02332 Compiler Construction
Mandatory Assignment: A Simple Hardware Simulator

Hand-out: 13. September 2022
Due: 14. October 2022 23:59
Hand-in via Learn platform in groups of 3-4 people
To hand in:

- All relevant source files (grammar and java)
- Your test examples.
- A small (1-2 page) report in PDF format that documents what you did for each task (including answers to questions of the task), possibly with code snippets.

A Hardware Description Language

We introduce a language for describing hardware circuits, and the task is to implement a simulator for that, i.e., that computes what this hardware would do for a given input. This is inspired a bit by existing hardware description languages, but a bit simplified and reduced to a few key concepts.

We have in principle three kinds of things:

- Signals (or wires) that at any time point have either a high value ("1") or a low value ("0").
- Combinatoric circuits: computing using logical functions on signals like conjunction (logical "and", written `&&`), disjunction ("or", written `||`), and negation ("not", written `!`).
- Latches, i.e., memory cells that are connected to one global clock, and at every clock tick read the current value of an incoming signal and output that until the next clock tick. For simplicity we do not consider here rising-edge vs. falling-edge etc., and they all initially have the value 0.

A simple example (see `01-hello-world.hw`):

```
.hardware helloworld
.inputs Reset
.outputs Oscillator
.latch Oscillator -> OscillatorM
.update
Oscillator = !OscillatorM && !Reset
.simulate
Reset=0000100
```

This describes a circuit with one input signal `Reset` and one output `Oscillator`. It has one latch that stores at each clock cycle the current value of `Oscillator` and outputs it on output `OscillatorM`. (Thus, initially, `OscillatorM` has value 0.) In every cycle the combinatoric circuit, i.e. `.update` section, computes the following function: The next value of the `Oscillator` is 1 if and only if the previous value `OscillatorM` and the `Reset` signal are both 0. Thus, if `Reset` is 0 all the time, then the oscillator will just alternate between 0 and 1 at each clock cycle, and whenever `Reset` is 1, the oscillator is also 0.

We require that the combinatoric circuit (i.e., the `.update` section) has no cycles (e.g. `O=!O` is forbidden), that every mentioned signal is either an input, or the output of a latch, or the result of a unique update. This means, there are no conflicting definitions for any signal, and there is a definition for every signal except inputs.

Finally, there is a `.simulate` section that gives a bitstring for every input signal; this bitstring means the value that the input signal should have during each clock cycle, so in the example, the `Reset` signal would only be set in the fifth cycle of the system.

The task of the simulator is now to compute the corresponding output signals also as such a bitstring. For this example we have:

```
0000100 Reset
1010010 Oscillator
```

where we can see that the Oscillator is alternating between 0 and 1 until the point where the `Reset` forces it to stay at 0. The first value of the `Oscillator` is 1, because the initial value of `OscillatorM` is 0 and reset is not set in the first cycle.

More complicated examples are included, namely a pedestrian traffic light that gives signals `Red` and `Green` for cars and `PRed` and `PGreen` for pedestrians, and has an input `Button` where pedestrians can request `PGreen`. However, the cars have green as long as the button is not pushed, and if the button is continuously pushed, the cars get green in the next phase. Finally, this is example is made more complicated where the cars have signals `Red`, `Yellow` and `Green`, and the traffic light has the cycle `Green, Yellow, Red, Red-Yellow, Green` and only during `Red`, the pedestrians have `PGreen`.

Week 3

We suggest to first try to get the example from the lecture in week 3 to run with ANTLR and Java. Please reach out to the teachers, if you have still issues to install it. You may well use this source code as a starting point for this project, since several things can be just adapted a bit. If you want to rename `impl` from the lecture to something else, note that there are many places in the grammar, `Makefile`, and `main.java`. Especially search and replace must be done with care since the Java keyword `implements` should not be replaced...

Also, note that if you change the grammar without also changing the `main.java` file accordingly, the Java compiler will complain. Since you may want to first play with the grammar a bit and use `make tree` which only compiles the ANTLR generated files, parses the first example file, and calls GRUN to show the parse tree. This is best for first testing and debugging your grammar.

Task 1: Design a grammar for our hardware description language. First, identify what are the tokens that the lexer should recognize, and what are the non-terminals of the grammar. It is a good idea to do this first on paper and only in a second step implement it in ANTLR. Consider the examples given, and possibly design another example. Implement your grammar in ANTLR and test it with the examples, using `make tree` to see the parse tree in GRUN.

Task 2: Check what ambiguities your grammar contains and whether they are resolved in the preferred way. In particular, negation binds stronger than conjunction, and conjunction binds stronger than disjunction. For instance

`A || !B && C` should mean `A || ((!B) && C)`

Check that this is parsed correctly on all examples, and check if there could be other ambiguities that are not covered by the examples.

Week 4

This week is about the design of the abstract syntax and implementing a visitor that transforms the parse trees into abstract syntax.

Suggestions: the abstract syntax for Boolean expressions can be similar to that of arithmetic expressions in our interpreter from the lecture. For latches and updates, we can just define data structures to hold the respective information. For the simulation inputs, it may make sense to have a structure `Trace` that holds the name of the signal and an array of Booleans (the value of the signal at different time points). The top level data structure could be circuit, and it could contain a `Trace` for every input and output signal, i.e., the input signal trace is given from the file, and the output signals will be computed by the simulator.

Task 3: Define Java classes for the Abstract Syntax; this does not yet have to include methods for computing anything, just member variables and the constructors.

Task 4: Write a visitor (similar to the `main.java` file from the lecture example) that generates the abstract syntax.

Week 5

Task 5 This week's task is to actually write the simulator, i.e., computing the values at each point in time.

Recommended design for this:

- For that we suggest to adapt the `Environment` class from the lecture, just to map all variables to `Boolean`, i.e., mapping each Signal to its current value.
- An `eval` function for boolean expressions of the combinatoric circuit.
- For latches have a method `initialize` that sets the value of the output signal to 0, and a method `nextCycle` that sets the value of the output value to the current value of the input signal.
- For each Update of the combinatoric circuit, a method `eval` that sets the value of the defined signal to the value that the given expression currently yields.
- For Traces: a `public String toString()` method to produce the output for that trace.
- For the circuit: to methods `initialize` and `nextCycle` that read the current input value and store it in the environment, that call the `initialize` or `nextCycle` of each latch, and then the `eval` of every update. Finally, it should store the value of all output signals appropriately. (Use again the `Trace` data structure!)
- The circuit should offer a method `runSimulator` that runs `initialize` and then n times `nextCycle` where n is the length of the simulator inputs. The simulator should quit with an error if for any input signal not bitstring is provided or if they do not all have the same length.

Task 6 Test that your simulator gives the correct outputs on all examples; you may well design your own examples.

Week 6

Task 6 There are several requirements on a hardware specification that the parser cannot check, for instance that there are no cyclic updates like `O=!O`. Implement checks for things like that and ensure understandable error messages are returned

Task 7 One may want to verify the following properties for the traffic light examples:

- Safety: the pedestrians have only green when the cars have red.
- Liveness: the cars never get stuck on red, even if the button is continuously pressed. In fact, I claim that in the simple traffic light, the cars have never red for two consecutive phase, in the more realistic example they never have red and/or yellow for four consecutive phases.

Implement these checks in the examples as an augmentation of the circuit by computing some output test signal that gives 1 at every timepoint where the checks are satisfied. This should be without modifying the simulator, but only modifying the examples.

Week 7

Lab week to complete the implementation and report.