Project C++

NanoTekSpice

Koalab koala@epitech.eu

Abstract: *The purpose of this project is to create a digital electronic simulator.*

Contents

Chapter I

Digital Electronic

There is variety of programming languages, C, Lisp, Basic, APL, Intercal... They all have their specificities and may be efficient in different ways. They mainly rely on a compiler to work. This compiler will transform the code which is written in almost human language to a more primitive form.

This primitive form is the assembly language if seen as its human readable form, or machine language if seen as the microprocessor sees it.

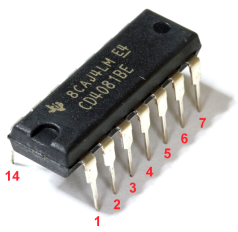
The same way there is a variety of programming languages, there is a lot of assembly languages: at least one per processor family, sometimes one per processor, sometimes several per processor.

Of course, knowing an assembly language for a microprocessor may not be enough to be efficient: a microprocessor is not a full machine. You may need to know how your machine works in order to create a usefull program in assembly: Which address for the graphic card? Which trap for this system call?

Under assembly languages and machine languages, there is the hardware itself. The hardware is built with digital electronic components: chipsets. Chipsets are tiny little functions with inputs and outputs that can be linked together to create more powerful functions, exactly like with software functions.

These functions are exclusively based on boolean logic. So it only gets and returns true or false values. In this project, you will have to create a graph. In this graph, nodes will be simulated digital electronic components, inputs or outputs.

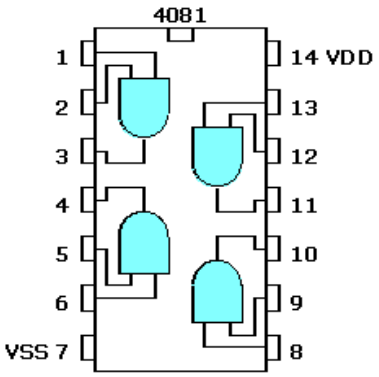
What does a chipset look like? Here is one:



Chapter II

Chipset

Here is an example of what's inside a simple chipsets: the 4081.



The 4081 is a little box that contains four AND gates. An AND gate has two inputs (A and B) and one output (S). The 4081 features 14 connectors.

- Pin 1 and pin 2 are inputs of one of the and gate, let's say gate 1.
- Pin 3 is the output of gate 1.
- Pin 4 is the output of gate 2.
- Pin 5 and 6 are input of the gate 2.
- Pin 7 (VSS) is here for electrical purpose. We will ignore it in this project.
- Pin 8 and 9 are inputs of gate 3.
- Pin 10 is the output of gate 3.
- Pin 11 is the output of gate 4.
- Pin 12 and 13 are inputs of gate 4.
- Pin 14 (VDD) is here for electrical purpose. We will ignore it in this project.

Here is the AND gate “truth table”:

A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

Note that this is absolutely equivalent to the following function:

```
1 bool and_gate(bool a, bool b)
2 {
3   return (a && b);
4 }
```

There is plenty of chipsets. Some of them are primitive, like the 4081, some others bring more complicated functions like counting or serialising functions. Microprocessors are one of the more complicated family of chipset existing by providing interpreters.

Of course, microprocessors are built with more simple chip, which happened to be built from components like the 4081 chipset. Everything can be reduced to boolean logic.

Chapter III

The Undefined State

The boolean algebra is composed of only two states: true and false. In the electronic world, true means VCC volt. VCC is the alimentation of the system. In a lot of case, VCC is 5 volt. In the electronic world, false means, in most cases, 0 volts.

It creates a problem we won't treat here: there is a lot of value under 0 and a lot above 5, and even an infinite between 0 and 5.

It also creates another problem we will have to treat: What if a component A needs to know the value S to compute its output, but S needs its output to be defined?

The answer cannot be true or false. The answer is that the value is undefined, a third state. So you will not compute pure boolean algebgra. You will have to integrate an undefined state.

Get out bool! Welcome enum!

Chapter IV

The project

NanoTekSpice is a logic simulator that is able to build a graph thanks to a configuration file, as well as inject values inside the graph to get results.

IV.1 The configuration file

IV.1.1 Example

Here is an example of configuration file. It contains a graph description.

```
#three inputs and gate
.chipsets:
input i0
input i1
input i2
4081 and0
output out

.links:
i0:1 and0:1
i1:1 and0:2

and0:3 and0:5
i2:1 and0:6

and0:4 out:1
```

IV.1.2 Description

The first part, which starts with the ".chipsets:" statement allows you to declare components that will be used by the program and name them. Some components may need a value. This value will be passed right after the name between parenthesis.

The second part, which starts with the ".links:" statement allows you to declare links between components. You have to precise which component you wish to link and which pin.

Space between keywords on the same line may be spaces or tabs. Newline terminates the statement.

Comments start with a '#' and finish with a newline. A comment can be either at the start of a line, or after an instruction.

Here are a few special components:

- **input** : Creates a value directly linked to the command line. The value may be 0 or 1 and must be given this way: ./nanotekspice circuit_file.ts input_name=input_value
- **clock** : Creates an input that works the same way as **input** except that its value is inverted after each simulation.
- **true** : This component have one pin that is always true.
- **false** : This component have one pin that is always false.
- **output** : Creates an output that writes its state on stdout at the end of each simulation. If there are several outputs, they must be ascii sorted.

IV.1.3 Errors

When one of the following cases happens, NanoTekSpice must raise an exception and stop the execution of the program neatly. It is forbidden to raise scalar exceptions. Moreover, exception classes must inherit from `std::exception` of the `STL`.

- The circuit file includes one or several lexical errors or syntactic errors.
- A component type is unknown
- A component name is unknown
- The given pin does not exist
- Several components share the same name
- Not every output is linked
- Missing input value on command line
- Unknown input specified by command line
- No chipset section
- No links section

Perhaps there are others errors. However, your simulator must never crash! (segfault, bus error, infinite loop ...)

IV.1.4 Execution

Your simulator must be able to run with a circuit file and inputs value passed as parameter. The simulator also reads on stdin a few commands:

- **exit** : It closes the program.
- **display** : It prints on stdout the value of all outputs sorted by name.
- **input=value** : It changes the value of an input. (Not a clock)
- **simulate** : It launches a simulation.
- **loop** : Simulate without prompting again until SIGINT.
- **dump** : Call the Dump method of every IComponent

Your simulator must launch one simulation directly after being launched, then display. Your simulator must handle CTRL+D correctly.

Now let's see some examples of execution.

```
cat -e or_gate.nts
.chipsets:
input a
input b
4071 or
output s
.links:
a:1 or:1
b:1 or:2
or:3 s:1
./nanotekspice or_gate.nts a=0 b=0
s=0
> a=1
> simulate
> display
s=1
> exit
C:\>
```

```
C:\> cat -e bad.nts
.chipsets:
input i
output o
.links:
C:\> ./nanotekspice bad.nts i=0
Output 'o' is not linked to anything.
C:\>
```

The error message is given as an example. Feel free to use yours instead.

IV.2 The parser

This project is probably your first try at parsing a configuration file. Thus, we will guide you through parsing steps. Here is the interface your parser must implement.

```
1  //
2  // Created by fourdr_b on 08/02/16.
3  //
4
5  #ifndef CPP_NANOTEKSPICE_PARSER_HPP
6  # define CPP_NANOTEKSPICE_PARSER_HPP
7
8  # include <string>
9  # include <vector>
10
11 namespace nts
12 {
13 enum class ASTNodeType : int
14 {
15    DEFAULT = -1,
16    NEWLINE = 0,
17    SECTION,
18    COMPONENT,
19    LINK,
20    LINK_END,
21    STRING
22 };
23
24 typedef struct s_ast_node
25 {
26   s_ast_node(std::vector<struct s_ast_node*> *children)
27     : children(children) { }
28   std::string    lexeme;
29   ASTNodeType    type;
30   std::string    value;
31   std::vector<struct s_ast_node*> *children;
32 } t_ast_node;
33
34 class IParser
35 {
36 public:
37   virtual void feed(std::string const& input) = 0;
38   virtual void parseTree(t_ast_node& root) = 0;
39   virtual t_ast_node *createTree() = 0;
40   virtual ~IParser() { }
41 };
42 }
43
44 #endif //CPP_NANOTEKSPICE_PARSER_HPP
```

The idea behind this parser is the same as presented by Victor Boudon in his slides about the 42sh parser (they will be given as annexes).

- **ASTNodeType** : The type of a node in the tree.
- **s_ast_node** : A node of the output tree.
- **Parser** : The actual interface.

- **feed(std string const&)** : Appends the string to the current input.
- **parseTree(t_ast_node&)** : Takes the AST root and goes through the whole tree.
- **createTree()** : Parses the input string to produce the ouput tree.

From this interface, one could wonder how to achieve a correct complete parser. Well, it is not that hard. You have to split the parsing in two steps : build the AST (Abstract Syntax Tree), then actually go through this tree, calling a function depending on the type of each node. As you might have understood from Victor's example, you will have to write / design your own grammar, which you will use to implement your parser. You will also have to write your own semantic actions, which are the functions associated with each one of the operators in your grammar. For example, imagine the grammar of a calculator, 'add' is a semantic action -> from a character ('+'), you extract a semantic meaning (the addition of two operands), and apply it.

The whole point of this exercise is to give you a basic understanding of what is a good parser. For reference, this is called a Recursive Descent Parser. Feel free to read anything you'd like about this kind of parsers, but be careful, you have to respect our interface.

Chapter V

Technical considerations

In order to help you with your code, we give you the following instructions that you have to respect. For each instruction, you must understand why this particular one is imposed. We are not providing these instructions randomly or to bore you to death!

V.1 The IComponent interface

Each of your component classes MUST implement the following IComponent interface:

```
// Kind of Advanced Language Assistant Laboratory 2006-2042
// Epitech 1999-2042
// Jason Brillante brilla_a brilla_b
//
// NanoTekSpice

#ifndef __ICOMPONENT_HPP__
# define __ICOMPONENT_HPP__

namespace nts
{
  enum Tristate
    {
      UNDEFINED = (-true),
      TRUE = true,
      FALSE = false
    };

  class IComponent
  {
  public:
    /// Compute value of the precised pin
    virtual nts::Tristate Compute(size_t pin_num_this = 1) = 0;

    /// Useful to link IComponent together
    virtual void SetLink(size_t pin_num_this,
            nts::IComponent &component,
            size_t pin_num_target) = 0;

    ///// Print on terminal the name of the component and
    //// the state of every pin of the current component
    /// The output won't be tested, but it may help you
    // as a trace.
    virtual void Dump(void) const = 0;

    virtual ~IComponent(void) { }
  };
}

#endif // __ICOMPONENT_HPP__
```

Component classes implementing the IComponent interface are the following ones. You can find manuals about these components in the same directory you found this subject.

- **4001** : Four NOR gates.
- **4008** : 4bits adder.
- **4011** : Four NAND gates.
- **4013** : Dual Flip Flop.
- **4017** : 10bits Johnson decade.
- **4030** : Four XOR gates.
- **4040** : 12 bits counter.
- **4069** : Six INVERTER gates.
- **4071** : Four OR gates.
- **4081** : Four AND gates.
- **4094** : 8bits shift register.
- **4514** : 4bits decoder.
- **4801** : Random access memory.
- **2716** : Read only memory. 2716 must takes a value when declared. In the .chipsets: section, you must precise a file that will contain the data written in the ROM: 2716 rom(cp_m.dat)

`Important : It is FORBIDDEN to manipulate pointers or references on each of these classes in your graph. You are allowed to manipulate pointers ONLY on IComponent.`

V.2 Creation of a new IComponent

You have to write a member function of a class **relevant** to your system that will allow you to create new components in a generic way. This function must have the following prototype:

```
1    IComponent * createComponent(const std::string &type, const std::string &value);
```

Depending on the value of the string passed as a parameter, the member function `createComponent` creates a new `IComponent` by calling one of the following `private` member function:

```
1    IComponent * create4001(const std::string &value) const;
2    IComponent * create4013(const std::string &value) const;
3    IComponent * create4040(const std::string &value) const;
4    // Etc...
```

In order to choose the right member function for the creation of the new `IComponent`, you MUST create and use an array (here, `vector` shows little interest) of pointers to member functions.

Chapter VI

Bonus

There is several bonuses available. Here are a few example:

- Terminal output component. Takes 8 bits as ascii code, and 1 bit which prints the character on stderr while switching from 0 to 1.
- Graphic output of the graph. (Feel free to add a .graphic: section inside your configuration file with graphics information.
- Graphic component “7seg”: Takes 4 bits as input. Display an hexadecimal number matching its entry.
- Graphic component “matrix”: Takes 8 bits as a color shade and takes 12 bits as pixel select. Output is a 64*64 pixels picture.
- i4004 support (Or any other microprocessor, even your own creation. Must be turing complete).
- Simulating a little computer.

Feel free to use SFML, SDL, Allegro, mlx or lapin for graphics. **For graphics only.**

Chapter VII

Instructions

You are more or less free to implement your machine how ever you want. However there are certain rules:

- Your program name must be “nanotekspice”.
- You must provide a static library “libnanotekspice.a” containing all your functions except the main function. This library will be used to test your IComponent system.
- All libraries except for `STL` are strictly prohibited. Graphic library can be used under bonus condition.
- You must use `STL` as much as you can. Your project must contains at least one container and one exception class from `STL`. The use of at least one algorithm from `STL` will be appreciated. Be very careful!
- The only functions of the `libc` that are authorized are those one that encapsulate system calls, and that don’t have a C++ equivalent.
- “split”, “strtowordtab”, and so on must **not** be used for parsing.
- Each value passed by copy instead of reference or by pointer must be justified. Otherwise, you’ll loose points.
- Each non `const` passed as parameter must be justified. Otherwise, you’ll loose points.
- Each member function or method that does not modify the current instance and which is not `const` must be justified. Otherwise, you’ll loose points.
- There are no `C++` norm. However if a code is reckoned to be unreadable or dirty, this code will be penalized. Be serious please!
- Keep an eye on this project instructions. It can change with time!

Chapter VIII

Turn in instructions

You must turn in you project in the repository provided by Epitech.
The repository name is cpp_nanotekspice.

Repository will be cloned at the exact hour of the end of the project, intranet `Epitech` being the reference.

Only the code from your repository will be graded during the oral defense.

Good luck!