Report Week 4

Achievements:

- Established a simple protocol for writing data to each and every core.
  The smallest unit of data is a 4-byte integer. Every string data entry has to be converted into 4 integers, where every integer stores 4 characters (8-bit each).
  The main purpose of this format is to simplify memory allocation, albeit at the expense of speed when loading data, since strings have to be converted to integers first
  Note: By default, strings are not allowed to have more than 16 characters (for now)
  Details of the data protocol can be seen in image 1 and 2.

Goals for next week:

- Implementing edges to enable communication structures between cores
- Once edges between cores are available, the first testing algorithm will be implemented
- Image 3 shows a possible core structure for accessing data and returning results
- Image 4 proposes an algorithm (count the number of data entries) for testing the connections. Potential problems, such as spike overload, can thus be detected early on. Furthermore, the result is already known before the computation, since we as the users know already from the beginning how many data entries we allocate to each core

Image 1 & 2

```python
def load_data_onto_vertices(total_number_of_cores, data):

    data_len = len(data) -1

    for x in range(0, total_number_of_cores):
        if x < data_len:
            front_end.add_machine_vertex(
                Vertex,
                {
                 "columns":     1,
                 "rows":        2,
                 "string_size": 16,
                 "flag":        [0], #0-string, 1-integer
                 "entries":     [[data[x][0],data[x][0]]]
                },
            label="Data packet at x {}".format(x))

'''
```

```c
//global variables holding the data pointers
struct header_info {

    unsigned int num_cols;
    /* number of columns
     * in the original csv file
     */
    unsigned int num_rows;
    /* number of rows
     * in the original csv file
     */
    unsigned int string_size;
    /* number of bytes that
     * are allocated for each individual string
     */
    unsigned int flag;
    /* flag has 32bits
     * each bit can be 0 or 1
     * 0 stands for string data
     * 1 stands for integer data
     * Example: if the first bit is 0,
     *          the first columns holds string data
     * Warning: There should be no more than 32 columns
     */

};
```

Image 3

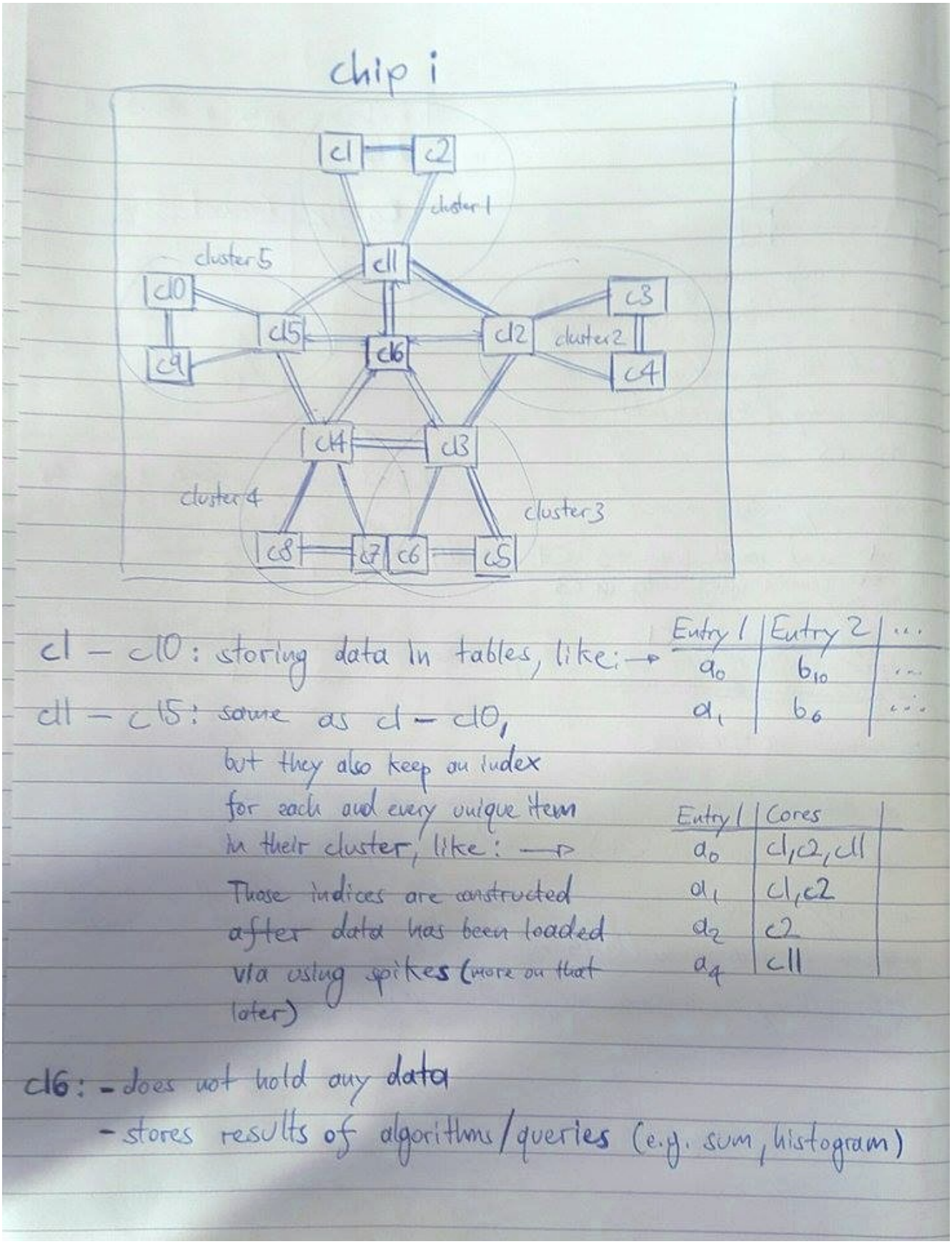
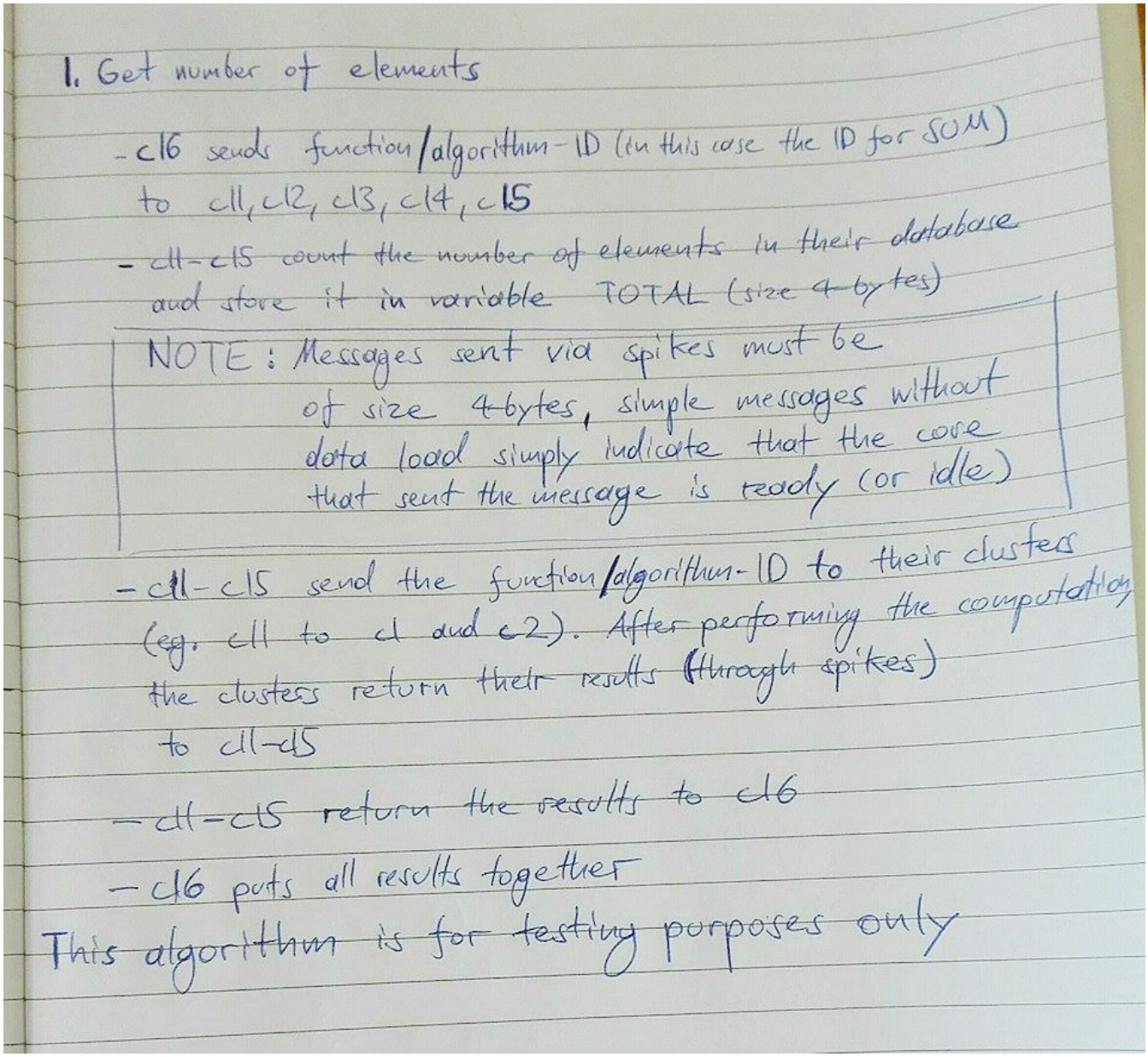

chip i
c1
c2
cluster 1
cluster 5
c11
c0
c3
c5
c12
cluster 2
c6
c9
c4
c14
c13
cluster 4
cluster 3
c8
c7
c6
c5
c1 – c10: storing data in tables, like: →
Entry 1 | Entry 2 | ...
a0 | b10 | ...
a1 | b6 | ...
c11 – c15: same as c1 – c10,
but they also keep an index
for each and every unique item
in their cluster, like: →
Entry 1 | Cores
a0 | c1, c2, c11
a1 | c1, c2
a2 | c2
a4 | c11
Those indices are constructed
after data has been loaded
via using spikes (more on that
later)
c16: - does not hold any data
- stores results of algorithms/queries (e.g. sum, histogram)

Image 4


1. Get number of elements
- c16 sends function/algorithm-ID (in this case the ID for SOM) to c11, c12, c13, c14, c15
- c11-c15 count the number of elements in their database and store it in variable TOTAL (size 4 bytes)
NOTE: Messages sent via spikes must be of size 4 bytes, simple messages without data load simply indicate that the core that sent the message is ready (or idle)
- c11-c15 send the function/algorithm-ID to their clusters (eg. c11 to c1 and c2). After performing the computation, the clusters return their results (through spikes) to c11-c15
- c11-c15 return the results to c16
- c16 puts all results together
This algorithm is for testing purposes only