B4 - Object-Oriented Programming

B-OOP-400

NanoTekSpice

Digital Electronics

3.0

EPITECH.

NanoTekSpice

binary name: nanotekspice
language: C++
compilation: via Makefile (all, clean, fclean, re), or CMake 3.17

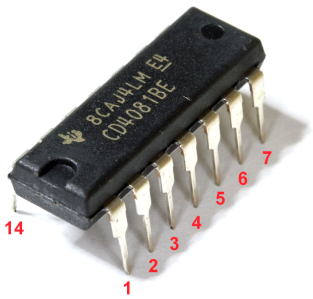
- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

There are many programming languages: C, Lisp, Basic, APL, Intercal…
Each has its specificities and may be efficient in a different way.
They generally rely on a compiler to work.
The compiler transforms code written in something similar to human language into a more primitive form.

This primitive form is called **assembly language** when displayed in a human-readable format, or **machine language** when displayed under the format read by the microprocessor.

Much like there are many programming languages, there are many assembly languages: at least one per processor family, sometimes even one per single processor.

Of course, knowing an assembly language for a microprocessor may not be enough to be efficient: processors aren't the only components in machines.
You may need to know how your machine works in order to create a useful assembly language: which address corresponds to the graphics card?
Which trap is assigned to this specific system call?



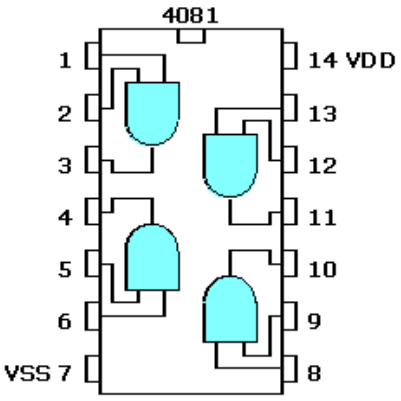
Hidden behind assembly and machine languages is the hardware itself.
Hardware is built from digital electronic components: chipsets.
Chipsets are tiny little functions with input and outputs that can be linked together to create more powerful functions, **just like in software**.

These functions are exclusively based on **boolean logic**, meaning the only inputs and outputs are `true` and `false`.

NanoTekSpice is a logic simulator that builds a graph (the nodes of which will be simulated digital electronic components, inputs or outputs) from a configuration file, and injects values into that graph to get results.

CHIPSETS

Here is an example of what's inside a simple chipset, the 4081:



The 4081 is a little box that contains four AND gates, with two inputs (A and B) and one output (S) each.
The 4081 features 14 connectors:

- pin 1 and pin 2 are inputs of one of the AND gates (gate 1),
- pin 3 is the output of gate 1,
- pin 4 is the output of gate 2,
- pin 5 and pin 6 are inputs of gate 2,
- pin 7 (VSS) is for electrical purposes; we will ignore it in this project,
- pin 8 and pin 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 pin 13 are inputs of gate 4,
- pin 14 (VOD) is for electrical purposes; we will ignore it in this project.

Here is the AND gate's **truth table**:

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

Note that this is exactly equivalent to the following function:

```
bool and_gate(bool a, bool b)
{
        return a && b;
}
```

There are plenty of chipsets. Some are primitive, like the 4081, while others represent more complicated functions which perform operations such as counting or serialising.
Microprocessors are one of the more complicated family of chipsets as they provide interpreters.

Of course, microprocessors are composed of simpler chipsets, **which are themselves composed of chipsets** like the 4081. Everything boils down to boolean logic.

The Undefined State

In boolean algebra, there are only two values: `true` and `false`.
In the electronic world, `true` means *VCC volt*. VCC is the power for the system, generally 5 volts.
`false`, in most cases, means 0 volts.

There is however, another case we have to take into account: what if a component A needs to know a value S to compute its output, but S hasn't been defined yet?

S cannot be set to `true` or `false`, as that would be guessing its value and could mean the rest of the system receives the wrong value.
The only solution is to say that S is `undefined`, a third state.
This means you won't be using pure boolean algebra, and will have to integrate an undefined state.

Bye bye `bool`, welcome `enum`!

The Configuration File

Here is a sample 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
```

The **first part**, starting with the **".chipsets:"** statement is used to declare and name components that will be used by the program.
Some components may need to be initialized with a value, which is provided after the name, between parenthesis.

The **second part**, starting with the **".links:"** statement is used to declare links between components.
Each link indicates which pin of which component are linked.

White space between keywords on a line may be spaces or tabs. Statements are newline-terminated.

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

Special Components

Here are a few special components:

- **input**: a component with a single pin directly linked to the command line. Its value is initialized to `undefined`.
- **clock**: a component that works like an **input**, except its value is inverted after each simulation.
- **true**: a component with a single pin that is always `true`.
- **false**: a component with a single pin that is always `false`.
- **output**: a component with a single pin used as the output of a circuit.

Errors

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

- The circuit file includes one or several lexical or syntactic errors.
- A component type is unknown.
- A component name is unknown.
- A requested pin does not exist.
- Several components share the same name.
- No chipsets in the circuit.

There may be other errors. However, your simulator must never crash (uncaught exception, segmentation fault, bus error, infinite loop…)!

Execution

Your simulator must be able to run with a circuit file passed as parameters.
The simulator also reads the standard input for the following commands:

- **exit**: closes the program.
- **display**: prints the current tick and the value of all inputs and outputs the standard output, each sorted by name in ASCII order.
- **input=value**: changes the value of an input. Possible `value` are `0`, `1` and `U`. This also apply to clocks.
- **simulate**: simulate a tick of the circuit.
- **loop**: continuously runs the simulation (`simulate`, `display`, `simulate`, …) without displaying a prompt, until SIGINT is received.
- **dump**: calls the `dump` method of every component. The output format is free.

Between commands, your program must display "`>`" as a prompt ('`>`' followed by a space).

When printing a value, if it is undefined, your program must display *"U"* (without the quotes).

When it reaches the end of its standard input (ever heard of `CTRL+D`?), the program must stop with status `0`.

Here are some sample executions:

Terminal

```
~/B-OOP-400> 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$
~/B-OOP-400> ./nanotekspice or_gate.nts
> b=0
> a=1
> simulate
> display
tick: 1
input(s):
  a: 1
  b: 0
output(s):
  s: 1
> exit
~/B-OOP-400> echo $?
0
```

Terminal

```
~/B-OOP-400> cat -e bad.nts
.chipsets:$
input i$
output o$
.links:$
a:1 o:1$
~/B-OOP-400> ./nanotekspice bad.nts
Unknow component name ‘a’.
~/B-OOP-400> echo $?
84
```

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

Terminal

```
~/B-OOP-400> cat -e clock.nts
.chipsets:$
clock cl$
output out$
.links:$
out:1 cl:1$
~/B-OOP-400> ./nanotekspice clock.nts
> display
tick: 0
input(s):
  cl: U
output(s):
  out: U
> cl=0
> display
tick: 0
input(s):
  cl: U
output(s):
  out: U
> simulate
> display
tick: 1
input(s):
  cl: 0
output(s):
  out: 0
> simulate
> display
tick: 2
input(s):
  cl: 1
output(s):
  out: 1
> simulate
> simulate
> simulate
> display
tick: 5
input(s):
  cl: 0
output(s):
  out: 0
> (CTRL+D)
~/B-OOP-400> echo $?
0
```

TECHNICAL CONSIDERATIONS

In order to help you through your implementation, we have designed the following instructions which you must follow.
It is important that you understand why each of these instructions exists.
We are not providing them for fun or to torture you!

THE `IComponent` INTERFACE

Each of your components **MUST** implement the following `IComponent` interface.

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

    class IComponent
    {
    public:
        virtual ~IComponent() = default;

        virtual void          simulate(std::size_t tick) = 0;
        virtual nts::Tristate compute(std::size_t pin) = 0;
        virtual void          setLink(std::size_t pin, nts::IComponent &other, std::
            size_t otherPin) = 0;
        virtual void          dump() const = 0;
    };
}
```

Here is a list of the component classes you must implement.
Manuals for these components are provided alongside this subject.

- **4001**: Four NOR gates.
- **4011**: Four NAND gates.
- **4030**: Four XOR gates.
- **4071**: Four OR gates.
- **4081**: Four AND gates.
- **4069**: Six INVERTER gates.
- **4008**: 4 bits adder.
- **4013**: Dual Flip Flop.
- **4017**: 10 bits Johnson decade.
- **4040**: 12 bits counter.
- **4094**: 8 bits shift register.
- **4512**: 8 channel data selector.
- **4514**: 4 bits decoder.
- **4801**: Random access memory.
- **2716**: Read only memory (memory initiliazed for `./rom.bin`).

You must also implement the **logger** component. It append the character computed from its inputs to the

file `./log.bin` on the positive-going edge of the `clock` when `inhibit` is false. More details in the provided NTS files.

It is FORBIDDEN to manipulate pointers or references to these classes in your graph.
You must **ONLY** manipulate pointers and references to `IComponent`.

CREATION OF A NEW `IComponent`

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

```
std::unique_ptr<nts::IComponent> createComponent(const std::string &type);
```

This function creates a new `nts::IComponent` according to its `type` parameter.
To do so, it could call one of the following `private` member functions:

```
std::unique_ptr<nts::IComponent> create4001() const noexcept;
std::unique_ptr<nts::IComponent> create4013() const noexcept;
std::unique_ptr<nts::IComponent> create4040() const noexcept;
// etc...
```

However, there are other techniques you may use to implement this **factory** (ever heard of **lambdas**?).

Bonus

This project is extensible with various bonuses.
Here are a few examples:

- Graphic output of the graph. Feel free to add a `.graphic` section to the configuration file with graphical information.
- Graphic component *"7seg"*. Uses 4 pins as inputs. Displays a hexadecimal number matching its input.
- Graphic component *"matrix"*. Uses 8 pins as a color shade and 12 bits as pixel selections. Outputs a 64x64 pixel picture.
- i4004 support (Or any other microprocessor, possibly one of your own design. Must be Turing complete).
- Simulating a little computer.

Feel free to use SFML, SDL, Allegro or any other library for 2D or 3D graphics.