University of Trento
Dipartimento di Ingegneria e Scienza dell'Informazione
" NOMADIC COMMUNICATIONS", Lab tutorial - Dr. F. Gringoli, A.A. 2013/2014

Exercise: heavy hitters

1. Exercise goals

After this exercise students should have acquired enough skills to count heavy hitters in a BSS.

2. Tutorial steps

1) **Counting heavy hitters** Counting "having hitters" is a basic building block for many mechanisms, e.g., improving the fairness, limiting nodes when exceeding a given transmission attempts threshold, detecting a Denial of Service attack, etc. As the resources in the wireless chip are limited, counting heavy hitters directly on the chip may be tricky: for this reason, counting in the chip is usually considered as a method for detecting hardest problems in real-time (e.g., reacting immediately to a single heavy hitter), leaving to the kernel the task of maintaining a complete list of heavy hitters. If we limit the number of heavy hitters we consider at the same time, e.g., 10, we can hence implement a simple search loop that for each incoming frame analyzes the entire heavy hitter table (that we keep in a free portion of the shared memory) and either 1) refresh the number of frames transmitted by a given node, 2) add a new node to a free entry in the table, or 3) replace the oldest entry in the table with a new node.

2) **Detecting transmitter address** Counting heavy hitters should be done while the frame is still being received so that the search code will not block the likely transmission of the acknowledgment at the end of the reception. For this reason in the following we focus again on the receiver code path, in particular on `rx_data_plus` that we know is executed *during* the frame reception. We have to first filter out frames not addressed to the station that is counting (the AP): this can be easily done inside `rx_data_plus` by inserting new code before the last line reported here below:

```
rx_data_plus:
        jext    COND_RX_COMPLETE, end_rx_data_plus
        jl      SPR_RXE_FRAMELEN, 0x01C, rx_data_plus
end_rx_data_plus:
        jl      SPR_RXE_FRAMELEN, 0x01C, rx_check_promisc
        jnext   COND_RX_RAMATCH, rx_ra_dont_match

    [<--------                add      code      here                -------->]

        jext    COND_TRUE, send_response
```

Only frames addressed to the counting/receiving station, in fact, match the Receiver Address test (RA, `COND_RX_MATCH`). We have now to extract the transmitter address (TA) that for a data frame is always the second one.

Question:

a. Is the number of bytes we wait (0x1c) enough for considering the second address in the rx buffer?

To access the TA (addr2) we can use the three 16-bit definitions RX_FRAME_ADDR2_1, RX_FRAME_ADDR2_2 and RX_FRAME_ADDR2_3 that together with the offset register 1 (`off1`, or `SPR_BASE1`) point to the part of the frame that is copied inside the share memory as reception goes ahead: for instance to load the first two bytes of TA in a free register use

```
          mov    [RX_FRAME_ADDR2_1,off1], r63
```

3) **Maintaining the heavy hitter table** A simple table layout could consider for each heavy hitter the following data (we always use 16-bit words as the shared memory is)

a. byte 0-1 of TA of this node
b. byte 2-3 of TA
c. byte 4-5 of TA
d. counter for this node

This layout requires 8 bytes for each node: if we limit the table to the ten heaviest hitters, the entire table takes as only as 80 bytes. However, there is a problem in this case: it is really hard to test if the implemented algorithm works or not, given the limited amount of nodes available for each group. For this reason in the following we consider only the TWO heaviest hitters, so that each group can use four nodes for testing the algorithm. Even if this would appear more clearly in the following, this requires a table with THREE entries: why? We also need two additional locations in the shared memory, one for temporarily preserving the value of the offset register that we will use to loop over the table, and another location to hold the number of valid entries in the table. A possible address choice could be

```
#define BASE_TABLE_ADDRESS     SHM(0xF00)     // start address of the table
#define VALID_ENTRIES          SHM(0xEF0)     // number of entries in the table
#define TEMP_OFFSET            SHM(0xEF2)     // temp save location for offXX reg
#define MAX_ENTRIES            3              // number of entries in the table
```

We then need to define the offset for each entry:

```
#define TBL_B01_OFFSET         SHM(0x00)      // TA
#define TBL_B23_OFFSET         SHM(0x02)      // TA
#define TBL_B45_OFFSET         SHM(0x04)      // TA
#define TBL_COUNTER            SHM(0x06)      // COUNTER
#define TBL_ENTRY_SIZE         SHM(8)         // size of the entry, 8 bytes
```

Questions:

a. As the counter is a 16-bit value, what is the maximum number of frames that we could count for every TA?
b. What kind of problems can cause this maximum count number? Try to propose simple solution for avoiding all nodes under monitoring may reach a sort of saturating condition (this will be implemented in **PART4** section of the code below).

4) **Looping over the table entries** In the following a simple loop is proposed, but many parts have been omitted: it is up to you to fill such parts with the required code. The omitted parts are: **PART1**, for checking if the TA of the frame being received matches that of the specific table entry under analysis during the loop iteration; **PART2**, for tracking the node already in the table that counts the lowest number of transmitted packets; **PART3**, for refreshing the table at the end of the loop; **PART4** for handling the case the node in the table reached the maximum number of frame that can be represented with a 16-bit value.

We can start with this simple loop skeleton:

```
          mov      SPR_BASE0, [TEMP_OFFSET]            // save current value of off0
          mov      BASE_TABLE_ADDRESS, SPR_BASE0       // initialize off0 at table start
          mov      0, r63                              // r63 is the loop register
          mov      0xFFFF, r62                         // r62 counts the lowest counting node
          mov      BASE_TABLE_ADDRESS, r61             // r61 tracks the lowest counting node

search_loop:                                           // loop start label
          je       r63, [VALID_ENTRIES], exit_search   // loop termination condition

PART1     [ add code to compare TA with the address at this position ]
          [ if match, jump to exit_search                            ]

PART2     [ add code to refresh r61 and r62 if needed                ]

          add      r63, 1, r63                         // switch to next entry
          add      SPR_BASE0, TBL_ENTRY_SIZE, SPR_BASE0 // point to next entry
          jext     COND_TRUE, search_loop              // continue the loop
exit_search:

PART3     [ add code to refresh the table                            ]

exit_final:                                            // final loop exit
          mov      [TEMP_OFFSET], SPR_BASE0            // restore the offset register we used
```

If the value of `r63` at the end of the loop (`exit_search`) is less than the number of valid entries, it means that the TA has been found in the table. In this case we should simply refresh the counter but we need to take into account that the counter may reach its maximum, so we need to add some code to handle this specific case: we hence have for **PART3**

```
          je       r63, [VALID_ENTRIES], not_found
          add      [TBL_COUNTER,off0], 1, [TBL_COUNTER,off0]
          jne      [TBL_COUNTER,off0], 0, exit_final
PART4     [ add code to handle the case the counter reached its max  ]
          [ and wrapped around                                       ]
not_found:
```

If instead the value of `r63` is equal to the number of nodes we have in the table, then there are two possibilities, 1) the number of nodes is less then the maximum number the table can hold: in this case we can add a new node at the end of the table; 2) the table is full, in this case we need to replace the entry with the lowest counting node tracked by `r61` with the new one.

Before starting implementing and running (at random ☺) this code in the firmware try to add the missing parts and ask the lab assistant to validate your choices.

Questions:

a. What is the purpose of `r62` in this code?
b. Can we remove it from the algorithm?

5) **Improving heavy hitters detection** An interesting improvement for the considered algorithm is to take time into account. This means that we can make each *hitter* aging by tracking the time we saw its last packet. To this end the table must be expanded as follows:

```
#define TBL_B01_OFFSET         SHM(0x00)      // TA
#define TBL_B23_OFFSET         SHM(0x02)      // TA
#define TBL_B45_OFFSET         SHM(0x04)      // TA
#define TBL_COUNTER            SHM(0x06)      // COUNTER
#define TBL_CLOCK_MSW          SHM(0x08)      // MSW of the entry last seen clock
#define TBL_CLOCK_LSW          SHM(0x0a)      // LSW of the entry last seen clock
#define TBL_ENTRY_SIZE         SHM(12)        // size of the entry, 12 bytes
```

We need to take at least 32-bit of the clock as the clock granularity is 1 microsecond and considering only 16-bit would require the frequent execution of an "anti-wrap" clock counter-mechanism. With 32-bit clock, instead, this can be completely avoided as the duration of the exercise is less then 2^32 - 1 microseconds ☺!!

In the following we will use two of the four registers that timestamp each new arrival, namely `LAST_RXTIME_WORD0` and `LAST_RXTIME_WORD1`, that are initialized by `rx_plcp` at each new frame arrival.

As now we have the concept of TA age, we can track the oldest entry (means: the one that was refreshed first in the table), and in case the table is already full and we have a new TA we can simply replace the oldest TA with the new one. We have to remove the previous usage of `r61` and `r62` and replace **PART2** with a new one that track the oldest flow, e.g., initialize `r61` and `r62` to zero, and we have to consider a new register, `r60` for tracking the address of the oldest TA:

```
mov      LAST_RX_TIME_WORD1, r62              // r62 tracks the MSW of the oldest flow
mov      LAST_RX_TIME_WORD0, r61              // r61 tracks the LSW of the oldest flow
mov      BASE_TABLE_ADDRESS, r60              // r60 tracks the lowest counting node
```

In **PART2** we have to compare `r61` and `r62` with the time-stamp stored in the entry, and if the entry is older than `r62/r61` we refresh the two registers and we load the current value of the `SPR_BASE0` into `r60`.

As the 32-bit values are splitted into couples of 16-bit registers, the comparison is tricky (it involves two subtractions and a test over the carry register): use the following snippet of code at your convenience.

```
// compare VAL1 and VAL2, use r63 as convenience register: store in RES = VAL1 - VAL2
          mov      VAL2LSW, r63
          sub.     VAL1LSW, r63, RESLSW
          mov      VAL2MSW, r63
          subc.    VAL1MSW, r63, RESMSW
          addc     0, 0, r63
// output:
//        r63 = 1, if VAL1 >= VAL2
//        r63 = 0, if VAL1 < VAL2
//        RES = VAL1 - VAL2
```

Also in this case before starting implementing and running the code (at random ☺) ask the lab assistant to validate your choices.