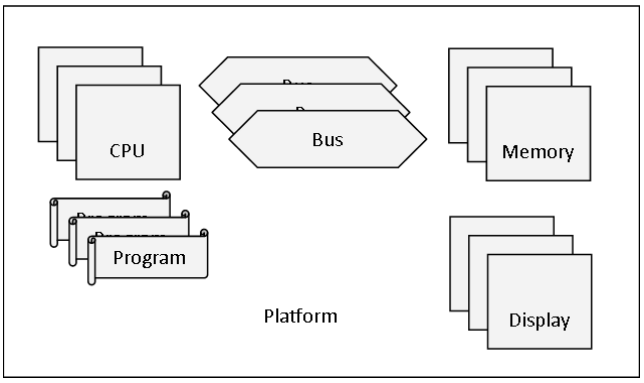
Introduction to Object-Oriented Programming with C++

Hardware platform simulator

You are going to model a simplified (and quite unrealistic) simulator of hardware components. The simulator will load textual definitions of hardware components; it will build a platform with the loaded components and simulate their behavior.

The platform is made of CPUs, busses, memories and displays. Each CPU runs a program.



The simulator has three main phases:

- *Phase 1*: The platform and all its components are loaded from files on the system. Each file contains a short text that defines a component.
- *Phase 2*: Some components need to be bind together (e.g. a memory binds to a bus), so all the bindings are made.
- *Phase 3*: The simulation runs.

The simulator runs for a given number of steps. The simulation principle is simple: at each step, the simulator "asks" the platform to `simulate()`.

Components

Each component, regardless of its type (CPU, memory…) knows how to `simulate()`.

All the components are loaded from text files. The general syntax of these files is as follows:

```
KEYWORD1: value1
KEYWORD2: value2
…
```

CPU

A CPU is a component and can simulate. It has a label. It contains and executes a program. It has a register where the computed values are stored. It also has a frequency, a number of cores and a currently active core (the cores of a CPU are never active simultaneously).

A CPU is loaded (created) from a text file, for example:

```
TYPE: CPU
LABEL: Main processing unit
CORES: 2
FREQUENCY: 4
PROGRAM: /path/to/program/file.txt
```

When loaded, the CPU loads its program.

Each time it is simulated (invocation of the `simulate()` method), the CPU "asks" *F* times the program to compute an instruction, and it writes each result in the register. *F* is the frequency of the CPU.

If the program has finished all the instructions, but the currently active core is not the last one, then the CPU (1) activates the next core, (2) "asks" the program to restart and to compute again the first instruction.
Otherwise, it does nothing.

A CPU can be `read()`. Each time it is read, it returns a DataValue (see later) which contains the first available value in the register. If there are no available values, then the returned DataValue has the *validity flag* set to `false`.

Bus

A Bus is a component and can simulate. It has a label and a width. It knows the source it is bound to. The source is a readable object that returns a DataValue each time it is read. The Bus contains *pending* DataValues and *ready* DataValues.

A Bus is loaded from a text file, for example:

```
TYPE: BUS
LABEL: My bus 1
WIDTH: 2
SOURCE: Main processing unit
```

The Bus can tell the label of its source (labels are meant to be unique in a platform). The Bus is also an object that can bind to a source object.

Each time it is simulated, the Bus moves all the *pending* values to *ready*, then it reads at most *W* times from its source, stopping as soon as an invalid value is obtained. *W* is the width of the Bus. All the read values (but the invalid one) are stored as *pending*.

A Bus can be read. Each time it is read, it returns the oldest DataValue that is ready. If there are no ready values, then it returns an *invalid* DataValue. The bus counts the number of times it is read.

Memory

A Memory is a component and can simulate. It has a label, a size and an access time. It knows the source it is bound to. The source is a readable object that returns a DataValue each time it is read. The Memory can store as many `double` values as its size. Each new value is stored at the next free location.

A Memory is loaded from a text file, for example:

```
TYPE: MEMORY
LABEL: DRAM 1
SIZE: 32
```

```
ACCESS: 3
SOURCE: My bus 1
```

The Memory can tell the label of its source. The Memory is also an object that can bind to a source object.

A Memory knows how to simulate, however it does not react every time: it reacts only one over *A* times. *A* is the access time of the Memory. When the memory reacts, it reads everything from its source. This means that the Memory keeps reading as long as valid values are obtained. It stores each valid value at the next free location.

How does the memory know the next free location? The memory is a circular buffer (*https://en.wikipedia.org/wiki/Circular_buffer*).

A Memory can be read. Each time it is read, it returns the next oldest DataValue. If there are no more values to read (i.e. all the stored values have been read once), then it returns an *invalid* DataValue.

Display

A Display is a component and can simulate. It has a refresh rate. It knows the source it is bound to. The source is a readable object that returns a DataValue each time it is read.

A Display is loaded from a text file, for example:

```
TYPE: DISPLAY
REFRESH: 8
SOURCE: DRAM 1
```

The Display can tell the label of its source. The Display is also an object that can bind to a source object.

A Display knows how to simulate, however it does not react every time: it reacts only one over *R* times. *R* is the refresh rate of the Display. When it reacts, it reads all the values from its source ("all" means that it stops as soon as it gets an *invalid* value). It prints all the read values to the standard output.

The Platform

A Platform contains other components. It is loaded from a text file, for example:

```
/path/to/cpu1/file.txt
/path/to/cpu2/file.txt
/path/to/bus/file.txt
/path/to/memory/file.txt
/path/to/display/file.txt
```

When loaded, the Platform loads all its components.

The Platform also handles the *phase 2* by making all the bindings.

Finally, the Platform knows how to simulate: it simply "asks" all its components to simulate. Order does not matter.

Caveat: Try to think in terms of usages (interfaces?) of the objects rather than in terms of object natures (CPU, Memory…).

Other elements

Data Value

A DataValue puts together a `double` `value` and a `boolean` flag `valid`. *Invalid* data values have the boolean flag set to `false`.

CPU Register

A Register is FIFO structure (*https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)*) with unlimited capacity, containing `double` values. It can tell whether it is empty or not. Each time it is read, it returns a value. Each time it is written, it stores a value.

CPU Program

A Program is made of Instructions. It loads entirely from a file, for example:

```
ADD 2 3
SUB 4.5 1.2
MUL 5.002 3
DIV 8.4 2
```

The instructions order matters and must be kept unchanged. The Program has an instruction pointer, initially set to the first instruction. Each time it is "asked" to `compute()`, the Program returns the current instruction, then moves the instruction pointer forward. If there are no more instructions, the Program returns the `NOP` instruction (see below).

A Program can tell whether it has executed all its instructions. A Program can also reset, which means rewinding the instruction pointer to the beginning.

Program Instruction

An Instruction has an operation code and two operands. There are five operation codes: `NOP`, `ADD`, `SUB`, `MUL`, `DIV`.

An Instruction can compute its result. The result of a `NOP` operation is always `0`, regardless of the values of the operands. The operands and the result are `double` values.

Your mission

Before rushing into the code, sketch the object model of your project. Draw a class diagram. No need to overcomplexify it: just sketch the classes, the most important relations and the most relevant methods. *Please consider discussing the object model with your lab supervisor*.

Implement the functions, classes, methods, etc. necessary to make the simulator work. Consider testing the components separately.

Load the provided `platform.txt` file and simulate for different numbers of steps.

Create your own platforms, load and simulate them.

To go further

Once the base milestones described above are met, you can go further with any of the items below. Order is not important, but chose wisely, as some developments may require more effort than others!

- Platform of platforms: a Platform is like other components, so it can be embedded into another platform. A Platform is now loaded from a file like this:

```
TYPE: PLATFORM
LABEL: Top platform
COMPONENT: /path/to/component1/file.txt
COMPONENT: /path/to/component2/file.txt
...
```

- You may have already added print traces to your objects. If not already done, this is the time. However, printing is controlled by a `verbose` flag, specified to the simulator. Each component prints at least the following pieces of information each time it is simulated:
  - Platform: label.
  - CPU: label, total number of cores, active core, frequency, instruction and its result.
  - Bus: label and number of accesses.
  - Memory: label, access time, number of unread values and number of free places.

- Do not hardcode the number of simulation steps: get it from the command line.

- How did you handle possible errors, e.g. a malformed file describing a component? Instead of simply printing traces, exiting the application or returning "default" values, make use of the exceptions (*http://www.cplusplus.com/doc/tutorial/exceptions/*). *Tricky*: If you have time, try also another approach: instead of *raising* exceptions, *return* them.
- Instead of fixing the number of operands for an instruction to 2, make it *variadic* (*https://en.wikipedia.org/wiki/Variadic*).

- Ignore spurious whitespaces in the files. For instance, you should be able to load this file:

```
 TYPE :  CPU

  LABEL: Main processing unit
CORES: 2
 FREQUENCY:  4

PROGRAM:  /path/to/prog/file.txt
```

- Add the notion of *simulation order*: add a *priority* to each component and a `PRIORITY` keyword to the description files. From now on, each simulation step takes into account the priority of the components. However, the order is not specified among components with the same priority.

- Memories can be initialized: add an `INIT` keyword to the description file of the memories, followed by some double values. For example: `INIT: 1.1 1.2 3.4 0.0123 9.87`.

- Memories can be of two types: RWM and ROM. Values inside ROM memories cannot be overwritten. When a ROM memory is read, it returns the next unread value. If all the values have been read, it returns an *invalid* value. Next time that it is read, it restarts from the beginning (it returns again the oldest value).
  Consider implementing this point alongside the previous one, otherwise how to you initialize ROM memories?

- Add a Serial component. It is a bit different from other components, because it binds to a "debuggable" component.
  - A "debuggable" component records all the instructions executed but not already debugged. When "asked" to debug, the component returns all the buffered instructions, then clears the buffer.
  - Make the CPU "debuggable".
  - Each time it simulates, the Serial writes all the instructions that it reads from its source to a file. Instructions are written like this: `SUB 3 4 = -1`.
  - The file describing a serial contains the `TYPE`, the `LABEL`, the `SOURCE` and the `TARGET` file.