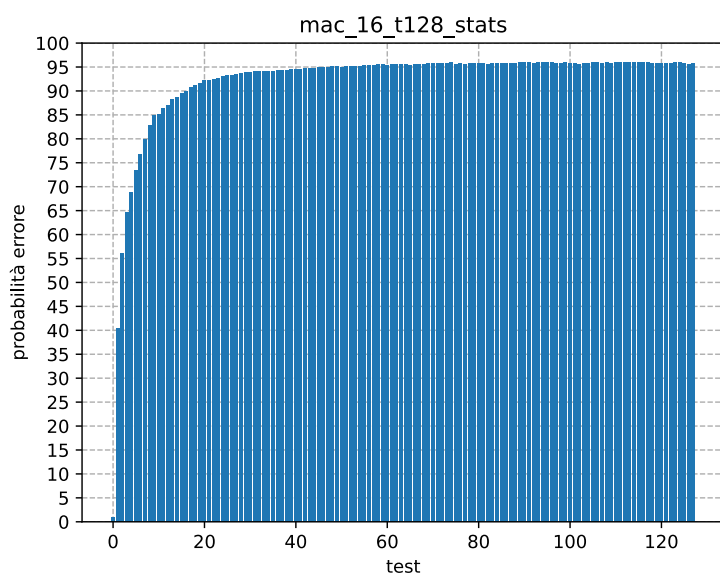
Abbiamo anche qui un comportamento simile a quello visto prima: indicativamente a 30 vettori la probabilità di errore va in saturazione.

6.3 Analisi del mac 16 bit

Vediamo il grafico per il circuito a 16 bit con 64 vettori di test:



Per curiosità vediamo anche il grafico con 128 vettori di test.



6.4 Ulteriori analisi

Ho provato con il mac a 32 e 64 bit ma *HOPE* dà un errore di segmentazione con l'opzione -F, i circuiti sono testabili normalmente ma per qualche motivo non riesce a salvare il file di tale opzione.