PROGETTAZIONE ED IMPLEMENTAZIONE, SU LOGICA PROGRAMMABILE, DI UNA CUCKOO HASH TABLE RICONFIGURABILE

INDICE

INTRODUZIONE

CAPITOLO1: SDN

CAPITOLO2: Flowblaze e progetto NETFPGA

CAPITOLO3: HASH TABLES e CUCKOO HASHING

-state of the art hashing (liste linkate)

-cuckoo e perché in hardware (constant lookup and insertion time)

-cuckoo double hashing, stash. 4 hash table per riempimento 99 circa della memoria

CAPITOLO4: Modulo VHDL per la riconfigurazione delle hash table

-hash table per la gestione chiavi del lookup e update extractor

-utilità di una hash table con più di 4k locazioni per migliorare riempimento

-logica dietro la riconfigurazione

-descrizione hardware.

-xilinx\_true\_dual\_port\_read\_first\_byte\_write\_2\_clock\_ram

-case combinatori in ingress e uscita

-5 diversi case di utilizzo

CAPITOLO5: Implementazione e Simulazione

CAPITOLO6: Reports

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 table con 4 algoritmi di hash diversi in modo da far tendere il riempimento della memoria al 99 percento (cita mitzenmacher e il doppio hashing). Ognuna di esse poggia su una block ram da 1024 entries da 256 bit l’una. Ciò permette ipoteticamente 4k keys salvate in hash table senza considerare il fatto che un riempimento completo della hash table ha probabilità quasi nulla. E’ anche vero che attraverso l’hash cuckoo e i 4 hash paralleli otteniamo la massimizzazione dell’utilizzazione della memoria ma ciò non si avvicina al caso di una hash table perfetta.

//Quest ultimo caso si avrebbe se e solo se a priori potessimo conoscere quali e quante finite chiavi andare ad allocare.

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 instanziare 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 32 bit. 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 dinamicamente 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 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 il bus S\_AXI in ingresso alla scheda, specificando l’indirizzo 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.

#AGGIUNGI SNAP DEL RESET DI FLOWBLAZE

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

-key­\_len = 128 bit ; value\_len = 128 bit –valore di default

-key\_len = 96 bit ; value\_len = 32 bit

-key\_len = 64 bit ; value\_len = 64 bit

-key\_len = 48 bit ; value\_len = 16 bit

-key\_len = 32 bit ; value\_len = 32 bit

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 instanziare e descritte nel paragrafo xilinx\_true\_dual\_port\_read\_first\_byte\_write\_2\_clock\_ram. 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 0x80ffff88 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) bit = 128 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 256x1024 / 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 creando virtualmente una ram allungata. In questo modo avremo 2048 locazioni da 128 bit ciascuna.

#DRAW.IO DI UNA RAM SMEZZATA E ALLUNGATA

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.

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.

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 determinato hashando la 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 l’ultimo bit dell’address, address\_a[0], serve per demuxare la word in ingresso tra le due parti della block ram mentre address\_a[10 downto 1] è usato come address della block ram. In questo modo abbiamo virtualmente allocato 2^11 locazioni di memoria con una ram che ne contiene 2^10.

Per realizzare ciò si è ricorso alla porta byte-wide write enable dell’ip della xilinx\_true\_dual\_port\_read\_first\_byte\_write\_2\_clock\_ram attraverso la quale è possibile specificare quale byte del dato in ingresso andare a salvare nella locazione di memoria puntata dal vettore di address. Si è dunque definito il valore del write enable in dipendenza del bit address\_a[0]. In questo modo se l’address punta alla zona di “destra” della memoria, il vettore di write enable sarà di conseguenza tale da permettere la scrittura solo di quella parte della cella di memoria. In figura vediamo come ciò sia stato eseguito attraverso un secondo case combinatorio nel quale vengono definiti data\_in\_ram\_a e we\_byte\_a.

#CODICE

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. Analogamente 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.

Per quanto riguarda le 2 uscite delle hash table si è applicata una simile implementazione ma con una leggera modifica. È stato indispensabile infatti ripartire le uscite nei relativi segnali di key\_out\_a, value\_out\_a, key\_out\_b e value\_out\_b attraverso il controllo non più dei due address in ingresso bensì degli stessi segnali di address rallentati di un colpo di clock. Il registro infatti è necessario 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 la chiave hashata) 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.