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

-hash tables flowblaze (perchè servono)

-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

-d-ary cuckoo 🡪 4 cuckoo per riempimento

-stash 🡪 abbassiamo la probabilità di insert failure di parecchio (esponenzialmente con la dimensione dello stash. 8 è più che sufficiente)

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 ingresso e uscita

-5 diversi case di utilizzo

CAPITOLO5: Implementazione e Simulazione

CAPITOLO6: Reports

Capitolo 3: Hash Tables

Come è già stato descritto nel capitolo 2, Flowblaze implementa una cuckoo hash table 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.

HASH TABLE

Immagine che contiene testo

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

Figura 1: Schema base di funzionamento dell'hashing. Le chiavi sono prese dall'universo U e mappate nelle celle della hash table attraverso una hash function

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 (Θ(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 (Θ(logn)).

Grazie invece ad una hash table potrei controllare la presenza o meno della parola con un solo confronto velocizzando di molto il processo (Θ(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 incombere in una collisione. In base allo scopo di utilizzo scelto per la hash table, ci sono diversi modi per risolvere le collisioni.

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 notabile 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.

CUCKOO HASHING

La cuckoo hashing è un’astrazione ideata da Rasmus Pagh e Flemming Friche Rodler presentata nel Journal of Algorithms nel 2004. 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 hashata usando la seconda hash function e posizionata nella seconda hash table. Se anche questa posizione sarà occupata da una chiave, il processo verrà ripetuto finché tutte le chiavi non avranno una propria locazione di Immagine che contiene testo

Descrizione generata automaticamentememoria.

Figura 2: 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 T1 e causare poi un kick in T2 in caso di collisione. Nell’esempio l’inserimento di x causa lo spostamento di y in T2 e di z in T1. 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 come suggerirebbe facilmente il buon senso. In figura infatti stiamo cercando di “stipare” Immagine che contiene testo

Descrizione generata automaticamente7 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 3: Loop cuckoo hashing

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

L’unico modo per risolvere una insertion failure consiste nel rihashare tutte le chiavi presenti nelle tabelle. Nonostante la probabilità che ciò accada non è elevata, si cerca di arginarne l’eventualità in quanto il rehash è molto “costoso” in ambito di performance.

Nel 2008 è stata proposta da Adam Kirsch, Michael Mitzenmacher 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.

Viene modificato l’algoritmo di inserzione nel seguente modo: Si controlla un fallimento o attraverso un prestabilito numero di kick massimi o 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.

Immagine che contiene testo

Descrizione generata automaticamenteIntroduciamo 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”. 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 kickata in una delle d hash table scelte randomicamente.

Figura 4: Space utilization cuckoo hash table in dipendenza delle d hash functions

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 separate di dimensione (1 + ε)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 randomicamente. Se la locazione dovesse essere occupata, verrebbero controllate le altre d-1 tabelle finchè non venga trovata una locazione libera. Se tutte e d le tabelle dovessero essere piene allora verrebbe kickata randomicamente 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 ε, 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 un 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, affinchè 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.

Immagine che contiene testo, screenshot

Descrizione generata automaticamenteImmagine che contiene testo

Descrizione generata automaticamente

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 106 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.

HASH E STASH FLOWBLAZE

#CUCKOO ENTITY E FORSE PURE LA MACCHINA A STATI

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.

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 deasserito. 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, verrano 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, affinchè 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 và 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.

#foto entity stash e hash table

--descrivere funzionamento ht e stash

Capitolo 4: Modulo Riconfiguratore

#entity reconfigurator

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 percento 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.

#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 -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 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 parti 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

Immagine che contiene screenshot

Descrizione generata automaticamenteImmagine che contiene screenshot

Descrizione generata automaticamenteImmagine che contiene screenshot

Descrizione generata automaticamenteImmagine che contiene screenshot

Descrizione generata automaticamente

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. 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 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.

#DRAW.IO DEI 5 CASE