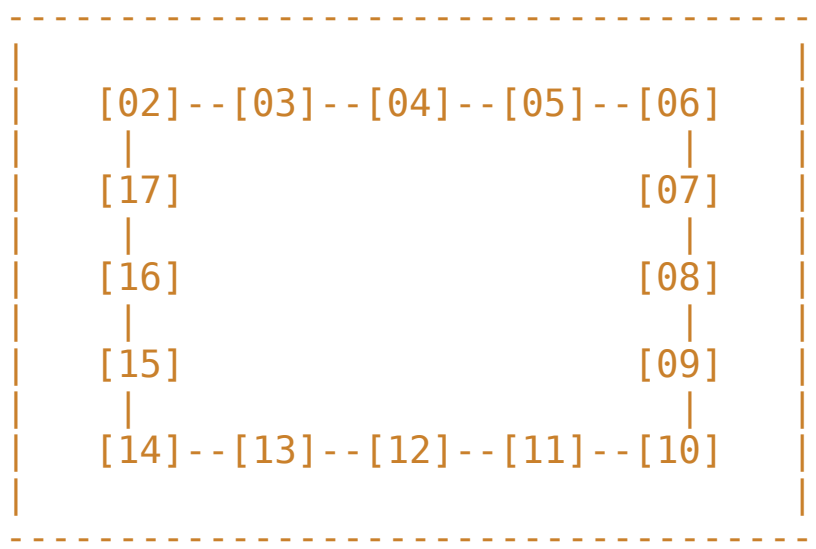
Report Week 6

Achievements:

Right now, 16 vertices on each chip form a ring structure with edges, which looks as follows:



After wiring up the edges, I implemented a simple function that counts the total number of data entries (the number of rows to be more precise) in all cores. Core 02 starts the entire process by sending a MCPL packet containing its number of data entries to core 03. Core 03 increments the received message by its number of data entries and sends it to core 04. The process repeats until core 02 is reached, which will write the content of the MCPL packet back to SDRAM:

```
2017-11-02 16:49:38 INFO: Time 0:00:00.369579 taken by RouterProvenanceGatherer
Getting profile data
|0%                          50%                         100%|
 ============================================================
2017-11-02 16:49:38 INFO: Time 0:00:00.008371 taken by ProfileDataGatherer
2017-11-02 16:49:39 INFO: 0, 0, 2 > 2
2017-11-02 16:49:39 INFO: 0, 0, 3 > 4
2017-11-02 16:49:39 INFO: 0, 0, 4 > 6
2017-11-02 16:49:39 INFO: 0, 0, 5 > 8
2017-11-02 16:49:39 INFO: 0, 0, 6 > 10
2017-11-02 16:49:39 INFO: 0, 0, 7 > 12
2017-11-02 16:49:39 INFO: 0, 0, 8 > 14
2017-11-02 16:49:39 INFO: 0, 0, 9 > 16
2017-11-02 16:49:39 INFO: 0, 0, 10 > 18
2017-11-02 16:49:39 INFO: 0, 0, 11 > 20
2017-11-02 16:49:39 INFO: 0, 0, 12 > 22
2017-11-02 16:49:39 INFO: 0, 0, 13 > 24
2017-11-02 16:49:39 INFO: 0, 0, 14 > 26
2017-11-02 16:49:39 INFO: 0, 0, 15 > 28
2017-11-02 16:49:39 INFO: 0, 0, 16 > 30
2017-11-02 16:49:39 INFO: 0, 0, 17 > 32
```

In this case, I put two data entries on every core before running the algorithm

Documentation (C code):

Disclaimer: This particular code is designed for the ring structure (specified above).
Different topologies require their own set of functions.

void start_processing()
//Given the header.function_id (supplied in the SDRAM), start_processing() selects
//which function to execute. Currently 1 → count, 2 → builds index table

//
// COUNT FUNCTION //
//

void count_function_start()
//if header.initiate_send is 1, this particular vertex is entitled to start the entire process
//by sending the first MCPL package to its neighbour in the ring
//The message is a 32bit integer that represents the number of rows in the data table
//on this particular vertex

void count_function_receive(uint payload)
//if header.initiate_send is 0, we have not reached our original vertex yet -
//therefore we increment the message by the number of rows in the data table on this
//particular vertex and send it to the next vertex. Then, we record the message to
//SDRAM, so that it can be seen in the output later on

//
// BUILD INDEX TABLE FUNCTION //
//

The main idea of this function is to assign a unique identifier of size 4 (bytes) to each unique string data item – while making sure that all of those data items are uniquely identified across all 16 cores within the ring without inconsistencies, such as the same data item having different identifiers on different cores.
String data entries consist of 16 bytes and can only be sent in chunks of four – clogging up spike traffic and slowing everything down. By putting the unique identifiers into place, information regarding the entries can be shared much quicker, since now only the 4 byte identifiers have to be passed around.
Furthermore, the main vertex (or the leader) will keep a dictionary of all indices and data entries in its memory. This will enable checking whether certain data entries exist and what type of id they use within this specific ring network. Once we get to the stage where several rings will be linked together, each leader has to know exactly what information is stored within the network that it is assigned to, be it the current ring or some other form of topology in the future.

```
struct index_info {

	unsigned int *id_index;
	/* Holds the unique identifier for each data entry
	 * Example: id_index[1] contains the unique id for the
	 * second data entry within SDRAM
	 * Currently this works only for one column
	 * Length: header.num_rows
	 */
	unsigned int *message;
	/* Holds 4 integers that make up a string
	 * Designed to take a string entry that has been forwarded
	 * by 4 distinct MCPL packages
	 */
	unsigned int message_id;
	/* Holds the unique id of string above
	 * Takes the id from an incoming MCPL package as well
	 */
	unsigned int messages_received;
	/* Keeps track of number of MCPL packages received
	 * if messages_received mod 5 = 0, a string data entry and its
	 * id have been received
	 */
	unsigned int index_complete;
	/* A flag that tells if the index on this vertex is complete
	 * Complete = 1; Incomplete = 0;
	 * Complete means that there are no indices left with value 0
	 */
	unsigned int max_index;
	/* A flag that tells if the index on this vertex is complete
	 * Complete = 1; Incomplete = 0;
	 * Complete means that there are no indices left with value 0
	 */

};

struct index_info local_index;
```

All index information is available globally through “struct index_info local_index”

void initialise_id_index()
//The first function that is executed by every core before any MCPL packages have
//been sent.

//Step 1: All vertices allocate memory to their index
→ local_index.id_index = malloc(sizeof(unsigned int) * header.num_rows)

//Step 2: All vertices create a buffer for receiving string messages
→ local_index.message = malloc(sizeof(unsigned int) * 4)

//Step 3: If this is the vertex leader:
Go through all entries in SDRAM via
“address = data_specification_get_data_address()”
and put unique IDs into the “local_index.id_index[i]”
Every ID in “local_index.id_index[i]” corresponds
to an entry in the SDRAM in the given order.
The algorithm makes sure that no two identical data entries get different IDs

//Step 3: If this is a normal vertex:
Set all Ids to 0 in “local_index.id_index[i]” for all I

void unique_id_function_start()
//This function is only invoked by the leader vertex, since only it fulfils the following
//condition: header.initiate_send == 1
//What it does is simply sending the first string entry to its neighbour vertex alongside
//its id. This requires 4 MCPLs for the string data entry and 1 MCPL for the id.

void unique_id_function_receive(uint payload)
//Whenever a string data entry is being sent to the current vertex, the vertex waits
//until it receives all 4 MCPLs for the string entry and the 1MCPL for the id. Once all
//information has arrived, it is stored in “local_index.message” and
//“local_index.message_id”. The function then invokes “update_index()”

void update_index()
//This function takes the newly arrived message from “local_index.message” and
//“local_index.message_id” and checks if any string data entries within its section of
//the SDRAM memory are identical with “local_index.message”. There are several
//scenarios:
//
//Scenario 1: There are data entries which are identical with local_index.message.
Their ids are still 0.
→ The ids of the data entries are updated to local_index.message_id
//
//Scenario 2: There are data entries which are identical with local_index.message.
Their ids are not 0 anymore.
→ The message has gone through all cores already.
Invoke message_reached_sender() to handle this

void message_reached_sender()

Vertex [02] – Leader Index Table	
Data Entry	**Unique ID**
United Kingdom	1
Germany	2
France	3
Germany	2
United Kingdom	1

Vertex [02] – Leader Lookup Table	
Data Entry	**Unique ID**
United Kingdom	1
Germany	2
France	3
Spain	4
Greece	5

Vertex [03] – Chain Index Table	
Data Entry	**Unique ID**
Spain	4
Spain	4
Germany	2
Germany	2
Germany	2

Vertex [04] – Chain Index Table	
Data Entry	**Unique ID**
Greece	5
Spain	4
United Kingdom	1
Germany	2
Germany	2