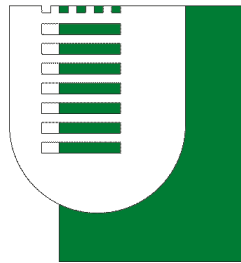


# TOR VERGATA



UNIVERSITÀ  
DEGLI STUDI  
DI ROMA

Macroarea di Ingegneria

Tesi di Laurea Triennale in Ingegneria Elettronica

*Progettazione ed implementazione, su logica  
programmabile, di una cuckoo hash table  
ricongfigurabile*

**Relatore:**

Prof. Giuseppe Bianchi

**Candidato:**

Alessandro Rivitti

**Correlatore:**

Prof. Salvatore Pontarelli

Dott. Aniello Cammarano

Anno Accademico 2018/2019

# Indice

INTRODUZIONE.....	5
CAPITOLO1: SDN .....	6
1.1    Introduzione Reti .....	7
1.2    Software Defined Networking.....	8
1.3    Control Layer Centralizzato.....	11
1.4    P4.....	13
CAPITOLO 2: FLOWBLAZE .....	17
2.1 Data Plane Stateful .....	17
2.2 FlowBlaze .....	22
2.3 XFSM .....	24
2.4 Machine Model .....	26
2.5 Hardware Design .....	28
2.6 Programmazione .....	29
2.7 NetFPGA.....	29
CAPITOLO 3: HASH TABLES.....	31
3.1 Hash Tables .....	31
3.2 Cuckoo Hashing.....	34
3.3 Hash Tables e Stash FlowBlaze.....	41
3.3.1 HT128new.vhd .....	44
3.3.2 Stash.vhd.....	46
3.4 Block Ram .....	47

CAPITOLO 4: MODULO RICONFIGURATORE .....	49
CAPITOLO 5: SIMULAZIONE E IMPLEMENTAZIONE .....	52
5.1 Test Bench Stand-Alone.....	60
CAPITOLO 6: SETUP E RISULTATI SPERIMENTALI .....	63
CONCLUSIONI E SVILUPPI FUTURI .....	72
BIBLIOGRAFIA .....	72

*“You miss 100% of the shots you don’t take      -Wayne Gretzky”*

-Michael Scott

# *Introduzione*

Le infrastrutture di rete inizialmente erano costituite da devices con hardware dedicato per implementare funzioni diverse (firewall, router, switches...). Avremmo avuto un device specifico che operava come switch e solamente come switch.

In un secondo momento, per incrementare la flessibilità dell'hardware e quindi abbassarne i costi, si è passato ad una implementazione software delle funzioni di rete aspecificando l'hardware. La potenza di questo tipo di astrazione, proposta come "network softwarization", è la possibilità di utilizzare hardware di produttori diversi per implementare qualunque tipo di funzione di rete. Tuttavia, sono state compromesse le performance dovendo eseguire molte operazioni a livello software. Nell'articolo che ha inizialmente proposto OpenFlow (protocollo basato sull'idea di SDN, Software Defined Networking), si parlava di rendere aperti i dispositivi di produttori diversi senza costringerli ad "aprire la scatola", ovvero rendere pubbliche le Intellectual Properties del loro hardware. Una rete dunque sarà realizzabile con un basso costo (non sarà necessario creare nuovo hardware per ogni specifica funzione) e usiamo i device già presenti come semplici dispositivi di forward.

È importante notare tuttavia come OpenFlow sia stato "imposto" e offre alcune limitazioni in ambito di programmabilità. Nonostante ciò, un possibile utilizzo di OpenFlow è tale da poter implementare funzioni stateful che riducono le lente operazioni che sarebbero affidate in software al piano di controllo. Potrebbe sembrare come un passo indietro rispetto alla proposta originale di SDN poiché riportando parte del control layer nel data plane ma è da notare come ciò non sia propriamente esatto. Attraverso macchine a stati gestite in hardware è possibile installare a livello di data plane funzioni complesse pur mantenendo la velocità del piano hardware. FlowBlaze si propone come semplice approccio per la programmazione di funzioni stateful in hardware.

Esiste un prototipo funzionante ed open-source di FlowBlaze implementato sulla NetFPGA, una smart NIC della Xilinx. Un importante elemento costitutivo di FlowBlaze, e in generale della maggior parte dei dispositivi di rete, sono le hash tables usate per salvare e cercare velocemente un dato. L'utilità di esse risiede principalmente nella possibilità di constatare la presenza o meno, all'interno della tabella, di un elemento senza doverla controllare tutta. Solitamente è comune usare hash tables "statiche" per quanto concerne numero di righe, dimensioni delle chiavi e dei valori associati, ovvero viene deciso a priori lo spazio di memoria che dovrà essere allocato per ognuna di esse. Per applicazioni che richiedono diverse tipologie di dati da salvare si incorre nel problema di celle staticamente allocate e inutilizzate.

Il mio lavoro di tesi si è basato sull'ottimizzazione della Hash Table usata da FlowBlaze per la gestione della State Table. In particolare, ho implementato un modulo per la riconfigurazione dinamica delle dimensioni della tabella, permettendo di ottimizzare la memoria utilizzata variando le dimensioni della chiave e dei valori da inserire nella tabella. È stato dunque per me necessario:

1. Studiare inizialmente l'architettura di FlowBlaze e l'implementazione su FPGA della cuckoo hash table
2. Ideare un blocco che rendesse dinamica la dimensione della memoria specificandone la dimensione di chiavi e valori da salvare
3. Simulare e sintetizzare su FPGA la HT dinamica come blocco a sé stante
4. Integrare nel prototipo FlowBlaze la HT riconfigurabile, incrementandone la flessibilità di utilizzo gestendo opportunamente la dimensione di chiavi e valori
5. Infine, simulare, sintetizzare e testare il funzionamento su FPGA di FlowBlaze integrato della mia implementazione della hash table

# Capitolo 1: SDN

Le reti odierne possono essere divise in tre piani funzionali: Application plane, Control plane e Data Plane

- **Application plane:** è il livello più alto di astrazione nel quale vengono specificati i servizi software che andranno ad essere implementati nella rete
- **Control plane:** Viene definita la topologia della rete che soddisfa l'applicazione definita a livello superiore ed essa è tradotta in azioni di match-action che andranno a popolare le flow-tables nel livello inferiore. Comunica con l'application layer attraverso una Northbound API (Application Programming Interface) e con il dataplane attraverso una Southbound API (North e South dipingono un'immagine della verticalità della rete. Sono spesso usati anche east e west quando parliamo di interconnessioni tra i vari control planes).
- **Data plane:** anche detto forwarding plane (livello di "inoltro" degli effettivi pacchetti) è il livello hardware dove avviene l'inoltro dei flussi da un'interfaccia di rete all'altra rispettando le regole definite nelle flow-tables.

## 1.1 Introduzione Reti

Nelle reti tradizionali i due piani, di controllo e dati, sono integrati negli stessi dispositivi. Nella realizzazione di una rete complessa avremo dunque un insieme di switch ognuno con il proprio control plane e il risultato di tale architettura è una struttura altamente decentralizzata. Ogni distributore di switch infatti utilizzerà dei protocolli proprietari per la gestione del traffico e per la popolazione delle flow-tables. Questo è stato considerato importante per la progettazione di Internet: sembrava il modo migliore per garantire l'elasticità della rete, che era un obiettivo progettuale cruciale. Tuttavia, una simile

architettura compromette inevitabilmente innovazioni ed evoluzioni delle infrastrutture di rete. Né è un esempio lampante quella che avrebbe dovuto essere una semplice transizione da IPv4 e IPv6 avviata più di un decennio fa e ancora largamente non adottata nonostante si trattasse di un semplice aggiornamento di protocollo. Analogamente un'infrastruttura di rete ideata da zero, seppur portasse con sé un incredibile miglioramento delle performance, sarebbe impossibile da mettere in pratica.

## 1.2 Software Defined Networking

La Software-Defined networking è un'astrazione che propone la separazione fisica tra il network control plane e il data plane introducendo un controller centralizzato. [1]

### Software Defined Network (SDN)

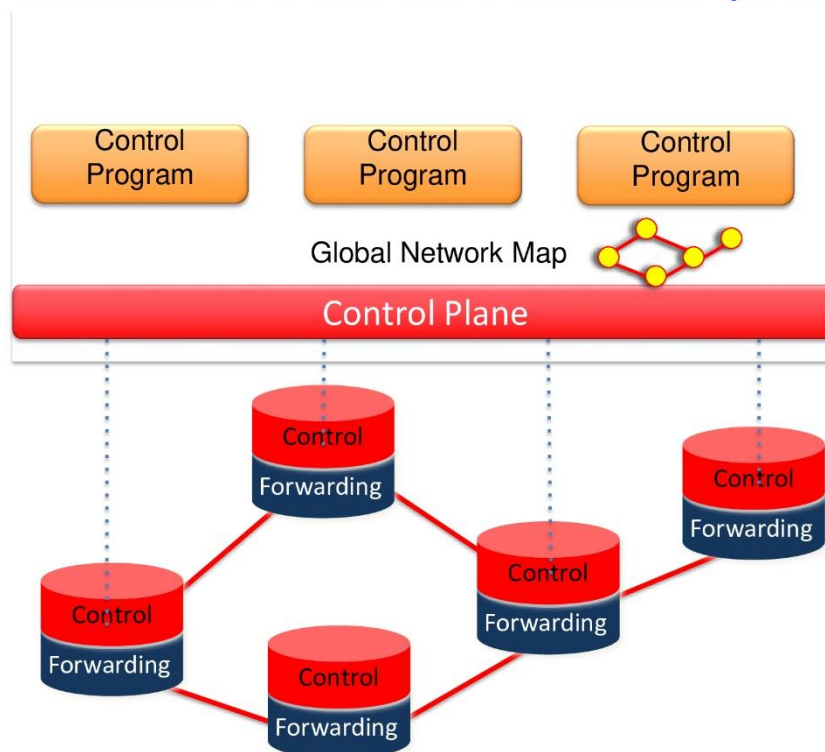


Figura 1: Software Defined Networking Astraction



Un enorme vantaggio è quello di spezzare l'integrazione verticale rendendo così possibili evoluzioni e riconfigurazioni protocollari. Un secondo vantaggio nella separazione logica dei due piani, non di minor importanza, si ha anche in ambito di performance. Vanno infatti presi in considerazione i "tempi di reazione" [2] associati a questi due layer:

- un control plane stabilisce in software le rotte per le connessioni punto-punto (ordine dei millisecondi);

- un data plane determina il comportamento dei pacchetti in ingresso ad uno switch (ordine dei nanosecondi).

Con un layer di controllo centralizzato vengono ridotte drammaticamente le operazioni da svolgere in quanto riduciamo la possibile ridondanza introdotta da operazioni ripetute da control plane integrati in switch vicini.

Il compito sebbene a primo impatto relativamente arduo, avendo infatti già detto che ogni vendor usa un protocollo proprietario diverso da quello di ogni altro competitor per comunicare a livello southbound tra controller e switch, è stato possibile attraverso una soluzione comune. Per centralizzare i controller in unico dispositivo infatti è stata necessaria l'introduzione di un nuovo protocollo che si potesse interfacciare con dispositivi di vendors diversi rendendo gli switch semplici dispositivi di inoltra (introducendo difatti l'espressione di "dumb switch" il cui unico scopo è quello di forward). Nel 2011 è stato ideato e presentato quindi il protocollo OpenFlow per rispondere a questa necessità. [3]

# Software Defined Network (SDN)

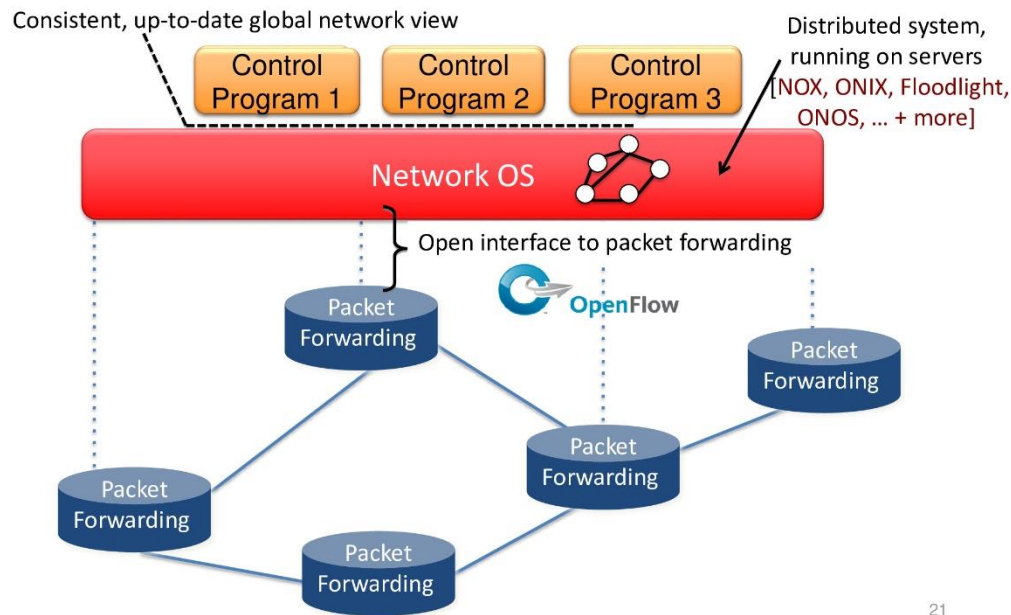


Figura 2: Controller centralizzato e switch "dumb" col solo scopo di packet forwarding

È da notare come un piano di controllo logicamente centralizzato non debba corrispondere ad un sistema fisicamente centralizzato. Una simile soluzione sarebbe di fatti poco scalabile e modificabile. Reti implementate in SDN infatti prevedono la possibilità di avere control planes fisicamente distribuiti.

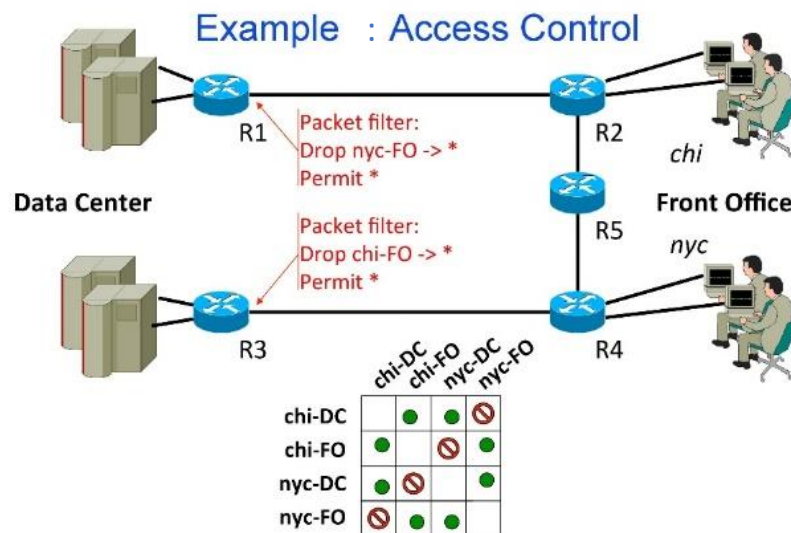
La separazione tra i due layer diventa quindi possibile avendo un'interfaccia di programmazione tra gli switches ed il controller SDN (southbound API) di cui ne è solo un esempio OpenFlow.

È di cruciale importanza notare che è propria questa la vera potenza di OpenFlow e in generale di SDN ovvero l'aver ideato un'astrazione di programmazione che non dipenda dal vendor e non imponga quindi la conoscenza dello specifico device operando a "scatola chiusa".

Uno switch OpenFlow ha una o più tabelle, definite flow-tables dove sono specificate regole di gestione dei pacchetti. Queste tabelle sono dette match-action tables (MAT) in quanto di un pacchetto in ingresso viene controllata una certa porzione (match), ad esempio l'header, e per tale valore viene controllato all'interno della tabella che tipo di azione (action) è associata con esso. In base all'applicazione specificata dal controller, uno switch OpenFlow può agire come router, firewall, load balancer... attraverso la semplice e unica operazione di popolazione delle tabelle.

## 1.3 Control Layer Centralizzato

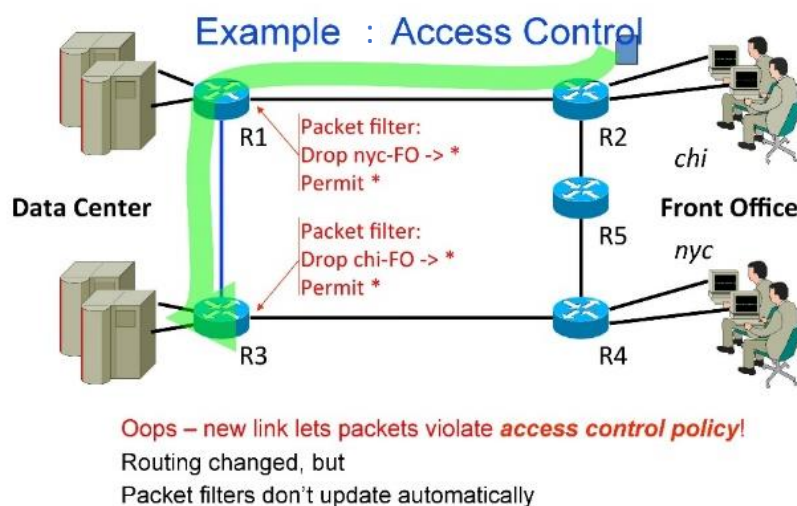
Riportiamo ora un esempio di forwarding di una normale rete e di come un SDN possa portare vantaggi in ambito di malfunzionamenti dati dalla non conoscenza globale della rete.



20

Figura 3: Esempio di rete governata da controller locali

Nell'esempio abbiamo due data center di due uffici interconnessi tra di loro da 5 switch. In R1 e R5 vengono inserite due regole nelle tabelle in modo da filtrare i pacchetti da un ufficio in ingresso al data center dell'altra città in modo che ogni data center sia accessibile solo dal relativo ufficio. Se ad esempio l'ufficio di Chicago volesse provare ad accedere al data center di New York, in ingresso ad R3 verrebbe controllato il campo del pacchetto relativo all'indirizzo sorgente e, soddisfacendo la regola inserita nel filtro, verrebbero scartati. In un secondo momento viene deciso di aggiungere un collegamento diretto tra gli switch nei pressi dei due data centers in modo da poter scambiare informazioni tra di essi. Si può notare come involontariamente si è creato un collegamento tale da rendere nulli entrambi i filtri. Come da immagine vediamo che l'ufficio di Chicago potrà ora accedere all'altro data center passando per lo switch del data center della propria azienda.



22

Figura 4: La politica di filtraggio è stata violata non avendo un controller centralizzato

Sarebbe necessario infatti un aggiornamento delle flow tables in modo da bloccare un tipo di traffico simile. Se tutti gli switch condividessero il control plane, si avrebbe una visione

globale del traffico e sarebbero state automaticamente riscritte le flow tables per garantire la regola definita nel control layer.

Un altro degli obiettivi di SDN è quello di gestire il traffico basandosi sul flusso e non sulla destinazione. Si parla di flusso quando viene inviata una sequenza di diversi pacchetti tra una determinata sorgente e una destinazione. È facile notare come per situazioni di questo tipo, tutti i pacchetti avranno inevitabilmente stesse politiche di servizio da parte dei dispositivi di forwarding. Sarà dunque necessario comunicare a livello di control plane (ordine di millisecondi) solo per il primo pacchetto e, dopo esser state popolate le flow tables, verranno processati tutti gli altri pacchetti attraverso il data plane (ordine dei nanosecondi).

Attraverso un approccio software-defined networking è possibile quindi snellire il traffico all'interno di una rete implementando una certa gestione definita nell'application layer in unico controller. Lo smart controller, attraverso OpenFlow, popolerà tutte le flow-tables degli switch ad esso annessi. Avendo un controller con conoscenza globale della rete avremo inoltre una semplificazione nello sviluppo di funzioni e applicazioni di rete più sofisticate.

Nonostante SDN e OpenFlow siano nati come studi accademici, hanno ricevuto un'attenzione nel campo dell'industria. Google, Facebook, Yahoo, Microsoft, Verizon e la Deutsche Telekom hanno investito nella ONF (Open Networking Foundation) con lo scopo di promuovere l'adozione di SDN attraverso standard comuni.

## 1.4 P4

Nel proseguire a descrivere passo passo il processo di sviluppo che hanno subito le infrastrutture di rete prima di introdurre FlowBlaze, introduciamo ora P4.

OpenFlow si basa su un preciso set di headers sul quale può operare per distinguere diversi protocolli. Questo set è stato accresciuto da 12 a 41 in pochi anni aumentando la complessità delle specifiche non apportando tuttavia una flessibilità nell'aggiunta di headers diversi da quelli del set stesso. In foto è riportato il numero di headers di OpenFlow associato alle relative versioni del protocollo fino alla versione OF 1.4. (Nel periodo in cui è stata scritta questa tesi, OpenFlow è stato aggiornato fino a OF 1.6 anche se questa versione è stata rilasciata solo internamente alla ONF).

Version	Date	Header Fields
OF 1.0	Dec 2009	12 fields (Ethernet, TCP/IP <sub>v4</sub> )
OF 1.1	Feb 2011	15 fields (MPLS, inter-table metadata)
OF 1.2	Dec 2011	36 fields (ARP, ICMP, IP <sub>v6</sub> , etc.)
OF 1.3	Jun 2012	40 fields
OF 1.4	Oct 2013	41 fields

*Figura 5: Storico versioni rilasciate da OpenFlow*

Con P4 si è cercato di migliorare e di generalizzare la programmazione dei device atti al processamento di pacchetti estraendo la dipendenza da un protocollo ben definito da un set. Esso è un linguaggio di programmazione di alto livello per configurare il processamento di pacchetti. Si basa su tre punti cardine: [4]

- Riconfigurabilità del campo: I programmatori devono avere la possibilità di cambiare il comportamento di processamento dei pacchetti da parte degli switch una volta che sono stati configurati.
- Indipendenza protocollare: come già accennato è indispensabile che gli switch non siano legati da un protocollo di rete ben specifico.

- Indipendenza del target: P4 si propone come linguaggio per descrivere funzioni di processamento di pacchetti indipendentemente dalle specifiche dell'hardware che verrà poi usato.

Per rispondere ai 3 capisaldi appena proposti, P4 permette in primo luogo al controller di riprogrammare il parser dei pacchetti in ingresso agli switch rispondendo al requisito di Riconfigurabilità. Per garantire una generalizzazione dei protocolli, inoltre, il parser sarà programmabile in modo da poter decidere specifiche porzioni dell'header da controllare e da confrontare in tabelle MAT. Infine, P4 si propone come una descrizione target-independent analogamente a come lo è C. Sarà poi il compilatore (come gdb nel caso di C) che adatterà il codice alla specifica CPU o, in P4, allo specifico switch.

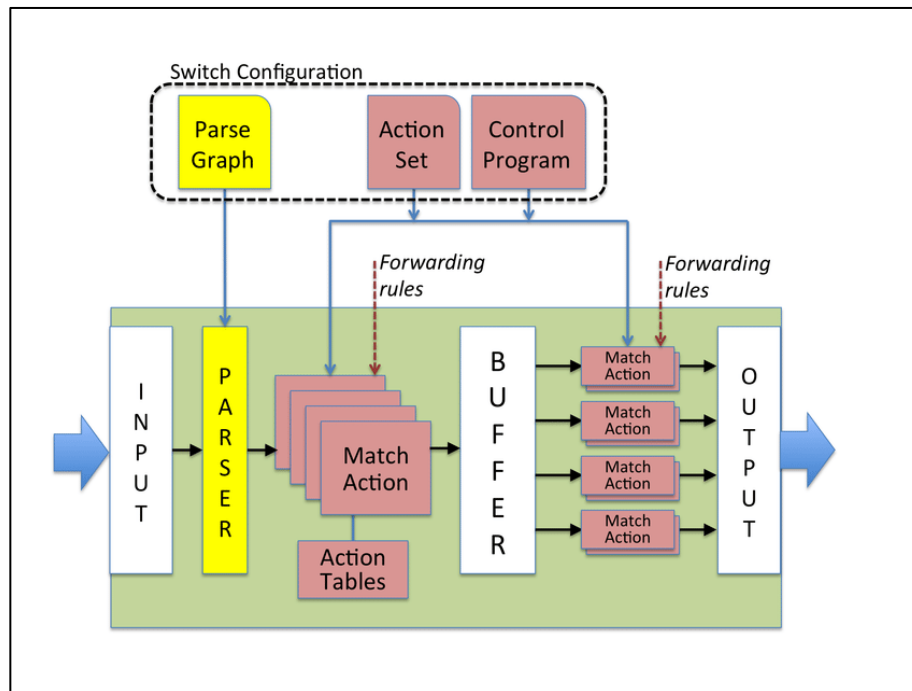


Figura 6: Switch P4. Parser programmabile in ingresso; pipeline di tabelle Match+Action

L'idea alla base di P4 è quella di avere switch che inoltrano pacchetti utilizzando un parser programmabile seguito da uno o più stadi di match-action in serie, parallelo o una

combinazione dei due. P4 ottimizza OpenFlow aggiungendo dunque un parser programmabile e non fisso, le MAT in parallelo oltre che in serie e azioni avulse da protocolli fissi.

Il pacchetto in ingresso allo switch è gestito in primo luogo dal parser mentre il corpo del pacchetto è salvato in un buffer separato. Il parser campiona ed estrae porzioni dall'header. Vengono quindi passati i campi selezionati alle match+action. Come possiamo vedere in figura dividiamo le MATs in ingress e egress. Le prime sono impiegate nel determinare le porte di uscita e la coda dove dovranno essere indirizzati i pacchetti mentre entrambe possono essere usate per modificare porzioni dell'header. Un pacchetto in ingresso ad una MAT di ingress potrà essere inoltrato (forward), replicato (nel caso in cui il pacchetto sia di tipo multicast oppure debba essere inviato al control plane), scartato (drop) oppure azionerà il control layer.

I pacchetti possono portare con sé del metadata ovvero delle informazioni aggiuntive che verranno controllati in stadi successivi delle MAT; un esempio di metadata può essere la porta 22 in una connessione SSH o 80 in una normale richiesta HTTP.



## Capitolo 2: FlowBlaze

### 2.1 Data Plane Stateful

Una macchina a stati è un'ottima astrazione per descrivere un problema relativamente complicato e renderlo più semplice e intuitivo. Sono molto efficaci inoltre nel descrivere funzioni di rete ("network functions").

Un data plane stateful può essere un enorme passo avanti rispetto all'astrazione iniziale della SDN dove il control layer ha il pieno controllo della gestione della rete. Non useremo più switches solo per l'inoltro di pacchetti ma implementeremo funzioni di rete anche molto complicate negli switch stessi, grazie alle macchine a stati, togliendo una gran parte del carico di lavoro che sarebbe stato affidato al sistema di controllo software. [5]

Facciamo ora un esempio di come una macchina a stati possa semplificare l'implementazione di funzioni di rete che richiederebbero molti accessi nel control layer (incrementando inevitabilmente la latenza complessiva). Prendiamo in considerazione il port knocking, un tipo di firewall molto utilizzato per il quale per accedere ad una determinata risorsa è necessario "bussare" in ordine ad una certa sequenza di porte. Solo il knocking eseguito con il giusto ordine abiliterà l'accesso ad una certa porta; se ciò non dovesse avvenire, la totalità dei pacchetti in ingresso verrà scartata. [6]

I pacchetti del "knocking" sono usati come effettivo codice d'accesso ed hanno il solo scopo di far avanzare un determinato flusso nella macchina a stati; vengono anch'essi infatti scartati.

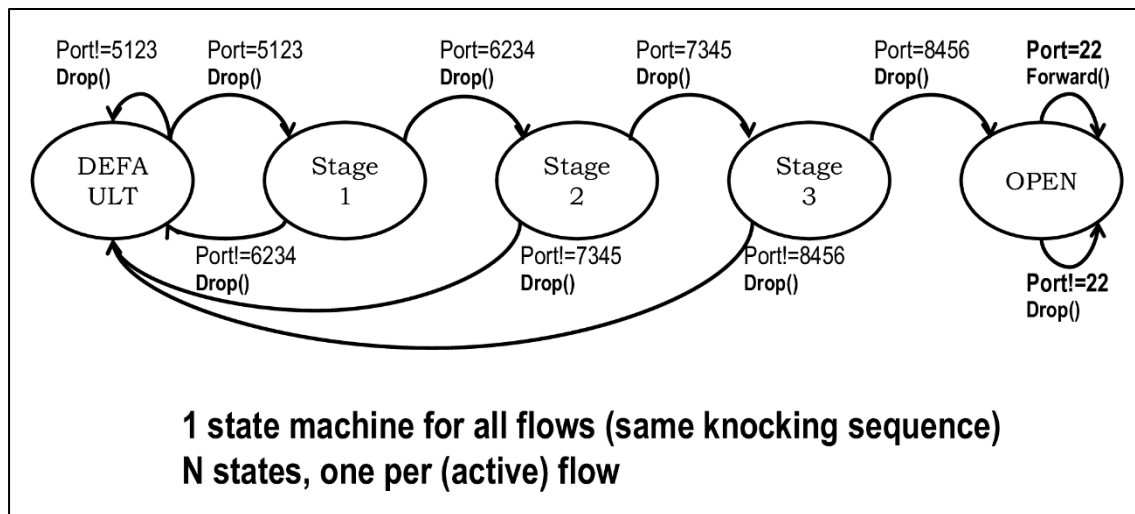


Figura 7: Port Knocking descritto attraverso una macchina a stati

Nell'immagine è mostrato un port knocking per l'abilitazione di una connessione ssh (porta 20) eseguito attraverso una macchina a stati.

Ad ogni flow in ingresso sarà associata una propria macchina a stati e per poter avviare una connessione ssh, un flow dovrà conoscere il "codice" dato dalla sequenza di porte 5123, 6234, 7345, 8456. Tutti i pacchetti saranno scartati fino a che un flusso non raggiunga lo stato OPEN e comunichi sulla porta 22.

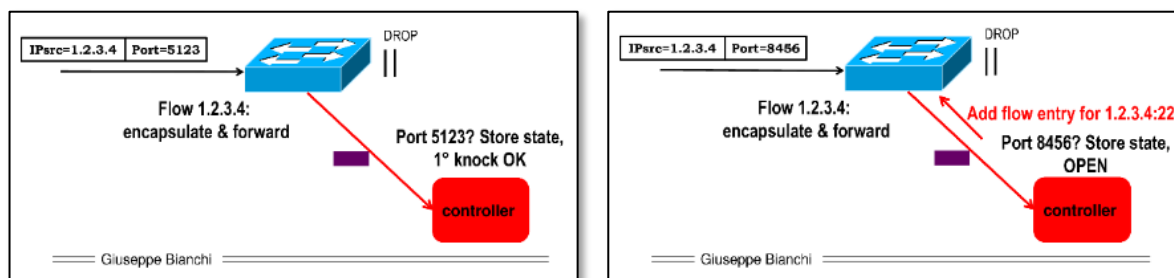


Figura 8: Port Knocking eseguito da uno switch OpenFlow

Se volessimo programmare un port-knocking in uno switch OpenFlow, ogni pacchetto in ingresso avrebbe dovuto essere inoltrato al control layer per poter aggiornare o meno le flow table dello switch. Infatti, come mostra la transizione nelle due figure, lo switch invierà ogni pacchetto al controller che aggiornerà o meno, per ogni flow, i vari knock effettuati (slide di sinistra). Solamente quando un flow avrà inviato l'intera sequenza di pacchetti allora il controller aggiungerà la rotta per l'inoltro di pacchetti alla porta 22 (slide di destra).

Vediamo invece come ciò sia implementabile a livello di data-plane attraverso una funzione stateful. Usiamo una eXtended Finite State Machine di cui chiariremo meglio il funzionamento più avanti ma di cui possiamo comunque apprezzare la logica di una macchina a stati. Utilizzeremo due principali tabelle, una State-Table e una XFSM-Table. La prima salverà lo stato di tutti i flussi in ingresso, mentre la seconda descriverà la classica relazione match-action con la sottile differenza per cui in parallelo ad un'azione sul pacchetto, eseguiremo anche un aggiornamento della State Table.

All'arrivo di un pacchetto in ingresso viene inizialmente controllato lo stato nella State-Table. Se il pacchetto dovesse appartenere ad un nuovo flusso, gli verrebbe assegnato lo stato di DEFAULT. Lo stato e la variabile in ingresso, (l'evento) che in questo caso è la porta, vengono inoltrati alla XFSM Table per eseguire il match-action.

## Putting all together

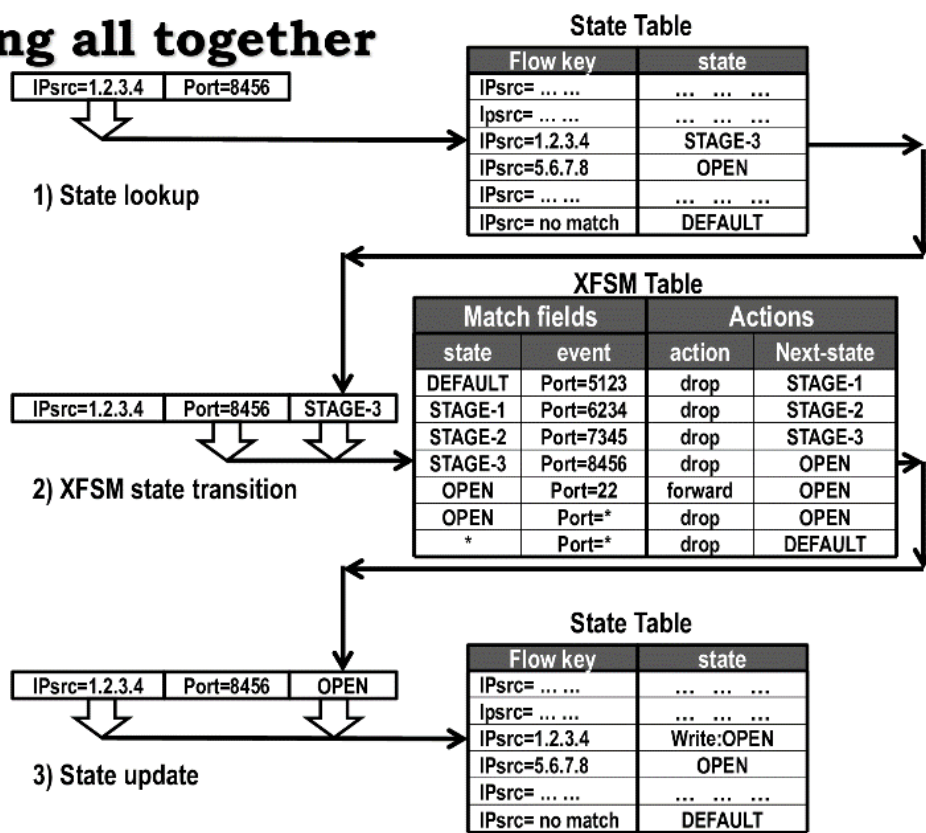


Figura 9: Port Knocking eseguito attraverso approccio stateful con l'uso di XFSM

Nell'esempio in figura abbiamo inviato l'ultimo knock necessario ad abilitare la connessione infatti lo stato del flusso in ingresso è STAGE3 e, avendo inviato un pacchetto sulla porta 8456 concorde con la sequenza, eseguirà un'azione di drop e aggiornerà lo stato in OPEN.

Notiamo come la state table ha necessità di avere dimensioni elevate per poter salvare lo stato di molti flussi diversi in ingresso. La XFSM Table invece specifica le regole di funzionamento per qualunque tipo di flusso in ingresso e non dovrà necessariamente avere molte entries salvate. Potremmo implementare la prima con una classica TCAM, ma per praticità di utilizzo useremo una hash table basata su RAM per salvare un elevato numero di entries. La seconda invece potrà essere realizzata in TCAM.

Come già accennato la performance di una XFSM rimane costante e non dipende dalla complessità della funzione implementata cosa che non può essere garantita da una normale astrazione match+action. Né sono da esempio gli usecases di impiego di FlowBlaze, che vediamo in foto, nei quali il Throughput è pressoché costante e notiamo invece l'impatto che hanno funzioni stateful su una NIC basata su Intel XEON. Quest'ultima infatti ha bisogno dell'utilizzo di almeno 3 cores per eguagliare il throughput che avrebbe per funzioni stateless.[7]

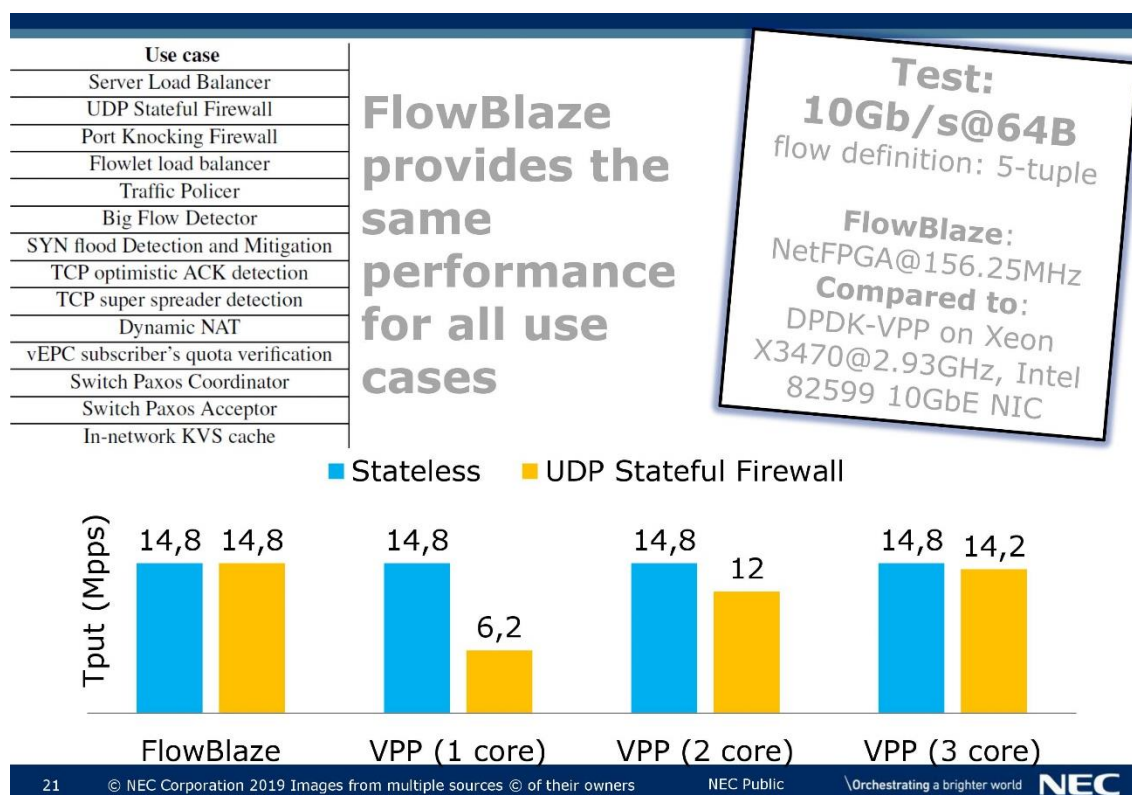


Figura 10: Report performance di funzioni stateful e stateless in FlowBlaze e NIC Xeon

## 2.2 FlowBlaze

Nel panorama odierno è sempre più comune vedere NIC (Network Interface Card) contenenti al loro interno anche una FPGA. Queste cosiddette SmartNICs hanno la possibilità di implementare funzioni di rete più o meno complesse. Programmare una SmartNIC come potrebbe essere la NETFPGA, tuttavia, può risultare complesso soprattutto per esperti di networking che non sono competenti nel basso livello di descrizione richiesto dagli HDLs (Hardware Description Languages) come possono essere VHDL e Verilog.

È richiesta una fase di progettazione, seguita da varie simulazioni e successivamente sintesi che possono richiedere diverse ore per dare esito positivo o meno di funzionamento in casi in cui, ad esempio, la sintesi non sia riuscita a soddisfare i constraints di timing richiesti.

FlowBlaze cerca di affrontare questo problema proponendosi come semplice approccio per la programmazione di funzioni di rete stateful o stateless per l'elaborazione di pacchetti in hardware. Lo scopo è stato quello di generalizzare le regole match/action tipiche di OpenFlow e P4 attraverso l'uso di XFSM (eXtended Finite State Machines) che operano direttamente nel dispositivo di switch, snellendo il carico di operazioni che verrebbero affidate al layer di controllo, eseguendo processamenti di pacchetti alla velocità del data plane. Va sottolineato come l'attenzione sia principalmente per funzioni che operano a livello di header del pacchetto, tipicamente quindi ai livelli L2-L4 della pila di rete come ad esempio firewalls, NATs e filtri di rete.

FlowBlaze si pone 4 obiettivi principali e vediamo come alcuni sistemi già esistenti ne soddisfino alcuni ma non tutti:

- R1: Elevate Performance: deve lavorare con funzioni che raggiungono throughput dai 40 ai 100 Gb/s mantenendo una latenza per processamento di un pacchetto non superiore di qualche  $\mu$ s.

- R2: Scalabilità della Macchina a stati: Il numero di flussi gestiti non deve influire la latenza di processamento. Sono richieste quindi funzioni che operano singolarmente per flusso e la capacità di salvare lo stato di un ampio numero di flussi (nell'ordine dei 100K).
- R3: Facilità di utilizzo: Non devono essere richieste al programmatore conoscenze di hardware e di HDLs e deve essere limitato l'impatto dell'hardware sulle varie funzioni di rete scelte.
- R4: Versatilità di utilizzo: si deve avere la possibilità di implementare funzioni di rete diverse e anche molto complesse

	High Perf	State Scal	Ease	Expresiv
General programming frameworks				
HDL	✓	✓	×	✓
HLS [52]	×	✓	✓	✓
ClickNP [42]	-	✓	-	✓
Match-action abstractions				
P4 [13]	-	-	✓	-
Domino [58]	✓	✓	-	-
OpenState [11]	✓	✓	✓	×
FAST [51]	×	✓	✓	×

Figura 11: State of the art di programmabilità dell'hardware di sistemi di rete

Nella tabella riportata confrontiamo lo stato dell'arte prendendo in considerazione alcuni sistemi già esistenti che soddisfano alcuni ma non tutti i requisiti sopra citati. Nella fattispecie i vari HDL (Verilog e VHDL) sono usati per programmare Smart-NIC basate su FPGA ma sono molto difficili da usare. Alcuni altri linguaggi di programmazione più semplici come OpenCL si basano su un più alto livello di astrazione ma non assicurano un'ottimizzazione nell'hardware che verrà poi implementato. Come già accennato invece sistemi basati su OpenFlow e P4 si basano esclusivamente su funzioni match+action privando la possibilità di programmare funzioni stateful.

## 2.3 XFSM

Per realizzare un data-plane stateful si è deciso di usare XFSM per evitare un problema principale che sarebbe potuto sorgere in hardware dall'impiego di una Finite State Machine classica. Si tratta della cosiddetta "state explosion" ovvero la necessità di dover allocare un numero eccessivamente elevato di transizioni per ogni singolo stato andando difatti a saturare la memoria. Per ogni stato infatti va definita la transizione allo stato successivo in relazione ad ogni possibile ingresso tale da definire la relazione

Transizione:  $\text{State} \times \text{Input} \rightarrow \text{Next State} \times \text{Output}$

Una eXtended Finite State Machine aggiunge una variabile D nella quale salveremo un'informazione per definire lo stato e la andremo ad aggiornare ad ogni accesso della macchina a stati. In questo modo ridurremo di molto il numero di stati necessari da allocare. La transizione sarà del tipo:

Transizione:  $\text{State} \times \text{Input} \times \text{Funzione} \rightarrow \text{Next State} \times \text{Output} \times \text{Update}$

Per fare un esempio riportiamo una macchina a stati di un'applicazione che identifica se un flusso è "grande" o meno andandone a contrassegnare tutti i pacchetti dopo il 100esimo. Ogni nodo corrisponde ad uno stato e ogni transizione dipende dall'evento, dal valore della funzione e produce un output e un update della tabella.



L'evento  $\text{pkt}(f_1)$  è relativo alla ricezione di un pacchetto del flusso preso in considerazione. Con  $\text{fwd}$  e  $\text{mark}$  sono definite delle descrizioni di altro livello di inoltramento del pacchetto e riscrittura di una porzione dell'header.

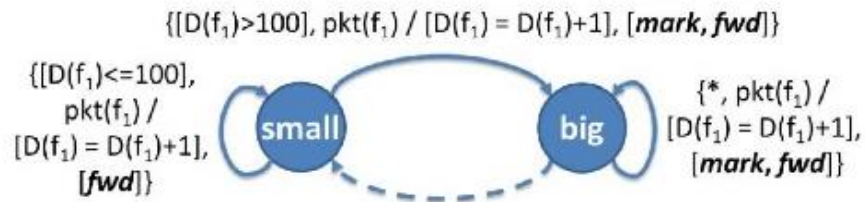


Figura 12: Esempio di XFSM e miglioramento dello "state explosion"

Il passaggio da state 2 a state 1 è tratteggiato per indicare il fatto che verrebbe eseguito da un timeout che scatterebbe dopo un certo lasso di tempo in cui non dovessero venir ricevuti pacchetti associati a quel determinato flusso nel quale verrebbe resettata la variabile  $D$ . Andando ad aggiornare ad ogni ingresso di un pacchetto associato al flusso  $f_1$  il counter salvato nella variabile  $D$ , possiamo implementare la macchina a stati con solo due stati per flusso. Iperboleggiandone il concetto, se invece avessimo voluto realizzare una simile applicazione senza una variabile associata allo stato, avremmo avuto la necessità di allocare 100 stati in cui transitavamo da uno all'altro all'arrivo di un nuovo pacchetto.

## 2.4 Machine Model

Come in una classica astrazione match-action si usano varie tabelle in serie in modo da creare una pipeline, così anche in FlowBlaze possiamo creare una pipeline di XFSMs dove l'output della XFSM (i) è l'input della XFSM (i+1).

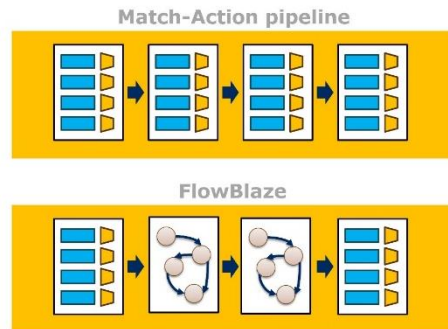


Figura 13: Pipeline di MATs e XFSM in FlowBlaze

Ad esempio, una prima macchina a stati può definire se un pacchetto sia “big” o “small” e utilizzare tale informazione come ingresso di un'altra state machine in sequenza. Più nello specifico con FlowBlaze viene ottimizzato il modello della pipeline di MATs. Gli header e i metadata dei pacchetti in ingresso vengono processati dagli elementi della pipeline per determinarne le azioni di inoltro.

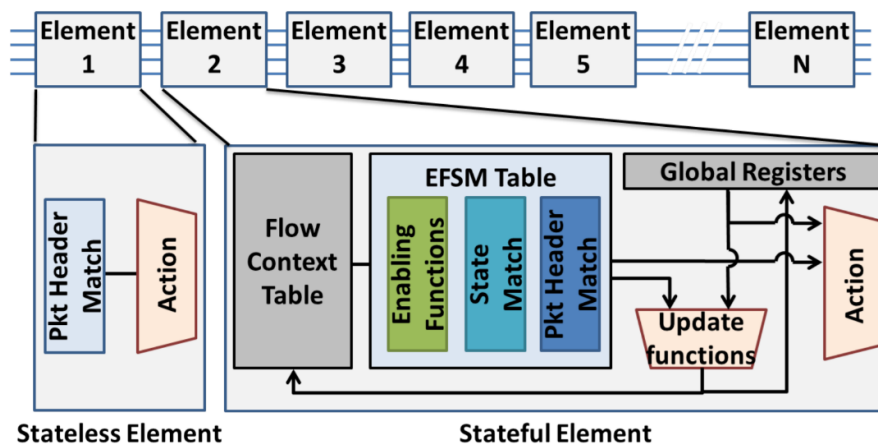


Figura 14: Schema a blocchi di un elemento Stateful in FlowBlaze

Ogni elemento della pipeline tuttavia può essere stateless, ovvero come una classica match-action table, oppure stateful nel quale useremo la definizione di XFSM. Quindi la pipeline sarà composta sia da elementi stateless sia da elementi stateful.

L'architettura di un elemento stateful, come mostrato nella figura del machine model, ha 2 differenze fondamentali rispetto ad una normale MAT: (1) prima di eseguire la normale operazione di match il pacchetto viene controllato dalla Flow-Context Table (che abbiamo precedentemente definito come State Table) nella quale ad un certo flusso in ingresso viene associato lo stato in cui si trova, e (2) all'operazione di match della XFSM table viene associata una azione e un aggiornamento della Flow Context Table nel quale verrà salvato il Next-State associato con il match.

Il processamento dei pacchetti prevede quattro passaggi sequenziali:

- **Flow Context Table:** Da un pacchetto in ingresso viene estratto un contesto (porzione di header e/o metadata) che verrà usato come search key nella flow context table. La chiave di ricerca viene specificata in fase di configurazione di FlowBlaze e ad ogni flusso viene associato uno stato e un array di registri. Se alla chiave cercata non è associata una entry nella tabella, ci sarà un contesto di DEFAULT che ne specificherà il comportamento.
- **XFSM Table:** L'header e il contesto appena estratto (stato e registri) sono passati alla XFSM table che opererà esattamente come una MAT, incapsulando o inoltrando il pacchetto, ma introducendo la possibilità di specificare un next-state e eseguire delle operazioni per aggiornare i registri nella Flow Context Table.
- **Update Functions:** In questo passaggio, in base alla transizione specificata nella XFSM, verranno eseguite operazioni che possono essere semplici somme o persino moltiplicazioni, ed infinite aggiornate le variabili globali, lo stato e i registri nella FCT.
- **Action:** Infine il pacchetto potrà o meno essere modificato e inoltrato all'elemento successivo della pipeline.

## 2.5 Hardware Design

Ci apprestiamo ora a dare una descrizione dettagliata dell'implementazione di un elemento stateful introdotto da FlowBlaze in quanto abbiamo già abbondantemente parlato, e in letteratura è ampiamente noto, come un elemento stateless sia implementabile con una TCAM e una RAM.

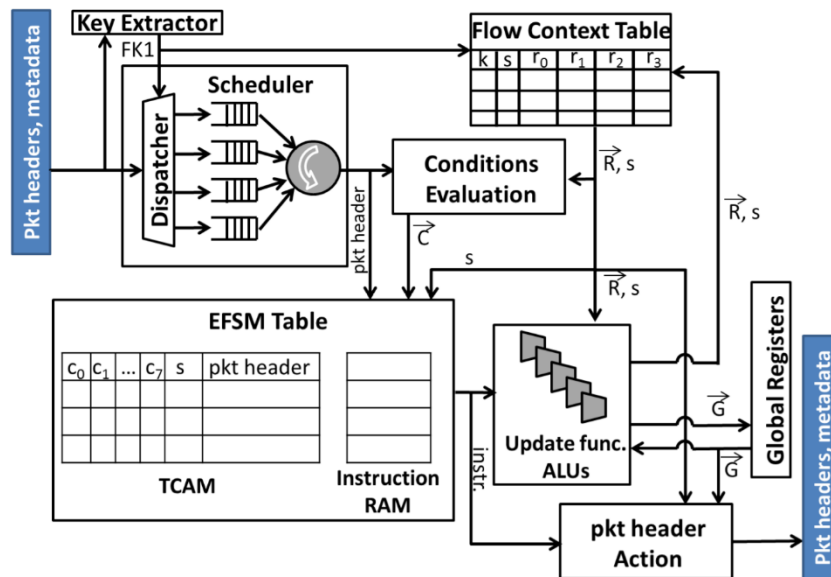


Figura 15: Schema a blocchi dell'implementazione hardware di FlowBlaze

L'header di un pacchetto è ricevuto e parsato dal key extractor (che opera come un parser programmabile). Generiamo così una chiave che contraddistinguerà un flusso nella XFSM. L'header è posto in una coda (in cui cercheremo di pipelineare flussi diversi) in base al valore della chiave. La chiave è quindi hashata e usata nella cuckoo hash table come search\_key per ottenere il corrispondente stato del flusso. Ad ogni chiave sarà associata, nella hash table, uno spazio di memoria che genericamente chiameremo value, nel quale salveremo stato e un insieme di registri utilizzabili. L'header, lo stato e il value sono serviti in ingresso alla XFSM realizzata da una TCAM e una RAM. A colpo di clock la TCAM valuterà gli ingressi

e darà in uscita l'indirizzo della sua Instruction RAM associata. L'Instruction RAM definirà l'azione da eseguire sul pacchetto, l'istruzione da passare all'ALU per aggiornare i valori della HT e i registri globali.

## 2.6 Programmazione

Per programmare FlowBlaze si usa XL (XFSM Language) un linguaggio di programmazione ad un livello di astrazione molto simile a quello di P4 tuttavia sviluppato per programmare funzioni di rete stateful. In fase di programmazione si definisce in XL la determinata funzione di rete che si vorrà implementare in FlowBlaze descrivendola come una macchina a stati partendo dal linguaggio d'alto livello. Verrà definito dunque il parser in ingresso e il tipo di azioni da eseguire. Il compilatore interpreterà il codice e restituirà un file che verrà usato per popolare i registri della FPGA attraverso protocollo AXI. I valori salvati all'interno di essi determineranno le politiche di match action della XFSM, le variabili globali o anche quali campi del pacchetto in ingresso analizzare.

## 2.7 NetFPGA

L'astrazione appena descritta di FlowBlaze è stata implementata sulla NetFPGA-SUME, una Smart-NIC equipaggiata da una FPGA e prodotta dall'azienda Digilent Inc. È nata con lo scopo di realizzare una scheda host PCIe in grado di supportare applicazioni a 100 Gb/s.

Il progetto NetFPGA fornisce una soluzione software, hardware e le infrastrutture per semplificare le fasi di progettazione, simulazione e test dei sistemi basati su piattaforme opensource ad alta velocità. NetFPGA si presenta alla comunità attraverso forum online, tutorial, eventi e campi estivi tutti organizzati dal team NetFPGA. Tutti i progetti sviluppati nell'ambito NetFPGA, sono open-source. Al giorno d'oggi NetFPGA ha prodotto 3 generazioni di schede equipaggiando prima una Xilinx Virtex-II Pro 50 sulla NetFPGA-1G, poi

una Virtex-5 sulla NetFPGA10G ed infine una Virtex-7 sulla NetFPGA-SUME, utilizzata per l'implementazione di FlowBlaze.

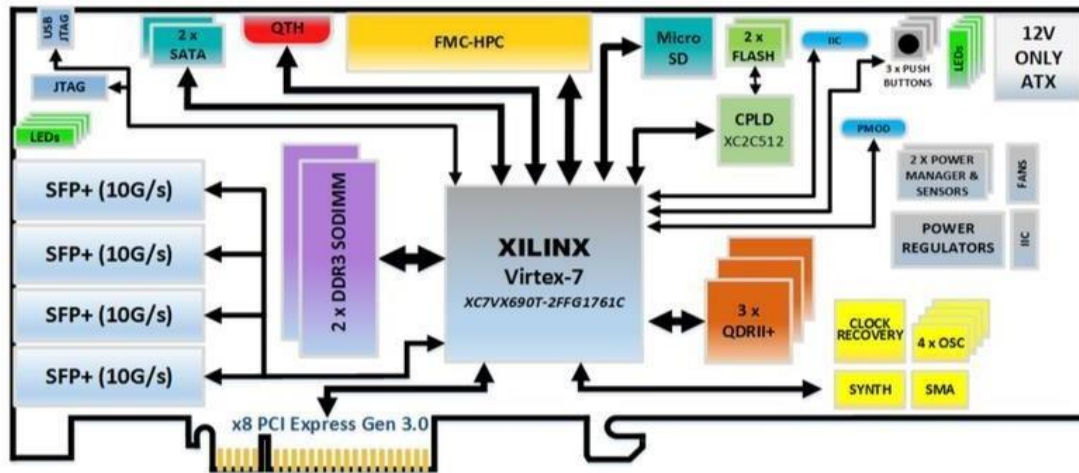


Figura 16: NetFPGA block diagram

Al centro della scheda è posizionata la FPGA Xilinx Virtex-7 690T e cinque sottoinsiemi di periferiche che permettono di completare la scheda. Inoltre, è stata aggiunta un'interfaccia seriale ad alta velocità composta da 30 collegamenti seriali che funzionano fino a una velocità di 13.1 Gb/s. Questi blocchi si collegano all'FPGA tramite 4 interfacce Ethernet 10Gb/s di tipo SPF+, due connettori di espansione e un connettore PCIe. Il secondo sottosistema è realizzato utilizzando un PCIe di ultima generazione 3.0, che interfaccia la scheda al dispositivo host permettendo i trasferimenti dei pacchetti tra la piattaforma e la scheda madre. Il sottosistema di memoria è realizzato attraverso una combinazione di SRAM e DRAM.

## Capitolo 3: Hash Tables

Come è già stato descritto nel capitolo 2, FlowBlaze implementa una cuckoo hash table per gestire le Flow Context Tables costituita da 4 tabelle che operano in parallelo e una memoria di appoggio definita stash di cui definiremo meglio lo scopo e l'impiego più avanti. La hash table ha lo scopo di gestire le chiavi generate dal look up extractor / update extractor in dipendenza dall'intestazione dei pacchetti in arrivo e alla maschera scelta in fase di programmazione della scheda.

### 3.1 Hash Tables

In informatica per hash table si intende un metodo di mappatura di una struttura dati tale da associare ad ogni dato un indice di un generico array.

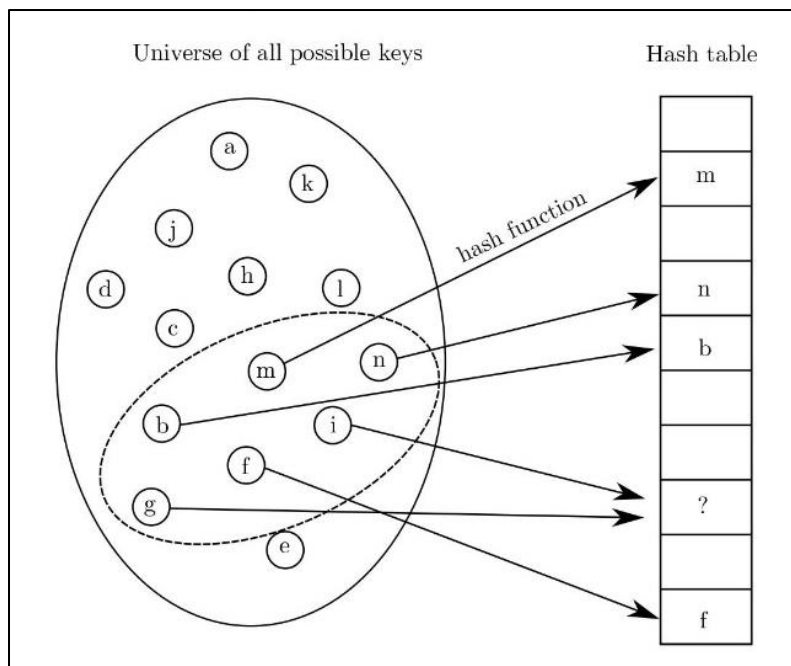


Figura 17: Schema base di funzionamento dell'hashing. Le chiavi sono prese dall'universo  $U$  e mappate nelle celle della hash table attraverso una hash function

L'associazione è fatta attraverso una hash function che, preso il dato come ingresso, restituisce l'indice dell'array.

Esse hanno svariate funzioni e impieghi ma il più semplice di tutti, e anche per spiegarne meglio il funzionamento attraverso un esempio, è usato per controllare se una data parola è presente o meno all'interno di un dizionario di parole proibite. Il metodo più semplice per eseguire questa operazione sarebbe controllare tutte le parole del dizionario e confrontarle con la nostra ma ciò impiegherebbe un tempo notevole in quanto il tempo di ricerca è proporzionale alle  $n$  parole del dizionario ( $\Theta(n)$ ). Se fosse ordinato alfabeticamente potrei eseguire quella che viene definita una "binary search" dimezzando ad ogni ricerca il numero di parole da confrontare. Questo è possibile paragonando la nostra parola con quella che si trova esattamente alla metà del "dizionario" e andando di volta in volta a stabilire se la parola che stiamo cercando si debba trovare prima o dopo di quella appena trovata. L'operazione è ripetuta dimezzando ad ogni iterazione il numero di elementi da controllare. Il tempo di ricerca di una binary search scala anch'esso con il numero di istanze da controllare ma logaritmicamente e non linearmente ( $\Theta(\log n)$ ).

Grazie invece ad una hash table potrei controllare la presenza o meno della parola con un solo confronto velocizzando di molto il processo ( $\Theta(1)$ ) avendo tuttavia un trade off in ambito di area e precisione che verrà spiegato più avanti. L'idea alla base è quella di usare una hash function che converta la chiave o qualunque dato in ingresso, in questo caso la nostra parola, in un numero relativamente piccolo che useremo come indice di un generico array di memoria, che chiameremo hash table, e controllare se a tale locazione è presente o meno la parola cercata. L'indice è dato in un set finito di valori che corrisponde al numero di locazioni da controllare

Per rendere fruibile la hash table è importante scegliere la hash function tale da mappare le chiavi con indici numerici relativamente piccoli ed equiprobabili. Tuttavia, è facile



immaginare come si ha la possibilità che due chiavi in ingresso alla hash function scelta diano lo stesso indice come risultato; in questo caso si parla di **collisione**.

Secondo il “Paradosso del compleanno”, prese 23 persone casualmente, la probabilità che due di esse abbiano lo stesso compleanno è pari al 50%. Questo risultato ci dà un’idea di quanto è facile incorrere in una collisione. In base allo scopo di utilizzo scelto per la hash table, ci sono diversi modi per risolvere le collisioni. [9]

Vanno distinti due casi di hashing noti come **chaining** e **open addressing**.

Il chaining fornisce un metodo di risoluzione delle collisioni basato sulle liste linkate. In fase di inserimento di chiavi all’interno della hash table, se una locazione risulta già occupata da una chiave diversa da quella che stiamo cercando di inserire, si appende ad essa quella che stiamo inserendo. In questo modo non avremo problemi di collisione in quanto avremo liste linkate per ogni locazione della hash table. In fase di ricerca delle chiavi andremo a scorrere la lista che si sarà formata a quell’indirizzo. È notevole come più chiavi linkiamo tra loro, più il tempo di look-up incrementerà.

L’idea alla base dell’open addressing invece è quella di avere una locazione per ogni chiave e quindi avere hash tables con dimensioni maggiore o uguale al numero di chiavi. Un fattore da tenere in conto nei vari tipi di open addressing è quanto efficientemente si utilizza la quantità di memoria allocata alla hash table. In caso di collisione, la nuova entry verrà inserita in un’altra locazione della hash table.

La soluzione più banale per la scelta di quale altra locazione usare per inserire una chiave che ha colliso sarebbe usare la prima locazione libera successiva a quella indicizzata. Questo metodo detto “probing” sia in fase di inserimento che in fase di ricerca può essere poco conveniente in quanto si potrebbero formare i cosiddetti “cluster” ovvero locazioni consecutive della tabella con un considerevole numero di chiavi lasciando pressoché vuota la restante parte della hash table. Inoltre, non abbiamo certezze sul numero di controlli che

vanno fatti sia in fase di ricerca di una chiave, sia in fase di scrittura nel tentativo di trovare la prima locazione libera di memoria.

È di interesse tuttavia notare come esista il “perfect hashing” ovvero il caso in cui siano note a priori e finite le chiavi. In una situazione del genere si allocherebbero per  $n$  chiavi un numero  $n$  di posti in memoria nella hash table e si sceglierebbe la hash function tale da mappare ogni chiave in una locazione diversa dalle altre.

## 3.2 Cuckoo Hashing

La cuckoo hashing è un’astrazione ideata da Rasmus Pagh e Flemming Frøde Rodler presentata nel Journal of Algorithms nel 2004 [10]. Lo scopo era quello di ideare un dizionario con tempo di look-up costante e pari al valore del worst-case. L’idea era di svolgere una dinamizzazione di un dizionario statico. Usando due hash tables con due hash functions distinte, una chiave in ingresso veniva salvata solamente in una delle due e non in entrambe. In fase di lookup dunque si controllavano entrambe le hash table e il tempo era dunque scandito da questa unica ricerca (che poteva o meno essere eseguita in parallelo in base al campo di applicazione e all’hardware). Capiamo subito come in tal senso è molto più performante di una soluzione a probing in quanto a tempo di look-up e di insert.

L’idea alla base del processo di insert spiega il nome di questa astrazione. Alcune specie di cuculo depongono le proprie uova nei nidi di altri uccelli le quali, alla nascita, scacciano le uova che già si trovavano nel nido. Allo stesso modo l’inserimento di una chiave in una delle due hash table che abbia una entry già salvata in quella posizione, causerà il “kick” della chiave presente a quella locazione. Quest’ultima verrà processata usando la seconda hash function e posizionata nella seconda hash table utilizzando come indice il risultato della seconda funzione di hash. Se anche questa posizione sarà occupata da una chiave, il

processo verrà ripetuto finché tutte le chiavi non avranno una propria locazione di memoria.

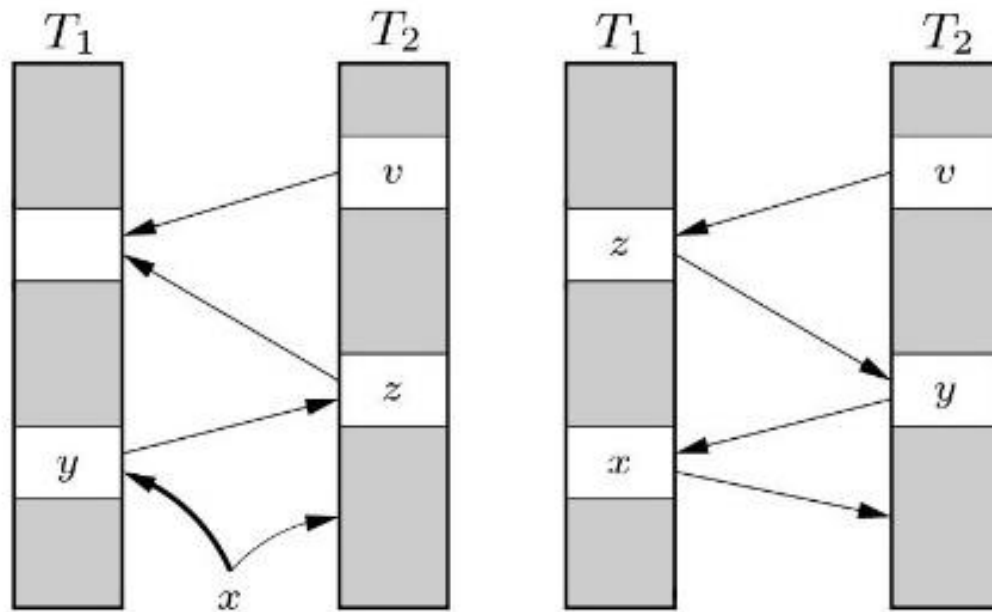


Figura 18: Insert della chiave  $x$  in una hash table cuckoo

In figura 2 viene mostrato come ogni chiave inserita ha potenzialmente la possibilità di migrare in una posizione dell'altra hash table e come ogni chiave in fase di inserimento possa essere inserita in una delle due tabelle. L'algoritmo ideato inizialmente era tale da inserire ogni chiave nella hash table  $T_1$  e causare poi un kick in  $T_2$  in caso di collisione. Nell'esempio l'inserimento di  $x$  causa lo spostamento di  $y$  in  $T_2$  e di  $z$  in  $T_1$ . Notiamo, come già detto, che in fase di lookup dovremo controllare la presenza della chiave nelle due locazioni che calcoleremo attraverso le due hash functions.

Esiste un problema fondamentale del cuckoo hashing e si ha quando un inserimento dà origine ad un loop. In tal senso avremo processi di kick che si andranno a ripetere all'infinito senza riuscire a completare l'insert. In matematica questo problema è definito come

“Pidgeonhole principle” per il quale  $n+k$  oggetti non possono essere ripartiti unicamente in  $n$  cassetti. Ad una simile conclusione ci si potrebbe anche arrivare usando il buon senso. In figura infatti stiamo cercando di “stipare” 7 chiavi in 6 locazioni di memoria. In un’applicazione della cuckoo hashing possiamo facilmente limitare l’infinita ripetizione del processo di kick imponendo un tetto massimo di ripetizioni prima di capire che effettivamente il loop non si chiuderà mai e dichiarare un Insert Failure.

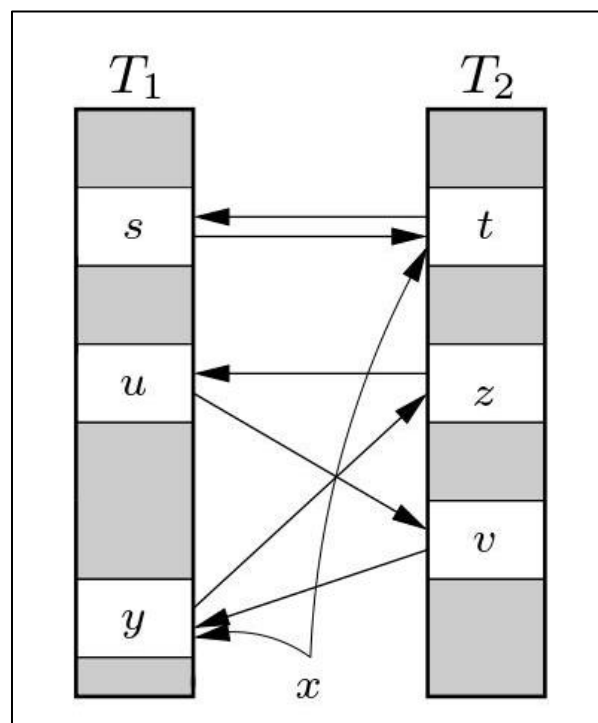


Figura 19: Loop cuckoo hashing

Pagh e Rodler dimostrano che prese due hash functions indipendenti tra loro da un’appropriata hash family, allora con probabilità  $1 - \Theta(1/n)$ , la procedura di inserzione allocherà con successo tutte le  $n$  chiavi con al massimo  $\alpha \log n$  spostamenti nell’inserimento di una qualsiasi chiave, presa una costante  $\alpha$  sufficientemente grande. In tal senso useremo  $\alpha \log n$  ripetizioni come valore di controllo per dichiarare insertion failure.

L'unico modo per risolvere una insertion failure consiste nel ricalcolare tutte le chiavi presenti nelle tabelle utilizzando diverse funzioni di hash. Nonostante la probabilità che ciò accada non sia elevata, si cerca di arginarne l'eventualità in quanto il rehash è molto "costoso" in ambito di performance.

Nel 2008 è stata proposta da Michael Mitzenmacher, Adam Kirsch e Udi Wieder una soluzione che eviterebbe il rehash della hash table nell'incombenza di un loop. Attraverso uno stash a dimensione costante e irrisoria, paragonata a quella della hash table, hanno dimostrato come la probabilità di insert failure viene drammaticamente ridotta. [11]

Viene modificato l'algoritmo di inserzione nel seguente modo: Si controlla un fallimento attraverso un prestabilito numero di kick massimi oppure dall'accertamento di un loop nel caso in cui un inserimento causi due accessi alla stessa locazione della medesima hash table, che andremo a chiamare col nome di vertice. Una volta accertato il loop dei kick si procede ad inserire una chiave nello stash in modo da spezzare il ciclo. Fintantoché un vertice ha un solo accesso, rimarremo nel caso di non loop. La prima entry che accederà per la seconda volta, durante la stessa insert, ad un vertice già incontrato, verrà salvata nello stash. In fase di implementazione si nota come sia più facile tenere traccia del numero di kick eseguiti piuttosto che il numero di accessi ad ogni vertice.

Introduciamo ora un'altra leggera modifica all'algoritmo di hashing che fu introdotto da Dimitris Fotakis in "Space Efficient Hash Tables with Worst Case Constant Access Time" per il quale viene introdotto il cosiddetto "Generalized Cuckoo Hashing" [12]. Egli introdusse la variabile  $d$  relativa al numero di hash functions e hash tables associate che non erano più solamente 2 come nell'idea originale di Pagh e Rodler. Tale astrazione portò a miglioramenti in ambito di efficienza nell'utilizzazione della memoria allocata alla hash table. Analogamente all'insert classico della cuckoo, in fase di collisione con una chiave, essa veniva rilocata in un'altra hash table. La chiave veniva spostata in una delle  $d$  hash tables scelte in maniera casuale.

Fotakis dimostra come la space utilization incrementa radicalmente aumentando  $d$ . I risultati riportati nel grafico ne sono la dimostrazione. Sono state scelte  $d$  hash tables diverse di dimensione  $(1 + \epsilon)n/d$  in modo da generalizzare l'efficienza dalla dimensione delle hash tables. Ogni inserzione usa un "random walk", ovvero un elemento viene collocato in una delle  $d$  hash tables in maniera casuale.

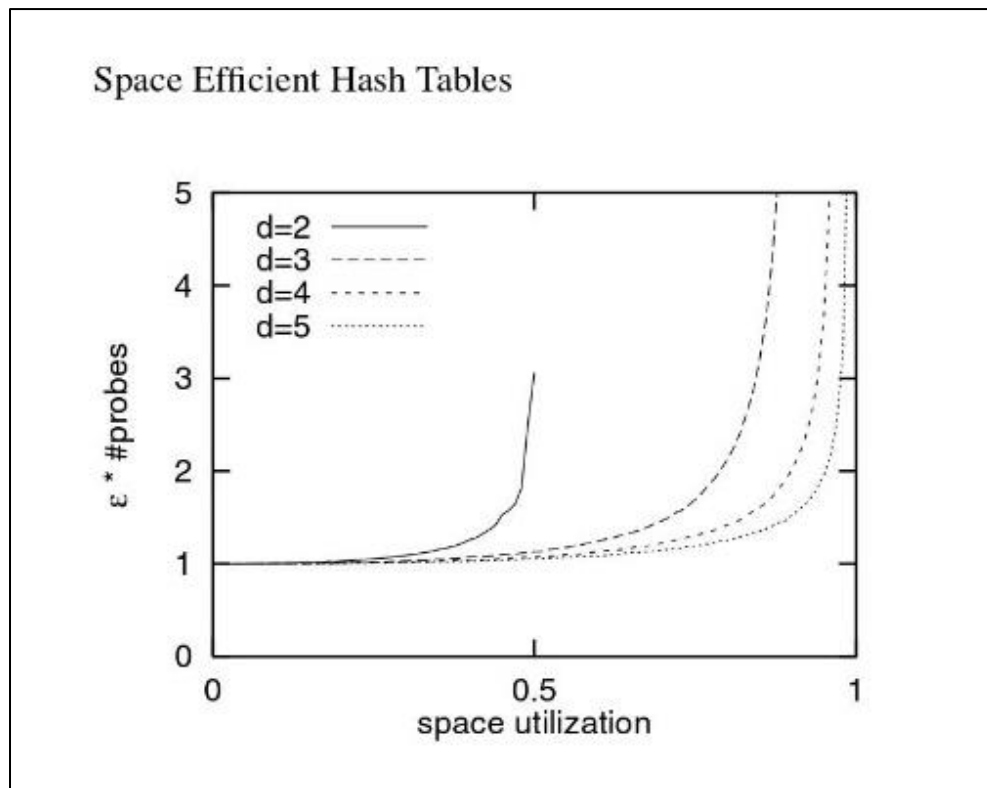


Figura 20: Space utilization cuckoo hash table in dipendenza delle  $d$  hash functions

Se la locazione dovesse essere occupata, verrebbero controllate le altre  $d-1$  tabelle finché non venga trovata una locazione libera. Se tutte e  $d$  le tabelle dovessero risultare piene allora verrebbe spostata una delle  $d$  entries e verrebbe ripetuto il processo di insert nelle  $d$  hash tables. Nel grafico è riportato il numero di probes (ripetizioni dell'algoritmo di insert)

necessarie per eseguire l'insert, normalizzate con  $\epsilon$ , variabile dipendente dalla dimensione delle tabelle.

Durante la simulazione veniva dichiarato insert failure e venivano arrestate le curve se un insert impiegava più di 1000 probes. Si denota dal grafico come all'aumentare di  $d$  aumenta drasticamente la space utilization e diminuisce l'insertion time fino a raggiungere una soglia (threshold) dipendente dalla  $d$  scelta. Vengono inoltre riportati i valori di massima utilizzazione dello spazio calcolati:

- $d = 2$               Space Utilization = 49%
- $d = 3$               Space Utilization = 91%
- $d = 4$               Space Utilization = 97%
- $d = 5$               Space Utilization = 99%

Consapevoli dei risultati riportati da Fotakis, nel paper precedentemente citato "More Robust Hashing: Cuckoo Hashing with a Stash" sono stati analizzati diversi impieghi dello stash in relazione al valore di  $d$ .

Attraverso simulazioni hanno mostrato come, affinché la cuckoo hashing non dia problemi di insert failure, è sufficiente uno stash di dimensione veramente esigua.

Riportiamo i risultati relativi a 2 e 3 hash tables con dimensione 1200 nel quale sono state inserite 1000 entries. Nell'esperimento, non appena venivano raggiunti i 100 spostamenti per unica insertion, veniva inserita una entry nello stash. In un milione di ripetizioni della simulazione riportiamo quante entries sono state salvate nello stash.

Stash Size	Standard	Modified	Stash Size	Standard	Modified
0	992812	992919	0	9989861	9989571
1	6834	6755	1	10040	10350
2	338	307	2	97	78
3	17	15	3	2	1
4	1	2	4	0	0

(a) 1000 items (b) 10000 items

Table 1: For  $d = 2$ , failures measured over  $10^6$  trials for 1000 items, requiring a maximum stash size of four (a), and failures measured over  $10^7$  trials, requiring a maximum stash size of three (b).

Stash Size	Standard	Modified	Stash Size	Standard	Modified
0	998452	998490	0	9964148	9964109
1	1537	1510	1	35769	35891
2	11	0	2	83	0

(a) 1000 items (b) 10000 items

Table 2: For  $d = 3$ , failures measured over  $10^6$  trials for 1000 items (a), and failures measured over  $10^7$  trials (b).

Figura 21: Risultati del test di riempimento dello stash in relazione al numero di hash tables e hash functions

Si evince dai grafici che stash con solo 4 locazioni di memoria sono sufficienti per abbassare la probabilità di insert failure al di sotto di  $10^{-6}$  in una normale cuckoo hashing con due hash tables e due hash functions. Nel secondo caso, avendo sperimentato con  $d = 3$  e attraverso la tecnica già discussa del “random walk” si nota come solo meno di 100 casi su  $10^6$  hanno salvato due entries nello stash mentre nessuno caso ha avuto bisogno di più di due locazioni in memoria nello stash.

Il risultato è di grande impatto in quanto togliamo quasi completamente la necessità di rihashare l’intera hash tabe nell’insert failure semplicemente allocando una piccola porzione di memoria ad uno stash d’appoggio che spezza gli eventuali loop.



### 3.3 Hash Tables e Stash FlowBlaze

In Flowblaze si è deciso di istanziare una cuckoo hash table con 4 hash functions e uno stash da 8 entries. Ogni hash table ha 1024 locazioni di memoria dove salviamo la coppia value + key. La praticità di implementare una cuckoo hash table con  $d = 4$  stà nella possibilità in hardware di svolgere operazioni di lookup in parallelo a tutte e 4 le hash tables e allo stash. Infatti operazioni di insert e lookup hanno tempo costante pari ad un colpo di clock. Per rendere possibile ciò lo stash è stato usato in maniera leggermente diversa dall'astrazione presentata.

```
entity cuckoo is
  generic (
    key_len : integer := 128;
    value_len : integer := 128;
    logsize : integer := 10
  );
  port (
    clock : in std_logic;
    reset : in std_logic;
    enable : in std_logic;

    --AKI interface
    we : in std_logic;
    rd : in std_logic;
    input_din : in std_logic_vector(1+key_len+value_len downto 0);
    addr : in std_logic_vector(logsize+1 downto 0);
    data_out : out std_logic_vector(255 downto 0);

    --Hash Interface
    pre_insert : in std_logic; -- is '1' 1 cc before insert
    insert : in std_logic; -- 1 to insert
    key : in std_logic_vector(key_len-1 downto 0); -- key to insert
    value : in std_logic_vector(value_len-1 downto 0); -- value to insert

    current_color : in std_logic_vector(2 downto 0);
    remove : in std_logic; -- 1 to remove
    search : in std_logic; -- 1 to search
    search_key : in std_logic_vector(key_len-1 downto 0); -- key to search
    hit : out std_logic;
    tot_num_entry_stash : out std_logic_vector(31 downto 0);
    num_entry_stash : out std_logic_vector(31 downto 0);
    num_present : out std_logic_vector(31 downto 0);
    clear_count_collision : in std_logic;
    count_collision : out std_logic_vector(31 downto 0);
    count_cuckoo_insert : out std_logic_vector(31 downto 0);
    count_item : out std_logic_vector(31 downto 0);
    evicted_entry : out std_logic_vector(31 downto 0);
    search_value : out std_logic_vector(value_len-1 downto 0); -- value associated with the searched value
    key_len_in : in std_logic_vector(clogb2(128) downto 0);
    value_len_in : in std_logic_vector(clogb2(128) downto 0)
  );
end cuckoo;
```

Figura 22: Entity e segnali di cuckoo.vhd

Attraverso una macchina a stati vengono gestiti segnali in ingresso di insert, search e remove. Non appena il segnale di insert viene asserito, si prova ad inserire la chiave

all'interno delle hash table provando in ordine dalla prima alla quarta per trovare una che non abbia una locazione occupata. Se dovessero essere occupate tutte e quattro, verrebbe scelta randomicamente quale delle quattro chiavi kickare. La chiave in ingresso verrà salvata nella locazione scelta mentre la chiave espulsa verrà salvata nella prima delle 8 locazioni libere dello stash. Operando in questo modo permettiamo l'insert a tempo costante e fisso che corrisponde ad un colpo di clock. In questo modo lo stash viene usato come locazione di appoggio per poi riprovare al colpo di clock successivo ad inserire la chiave appena espulsa. E' importante notare come lo stash proverà ad inserire chiavi all'interno delle hash tables solo nel momento in cui il segnale di insert globale è deassertito. L'insert globale dà infatti priorità alle chiavi presenti sulla porta in ingresso alla hash table rispetto alle chiavi salvate nello stash. Se dovessero infatti capitare due insert su due fronti di clock consecutivi ed in entrambi i casi le 4 hash tables hanno le locazioni occupate, verranno salvate due chiavi kickate dalle HT nello stash.

Il vero e proprio meccanismo di cuckoo in questo caso avviene non più tra le singole Hash tables ma viene usato lo stash come punto di appoggio in modo da "mascherare" la variabilità del tempo di insert (in questo caso sarà dato da un multiplo di colpi di clock). Lo stash quindi avrà il doppio compito di risolvere le poco probabili ma problematiche "insert failures" e di essere usato come registro di appoggio tra un kick e l'altro.

#fai un disegno DRAW.IO tipo figura 2 ma con lo stash in mezzo ad ogni kick

Il segnale di search opera in parallelo tra hash tables e stash in quanto, affinché il cuckoo abbia senso dovremo sempre assicurare che una generica chiave sia presente solamente in uno tra le 4 HT e lo Stash. Indipendentemente da ciò stash e hash tables hanno ognuna un segnale di hit in uscita che indica la presenza o meno della chiave cercata. A colpo di clock la macchina a stati controlla se uno dei due segnali è positivo e indica o meno la presenza della chiave nella coppia Hash Tables e Stash.

Per implementare il remove si è adoperato un bit di value che indichi o meno la validità di una generica entry. In questo modo se una entry di una Hash table ha questo bit a 1 vuol dire che la entry è valida e verrà presa in considerazione. In caso contrario quella locazione di memoria è sovrascrivibile da una nuova chiave in ingresso. Il processo di remove infatti consiste nella sovrascrittura del bit di controllo a 0. Questo principio è valido sia per lo stash che per le HT.

Il bit in questione è importante anche in fase di insert e search, infatti in entrambi i casi esso viene controllato per sovrascrivere una locazione di memoria o per accettare come effettivamente presente una data chiave.

### 3.3.1 HT128new.vhd

Il modulo hash table si compone di due principali interfacce che lavorano in parallelo senza interferirsi a vicenda. Per rendere ciò possibile sono state usate le ram dual port. Le due porte delle ram, che identificheremo con i suffissi a e b per i relativi segnali, sono indipendenti tra loro.

```
entity ht128newa is
generic(
    key_len_max : integer:= 128;
    value_len_max: integer:= 126
);
Port (
    clock : in std_logic;
    reset : in std_logic;
    search : in std_logic;

    --AXI interface
    we: in std_logic;
    rd: in std_logic;
    input_din: in std_logic_vector(1 + key_len_max+value_len_max downto 0);
    --addr :in std_logic_vector(11 downto 0);
    addr :in std_logic_vector(13 downto 0);
    data_out :out std_logic_vector(1 + key_len_max+value_len_max downto 0);

    --Hash Interface
    remove : in std_logic; -- 1 to remove
    insert : in std_logic; -- 1 to insert
    update : in std_logic; -- 1 to update
    key : in std_logic_vector(key_len_max-1 downto 0); -- insert key
    value : in std_logic_vector(value_len_max-1 downto 0); -- insert value
    search_key : in std_logic_vector(key_len_max-1 downto 0); -- search key

    key_len: in std_logic_vector(clogb2(key_len_max) downto 0);
    value_len: in std_logic_vector(clogb2(value_len_max + 2) downto 0);

    kicked : out std_logic; -- 1 if a key has been kicked out
    kicked_value : out std_logic_vector(value_len_max-1 downto 0); -- kicked value
    kicked_key : out std_logic_vector(key_len_max-1 downto 0); -- kicked key

    out_count_collision: out std_logic_vector(31 downto 0);
    clear_count_collision: in std_logic;
    out_count_item: out std_logic_vector(31 downto 0);

    match : out std_logic; -- 1 if hit on query
    output : out std_logic_vector(value_len_max-1 downto 0) -- value associated with the search key
);
end ht128newa;
```

Figura 23: Entity e segnali della hash table dinamica

Dei 128 bit della chiave, vengono prese porzioni e hashate in 4 diversi modi ma analoghe tra loro. La logica dietro alle hash functions è l'utilizzo della XOR. Presi due ingressi con distribuzione pressochè casuale infatti, usando una XOR si ottiene una nuova distribuzione,

anch'essa casuale, ma che dipende da entrambi gli ingressi. Ciò si può dimostrare studiando una semplice XOR a 2 bit.

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Essa avrà equiprobabilità di essere 0 o 1 con qualunque tipo di ingresso. Possiamo quindi utilizzare la XOR per combinare distribuzioni di probabilità uniformi.

Le 4 hash functions sono infatti operazioni di xor tra porzioni della chiave a 128 bit in modo da generare un vettore a 12 bit in grado di indirizzare al massimo 4096 locazioni.

In ingresso alla hash table abbiamo due chiavi: key e search\_key. Infatti come già accennato abbiamo usato ram dual port in modo da poter eseguire un'operazione di insert insieme ad un'operazione di search. Questo torna necessario soprattutto se vogliamo eseguire una search mentre il processo di insert di una entry si deve ancora concludere nel caso in cui abbia colliso molte volte.

### 3.3.2 Stash.vhd

Della hash table sono di interesse i segnali in uscita kicked, kicked\_key e kicked\_value. Essi infatti sono gli ingressi dello stash. In particolar modo kicked corrisponde all'insert dello stash e kicked\_key e kicked\_value ai rispettivi key e value. Lo stash, date le poche locazioni richieste, è implementato con una Distributed Ram ovvero è composta di soli registri.

```
ST: entity work.stash generic map (key_len,value_len+3)
port map (
    clk      => clock,
    reset    => reset,
    insert   => kicked,
    key      => kicked_key,
    --value  => kicked_value,
    value(value_len+2 downto value_len) => (others=>'0'), --kicked_color,
    value(value_len-1 downto 0) => kicked_value,
    not_empty => stash_not_empty,
    search    => search_FSM,
    update    => update_stash,
    remove    => remove_FSM,
    key_out   => key_out_stash,
    --value_out => value_out_stash,
    value_out(value_len+2 downto value_len) => color_out_stash,
    value_out(value_len-1 downto 0) => value_out_stash,
    search_key => search_key_FSM,
    search_value(value_len+2 downto value_len) => search_color_stash,
    search_value(value_len-1 downto 0) => search_value_stash,
    num_entry => num_entry_stash,
    num_present => num_present,
    tot_num_entry => tot_num_entry_stash,
    evicted_entry => evicted_entry,
    hit        => hit_stash
);
```

Figura 24: Component e segnali di stash.vhd

Questo ci dà il vantaggio che possiamo controllare l'intero stash a colpo di clock, cosa che non può essere fatta con le hash tables. Essendo le distributed RAM molto pesanti nell'architettura, sarebbe stato impossibile istanziare delle hash tables con esse.

Avendo completo accesso a tutte le entries a colpo di clock, essendo una distributed ram, lo stash ha un segnale in uscita, **stash\_not\_empty**, indispensabile per il funzionamento della cuckoo. Questo è infatti il segnale che permette la scrittura da stash ad hash. Finchè esso è 1, ovvero lo stash ha delle entries salvate in esso, la hash table proverà ad inserirle. Come

già detto prima, tuttavia, è necessario che questo insert avvenga con minor priorità rispetto ad un insert globale per il semplice fatto che una chiave sulla porta d'ingresso potrebbe venir persa al colpo di clock successivo mentre lo stash funge da elemento di memoria.

### 3.4 Block Ram

In base a quanto anticipato e alle necessità di progetto, sono state scelte le RAM da istanziare. Come descritto nel “7 Series FPGAs Memory Resources Manual” [13], Virtex 7 mette a disposizione una vasta rete di block ram attraverso le quali possono essere implementate celle di memoria più o meno complicate. Le Block-Ram dual\_port si basano

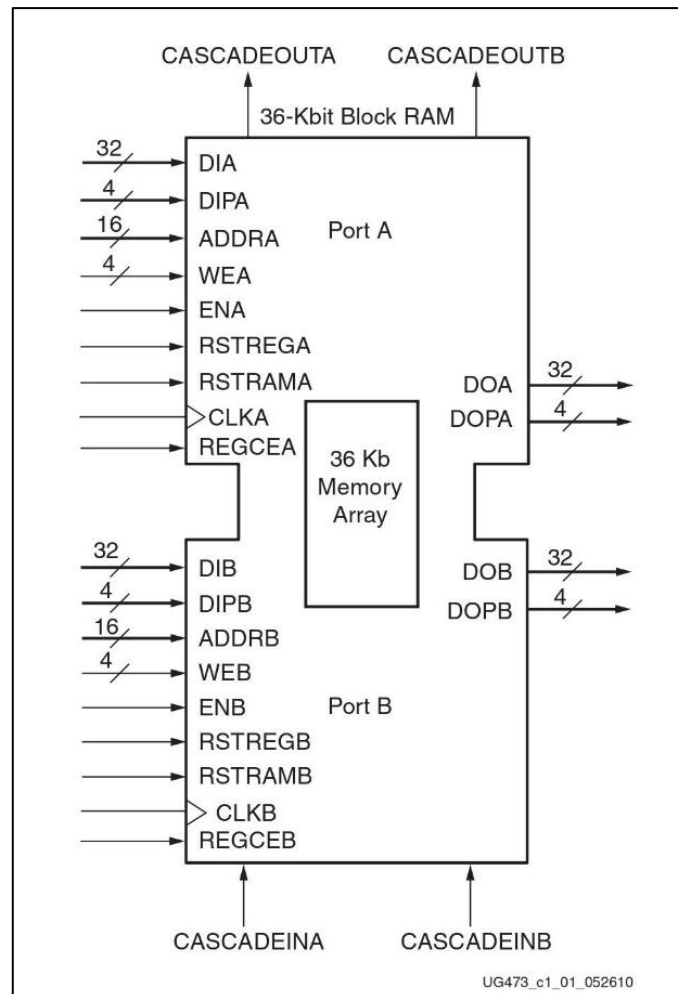


Figura 25: Blocco elementare della 36-Kbit Block RAM della famiglia Virtex7

su due blocchi elementari che differiscono tra di loro solo per le dimensioni: 36-Kbit Block RAM, 18-Kbit Block RAM.

Le memorie mettono a disposizione due porte di accesso tra loro indipendenti, A e B. La struttura è completamente simmetrica e intercambiabile. I dati possono essere scritti su una o entrambe le porte e letti su una o entrambe le porte. Da qui deriva la nomenclatura `true_dual_port`. Ogni porta, come mostrato in figura, ha il proprio address, data in, data out, clock e write enable. Inoltre, notiamo le due porte `CASCADEINA` e `CASCADEINB` che sono utilizzate per creare una rete di blocchi elementari in grado di soddisfare i requisiti di progetto richiesti. Esiste la possibilità di “collisione” anche se non ci riferiamo alla collisione della hash table ma all’effettiva indeterminazione dei bit nel caso in cui entrambe le porte indirizzano la stessa cella di memoria in fase di scrittura. Tuttavia, nel nostro caso, non sarà necessario badare a ciò dato che useremo una sola delle due porte per scrivere ed entrambe per leggere la ram. L’ip della xilinx dà anche la possibilità di sfruttare le porte asincronamente tra di loro adoperando due segnali di clock ma nuovamente questo non è il nostro caso di utilizzo e ci limiteremo ad usare un clock condiviso.

In FlowBlaze si è deciso di utilizzare 4 hash tables da 1024 locazioni da 256 bit l’una. Sono state dunque istanziate 8 36-Kbit Block RAM in parallelo per ogni HT. Infatti, ogni cella elementare supporta word in ingresso di massimo 32 bit e l’utilizzo in parallelo di 8 di esse permette un data in da 256 bit. Si nota come abbiamo usato un solo layer di blocchi dato che con una 36Kbit e 32 bit di data in avremo esattamente 1024 locazioni di memoria.



## *Capitolo 4: Modulo Riconfiguratore*

In questo capitolo verrà mostrato come è stato implementato il modulo atto alla riconfigurazione delle hash table in relazione alla lunghezza delle entries. Procederemo descrivendo inizialmente l'astrazione di ciò che si è voluto raggiungere e successivamente come ciò sia stato implementato in hardware.

Le 4 hash table di FlowBlaze agiscono in parallelo e sono analoghe tra loro. Come già detto precedentemente è stato necessario l'utilizzo di 4 hash tables con 4 algoritmi di hash diversi in modo da far tendere il riempimento della memoria al 97 per cento circa. Ognuna di esse poggia su una block ram da 1024 entries da 256 bit l'una. Ciò permette ipoteticamente 4k keys salvate.

Ogni hash table salva in ognuna delle sue entries la coppia key e value da 128 bit l'una associando così ad ogni chiave una locazione di memoria. Si è notato come istanziare una dimensione fissa di 128 bit per la chiave sia poco efficiente dato che non tutte le applicazioni di FlowBlaze richiedono una dimensione massima della chiave. Basti prendere l'esempio di un indirizzo mac che necessita di una entry di soli 48 bit (6 byte). Ciò risulterebbe in uno spreco di celle di memoria dato che la maggior parte dei bit delle word risulterebbero inutilizzati.

Si è dunque cercato un modo per rendere dinamica e riconfigurabile la dimensione delle chiavi da salvare in modo da migliorare il riempimento delle hash table. Ciò porterebbe oltre ad un numero maggiore di entries salvabili, anche ad una inferiore probabilità di collisione.

Attraverso l'aggiunta di due segnali in ingresso all'hash table, rispettivamente `key_len` e `value_len` definiamo run-time la dimensione di bit scelta per tale applicazione. Nella fattispecie sono stati implementati 5 casi d'utilizzo diversi ma con la stessa logica e con una leggera modifica se ne possono implementare anche di altri. I valori di `key_len` e `value_len`,

salvati in un registro, possono essere modificati durante l'utilizzo di FlowBlaze o semplicemente durante la configurazione iniziale dello stesso (come riportato nella fase di test della scheda) utilizzando in entrambi i casi il bus S\_AXI in ingresso alla scheda, specificando l'indirizzo del registro alla porta a 32 bit S\_AXI\_AWADDR e il valore sempre a 32 bit sulla porta S\_AXI\_WDATA. Di default hanno entrambi valore di 128 salvato in binario nei 16 LSB del registro come mostrato in figura.

```

REG_WRITE_PROCESS: process(ACLK)
begin
    if (ACLK'event and ACLK = '1') then
        clear_count_collision <='0';
        if (ARESETN = '0') then

            key_len_in  <=x"80";
            value_len_in<=x"80";

            -- "XXXXXXXXXXXXXXXXXXXX"

        if address =x"80ffff88" then
            key_len_in  <=s_axi_in.AXI_WDATA(7 downto 0);
            value_len_in<=s_axi_in.AXI_WDATA(15 downto 8);
        end if;

        s_axi_out.AXI_RDATA <= release_number      when address=x"80000000" else
            x"0000"&value_len_in & key_len_in      when address =x"80ffff88" else

```

Figura 26: In ordine dall'alto al basso: valore di default assegnato al reset della scheda, processo di scrittura, processo di lettura del registro associato ai segnali key\_len e value\_len

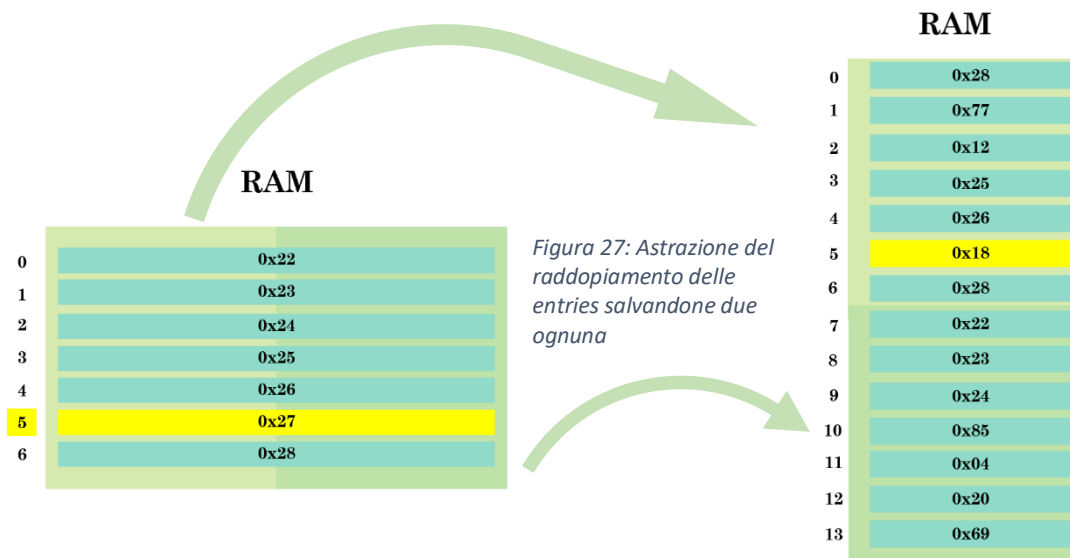
La dimensione della key in ingresso e la conseguente dimensione del value possono essere le seguenti:

- |                     |                     |                     |
|---------------------|---------------------|---------------------|
| • key_len = 128 bit | value_len = 128 bit | -default 4K entries |
| • key_len = 96 bit  | value_len = 32 bit  | -8K                 |
| • key_len = 64 bit  | value_len = 64 bit  | -8K                 |
| • key_len = 48 bit  | value_len = 16 bit  | -16K                |
| • key_len = 32 bit  | value_len = 32 bit  | -16K                |

Come si può notare i valori che i due segnali possono assumere sono scelti in modo da riempire il più possibile le ram che andremo ad istanziare e descritte nel paragrafo precedente. Inoltre, dato che le word che andremo a salvare sono formate dalla coppia value + key, è evidente vedere come le dimensioni delle word nonostante i 5 casi, siano solamente 3.

Lo scopo dunque è, attraverso il valore salvato nel registro di key\_len e value\_len, formare word in ingresso alla ram di dimensione data dalla somma delle due dimensioni ed indirizzare poi un numero di word dato dalla dimensione della ram diviso la dimensione della parola.

Per fare un esempio prendiamo in considerazione il caso in cui nel registro all'indirizzo 0x80fff88 sia salvato il valore 0x2060 che corrisponderà dunque a valori decimali di value\_len = 32 e key\_len = 96. Le word che andremo a salvare nella hash table avranno dimensione  $(32 + 96) \text{ bit} = 128 \text{ bit}$ . Come già precisato le block ram scelte hanno dimensione di word 256 con 1024 locazioni di memoria. Per adempiere al compito preposto dunque vogliamo salvare  $256 \times 1024 / 128$  entries ovvero il doppio di quante ne avremmo potute salvare senza definire le dimensioni a priori. In figura possiamo vedere come tale compito corrisponda idealmente a "dividere" la ram in due parti creando virtualmente una ram allungata. In questo modo avremo 2048 locazioni da 128 bit ciascuna.



## Capitolo 5: Simulazione e Implementazione

Traducendo questa astrazione in hardware si è deciso di implementare queste ram virtuali usando i valori di `key_len` e `value_len` come segnali di controllo di un case che definisca i vettori effettivamente in ingresso alle ram fisiche.

```
entity Reconfigurator2 is
generic (
  NB_COL    : integer := 32;           -- Specify number of columns (number of bytes)
  COL_WIDTH : integer := 8;           -- Specify column width (byte width, typically 8 or 9)
  RAM_DEPTH : integer := 1024;        -- Specify RAM depth (number of entries)
  INIT_FILE : string := "";           -- Specify name/location of RAM initialization file if using one (leave blank if not)
  ADDED_BITS : integer := 2;          -- ADDED BITS = 2 indirizzo al massimo 4 volte il numero di words
);

Port (
  key: in std_logic_vector(NB_COL*COL_WIDTH/2 - 1 downto 0); --128 bit massimi
  value: in std_logic_vector(NB_COL*COL_WIDTH/2 - 3 downto 0); --126 bit massimi

  control_in: in std_logic_vector(1 downto 0);
  control_out_a, control_out_b: out std_logic_vector(1 downto 0);

  data_a, data_b: in std_logic_vector(NB_COL*COL_WIDTH - 1 downto 0);

  key_len, value_len: in std_logic_vector(7 downto 0);
  we_a, we_b: in std_logic;
  switch: in std_logic;
  address_a: in std_logic_vector((clogb2(RAM_DEPTH) + ADDED_BITS - 1) downto 0);
  address_b: in std_logic_vector((clogb2(RAM_DEPTH) + ADDED_BITS - 1) downto 0);
  clk: in std_logic;
  -- data_out_a: out std_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
  -- data_out_b: out std_logic_vector(NB_COL*COL_WIDTH-1 downto 0);

  key_out_a: out std_logic_vector(NB_COL*COL_WIDTH/2 - 1 downto 0); --128 bit massimi
  value_out_a: out std_logic_vector(NB_COL*COL_WIDTH/2 - 1 downto 0); --128 bit massimi

  key_out_b: out std_logic_vector(NB_COL*COL_WIDTH/2 - 1 downto 0); --128 bit massimi
  value_out_b: out std_logic_vector(NB_COL*COL_WIDTH/2 - 1 downto 0); --128 bit massimi

  -- quartimod: in std_logic -- 1 se le word le scrivo un quarto e tre quarti, 0 altrimenti dove sono met
);
end Reconfigurator2;
```

Figura 28: Entity Reconfigurator

Nella fattispecie i segnali di `key` e `value` in ingresso alle hash table sono `std_logic_vector(127 downto 0)` ovvero hanno dimensione fissa massima a 128 bit e in base a `key_len` e `value_len` vengono presi in considerazione solo gli LSB utili. In questo modo è facile reindirizzare solo i bit utili di `key` e `value` per formare le word in ingresso alla ram. Nell'esempio dunque

prenderemo solo gli ultimi 32 bit di value (ovvero value(31 downto 0)) e gli ultimi 96 bit di key (key(95 downto 0)) e formeremo una parola da 128 bit.

Come secondo step si è cercato di indirizzare un numero maggiore di entries e un modo per scrivere quest'ultime senza andare ad intaccare quelle precedentemente salvate. Un'unica soluzione ha risposto ad entrambi i requisiti: un case su determinati bit dell'indirizzo in ingresso.

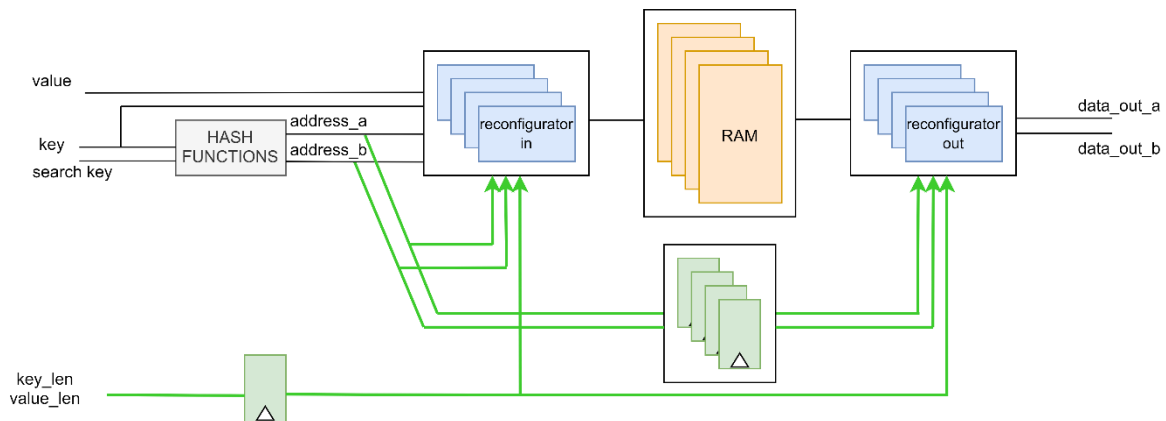


Figura 29: Schema a blocchi Reconfigurator

In base alla dimensione complessiva della word da salvare (che notiamo essere un valore tra 256,  $256/2 = 128$ , e  $256/4 = 64$ ) distinguiamo 3 casi in cui:

- ad ogni word della ram corrisponde una entry della hash table;
- ad ogni word della ram corrispondono 2 entries della hash table;
- ad ogni word della ram corrispondono 4 entries della hash table.

Nel primo caso i 10 bit meno significativi dell'address della hash table, determinato attraverso la hash function della chiave in ingresso, corrispondono all'address che indirizza effettivamente la block ram. Avremo dunque 1024 entries da 256 bit l'una.

Nell'esempio che stiamo considerando invece, che corrisponde al caso di due entries per word, l'ultimo bit dell'address, `address_a[0]`, serve per demuxare la word in ingresso tra le

due parti della block ram. Svolgerà il compito di effettivo indirizzo della ram `address_a[10 downto 1]`. In questo modo abbiamo virtualmente allocato  $2^{11}$  locazioni di memoria con una ram che ne contiene  $2^{10}$ .

È importante notare come sia architetturalmente facile dividere le word in fase di lettura usando gli LSB dell'address. La soluzione è infatti quella di usare un mux per abilitare sul bus d'uscita la porzione di parola interessata. Ciò è concettualmente più complicato da implementare in fase di scrittura. Una block ram che scriva una parola ad un certo indirizzo, sovrascrive tutte le celle di memoria associate a quell'indirizzo. Dato che lo scopo è scrivere solo porzioni di word, che corrispondono a word di dimensione inferiore, è stato indispensabile trovare un modo per scrivere entries senza toccare quelle già salvate a quel dato indirizzo.

Una soluzione sarebbe potuta essere quella di “comporre” la parola in ingresso leggendo in un primo momento la word salvata a tale indirizzo e poi rimettendo in ingresso i bit da lasciare intatti assieme alla nostra effettiva entry da salvare. Una implementazione possibile è quella di usare un registro e impiegare due colpi di clock ad ogni insert ma ciò è impensabile. La ip della RAM della Xilinx ha la possibilità di essere implementata attraverso 3 modalità di scrittura diverse che differiscono tra loro in ambito di power efficiency e area e una di queste semplifica il problema del doppio colpo di clock in scrittura. Nello specifico sono:

- `WRITE_FIRST`
- `READ_FIRST`
- `NO_CHANGE`

Durante la scrittura infatti:

- In `write_first` abbiamo sul bus d'uscita la parola appena scritta in memoria

- In read\_first (di cui riportiamo l'andamento in figura) avremo l'ultima parola salvata all'indirizzo che stiamo scrivendo
- In no\_change finché il we rimane attivo in uscita sarà pronta l'ultima parola a cui abbiamo fatto accesso in fase di lettura (viene usato un latch).

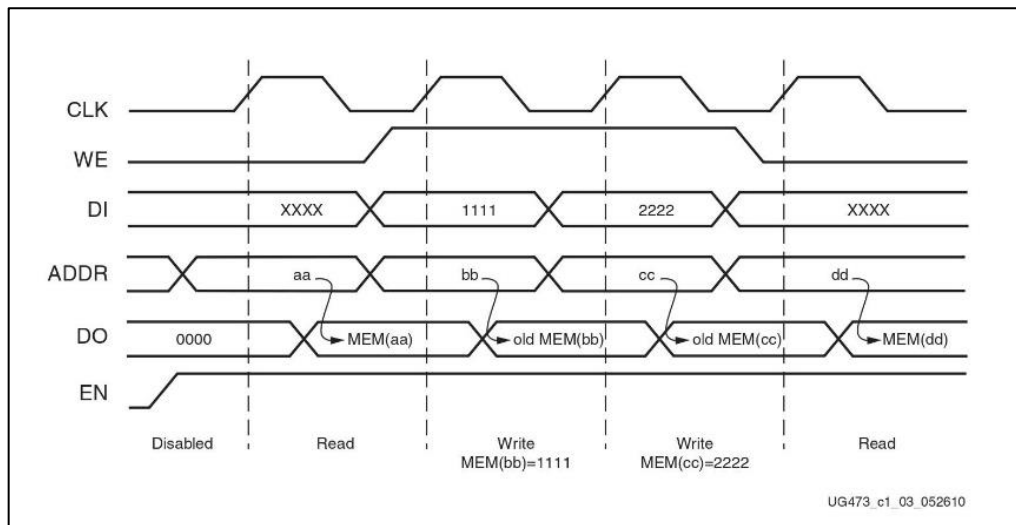


Figura 30: Andamento Read First

Utilizzando il READ\_FIRST potremmo scrivere a colpo di clock la parola in ingresso usando il data\_out della ram per ricostruire i bit che non vogliamo intaccare. Tuttavia, in questa modalità di utilizzo le ram hanno un power consumption maggiore del 15% rispetto alle altre due modalità di utilizzo.

Si è ricorso dunque ad un altro stratagemma di più semplice astrazione ovvero alla porta byte-wide write enable della Block RAM descritta in precedenza. Attraverso essa è possibile specificare quale byte del dato in ingresso andare a salvare nella locazione di memoria puntata dal vettore di address.

In figura è riportato un estratto del "7 Series FPGAs Memory Resources Manual" nel quale è mostrato un esempio di utilizzo.

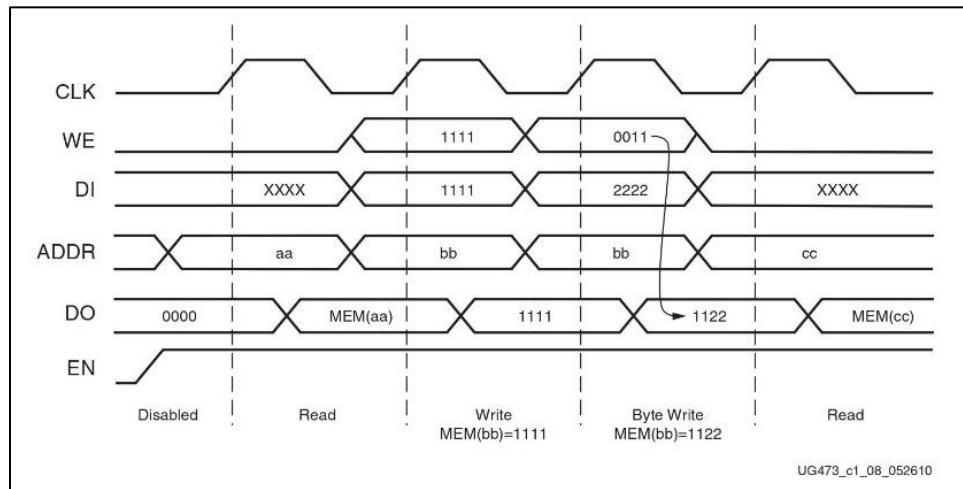


Figura 31:Forme d'onda write enable a byte

L'esempio si basa su una generica porzione di RAM con data\_in a 32 bit (DI è riportato come rappresentazione decimale dei valori che assumono ognuno dei suoi quattro byte).

Al terzo fronte positivo di clock, sulla porta del write-enable infatti al valore 0011 è associata la scrittura dei soli ultimi 2 dei 4 byte asseriti sul vettore di dati in ingresso. Verranno dunque sovrascritti gli ultimi 16 dei 32 bit della word associata all'indirizzo "bb" cosicché in memoria, e conseguentemente in uscita avremo la word composta da due diverse scritture.

Analogamente a quanto appena mostrato ci siamo serviti di 32 bit di write enable in modo da poter selezionare byte a byte quale dei 256 bit della word vanno effettivamente salvati in memoria. Nell'esempio di funzionamento che stiamo prendendo in considerazione, in base al valore che assume il bit address\_a[0] andremo a puntare alla zona di "destra" o di "sinistra" della memoria andando a definire il vettore di write enable in due modi:

- address\_a[0] = 0 → we\_a = 0x 0000 FFFF
- address\_a[0] = 1 → we\_a = 0x FFFF 0000



Un analogo discorso è possibile farlo per i due casi in cui ogni word della ram corrisponde a 4 entries dell'hash table. Avremo non 2 ma 4 zone distinte della ram demuxate questa volta dagli ultimi 2 bit di address\_a. Conformemente a prima infatti il vettore che effettivamente indirizzerà la block ram sarà costituito da address\_a[11 downto 2] mentre il case combinatorio sarà controllato da address\_a[1 downto 0]. Avendo effettuato questo tipo di "divisione" della memoria potremo indirizzare quattro volte il numero delle entries con keys di grandezza 48 o 32 bit. La relazione LSB dell'address e il vettore di write enable sarà eseguita nel seguente modo:

- address\_a[1:0] = 00            →    we\_a = 0x 0000 00FF
- address\_a[1:0] = 01            →    we\_a = 0x 0000 FF00
- address\_a[1:0] = 10            →    we\_a = 0x 00FF 0000
- address\_a[1:0] = 11            →    we\_a = 0x FF00 0000

Per quanto riguarda le 2 uscite delle hash table, key e value, si è applicata una simile implementazione ma con una leggera modifica. Come prima abbiamo scelto, in base al numero di entries da salvare per ogni word, il numero di bit dell'address da usare in un case combinatorio. Nel case vengono ripartite, come già accennato prima, le uscite nei relativi segnali di key\_out\_a, value\_out\_a, key\_out\_b e value\_out\_b. Tuttavia, il controllo non è più effettuato sui due address in ingresso bensì sugli stessi segnali di address rallentati di un colpo di clock. Si sono infatti aggiunti due registri in modo da ottenere i due segnali in uscita address\_out\_a e address\_out\_b. I registri infatti risultano necessari per evitare loop combinatori che si sarebbero venuti a formare. Ciò è dovuto principalmente dal fatto che l'address in ingresso è combinatorio con la chiave in ingresso (ricordo che l'indirizzo non è altro che l'output della funzione di hash applicata alla chiave) e che la chiave in uscita è usata nella macchina a stati delle hash table. Tuttavia, l'aggiunta di un registro tra address\_a e address\_out\_a non modifica in alcun modo la logica di funzionamento. Infatti, analogamente a come abbiamo ritardato l'indirizzo in ingresso di un colpo di clock, così le

parole in uscita alla ram sono sincrone con esso in quanto impiegano un colpo di clock per asserirsi sul bus d'uscita.

Di seguito sono riportati schemi di principio di funzionamento delle hash table riconfigurabili.

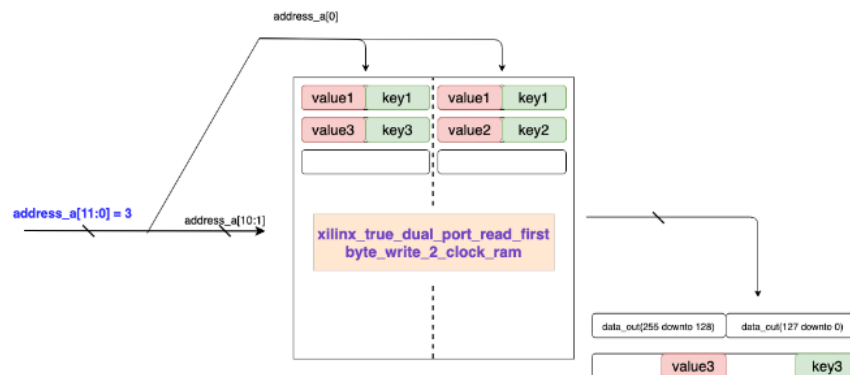


Figura 32: Hash Table Riconfigurabile, ad ogni word corrispondono 2 entries

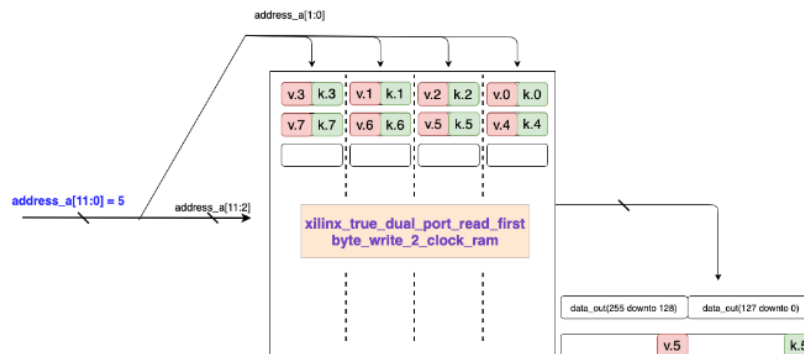


Figura 33: Hash Table Riconfigurabile, ad ogni word corrispondono 4 entries

Va aggiunta un'altra considerazione. Come si può notare nell'entity del modulo Reconfigurator2 , ci sono tre segnali di cui ancora non abbiamo definito lo scopo:

- `control_in`
- `control_out_a`

- control\_out\_b

Come già spiegato precedentemente, in fase di implementazione della hash table si è riservato un bit delle entries per indicare o meno la validità della chiave associata a quella word. Rendendo dinamica la dimensione delle memorie ha perso di significato l'utilizzo di una posizione fissa all'interno del value per salvare tale informazione.

Se infatti precedentemente il bit utilizzato era value(127) ovvero il MSB, ciò perderebbe di senso in tutti gli altri case in cui a priori associamo dimensione di value inferiori da quella massima. Per risolvere tale problema, si è resa dinamica la posizione di tale bit.

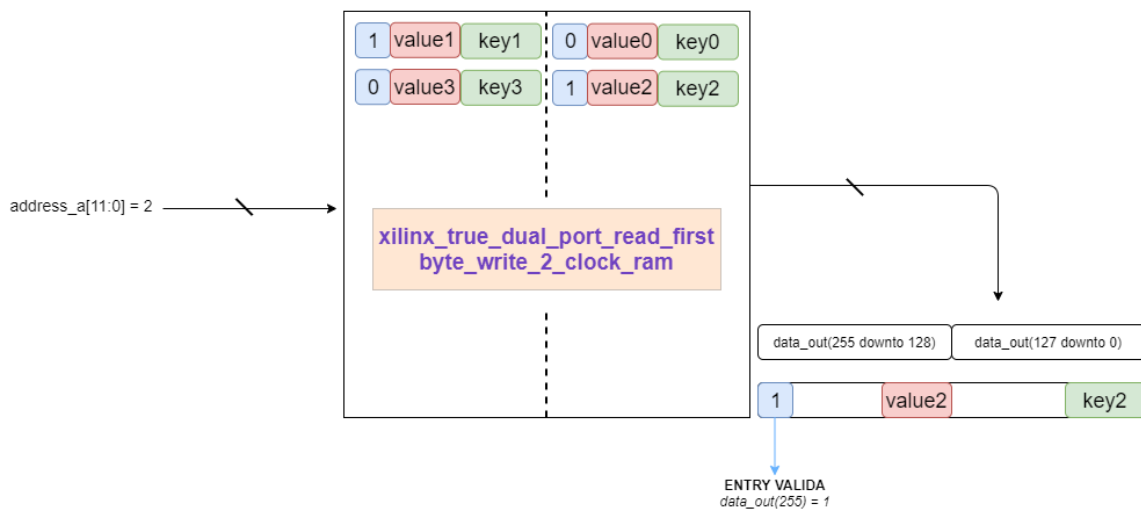


Figura 34: Schema a blocchi dove è evidenziato il bit di presenza delle entries

Una visione di ciò che è stato implementato è riportata nel disegno. Utilizzando control\_out abbiamo reindirizzato il bit di interesse in uscita nella posizione fissa controllata dalla macchina a stati. Il bit di interesse ovviamente dipende anch'esso dalla divisione della ram. Nel nostro esempio in cui ogni hash table ha il doppio delle entries, i due bit di interesse diventerebbero i due value(63) associati ad ogni word della ram ovvero data\_out\_ram(255) e data\_out\_ram(127). Nel case quindi, in base all'indirizzo scelto, reindirizzeremo in uscita uno dei due bit in control\_out.

Abbiamo usato due segnali di control out e uno solo di control in nuovamente per il fatto che la nostra hash table dual port usa una sola delle due porte in scrittura mentre entrambe vengono usate in lettura.

Nella macchina a stati sono poi state ricomposte le word a 256 bit allocando, indipendentemente dalla dimensione scelta, i bit da 127 a 0 alla key e dal 255 a 128 al value. In questo modo è stato facile operare delle operazioni di search sapendo quali bit confrontare del data\_out indipendentemente dalle dimensioni scelte.

## 5.1 Test Bench Stand-Alone

Si è svolta poi una fase di simulazione per accertarsi che il blocco da me implementato funzionasse stand-alone prima di poter essere inglobato in FlowBlaze. Per far ciò si è eseguita una fase di test bench utilizzando il simulatore messo a disposizione da Vivado stesso. Lo scopo è stato quello di testare il funzionamento di una singola RAM true dual port nei 5 diversi case di utilizzo definiti dai valori salvati in key\_len e value\_len.

La simulazione consiste nell'assegnazione iniziale di un valore ad entrambi i registri di key\_len e value\_len. Successivamente si è eseguito il test standard per controllare il funzionamento delle ram che consiste nello scrivere tutte le locazioni di memoria e poi rileggerle tutte. Per rendere facile il debug di una simile simulazione, ad ogni locazione di memoria si è scritto l'indirizzo puntato così in fase di lettura ad indirizzo x ci aspetteremo di trovare salvata esattamente la word x.

```

--CAMBIO DIMENSIONE
--control_in_s <= "11";
key_len_s <= "00100000";
value_len_s <= "00100000";
key_s <= (others => '0');
value_s <= (others => '0');
address_a_s <= (others => '0');
address_b_s <= (others => '0');
we_s <= '1';
wait for 1000 ns;

--SCRITTURA DIMENSIONE 2 (4096x64)
for i in 0 to 4094 loop
    address_a_s <= std_logic_vector(to_unsigned(i,12));
    --address_b_s <= std_logic_vector(to_unsigned(4094 - i,12));
    address_b_s <= std_logic_vector(to_unsigned(i,12));
    control_in_s <= std_logic_vector(to_unsigned(i,2));
    key_s <= key_s + '1';
    value_s <= value_s + '1';
    wait for 50 ns;
end loop;
we_s <= '0';
address_a_s <= (others => '0');
address_b_s <= (others => '0');
wait for 100 ns;

-- LETTURA--
for i in 0 to 4094 loop
    address_a_s <= std_logic_vector(to_unsigned(i,12));
    -- address_b_s <= std_logic_vector(to_unsigned(4094 - i,14));
    address_b_s <= std_logic_vector(to_unsigned(4094 - i,12));
    wait for 50 ns;
end loop;

```

Figura 35: Process di scrittura e lettura del test bench della ram riconfigurabile

Il test è stato eseguito con due process principali. Uno di scrittura in cui è stata scritta la ram usando la porta B e l'altro di lettura dove è stata scorsa tutta la memoria attraverso entrambe le porte ad indirizzi diversi. Inoltre, sono stati assegnati in ingresso valori casuali sulla porta control\_in per poi controllarne il corretto funzionamento su entrambe le porte control\_out.

In figura è riportato un esempio di entrambi i process con key e value a dimensione 64 bit. Negli altri cases sono state modificate le dimensioni dei cicli for per essere concorde con il numero complessivo di entries da salvare e ovviamente sono stati assegnati valori diversi a key\_len\_s e value\_len\_s (con il suffisso s sono stati indicati i segnali durante la fase di test bench). La simulazione si compone quindi di 5 ripetizioni del ciclo mostrato in figura.

Presentiamo ora i risultati ottenuti da tale test e notiamo come il funzionamento è corretto.

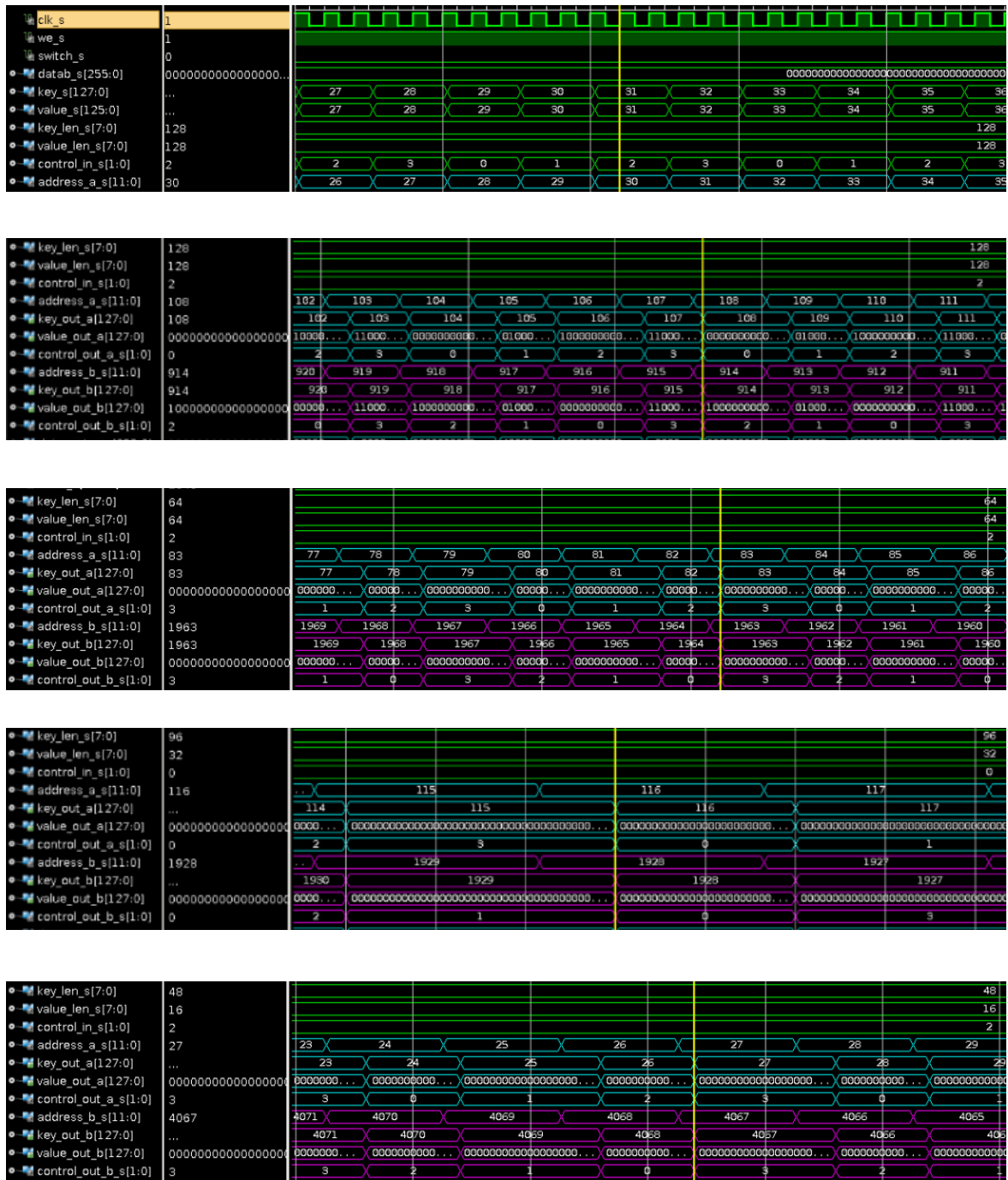


Figura 36: dall'alto al basso:

- 1: scrittura di words a 256 bit
- 2: lettura di words a 256 bit
- 3: lettura di words a 128 bit
- 4: lettura di words a 128 bit
- 5: lettura di words a 64 bit

Si faccia riferimento ai valori dei segnali key\_len e value\_len; la loro somma darà il valore della lunghezza della word salvata

## *Capitolo 6: Setup e risultati sperimentali*

L'ultima fase che ha contraddistinto il mio lavoro di tesi è stata quella di incorporare il mio modulo VHDL all'interno di FlowBlaze e testarne il funzionamento. Essendo FlowBlaze opensource e disponibile su GitHub, il primo passo è stato quello di ricrearne una copia funzionante.

Per fare ciò è stato necessario ricreare un ambiente di sviluppo necessario per interfacciare la NetFPGA SUME. Nella "Getting Started Guide" della repository su GitHub della scheda, è spiegato come. La NetFPGA SUME può essere solamente interfacciata dalla versione 2016.4 della Xilinx Vivado Design Suite su un sistema Unix quindi come primo passo è stato necessario lavorare su una distribuzione di xUbuntu e installare la richiesta versione del software della Xilinx.

Solamente dopo essersi registrati al programma per i beta testers della scheda, è stato possibile scaricare i file necessari per importare l'ambiente di sviluppo dalla cartella della NetFPGA-SUME-live [14]. Continuando a seguire la guida messa a disposizione dagli sviluppatori, attraverso un make file, è stato generato un progetto Vivado importando le specifiche della scheda e gli IpCore standard per interfacciarla. Successivamente, all'interno della cartella appena creata, sono stati importate le sources di FlowBlaze [15] analogamente a come è stato fatto per la NetFPGA. Attraverso un secondo makefile è stato possibile ricreare il progetto di FlowBlaze su cui è stata poi aggiunta la Hash Table da me modificata.

All'interno dell'IDE di Vivado ho aggiunto e collegato la hash table al posto di quella già in uso. Sono state quindi importate le seguenti sources di cui l'ultima corrisponde al template standard di Vivado per definire la Block RAM precedentemente descritta:

- Reconfigurator2\_1.vhd
- ht128new.vhd
- xilinx\_true\_dual\_port\_read\_first\_byte\_write\_2\_clock\_ram.vhd

La gerarchia che vogliamo realizzare è la seguente partendo dall'alto considerando solamente i moduli che ci interessano:

```
top.v
|
+-- nf_datapath.v
    |
    +-- FlowBlaze156_2.vhd
        |
        +-- FlowBlaze_core_new.vhd
            |
            +- cuckoo.vhd
                |
                +- ht128new.vhd
                    |
                    |
                    |   +- Reconfigurator2_1.vhd
                    |       +- Xilinx_true_dual_port_read_first_byte_write.vhd
                    |       |
                    |       +- Reconfigurator2_1.vhd
                    |           +- Xilinx_true_dual_port_read_first_byte_write.vhd
                    |           |
                    |           +- Reconfigurator2_1.vhd
                    |               +- Xilinx_true_dual_port_read_first_byte_write.vhd
                    |               |
                    |               +- Reconfigurator2_1.vhd
                    |                   +- Xilinx_true_dual_port_read_first_byte_write.vhd
                    |                   |
                    |                   +- Reconfigurator2_1.vhd
                    |                       +- Xilinx_true_dual_port_read_first_byte_write.vhd
                    |                       |
                    |                       +- stash.vhd
```

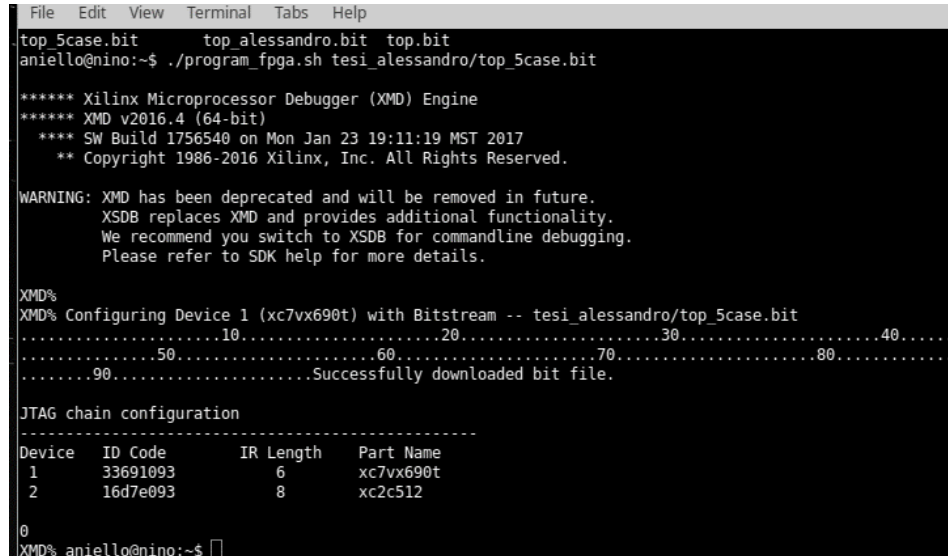


Nel modulo cuckoo.vhd è stata dichiarata l'entity relativa alla hash table, quindi mi sono limitato a rimuovere quella presente e ad inserire la mia. Essa richiamerà automaticamente le altre due sources generando le 4 hash tables volute. È stato inoltre necessario introdurre, sia in cuckoo.vhd che in FlowBlaze\_core\_new.vhd, i due segnali di cui abbiamo già abbondantemente specificato lo scopo: key\_len e value\_len. In particolar modo nel modulo superiore è stato aggiunto un nuovo registro a 32 bit per salvare i valori scritti dall'esterno attraverso il bus S\_AXI, che si basa sul protocollo AXI4\_Lite, di cui ci serviremo dei soli ultimi 16 bit: 8 e 8 per i due segnali. Questi registri vengono scritti in fase di configurazione ma si ha la possibilità di aggiornarli anche durante l'utilizzo della scheda. Lo stesso bus AXI è usato in fase di configurazione della scheda ad esempio per popolare la XFSM, in accordo con le regole specificate in fase di programmazione, nella seconda TCAM di FlowBlaze o anche per scrivere il valore iniziale che dovranno assumere i registri globali. In fase di utilizzo della scheda è possibile inoltre rileggere il valore salvato nei registri specificando il valore di address a cui si vuole accedere sul bus S\_AXI\_ARADDR e leggendo S\_AXI\_RDATA.

Una volta aggiunto il nuovo registro è stato settato il valore salvato di default durante la fase di reset della scheda. In particolare, come valore di reset è stato assegnato ad entrambi i segnali il valore in binario di 128. Unita la hash table a FlowBlaze, si è passato a testarne il funzionamento attraverso il test di utilizzo della scheda. Nel dettaglio, abbiamo generato un file di segnali di stimoli AXI da utilizzare in ingresso alla scheda chiamato test0.axi. All'interno di esso è stata specificata una trama di controllo per testare il funzionamento della scheda. Nello specifico, per testare il corretto funzionamento della hash table, è stato controllato a determinati intervalli l'andamento del suo segnale di match in uscita.



far assumere ai registri. Lo script `rwaxi` è stato poi usato per caricare tali valori sulla scheda attraverso il bus `S_AXI`.



```
File Edit View Terminal Tabs Help
top_5case.bit top_alessandro.bit top.bit
aniello@nino:~$ ./program_fpga.sh tesi_alessandro/top_5case.bit

***** Xilinx Microprocessor Debugger (XMD) Engine
***** XMD v2016.4 (64-bit)
**** SW Build 1756540 on Mon Jan 23 19:11:19 MST 2017
** Copyright 1986-2016 Xilinx, Inc. All Rights Reserved.

WARNING: XMD has been deprecated and will be removed in future.
XSDb replaces XMD and provides additional functionality.
We recommend you switch to XSDb for commandline debugging.
Please refer to SDK help for more details.

XMD%
XMD% Configuring Device 1 (xc7vx690t) with Bitstream -- tesi_alessandro/top_5case.bit
.....10.....20.....30.....40.....
.....50.....60.....70.....80.....
.....90.....Successfully downloaded bit file.

JTAG chain configuration
-----
Device ID Code IR Length Part Name
1 33691093 6 xc7vx690t
2 16d7e093 8 xc2c512
0
XMD% aniello@nino:~$
```

Figura 38: Programmazione della FPGA attraverso lo script `program_fpga.sh`

Per testare il progetto, è stata montata la NetFPGA su una delle interfacce PCI di un server. Due delle quattro periferiche di rete sono state collegate con due connettori ottici a due porte ethernet della NIC del server stesso. Lo scopo del test è stato quello di emulare un normale utilizzo da switch di FlowBlaze per controllarne solamente la corretta funzionalità e non tanto testarne le performance attraverso uno stress test. Dunque, non sono state eseguite misurazioni sul Load Factor e sul numero di probes massimi in fase di saturazione della HT.

Nella fattispecie il test si è basato su un ping in uscita da una porta della NIC e in ingresso a FlowBlaze. Quest'ultimo ha poi inoltrato il flusso nella seconda porta e abbiamo riletto il traffico in ingresso alla seconda interfaccia di rete del server. Il server poi ha risposto al ping rimandandolo in ingresso a FlowBlaze il quale lo ha poi reindirizzato sulla prima porta del server.

Abbiamo controllato il traffico generato dal ping e abbiamo riletto la risposta dalla prima interfaccia del server ed inoltre ci è interessato rileggere il traffico in ingresso e uscita alla seconda porta. A fine test inoltre sono state rilette le entries valide nelle 4 hash tables attraverso il bus S\_AXI attraverso uno script in bash.

Le due interfacce di rete sono state create virtualmente all'intero del server attraverso uno script in modo tale da isolarle internamente dal sistema operativo. Se non fossero state virtualizzate, un ipotetico ping inviato all'ip 2 dall'ip 1, sarebbe stato gestito internamente dall'os avendo esso infatti conoscenza di entrambi, senza farlo passare attraverso la NetFPGA.

Come primo passo abbiamo assegnato due mac address alle due interfacce di rete del server e abbiamo assegnato due indirizzi ip:

MAC: AA:AA:AA:AA:AA      ip:10.10.10.10      ad eth2

MAC: BA:BA:BA:BA:BA      ip:10.10.10.11      ad eth3

L'operazione è stata eseguita con i due seguenti comandi:

```
# ifconfig eth2 hw ether aa:aa:aa:aa:aa
```

```
# ifconfig eth3 hw ether ba:ba:ba:ba:ba
```

```
# ip a add 10.10.10.10 dev eth2
```

```
# ip a add 10.10.10.11 dev eth3
```

Successivamente sono state aggiunte le rotte in modo tale che il server possa reindirizzare il traffico destinato ad un certo ip su un'interfaccia specificata; nel nostro caso abbiamo imposto che il traffico in uscita destinato all'ip 10.10.10.11 fosse reindirizzato e deviato sull'interfaccia eth2. Così facendo il traffico in uscita da FlowBlaze arriva ad eth3 contraddistinta dall' ip cercato; in questo modo risponderà al ping e verrà riletto su eth2.

Questa operazione è stata eseguita attraverso la seguente opzione del comando ip dove è specificato l'ip e la maschera; nello specifico con 32 ci riferiamo ad una maschera "nulla" ovvero stiamo aggiungendo la rotta per uno e un solo indirizzo ip senza dare una wildcard a nessun campo:

```
# ip route add 10.10.10.11/32 dev eth2
```

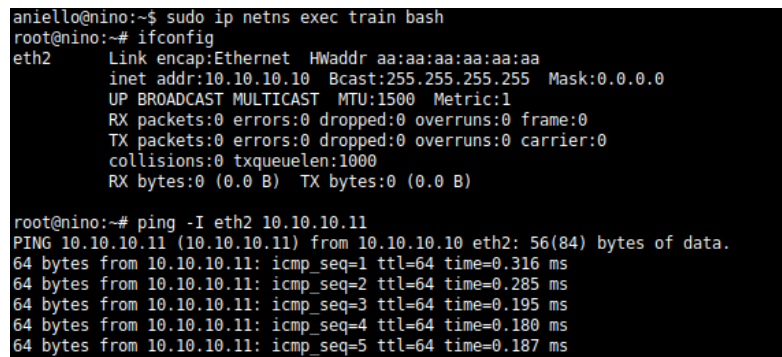
In seguito, è stato necessario popolare le tabelle ARP in modo tale che al server fosse nota l'associazione MAC e IP attraverso nuovamente il comando ip:

```
# ip neigh add 10.10.10.10 dev eth2
```

```
# ip neigh add 10.10.10.11 dev eth3
```

Infine, il ping è stato inviato dalla interfaccia di rete virtuale associata ad eth2 attraverso il seguente comando:

```
# ping -I eth2 10.10.10.11
```



```
aniello@nino:~$ sudo ip netns exec train bash
root@nino:~# ifconfig
eth2      Link encap:Ethernet  HWaddr aa:aa:aa:aa:aa:aa
          inet addr:10.10.10.10  Bcast:255.255.255.255  Mask:0.0.0.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@nino:~# ping -I eth2 10.10.10.11
PING 10.10.10.11 (10.10.10.11) from 10.10.10.10 eth2: 56(84) bytes of data.
64 bytes from 10.10.10.11: icmp_seq=1 ttl=64 time=0.316 ms
64 bytes from 10.10.10.11: icmp_seq=2 ttl=64 time=0.285 ms
64 bytes from 10.10.10.11: icmp_seq=3 ttl=64 time=0.195 ms
64 bytes from 10.10.10.11: icmp_seq=4 ttl=64 time=0.180 ms
64 bytes from 10.10.10.11: icmp_seq=5 ttl=64 time=0.187 ms
```

Figura 39: invio di un ping da eth2 a eth3

Avendo dato esito positivo, si è successivamente passato a testarne il funzionamento a diverse dimensioni. Ciò è stato eseguito cambiando il valore salvato nel registro che abbiamo allocato per key e value len. Nel dettaglio, si è ricaricato il bitstream, iniettati nuovamente i valori nei registri in accordo con la XFSM programmata a priori in XL ed infine siamo andati a riscrivere solamente il registro all'indirizzo AXI 0x80ffff88 attraverso lo script

rwaxi come mostrato in foto. Nuovamente il test è stato quello di mostrare che il ping funzionasse in modo corretto ovvero che nelle hash tables venissero correttamente salvati gli stati relativi ai due flussi contraddistinti dal mac address. Più in dettaglio nella hash table troveremo salvate in key il MAC address di destinazione e in value la porta d'uscita di FlowBlaze associata a tale indirizzo.

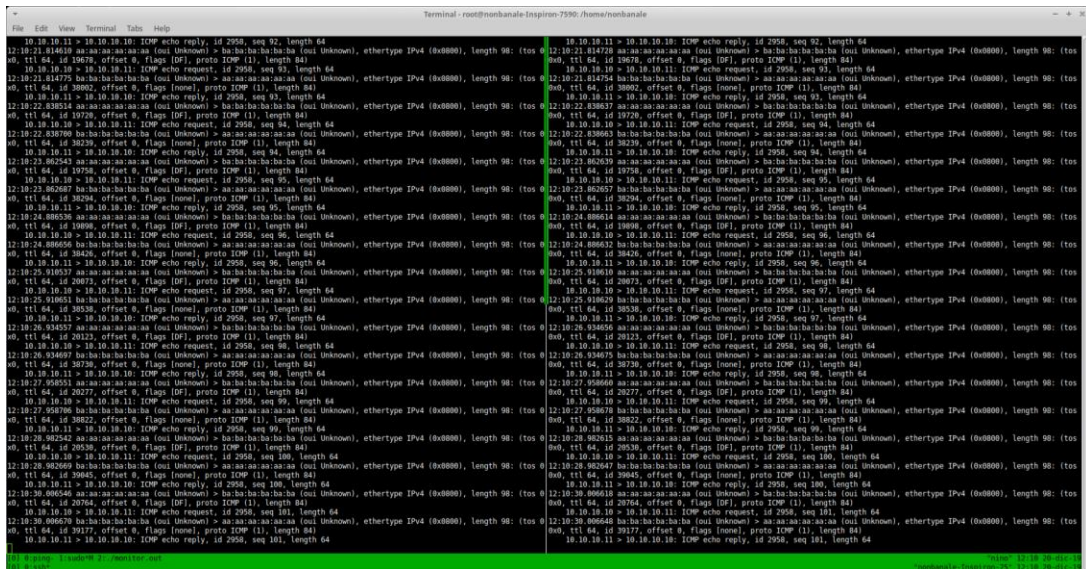


Figura 40: In figura è mostrata la totalità dei pacchetti in ingresso e in uscita da entrambe le interfacci di rete

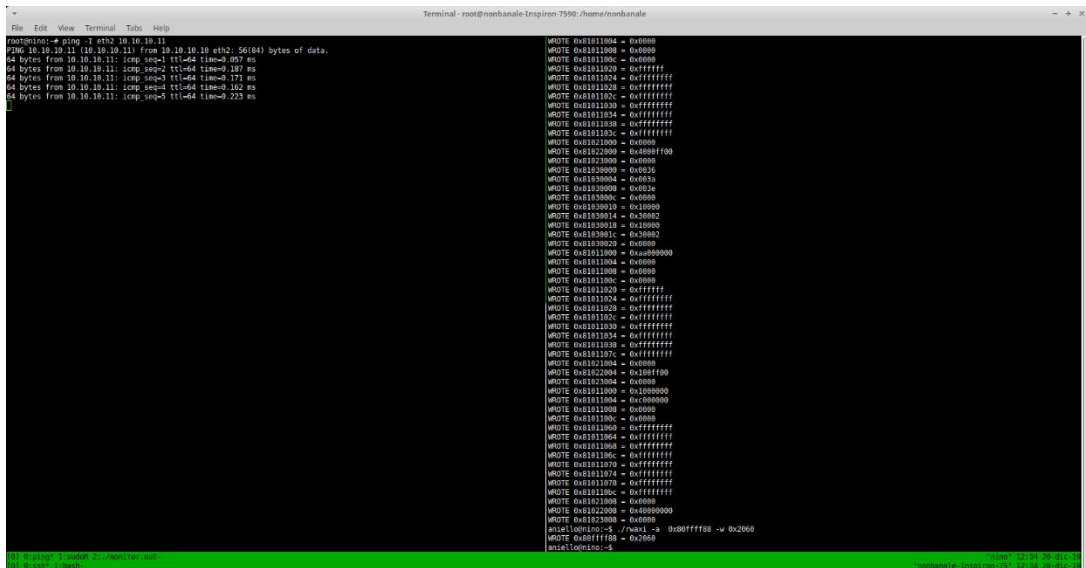


Figura 41: A destra è riportata la fase di programmazione di FlowBlaze seguita dalla scrittura del registro key e value len (key\_len è a 96 bit mentre value len è a 32 bit). A sinistra vediamo come il ping dia esito positivo

```

aniello@nino:~$ ./stampHT2nd.out
===== BITSTREAM =====
19/12/2019 (20:20:35)
=====
searching on HT
-----HT0-----
81101540: AAAAAAAAA 0000AAAA 00000000 00000000 | 00000000 00000000 00000000 C0000000
81101740: BABABABA 0000BABA 00000000 00000000 | 00000000 00000000 00000000 C0000000
Valid entries: 2
-----HT1-----
81109540: AAAAAAAAA 0000AAAA 00000000 00000000 | 00000000 00000000 00000000 C0000000
81109740: BABABABA 0000BABA 00000000 00000000 | 00000000 00000000 00000000 C0000000
Valid entries: 2
-----HT2-----
81111540: AAAAAAAAA 0000AAAA 00000000 00000000 | 00000000 00000000 00000000 C0000000
81111740: BABABABA 0000BABA 00000000 00000000 | 00000000 00000000 00000000 C0000000
Valid entries: 2
-----HT3-----
81119540: AAAAAAAAA 0000AAAA 00000000 00000000 | 00000000 00000000 00000000 C0000000
81119740: BABABABA 0000BABA 00000000 00000000 | 00000000 00000000 00000000 C0000000
Valid entries: 2
Total count: 8
aniello@nino:~$

```

Figura 42: Uno script ha facilitato la lettura delle hash tables dopo il test effettuato utilizzando il bus AXI. Notiamo i MAC address di entrambe le interfacce salvate nelle entries valide

```

=====
=====ETH=====
INTERFACEID | PKTIN | PKTOUT |
01 | 1 | 1 |
04 | 0 | 0 |
10 | 0 | 0 |
40 | 1 | 1 |
-----
TOT | 2 | 2 |

```

Figura 43: Attraverso un altro script abbiamo monitorato i pacchetti in ingresso ed in uscita alla NetFPGA. In particolare, notiamo come le due interfacce, 01 e 40, abbiano pacchetti in ingresso e uscita dati dal ping in corso.

## *Conclusioni e sviluppi futuri*

Il mio lavoro di tesi ha cercato di affrontare il problema di staticità legato alle dimensioni delle chiavi e dei valori salvati nelle hash tables del prototipo di FlowBlaze. A tal fine ho implementato un modulo per la dinamizzazione, anche in fase di utilizzo, di tali dimensioni. Si è dunque incrementata la flessibilità di utilizzo portando miglioramenti in ambito di “space efficiency” aumentando il numero massimo di entries salvabili pur mantenendo le stesse dimensioni delle memorie allocate, e si sono avute ottimizzazioni anche in ambito di minor probabilità di collisione.

Per realizzare ciò mi sono inizialmente documentato su FlowBlaze e ne ho studiato l’architettura usata. Ho ideato il modulo “Reconfigurator” in un linguaggio di basso livello di descrizione noto come VHDL per la programmazione di schede FPGA con logica programmabile. Ho eseguito una fase di test “stand-alone” del mio modulo e successivamente l’ho integrato nel prototipo di FlowBlaze. È seguita quindi una fase di simulazione e successivamente sintesi di FlowBlaze arricchito della mia hash table sulla NetFPGA-SUME, una SmartNIC dotata di FPGA. Infine, è stato eseguito un test su scheda che emulasse un caso di utilizzo da semplice switch tra due interfacce di rete e la NetFPGA.

È di interesse notare come la soluzione da me proposta sia applicabile e replicabile in altri ambiti di utilizzo delle hash tables, strumento spesso adoperato nelle applicazioni di rete o informatiche. Lo stesso modulo VHDL può essere implementato in qualunque tipo di SmartNIC con FPGA così come l’astrazione alla base di esso può essere riadattata in altri ambiti.

Raccomandazioni per ulteriori sviluppi futuri potrebbero essere:

- aumentare il numero massimo di locazioni di memoria utilizzando le RAM DDR3 esterne presenti sulla scheda al posto delle Block RAM utilizzate a disposizione sulla FPGA.
- utilizzare SmartNIC di generazioni successive alla NetFPGA con frequenze di clock più elevate e maggior numero di LUTs come ad esempio le schede UltraScale+ della Xilinx o anche schede della Altera.



# *Bibliografia*

1. Software-Defined Networking: A Comprehensive Survey, 8 Oct 2014
2. Mohammad Alizadeh, Massachusetts Institute of Technology corso 6.888 del 2016  
“Advanced Topics in Networking” Lezione 14, “Software Defined Networking”  
<https://people.csail.mit.edu/alizadeh/courses/6.888/>
3. Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, Jonathan Turner “OpenFlow: Enabling Innovation in Campus Networks”
4. Barefoot Networks, Intel, Stanford University, Princeton University, Google, Microsoft Research “P4: Programming Protocol-Independent Packet Processors”
5. G. Bianchi, M. Bonola, A. Capone, C. Cascone, “OpenState: programming platform-independent stateful OpenFlow applications inside the switch.” CCR Aprile 2014.
6. G. Bianchi, M. Bonola, A. Capone, C. Cascone, S. Pontarelli, D. Sanvito, “Revisiting control/data plane separation in Software Defined Networking” keynote lecture ICETE 2016
7. S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, G. Bianchi, “FlowBlaze: Stateful Packet Processing in Hardware” NSDI '19
8. NetFPGA: <https://netfpga.org/site/#/systems/1netfpga-sume/details/>

9. Michael Mitzenmacker “Probability and Computing Randomized Algorithms and Probabilistic Analysis”
10. Rasmus Pagh e Flemming Friche Rodler “Cuckoo Hashing” Journal of Algorithms May 2004
11. Adam Kirsch, Michael Mitzenmacher, Udi Wieder, “More Robust Hashing: Cuckoo Hashing with a Stash”
12. Dimitris Fotakis, Rasmus Pagh, Peter Sanders and Paul Spirakis, “Space Efficient Hash Tables with Worst Case Constant Access Time” 2005 Theory of Computing Systems
13. Xilinx: 7 Series FPGAs Memory Resources User Guide  
[https://www.xilinx.com/support/documentation/user\\_guides/ug473\\_7Series\\_Memory\\_Resources.pdf](https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf)
14. NetFPGA-SUME open source repo, <https://github.com/NetFPGA/NetFPGA-SUME-public/>
15. FlowBlaze open source repo, <https://github.com/axbryd/FlowBlaze>



