White Rabbit Switch Software

Routing Table Unit Technical Specification

Version: 0.1

Juan Luis Manas (Integrasys)
Miguel Baizán (Integrasys)

January 2011

White Rabbit Routing Table Unit (RTU@SW)

INTRODUCTION

This document first describes the reference functional and non-functional requirements for the WR routing table unit (RTU@SW module), based on the WR specifications and the 802.1D standard. These requirements consider exclusively implementation of the Basic Filtering Services. Then, the high-level reference architecture with the main modules involved in routing table management –both in HW and SW- is outlined. Finally, last section introduces the testing environment and test cases, and presents functional test results.

REQUIREMENT ANALYSIS

FUNCTIONAL REQUIREMENTS

ID	Description
FR1	RTU shall support Basic Filtering Services (802.1D 6.6.5)
FR2	RTU@SW shall support explicit configuration of static filtering information (802.1D 7.1.2)
FR3	RTU@SW shall support automatic learning of dynamic filtering information for unicast destination addresses[1]. (802.1D 7.1.2)
FR4	RTU@SW shall support aging out dynamic filtering information that has been learned. (802.1D 7.1.2)
FR5	RTU shall manage the forwarding and learning performed by each Port dynamically. (802.1D 7.4)
FR6	The learning process shall create or update dynamic filtering entries if and only if: - the receiving port is in appropriate state - the source address is unicast - there are no superior static filtering rules for the MAC - the filtering database is not full (See 802.1D 7.8)
FR7	When the filtering database is full, an existing entry <u>MAY</u> be removed to make space.
FR8	Static filtering information shall not be removed by any aging mechanism. (802.1D 7.9)

[1] Through observation of source addresses of network traffic

FR9	Static and dynamic filtering entries shall comprise: (802.1D 7.9) - A MAC address - A Port Map
FR10	The filtering database shall support the creation, updating and removal of dynamic filtering entries. (802.1D 7.9)
FR11	Static filtering entries shall support at least unicast and group MAC addresses and FORWARD/DROP control elements.
FR12	The filtering database shall contain, at most, one dynamic filtering entry for a given MAC address (802.1D 7.9.2)
FR13	The Aging Time MAY be set by management.
FR14	Aging shall be managed on RTU@SW (wrsw_rtu_wb.vhd)
FR15	A MAC Address can be in a static filtering entry, a dynamic filtering entry, both or neither (802.1D 7.9.5)
FR16	Static Filtering Entries shall be stored permanently.
FR17	The filtering database shall be initialised with static filtering entries.
FR18	Entries shall be added and removed from under explicit management control.
FR19	Reserved addresses shall be configured in the database (See 802.1D table 7.10 and section 7.12.4)
FR20	Management shall not provide capabilities to modify or remove reserved addresses from databases.(802.1D 7.12.6)
FR21	RTU@SW shall monitor the RTU Port Status information as required by the learning process. (802.1D 7.3)

NON-FUNCTIONAL REQUIREMENTS

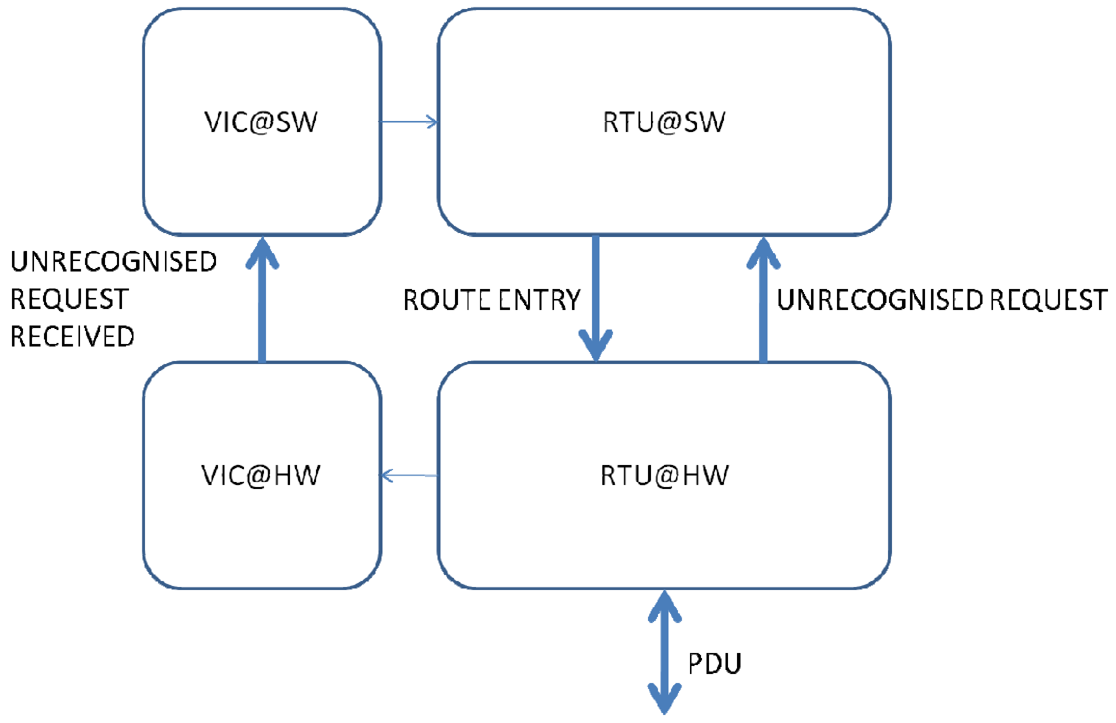
ID	Description
NFR1	RTU@SW shall be prepared for future support of Extended Filtering Services
NFR2	RTU@SW shall be prepared for future support of Remote Management

<u>GENERAL ARCHITECTURE</u>

RTU OPERATION

RTU takes packet source & destination MAC addresses, VLAN ID and priority and decides where and with what final priority (after evaluating the per MAC-assigned priorities, per-VLAN priorities, per-port and per-packet) the packet shall be routed.

The WR RTU consists of a number of hardware and software modules which cooperate to fulfil the abovementioned requirements. RTU@HW handles the switching process while RTU@SW handles de learning and aging processes, managing the filtering database and VLAN table. RTU interruptions are captured at kernel space using the WR Vectored Interrupt Controller (VIC).



At a high level, the normal RTU operation involves the following steps:

0. RTU is initialised
1. RTU@HW receives PDU from SRC MAC to DST MAC
2. If SRC MAC is not in the MAC table, RTU@HW places an UNRECOGNISED REQUEST in the learning queue (UFIFO) and triggers an RTU interruption.
3. If DST MAC is not in the MAC table, RTU@HW broadcasts the received PDU on every port (except the one on which it was received), places an UNRECOGNISED REQUEST in the UFIFO and triggers an RTU interruption.
4. The RTU@SW learning process captures RTU interruption and reads the UNRECOGNISED REQUEST from the learning queue.
5. RTU@SW creates (or updates, if required) a filtering entry in the filtering database[2].
6. The RTU@SW aging process keeps track of the age of each entry.
7. When an entry is too old, the RTU@SW aging process removes the entry from the filtering database.

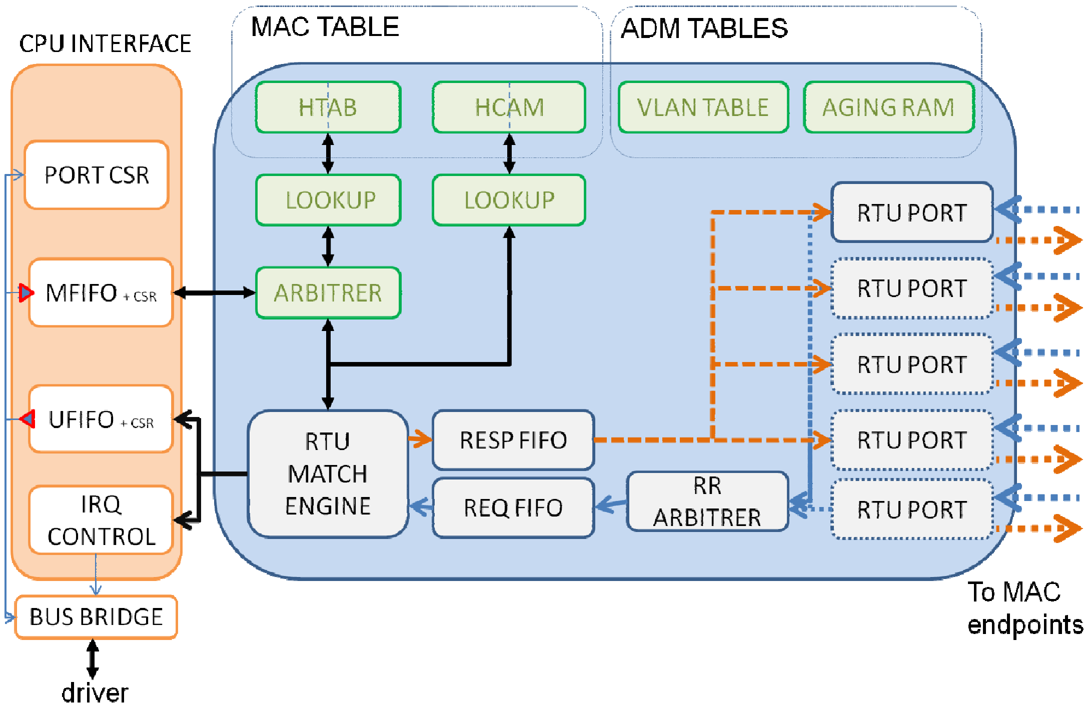
[2] Filtering information is commited to RTU@HW ZBT SRAM via MFIFO. RTU@SW also keeps a filtering database cache.

BLOCK DIAGRAM

This section briefly describes the main RTU blocks for both the RTU@HW and RTU@SW.

RTU@HW

RTU@HW perfoms the core WR packet switching functions based on information provided by the RTU@SW. RTU@SW gains RW access to the RTU@HW components either through direct mapped memory (HCAM, VLAN table, AGING RAM, CSRs), or FIFOs (MFIFO, UFIFO) enabled by the Wishbone bus bridge. Detailed information on the RTU@HW specification is available at the OHWR repository[3].

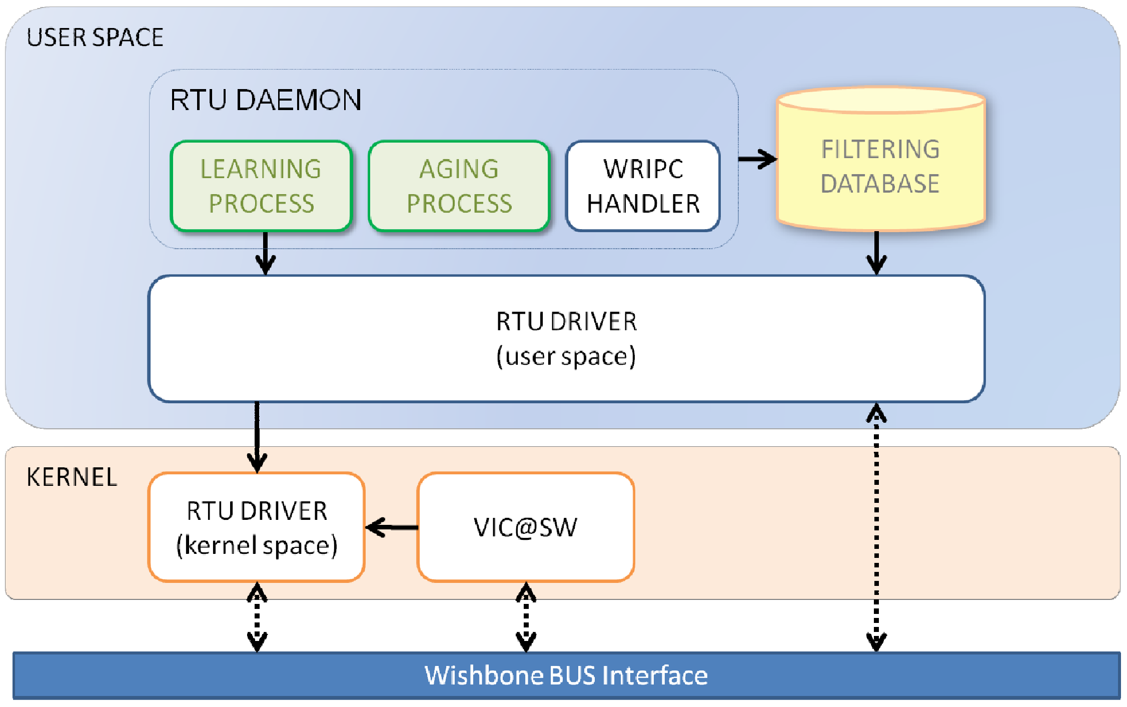


- HTAB: Main hash table storing routing information. Two memory banks. (ZBT SRAM)
- HCAM: Hash collisions memory. Two memory banks (BRAM)
- RR ARBITRER: Round Robin Arbitrer
- AGING RAM: Provides a map of entries accessed by RTU.
- VLAN TABLE: Provides VLAN related info.
- MFIFO: Main hashtable CPU access FIFO
- UFIFO: Unrecognized request FIFO
- CSR: Control/Status Register used for FIFO monitoring and control

[3] See wrsw_rtu_spec.txt for a detailed description of RTU interfaces and data structures.

- PORT CSR: Control/Status Register for Port configuration

RTU@SW



The RTU@SW comprises the following modules:

- RTU DAEMON, which handles the learning and aging processes and manages the filtering and permanent databases. The RTU deamon processes concurrent operation is based on standardised IEEE POSIX 1003.1c pthread libraries.

- The LEARNING PROCESS, continuously awaits for incoming unrecognised requests. Once an RTU interruption request signallig such event is received at the software level, the learning process is responsible for gathering and examining the received request, and create or update a dynamic filtering entry in the filtering database.

- The AGING PROCESS, runs periodically in order to remove old entries from the filtering database. The parameters that control the aging behaviour are the *aging resolution* and *aging time*, which determine the minimum resolution of the entry *last access time* and the maximum time that a non-accesssed entry can be stored in the filtering database.

- The WRIPC HANDLER, provides remote access to the filtering database. Currently used for inspection of stored static and dynamic filtering entries.

- The FILTERING DATABASE, offers the API for the creation and removal of static and dynamic filtering entries, hiding hardware related implementation details to the other RTU modules. It maintains a cache of the HTAB/HCAM and VLAN tables - which are implemented at the RTU@HW. The filtering database supports concurrent access to filtering entries.

- The RTU DRIVER (user space) provides the low level read/write access methods required to communicate with the RTU@HW modules (UFIFO, MFIFO, HCAM, AGING RAM for main hashtable, AGING register for HCAM, RTU Global Control Register and Port Control Register). RTU interruption requests are made available to user space by means of blocking IOCTL calls to the misc device representing the RTU DRIVER at kernel space.

- The RTU DRIVER (kernel space) handles RTU interruption requests. Applies the gnurabbit (at http://www.ohwr.org) misc device concepts to make RTU UFIFO interrupts available to user space.

The RTU@SW operation involves the following steps:

RTU SET-UP

1. The bootstrap process installs RTU driver (kernel space) module[4].
2. RTU driver at kernel space registers interruption handler at VIC@SW for RTU related interruption.
3. RTU daemon initialises RTU@HW
4. RTU daemon sets up filtering database and starts up the learning, aging and wripc processes. It populates the filtering database with static filtering entries.

NOTIFY UNRECOGNISED REQUEST

1. VIC@SW detects RTU interruption.
2. VIC@SW invokes RTU interruption handler (RTU driver at kernel space).
3. RTU interruption handler awakes (if necessary) any process awaiting to read an unrecognised request from UFIFO.
4. RTU driver (user space) gets next unrecognised request from UFIFO.

[4] RTU driver is independent from RTU daemon and is therefore registered apart.

UPDATE ROUTING TABLE

1. The RTU daemon learning process gets unrecognised request (ureq) from UFIFO[5]
2. If MAC address is NOT in the filtering database and the filtering database is not full, the learning process creates or updates the corresponding filtering entry in the filtering database.
3. The filtering database stores a copy of the filtering entry in the local cache and invokes RTU driver (user space) to write the filtering entry into the HTAB (via MFIFO) or HCAM, as required.

AGING OUT FILTERING ENTRIES

1. Periodically[6], the aging process flushes old entries from the filtering database.
2. The filtering database takes a snapshot of accessed entries during the last *aging resolution* period and updates the last access timestamp for each accessed entry.
3. Each *aging resolution* period, the *last access* timestamp is checked for all dynamic entries stored at the filtering database. If entry was not access for the aging time period, the entry is removed from the database. Therefore, an entry can remain in the filtering database at most for a period of AGING_TIME + AGING_RESOLUTION.

SYSTEM TESTS

TEST PROCEDURE

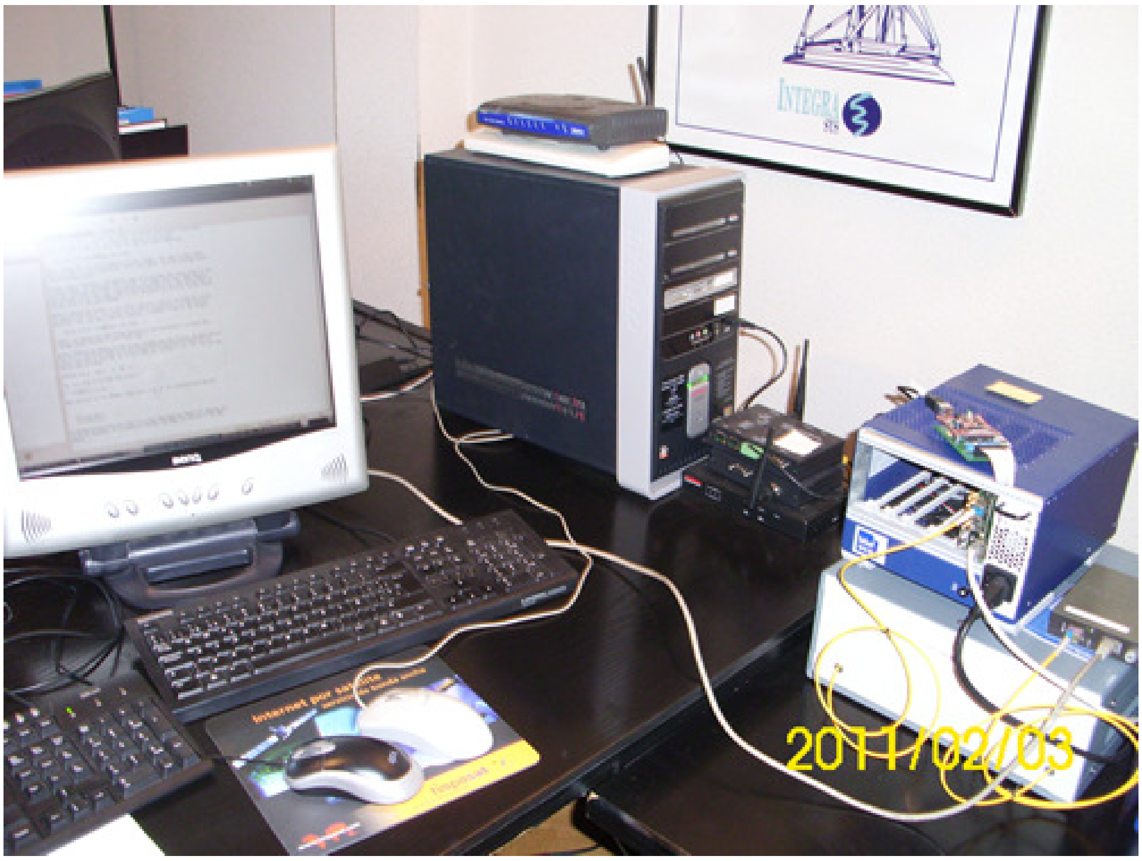
In order to test RTU@SW operation, an Ethernet frame generator[7] was used to produce ETH frames with pre-assigned source and destination MAC addresses. Specific MAC addresses were obtained to fulfil test case requirements in terms of the desired number of MAC hash collisions.

Each test case is manually run and behaviour of RTU is observed. If, once that an Ethernet frame arrives to the WR switch, the RTU correctly updates the aging map for the corresponding filtering database entry, the test case is considered a success.

[5] The RTU daemon remains blocked, awaiting unrecognised requests in UFIFO
[6] Each aging resolution time
[7] (based on the use of rawsockets)

TESTBED CONFIGURATION

The Ethernet frame generator was run on a Linux Ubuntu 10.04.1 PC with processor AMD Turion X2 Dual Core. Generated Ethernet frames were sent from a Broadcom Netlink Gigabit Ethernet card to a WR switch node v2.0, using a Trendnet TFC-1000 MGB optical converter to connect to one of the available SFP uplink ports (wru1).



TEST RESULTS

Test Case #1	
Name	Create HTAB entries
Goal of the test case	Check that entries are correctly stored at HTAB buckets, according to the number of hash collisions previously observed.
Scenario description	An ethernet frame generator was used to send a predefined number of ethernet frames with MAC addresses that produced a given number of hash collisions. The ethernet frames were periodically re-sent to the WR switch. The test

	case was repeated for different numbers of hash collisions between (0 to 3)
Test results	Entries were stored in HTAB buckets numbered 0 to 3 for the hash collisioning source MAC addresses. Position was correctly determined based on the number of collisions. RTU@HW correctly accessed to HTAB entries.

Test Case #2	
Name	Create HCAM entry
Goal of the test case	Check that, once that HTAB is full for a given hash, a new collisioning entry is correctly stored at HCAM. Check that last entry at HTAB is updated to point to following entry at HCAM.
Scenario description	Same scenario as the Test Case #1. Ethernet frame generator was run to produce frames with 4 hash collisions, thus forcing the RTU to store one entry at HCAM.
Test results	First four collisioning entries were stored in HTAB buckets numbered 0 to 3 for the hash collisioning source MAC addresses. The fifth collisioning frame was stored at bucket 0 of HCAM. Last entry at HTAB was updated: go_to_cam and cam_addr fields were updated to point to HCAM addr 0. RTU@HW correctly accessed both HTAB and HCAM entries.

Test Case #3	
Name	Create multiple HCAM entries
Goal of the test case	Check that multiple collisioning entries are correctly stored at HCAM.
Scenario description	Same scenario as the Test Case #1. Ethernet frame generator was run to produce frames with up to 36 hash collisions, thus forcing the RTU to store up to 32 entries at HCAM.
Test results	The process of updating the HCAM list to append new entries to an existing entry list was observed. The process involved updating the last HCAM entry, which was no longer the last one storing the new entry at the end of the existing list HCAM. RTU@HW correctly accessed both HTAB and HCAM entries.

Test Case #4	
Name	Create multiple HCAM entries for multiple collisioning hash

	lists
Goal of the test case	Check correctness of the algorithm that calculates the position of an appropriate empty bucket in HCAM for a new collisioning list.
Scenario description	Multiple ethernet frames generator processes were run in order to force the use of HCAM for storing multiple collisioning hash lists.
Test results	Operation of the find_empty_bucket() algorithm was verified. The method correctly returned the bucket which was at the middle of the longest free HCAM fragment, thus permiting any existing previous list to continue increasing.

Test Case #5	
Name	Fill filtering database
Goal of the test case	Check RTU behaviour when the filtering database gets full.
Scenario description	Ethernet frame generator was run to produce frames with more than 36 hash collisions. This fills up the HCAM and therefore no new hash collisioning entry can be stored in the database.
Test results	Once that the first 36 were stored at HTAB (4) and HCAM (32), new incoming unrecognised requests were ignored. The RTU just output an error message “filtering database full” and continued running.

Test Case #6	
Name	Delete HTAB entries (no HCAM contents)
Goal of the test case	Check that HTAB entries are correctly shifted to fill the position left by the removed entry.
Scenario description	The ethernet frame generator was programmed to initially send a list of collisioning hash frames (1 to 3). After a given period of time, all the frames were re-sent to the WR switch, except for the frame that would correspond to the HTAB bucket to delete. As a consequence, the aging process deleted such entry.
Test results	In all cases the aging process correctly removed the entry corresponding to the eth frame that was not resent. MFIFO write operations corresponding to shifting the HTAB buckets following the removed entry were observed. RTU@HW correctly accessed all the remaining HTAB entries.

Test Case #7	
Name	Delete HTAB entries (HCAM contents)
Goal of the test case	Check that HTAB entries are correctly shifted to fill the position left by the removed entry.
Scenario description	The ethernet frame generator was programmed to initially send a list of collisioning hash frames (1 to 3). After a given period of time, all the frames were re-sent to the WR switch, except for the frame that would correspond to the HTAB bucket to delete. As a consequence, the aging process deleted such entry.
Test results	In all cases the aging process correctly removed the entry corresponding to the eth frame that was not resent. MFIFO write operations corresponding to shifting the HTAB buckets following the removed entry were observed. RTU@HW correctly accessed all the remaining HTAB entries.

Test Case #8	
Name	Delete HTAB entries (HCAM contents)
Goal of the test case	Check that HTAB entries are correctly shifted to fill the position left by the removed entry. Check that first HCAM entry is moved to HTAB and HTAB pointer to HCAM is appropriatey updated.
Scenario description	The ethernet frame generator was programmed to initially send a list of 5 or 6 collisioning hash frames. After a given period of time, all the frames were re-sent to the WR switch, except for the frame that would correspond to the HTAB bucket to delete. As a consequence, the aging process deleted such entry.
Test results	MFIFO write operations corresponding to shifting the HTAB buckets following the removed entry were observed. The HCAM write operation corresponding to cleaning the first HCAM entry was observed. RTU@HW correctly accessed all the remaining HTAB and HCAM entries.

Test Case #9	
Name	Delete first HCAM entry.
Goal of the test case	Check that the last HTAB entry is updated to point to the

	following entry at HCAM. Check that the HCAM entry is correctly removed.
Scenario description	The ethernet frame generator was programmed to initially send a list of 5 or 6 collisioning hash frames. After a given period of time, all the frames were re-sent to the WR switch, except for the frame that corresponding to the first HCAM bucket.
Test results	MFIFO write operation corresponding to updating the last HTAB bucket was verified. The first entry at the HCAM list was correctly cleaned. RTU@HW correctly accessed all the remaining HTAB and HCAM entries.

Test Case #10	
Name	Delete intermediate HCAM entry.
Goal of the test case	Check that HCAM entries are correctly shifted to fill the bucket corresponding to the removed entry.
Scenario description	Same scenario as in Test Case #9, considering a number of collisions up to 36. Several intermediate positions to remove were checked (6 to 35).
Test results	HCAM write operations for all the entries following the one to remove were observed. As a result, all the HCAM list was correctly shifted. RTU@HW correctly accessed all the remaining HTAB and HCAM entries.

Test Case #11	
Name	Delete last HCAM entry.
Goal of the test case	Check that the HCAM entry previous to the last one is marked as the end of the list.
Scenario description	Same scenario as in Test Case #9. In this case, the position to remove was always corresponding to the last collisioning frame sent by the ethernet frame generator.
Test results	The low level HCAM write operation updating the end_of_bucket bit of the entry previous to the last one was verified. The last HCAM entry was correctly cleaned.