CFG reconstruction

Level 0: Manual assembly code parser and CFG builder

- We might distinguish two types of basic blocks :
  - basic blocks that end with one of the branching instructions : `beq, bne, b, bl`.
  - basic blocks for which the address of the first instruction correspond to a branching address from one or more previous basic blocks.
- One convenient data structure for manual parsing, is to use arrays in python such that :
  - One array will contain the sequence of start addresses of the basic blocks.
  - A second array will contain the corresponding sequence of sets (up to two elements) of branching addresses.
  - We consider the index if elements in both arrays as their corresponding basic block identifier.
- The algorithm we use to ecode the graph is as follows :

```
find the beginning of the ControleurPorte_step function.
branch_addresses_set := $\emptyset$
while not the end of the function do
    fill the start address of the current basic block;
    add this same address as a next address of the previous basic block (if exists);
    while not the end of the current basic block do
        foreach instruction do
            if instruction address is in branch_addresses_set then
                mark the end of the current basic block;
            end
            if instruction word contains one of the operations beq, bne, b, bl then
                mark the end of the current basic block;
                branch_addresses_set.APPEND(instruction.branch_address);
            end
        end
    end
end
```

Level 1: Assembly code parser

At this level (as we aim to automate the parsing), we are going to use data structures in the C language

- The main idea of the functionning of the parser, is to read the assembly code file line by line, and checks the existance of indications about the limits of a basic block. The major steps of its functionning is as follows :
  1. Find the beginning of the `ControleurPorte_step` function.
  2. Detect the limits of the basic blocks based on the idicators cited above.
  3. Create a data structure instance for each basic block that is detected.
  4. Stop parsing when the end of the function is detected.
- We use some global variables and flags to keep the state of the analysis clear. We rely on these flags to decide about the action to be taken at every loop iteration.
- The main difficulty remain in the follow-up of the multiple flags of the parser such that the implementation rmain correct and may be generalized.

Level 2: Graph reconstruction

- The notion of target we have chosen is a pointer to a data structure. In fact, we will have for each basic block two pointers. One of the pointers will next be hadeled so that it refers to the instance of basic block corresponding to the branching address, and the second one will refer to the continue address (the address of the following basic block if the branch operation is not performed or doesn't exist at the end of the basic block).
  The following is the data structure we use:

  ```
  typedef struct basic_block {
      int id;
      int size;
      char addr_start[5];
      struct basic_block *bb_branch;
      struct basic_block *bb_continue;
  } basic_block;
  ```

  The algorithm we choose is based on the pseudo code mentioned above.
  We choose to use this algorithm with this data structure because it seems to be more optimal in terms of memory usage ; the idea is to only preserve the independant elements, so that we can deduce other ones. For example :

  - From the "*bb_branch" we can access all the elements of the next basic block ; so we avoid to having them instantiated twice.
  - From the "size" and "addr_start" we can compute the address of the end of the basic block ; knowing that each instruction occupies 4 bytes.

- There are two main difficulties :

  - The first difficulty is related to linking the nodes in the graph (by assigning basic blocks addresses) at the same time the nodes are created. So, we have splitted the grapth reconstruction operation from the first steps, and we have chosen the most trivial way to doing it.
  - The second difficulty remains in the fact that some branching addresses are not within the scope of the ControleurPorte_step function. Which means that finding their corresponding basic blocks information resuires to switch to other parts of the file. But we have chosen to keep it simple and only preserve the branching address information in a global array (whose indexes refer to the same as the basic block indentifiers).

- The dynamic addressing consists in replacing the real start address of the basic block by an offset that allows to re-compute the address depending on a previous basic block address. Such an implementation must be handled properly so that the CFG reconstruction stay consistant.

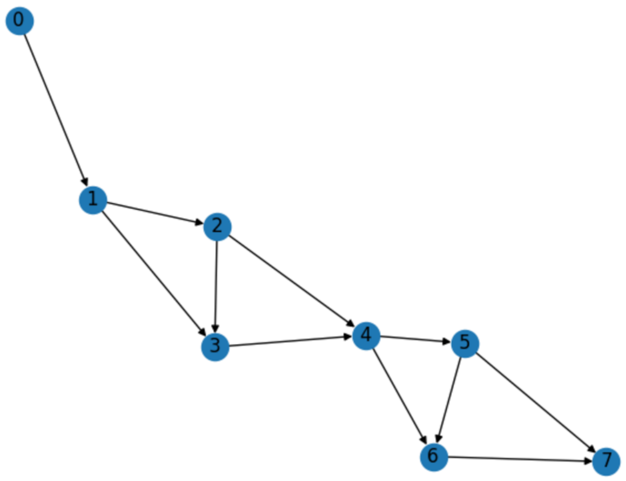

Figure 1: Plotting the first 7 basic blocks from the reconstructed CFG (resulted by matplotlib based on the created csv file, representing a reduced CFG format)