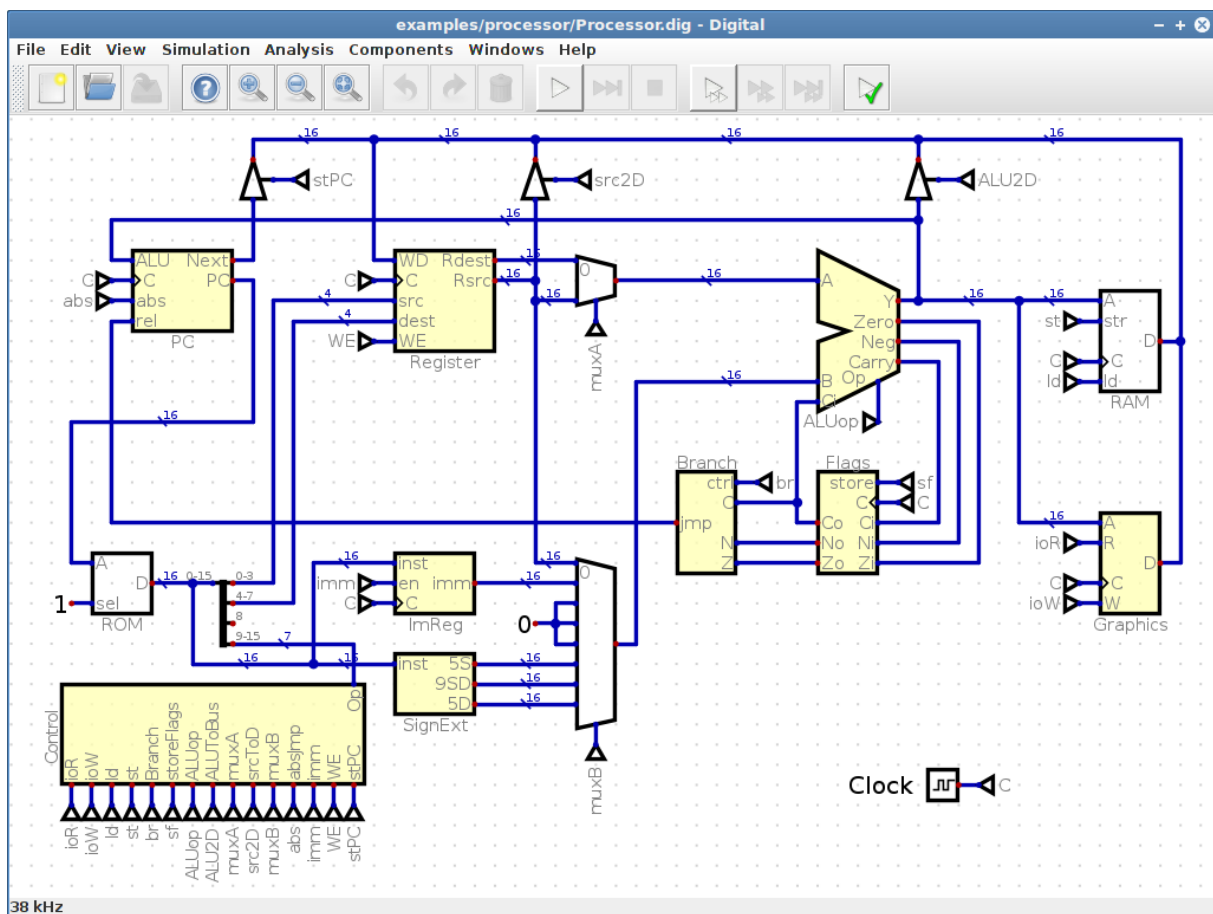


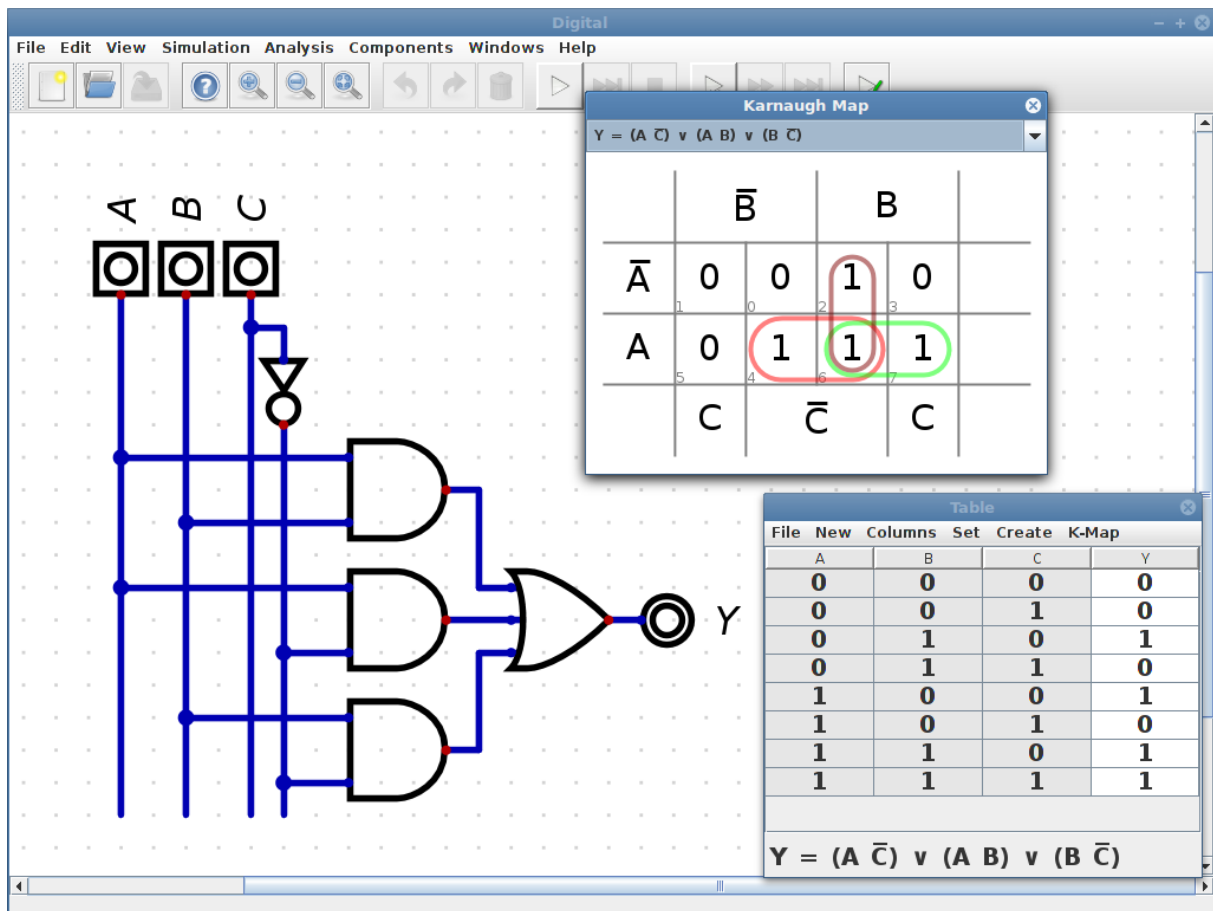
Download

build passing  codecov 57%

Digital

Digital is an easy-to-use digital logic designer and circuit simulator designed for educational purposes.





Download and Installation

There is no installation required, just unpack the *Digital.zip* file, which is available for download. On Linux start the shell script and on Windows and MacOS the JAR file can be started directly. A Java Runtime Environment (at least JRE 8) is required to run Digital. On Windows the easiest way to get Java is to install the binaries provided by the Eclipse Temurin project.

If there are any problems starting Digital on your system, please try to run Digital from a command line within the *Digital* folder:

```
1 java -jar Digital.jar
```

Features

These are the main features of Digital:

- Visualization of signal states with measurement graphs.

-
- Single gate mode to analyze oscillations.
 - Analysis and synthesis of combinatorial and sequential circuits.
 - Simple testing of circuits: You can create test cases and execute them to verify your design.
 - Many examples: From a transmission gate D-flip-flop to a complete (simple) MIPS-like single cycle CPU.
 - Includes a simple editor for finite state machines (FSM). A FSM can then be converted to a state transition table and a circuit implementing the FSM (See screenshot).
 - Contains a library with the most commonly used 74xx series integrated circuits.
 - Supports generic circuits. This allows the creation of circuits that can be parameterized when used. In this way, it is possible, for e.g., to create a barrel shifter with a selectable bit width.
 - Good performance: The example processor can be clocked at 120 kHz.
 - Supports large circuits: The “Conway’s Game of Life” example consists of about 2400 active components and works just fine.
 - It is possible to use custom components which are implemented in Java and packed in a jar file. See this example for details.
-
- Simple remote TCP interface which e.g. allows an assembler IDE to control the simulator.
 - Components can be described using VHDL or Verilog. The open source VHDL simulator ghdl needs to be installed to simulate a VHDL defined component, and the open source Verilog simulator Icarus Verilog is required to simulate a Verilog defined component.
 - A circuit can be exported to VHDL or Verilog. There is also direct support for the BASYS3 Board and the TinyFPGA BX board. See the documentation for details. The examples folder contains a variant of the example CPU, which runs on a BASYS3 board.
 - Direct export of JEDEC files which you can flash to a GAL16v8 or a GAL22v10. These chips are somewhat outdated (introduced in 1985!) but sufficient for beginners exercises, easy to understand and well documented. Also the ATF150x chips are supported which offer up to 128 macro-cells and in system programming. See the documentation for details.
 - SVG export of circuits, including a LaTeX/Inkscape compatible SVG version (see ctan)
 - No legacy code.
 - Good test coverage (about 80%; Neither the GUI tests nor the HDL simulator integration tests are running on the Travis-CI build servers, so CodeCov measures only about 50%). Almost all examples contain test cases which ensure that they work correctly.

The latest changes that have not yet been released are listed in the release notes. You can find the latest pre-release builds [here](#). In the pre release builds the automated GUI tests are usually not executed. All other tests, including the HDL tests, were executed without errors.

Documentation

The documentation is available in English, German, Spanish, Portuguese, French, Italian and simplified Chinese. It is still very incomplete but it contains a chapter “First Steps” which explains the basic usage of Digital. The documentation also contains a list of available 74xx chips and a list of available keyboard shortcuts.

Translations

So far Digital is available in English, German, Spanish, Portuguese, French, Italian and simplified Chinese. If someone wants to add a new translation, please let me know. I can provide you with a special file for translation. This file is much easier to translate than the files used directly by Digital. So you don't have to deal with GitHub or the Java source code. Simply add the respective translation of the English text to this file and send it back to me. If you want to know how to create the necessary files yourself, see [here](#).

Comments

If you want to send a bug report or feature request please use the GitHub issue tracker. This helps me to improve Digital, so do not hesitate. If you have general questions, you can also use the new GitHub Discussions to ask your questions without creating an issue.

It's also possible to send a private message to digital-simulator@web.de.

Motivation

Prior to the development of Digital, I used Logisim, developed by Carl Burch. If you are familiar with Logisim you will recognize the wire color scheme.

Logisim is a excellent and proven tool for teaching purposes, that has been actively developed until 2011. In 2013 Carl Burch has started the development of a new simulator called Toves. In his blog he explained why he decided to develop a new simulator instead of improving Logisim. In short: In his opinion, there are weaknesses in Logisim's architecture that are too difficult to overcome. Unfortunately, the development of Toves was discontinued at a very early stage.

In 2014, Carl Burch finally discontinued the development of Logisim. Since Logisim was released as open source, there are a number of forks to continue the work on Logisim:

-
- Logisim-evolution by people of a group of swiss institutes (Haute École Spécialisée Bernoise, Haute École du paysage, d'ingénierie et d'architecture de Genève, and Haute École d'Ingénierie et de Gestion du Canton de Vaud)
 - Logisim by Joseph Lawrance at Wentworth Institute of Technology, Boston, MA
 - Logisim-iitd from the Indian Institute of Technology Delhi
 - Logisim from the CS3410 course of the Cornell University

But as far as I know, these projects do not work on solving the architectural difficulties. They are more about adding features and fixing bugs. In Logisim-evolution, for example, a VHDL/Verilog export and a really nice FPGA board integration was added.

So I also decided to implement a new simulator completely from scratch and started the implementation of Digital in March 2016. In the meantime a development level has been reached which is comparable to Logisim. In some areas (performance, testing of circuits, circuit analysis, hardware support) Logisim has already been exceeded.

Below I would like to explain briefly the reasons which led me to start a new development:

Switch On

In Logisim there is no real “switching on” of a circuit. The simulation is running also while you are modifying it. This causes sometimes an unexpected behaviour. So it is possible to build a simple master-slave flip-flop which works fine. But after a circuit reset the flip-flop does not work anymore. Since the circuit is not switched on, there is no settling time to bring the circuit to a stable condition after its completion. A master-slave JK-flip-flop can only be implemented with a reset input, and this reset input needs to be activated to make the circuit operational.

To understand how Digital deals with this issue, you have to look at how the simulation works in Digital: Digital uses an event based simulator approach, i.e. each time a gate undergoes a change at one of its inputs, the new input states are read, however, the outputs of the gate are not updated instantly. Only when all gates involved have read their inputs, the outputs of all gates are updated. All gates seem to change synchronously, i.e. they seem to have all the exact same gate delay time. However, an undesirable feature of this approach is that even a simple RS flip-flop might not be able to reach a stable state. The same problem Logisim has.

To solve that problem, the “switching on” is introduced and a different simulation mode is used during the settling time right after switching on the circuit: Each time a gate undergoes a change at one of its inputs all gate inputs are read and their outputs are updated immediately. This happens gatewise in random order until no further changes occur and the circuit reaches a stable state. The gates appear to have random delay times now. This way, a master-slave flip-flop reaches a stable state after “switch on”, however, the final state is still undefined.

To start a circuit in a defined state a special reset gate is used. This gate has a single output which is low during settling time and goes high when settling time is over.

A disadvantage of this approach is the fact that a running simulation cannot be changed. In order to do so, the circuit needs be switched off, modified and switched on again. However, this procedure is also advisable for real circuits.

Oscillations

With Logisim it is hard to find the root cause for oscillating circuits. If Logisim detects an oscillation, a corresponding message is issued, but it is not possible to investigate the cause in more detail, so it is difficult to understand what happens.

The synchronous update of all gates, which have seen a change at one of their inputs may also cause oscillations in Digital. In such a case, the oscillation is detected and simulation stops. However, there is also a single gate mode which allows to propagate a signal change gate by gate. This feature allows to follow the way through the circuit. After each step, all gates with a change at one of their inputs are highlighted. This way you can see how a signal change propagates in a circuit, thus you are able to find the root cause of an oscillation.

Embedded circuits

Similar to Logisim, Digital also allows to embed previously saved circuits in new designs, so hierarchical circuits can be created. However, in Digital embedded circuits are included as often as the circuit is used. This is similar to a C program in which all function calls are compiled as inlined functions. And this is also similar to a real circuit: Each sub circuit is “physically present” as often as it is used in the design. Although this approach increases the size of the data structure of the simulation model in memory, it simplifies the simulation itself. Thus, for example, the inputs and outputs of an embedded circuit are not specifically treat, they simply don’t exist anymore after the formation of the simulation model. Even bidirectional connections can be implemented easily. Because of that approach for instance a embedded AND gate in a sub circuit behaves exactly like an AND gate inserted at top level although there is actually no difference between these two variants from the simulation models perspective. Logisim works somewhat different, which sometimes leads to surprises like unexpected signal propagation times and which makes it difficult to use bidirectional pins.

Performance

If a complete processor is simulated, it is possible to calculate the simulation without an update of the graphical representation. A simple processor (see example) can be simulated with a 120kHz clock

(Intel® Core™ i5-3230M CPU @ 2.60GHz), which is suitable also for more complex assembly exercises like Conway's Game of Life. There is a break gate having a single input. If this input changes from low to high this quick run is stopped. This way, an assembler instruction BRK can be implemented, which then can be used to insert break points in assembly language programs. So the debugging of assembly programs becomes very simple.

Debugging

In Logisim there is no easy way to debug an assembly program in a simulated processor. Digital offers a simple TCP-based remote control interface, so an assembler IDE can be used to control the simulator and load assembly programs into the simulated processor, start the program, perform single steps and so on. If a "single step" or a "run to next BRK instruction" is triggered by the assembly IDE, the actual used address of the program memory is returned to the assembler IDE. This allows the assembler IDE to highlight the actual executed instruction. In this way it is very easy to debug an assembly program executed by a simulated processor.

Circuit Synthesis

Logisim is able to generate combinatorial circuits from a truth table and vice versa. In Digital, this is also possible. In addition, a sequential circuit can be generated from an appropriate state transition table. You can specify both the transition circuit and the output circuit. The minimization of the expressions is done by the method of Quine and McCluskey. The truth table also can be derived from a circuit which contains simple combinatorial logic, D flip-flops or JK flip-flops, including the generation of the state transition table. Note, however, that a flip-flop build of combinatorial gates is not recognized as such. The analysis of sequential circuits only works with purely combinatorial logic combined with the build-in D or JK flip-flops. Once a truth table or state transition table has been created, a JEDEC file can be exported for a GAL16v8 or a GAL22v10. After that, this file can be flashed onto an appropriate GAL. As mentioned above these GALs are quite old but with 8/10 macro-cells sufficient for beginners exercises. If more macro-cells are required, see the PDF documentation for details on how to set up Digital to support the ATF1502 and ATF1504 CPLDs which offer 32/64 macro-cells and In-System Programming. It is also possible to export a circuit to VHDL or Verilog to run it on an FPGA. But the necessary HDL synthesis is sometimes a bit time-consuming and in my experience slows down the workflow in a lab exercise too much, especially if only simple circuits are required and the students change the circuit over and over again.

How do I get set up?

If you want to build Digital from the source code:

- At first clone the repository.
- A JDK (at least JDK 8) is required (either the Oracle JDK or OpenJDK)
- maven is used as build system, so the easiest way is to install maven.
- After that you can simply run `mvn install` to build Digital.
- Run `mvn site` to create a findbugs and a JaCoCo code coverage report.
- Most IDEs (Eclipse, NetBeans, IntelliJ) are able to import the `pom.xml` to create a project.

Contribution guidelines

- If you want to contribute, please open a GitHub issue first.
 - A discussion should avoid duplicate or unnecessary work.
 - Before you send a pull request, make sure that at least `mvn install` runs without errors.
- Don't introduce new findbugs issues.
- Try to keep the test coverage high. The target is a minimum of 80% test coverage.
- So far, there are only a few GUI tests, so that the overall test coverage is only slightly below 80%. Try to keep the amount of untested GUI code low.

Credits

Many thanks to the following persons for their help:

- Ivan de Jesus Deras Tabora from the Universidad Tecnológica Centroamericana in Honduras has implemented the verilog code generator and almost all the necessary verilog templates.
- Theldo Cruz Franqueira from the Pontifícia Universidade Católica de Minas Gerais in Brazil has provided the Portuguese translation.
- Ángel Millán from the Instituto de Educación Secundaria les Virgen de Villadiego in Peñafior (Sevilla), Spain has provided the Spanish translation.
- XinJun Ma (@itviewer) has provided the Chinese translation.
- Nicolas Maltais (@maltaish) has provided the French translation.
- Luca Cavallari (@psiwray) has provided the Italian translation.

Additional Screenshots

